Emmanuel Jeannot
Raymond Namyst
Jean Roman (Eds.)

LNCS 6852

Euro-Par 2011 Parallel Processing

17th International Conference, Euro-Par 2011
Bordeaux, France, August/September 2011
Proceedings, Part I

1 Part I

Springer

Lecture Notes in Computer Science 6852

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Emmanuel Jeannot Raymond Namyst
Jean Roman (Eds.)

Euro-Par 2011 Parallel Processing

17th International Conference, Euro-Par 2011
Bordeaux, France, August 29–September 2, 2011
Proceedings, Part I

Springer

Volume Editors

Emmanuel Jeannot
INRIA
351, Cours de la Libération
33405 Talence Cedex, France
E-mail: emmanuel.jeannot@inria.fr

Raymond Namyst
Université de Bordeaux, INRIA
351, Cours de la Libération
33405 Talence Cedex, France
E-mail: raymond.namyst@labri.fr

Jean Roman
Université de Bordeaux, INRIA
351, Cours de la Libération
33405 Talence Cedex, France
E-mail: jean.roman@inria.fr

ISSN 0302-9743
ISBN 978-3-642-23399-9
DOI 10.1007/978-3-642-23400-2

e-ISSN 1611-3349
e-ISBN 978-3-642-23400-2

Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011934379

CR Subject Classification (1998): F.1.2, C.3, C.2.4, D.1, D.4, I.6, G.1, G.2, F.2, D.3

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Euro-Par is an annual series of international conferences dedicated to the promotion and advancement of all aspects of parallel and distributed computing.

Euro-Par covers a wide spectrum of topics from algorithms and theory to software technology and hardware-related issues, with application areas ranging from scientific to mobile and cloud computing.

Euro-Par provides a forum for the introduction, presentation and discussion of the latest scientific and technical advances, extending the frontier of both the state of the art and the state of the practice.

The main audience of Euro-Par are researchers in academic institutions, government laboratories and industrial organizations. Euro-Par's objective is to be the primary choice of such professionals for the presentation of new results in their specific areas. As a wide-spectrum conference, Euro-Par fosters the synergy of different topics in parallel and distributed computing. Of special interest are applications which demonstrate the effectiveness of the main Euro-Par topics.

In addition, Euro-Par conferences provide a platform for a number of accompanying, technical workshops. Thus, smaller and emerging communities can meet and develop more focussed topics or as-yet less established topics.

Euro-Par 2011 was the 17th conference in the Euro-Par series, and was organized by the INRIA (The French National Institute for Research in Computer Science and Control) Bordeaux Sud-Ouest center and LaBRI (Computer Science Laboratory of Bordeaux). Previous Euro-Par conferences took place in Stockholm, Lyon, Passau, Southampton, Toulouse, Munich, Manchester, Paderborn, Klagenfurt, Pisa, Lisbon, Dresden, Rennes, Las Palmas, Delft and Ischia. Next year the conference will take place in Rhodes, Greece. More information on the Euro-Par conference series and organization is available on the wesite http://www.europar.org.

The conference was organized in 16 topics. This year we introduced one new topic (16: GPU and Accelerators Computing) and re-introduced the application topic (15: High-Performance and Scientific Applications). The paper review process for each topic was managed and supervised by a committee of at least four persons: a Global Chair, a Local Chair, and two Members. Some specific topics with a high number of submissions were managed by a larger committee with more members. The final decisions on the acceptance or rejection of the submitted papers were made in a meeting of the Conference Co-chairs and Local Chairs of the topics.

The call for papers attracted a total of 271 submissions, representing 41 countries (based on the corresponding authors' countries). A total number of 1,065 review reports were collected, which makes an average of 3.93 review reports per paper. In total 81 papers were selected as regular papers to be presented at

the conference and included in the conference proceedings, representing 27 countries from all continents, an yielding an acceptance rate of 29.9%. Three papers were selected as distinguished papers. These papers, which were presented in a separate session, are:

1. Lakshminarasimhan, Neil Shah, Stephane Ethier, Scott Klasky, Rob Latham, Rob Ross and Nagiza F. Samatova "Compressing the Incompressible with ISABELA: In-situ Reduction of Spatio-Temporal Data"
2. Aurelien Bouteiller, Thomas Herault, George Bosilca and Jack J. Dongarra "Correlated Set Coordination in Fault-Tolerant Message Logging Protocols"
3. Edgar Solomonik and James Demmel "Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms".

Euro-Par 2011 was very happy to present three invited speakers of high international reputation, who discussed important developments in very interesting areas of parallel and distributed computing:

1. Pete Beckman (Argonne National Laboratory and the University of Chicago), "Facts and Speculations on Exascale: Revolution or Evolution?"
2. Toni Cortes Computer Architecture Department (DAC) in the Universitat Politècnica de Catalunya, Spain), "Why Trouble Humans? They Do Not Care"
3. Alessandro Curioni (IBM, Zurich Research Laboratory, Switzerland), "New Scalability Frontiers in Ab-Initio Molecular Dynamics"

In this edition, 12 workshops were held in conjunction with the main track of the conference. These workshops were:

1. CoreGRID/ERCIM Workshop on Grids, Clouds and P2P Computing (CGWS 2011)
2. Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar 2011)
3. High-Performance Bioinformatics and Biomedicine (HiBB)
4. System-Level Virtualization for High Performance Computing (HPCVirt 2011)
5. Algorithms and Programming Tools for Next-Generation High-Performance Scientific Software (HPSS 2011)
6. Managing and Delivering Grid Services (MDGS)
7. UnConventional High-Performance Computing 2011 (UCHPC 2011)
8. Cloud Computing Projects and Initiatives (CCPI)
9. Highly Parallel Processing on a Chip (HPPC 2011)
10. Productivity and Performance (PROPER 2011)
11. Resiliency in High-Performance Computing (Resilience) in Clusters, Clouds, and Grids
12. Virtualization in High-Performance Cloud Computing (VHPC 2011)

The 17th Euro-Par conference in Bordeaux was made possible thanks to the support of many individuals and organizations. Special thanks are due to the authors of all the submitted papers, the members of the topic committees, and all

the reviewers in all topics, for their contributions to the success of the conference. We also thank the members of the Organizing Committee and people of the Sud Congrès Conseil. We are grateful to the members of the Euro-Par Steering Committee for their support. We acknowledge the help we had from Dick Epema of the organization of Euro-Par 2009 and Pasqua D'Ambra and Domenico Talia of the organization of Euro-Par 2010. A number of institutional and industrial sponsors contributed toward the organization of the conference. Their names and logos appear on the Euro-Par 2011 website http://europar2011.bordeaux.inria.fr/

It was our pleasure and honor to organize and host Euro-Par 2011 in Bordeaux. We hope all the participants enjoyed the technical program and the social events organized during the conference.

August 2011

Emmanuel Jeannot
Raymond Namyst
Jean Roman

Organization

Euro-Par Steering Committee

Chair

Chris Lengauer	University of Passau, Germany

Vice-Chair

Luc Bougé	ENS Cachan, France

European Respresentatives

José Cunha	New University of Lisbon, Portugal
Marco Danelutto	University of Pisa, Italy
Emmanuel Jeannot	INRIA, France
Paul Kelly	Imperial College, UK
Harald Kosch	University of Passau, Germany
Thomas Ludwig	University of Heidelberg, Germany
Emilio Luque	Autonomous University of Barcelona, Spain
Tomàs Margalef	Autonomous University of Barcelona, Spain
Wolfgang Nagel	Dresden University of Technology, Germany
Rizos Sakellariou	University of Manchester, UK
Henk Sips	Delft University of Technology, The Netherlands
Domenico Talia	University of Calabria, Italy

Honorary Members

Ron Perrott	Queen's University Belfast, UK
Karl Dieter Reinartz	University of Erlangen-Nuremberg, Germany

Observers

Christos Kaklamanis	Computer Technology Institute, Greece

Euro-Par 2011 Organization

Conference Co-chairs

Emmanuel Jeannot	INRIA, France
Raymond Namyst	University of Bordeaux, France
Jean Roman	INRIA, University of Bordeaux, France

Local Organizing Committee

Olivier Aumage	INRIA, France
Emmanuel Agullo	INRIA, France
Alexandre Denis	INRIA, France
Nathalie Furmento	CNRS, France
Laetitia Grimaldi	INRIA, France
Nicole Lun	LaBRI, France
Guillaume Mercier	University of Bordeaux, France
Elia Meyre	LaBRI France

Euro-Par 2011 Program Committee

Topic 1: Support Tools and Environments

Global Chair

Rosa M. Badia	Barcelona Supercomputing Center and CSIC, Spain

Local Chair

Fabrice Huet	University of Nice Sophia Antipolis, France

Members

Rob van Nieuwpoort	VU University Amsterdam, The Netherlands
Rainer Keller	High-Performance Computing Center Stuttgart, Germany

Topic 2: Performance Prediction and Evaluation

Global Chair

Shirley Moore	University of Tennessee, USA

Local Chair

Derrick Kondo	INRIA, France

Members

Giuliano Casale	Imperial College London, UK
Brian Wylie	Jülich Supercomputing Centre, Germany

Topic 3: Scheduling and Load-Balancing

Global Chair

Leonel Sousa	INESC-ID/Technical University of Lisbon, Portugal

Local Chair

Frédéric Suter	IN2P3 Computing Center, CNRS, France

Members

Rizos Sakellariou	University of Manchester, UK
Oliver Sinnen	University of Auckland, New Zealand
Alfredo Goldman	University of São Paulo, Brazil

Topic 4: High Performance Architectures and Compilers

Global Chair

Mitsuhisa Sato	University of Tsukuba, Japan

Local Chair

Denis Barthou	University of Bordeaux, France

Members

Pedro Diniz	INESC-ID, Portugal
P. Saddayapan	Ohio State University, USA

Topic 5: Parallel and Distributed Data Management

Global Chair

Salvatore Orlando	Università Ca' Foscari Venezia, Italy

Local Chair

Gabriel Antoniu	INRIA, France

Members

Amol Ghoting	IBM T. J. Watson Research Center, USA
Maria S. Perez	Universidad Politecnica de Madrid, Spain

Topic 6: Grid, Cluster and Cloud Computing

Global Chair

Ramin Yahyapour	TU Dortmund University, Germany

Local Chair

Christian Pérez	INRIA, France

Members

Erik Elmroth	Umeå University, Sweden
Ignacio M. Llorente	Complutense University of Madrid, Spain
Francesc Guim	Intel, Portland, USA
Karsten Oberle	Alcatel-Lucent, Bell Labs, Germany

Topic 7: Peer to Peer Computing

Global Chair

Pascal Felber	University of Neuchâtel, Switzerland

Local Chair

Olivier Beaumont	INRIA, France

Members

Alberto Montresor	University of Trento, Italy
Amitabha Bagchi	Indian Institute of Technology Delhi, India

Topic 8: Distributed Systems and Algorithms

Global Chair

Dariusz Kowalski	University of Liverpool, UK

Local Chair

Pierre Sens	University Paris 6, France

Members

Antonio Fernandez Anta	IMDEA Networks, Spain
Guillaume Pierre	VU University Amsterdam, The Netherlands

Topic 9: Parallel and Distributed Programming

Global Chair

Pierre Manneback	University of Mons, Belgium

Local Chair

Thierry Gautier	INRIA, France

Members

Gudula Rünger	Technical University of Chemnitz, Germany
Manuel Prieto Matias	Universidad Complutense de Madrid, Spain

Topic 10: Parallel Numerical Algorithms

Global Chair

Daniela di Serafino	Second University of Naples and ICAR-CNR, Italy

Local Chair

Luc Giraud	INRIA, France

Members

Martin Berzins	University of Utah, USA
Martin Gander	University of Geneva, Switzerland

Topic 11: Multicore and Manycore Programming

Global Chair

Sabri Pllana	University of Vienna, Austria

Local Chair

Jean-François Méhaut	University of Grenoble, France

Members

Eduard Ayguade	Technical University of Catalunya and Barcelona Supercomputing Center, Spain
Herbert Cornelius	Intel, Germany
Jacob Barhen	Oak Ridge National Laboratory, USA

Topic 12: Theory and Algorithms for Parallel Computation

Global Chair

Arnold Rosenberg	Colorado State University, USA

Local Chair

Frédéric Vivien	INRIA, France

Members

Kunal Agrawal	Washington University in St Louis, USA
Panagiota Fatourou	University of Crete, Greece

Topic 13: High Performance Network and Communication

Global Chair

Jesper Träff	University of Vienna, Austria

Local Chair

Brice Goglin	INRIA, France

Members

Ulrich Bruening	University of Heidelberg, Germany
Fabrizio Petrini	IBM, USA

Topic 14: Mobile and Ubiquitous Computing

Global Chair

Pedro Marron	Universität Duisburg-Essen, Germany

Local Chair

Eric Fleury	INRIA, France

Members

Torben Weis	University of Duisburg-Essen, Germany
Qi Han	Colorado School of Mines, USA

Topic 15: High Performance and Scientific Applications

Global Chair

Esmond G. Ng	Lawrence Berkeley National Laboratory, USA

Local Chair

Olivier Coulaud	INRIA, France

Members

Kengo Nakajima	University of Tokyo, Japan
Mariano Vazquez	Barcelona Supercomputing Center, Spain

Topic 16: GPU and Accelerators Computing

Global Chair

Wolfgang Karl	University of Karlsruhe, Germany

Local Chair

Samuel Thibault	University of Bordeaux, France

Members

Stan Tomov	University of Tennessee, USA
Taisuke Boku	University of Tsukuba, Japan

Euro-Par 2011 Referees

Muresan Adrian
Kunal Agrawal
Emmanuel Agullo
Toufik Ahmed
Taner Akgun
Hasan Metin Aktulga
Sadaf Alam
George Almasi
Francisco Almeida
Jose Alvarez Bermejo
Brian Amedro
Nazareno Andrade
Artur Andrzejak
Luciana Arantes
Mario Arioli
Ernest Artiaga
Rafael Asenjo
Romain Aubry
Cédric Augonnet
Olivier Aumage
Eduard Ayguade
Rosa M. Badia
Amitabha Bagchi
Michel Bagein
Enes Bajrovic
Allison Baker
Pavan Balaji
Sorav Bansal
Jorge Barbosa
Jacob Barhen
Denis Barthou
Rajeev Barua
Francoise Baude
Markus Bauer
Ewnetu Bayuh Lakew
Olivier Beaumont
Vicenç Beltran
Joanna Berlinska
Martin Berzins
Xavier Besseron
Vartika Bhandari
Marina Biberstein
Paolo Bientinesi
Aart Bik
David Boehme
Maria Cristina Boeres
Taisuke Boku
Matthias Bollhoefer
Erik Boman
Michael Bond
Francesco Bongiovanni
Rajesh Bordawekar
George Bosilca
François-Xavier Bouchez
Marin Bougeret
Aurelien Bouteiller
Hinde Bouziane
Fabienne Boyer

Ivona Brandic
Francisco Brasileiro
David Breitgand
Andre Brinkman
François Broquedis
Ulrich Bruening
Rainer Buchty
j. Mark Bull
Aydin Buluc
Alfredo Buttari
Edson Caceres
Agustin Caminero
Yves Caniou
Louis-Claude Canon
Gabriele Capannini
Pablo Carazo
Alexandre Carissimi
Giuliano Casale
Henri Casanova
Simon Caton
José María Cela
Christophe Cerin
Ravikesh Chandra
Andres Charif-Rubial
Fabio Checconi
Yawen Chen
Gregory Chockler
Vincent Cholvi
Peter Chronz
IHsin Chung
Marcelo Cintra
Vladimir Ciric
Pierre-Nicolas Clauss
Sylvain Collange
Denis Conan
Arlindo Conceicao
Massimo Coppola
Julita Corbalan
Herbert Cornelius
Toni Cortes
Olivier Coulaud
Ludovic Courtès
Raphael Couturier
Tommaso Cucinotta
Angel Cuevas
Pasqua D'Ambra
Anthony Danalis
Vincent Danjean
Eric Darve
Sudipto Das
Ajoy Datta
Patrizio Dazzi
Pablo de Oliveira Castro
César De Rose
Ewa Deelman
Olivier Delgrange
Alexandre Denis
Yves Denneulin
Frederic Desprez
Gérard Dethier
Daniela di Serafino
François Diakhaté
James Dinan
Nicholas Dingle
Pedro Diniz
Alastair Donaldson
Antonio Dopico
Matthieu Dorier
Niels Drost
Maciej Drozdowski
Lúcia Drummond
Peng Du
Cedric du Mouza
Vitor Duarte
Philippe Duchon
Jörg Dümmler
Alejandro Duran
Pierre-François Dutot
Partha Dutta
Eiman Ebrahimi
Rudolf Eigenmann
Jorge Ejarque Artigas
Vincent Englebert
Dominic Eschweiler
Yoav Etsion
Lionel Eyraud-Dubois
Flavio Fabbri
Fabrizio Falchi
Catherine Faron Zucker
Montse Farreras

Panagiota Fatourou
Hugues Fauconnier
Mathieu Faverge
Gilles Fedak
Dror G. Feitelson
Pascal Felber
Florian Feldhaus
Marvin Ferber
Juan Fernández
Antonio Fernández Anta
Ilario Filippini
Salvatore Filippone
Eric Fleury
Aislan Foina
Pierre Fortin
Markos Fountoulakis
Rob Fowler
Vivi Fragopoulou
Felipe França
Emílio Francesquini
Sébastien Frémal
Davide Frey
Wolfgang Frings
Karl Fuerlinger
Akihiro Fujii
Nathalie Furmento
Edgar Gabriel
Martin Gaedke
Georgina Gallizo
Efstratios Gallopoulos
Ixent Galpin
Marta Garcia
Thierry Gautier
Stéphane Genaud
Chryssis Georgiou
Abdou Germouche
Michael Gerndt
Claudio Geyer
Amol Ghoting
Nasser Giacaman
Mathieu Giraud
Daniel Gmach
Brice Goglin
Spyridon Gogouvitis
Alfredo Goldman
Maria Gomes
Jose Gómez
Jose Gonzalez
José Luis González García
Rafael Gonzalez-Cabero
David Goudin
Madhusudhan Govindaraju
Maria Gradinariu
Vincent Gramoli
Fabíola Greve
Laura Grigori
Olivier Gruber
Serge Guelton
Gael Guennebaud
Stefan Guettel
Francesc Guim
Ronan Guivarch
Jens Gustedt
Antonio Guzman Sacristan
Daniel Hackenberg
Azzam Haidar
Mary Hall
Greg Hamerly
Qi Han
Toshihiro Hanawa
Mauricio Hanzich
Paul Hargrove
Masae Hayashi
Jiahua He
Eric Heien
Daniel Henriksson
Ludovic Henrio
Sylvain Henry
Francisco Hernandez
Enric Herrero
Pieter Hijma
Shoichi Hirasawa
Torsten Hoefler
Jeffrey Hollingsworth
Sebastian Holzapfel
Mitch Horton
Guillaume Houzeaux
Jonathan Hu
Ye Huang
Guillaume Huard

Fabrice Huet
Kévin Huguenin
Sascha Hunold
Costin Iancu
Aleksandar Ilic
Alexandru Iosup
Umer Iqbal
Kamil Iskra
Takeshi Iwashita
Julien Jaeger
Emmanuel Jeannot
Ali Jehangiri
Hideyuki Jitsumoto
Josep Jorba
Prabhanjan Kambadur
Yoshikazu Kamoshida
Mahmut Kandemir
Tejas Karkhanis
Wolfgang Karl
Takahiro Katagiri
Gregory Katsaros
Joerg Keller
Rainer Keller
Paul Kelly
Roelof Kemp
Michel Kern
Ronan Keryell
Christoph Kessler
Slava Kitaeff
Cristian Klein
Yannis Klonatos
Michael Knobloch
William Knottenbelt
Kleopatra Konstanteli
Miroslaw Korzeniowski
Dariusz Kowalski
Stephan Kraft
Sriram Krishnamoorthy
Diwakar Krishnamurthy
Rajasekar Krishnamurthy
Vinod Kulathumani
Raphael Kunis
Tilman Küstner
Felix Kwok
Dimosthenis Kyriazis
Renaud Lachaize
Ghislain Landry Tsafack
Julien Langou
Stefan Lankes
Lars Larsson
Alexey Lastovetsky
Guillaume Latu
Stevens Le Blond
Bertrand Le Cun
Erwan Le Merrer
Adrien Lèbre
Rich Lee
Erik Lefebvre
Arnaud Legrand
Christian Lengauer
Daniele Lezzi
Wubin Li
Charles Lively
Welf Loewe
Sebastien Loisel
João Lourenço
Kuan Lu
Jose Luis Lucas Simarro
Mikel Lujan
Ewing Lusk
Piotr Luszczek
Ignacio M. Llorente
Jason Maassen
Edmundo Madeira
Anirban Mahanti
Scott Mahlke
Sidi Mahmoudi
Nicolas Maillard
Constantinos Makassikis
Pierre Manneback
Loris Marchal
Ismael Marín
Mauricio Marin
Osni Marques
Erich Marth
Jonathan Martí
Xavier Martorell
Naoya Maruyama
Fabien Mathieu
Rafael Mayo

Abdelhafid Mazouz
Jean-François Méhaut
Wagner Meira
Alba Cristina Melo
Massimiliano Meneghin
Claudio Meneses
Andreas Menychtas
Jose Miguel-Alonso
Milan Mihajlovic
Alessia Milani
Cyriel Minkenberg
Neeraj Mittal
Flávio Miyazawa
Hashim Mohamed
Sébastien Monnet
Jesus Montes
Alberto Montresor
Shirley Moore
Matteo Mordacchini
Jose Moreira
Achour Mostefaoui
Miguel Mosteiro
Gregory Mounié
Xenia Mountrouidou
Hubert Naacke
Priya Nagpurkar
Kengo Nakajima
Jeff Napper
Akira Naruse
Bassem Nasser
Rajib Nath
Angeles Navarro
Philippe O. A. Navaux
Marco Netto
Marcelo Neves
Esmond Ng
Yanik Ngoko
Jean-Marc Nicod
Bogdan Nicolae
Dimitrios Nikolopoulos
Sébastien Noël
Ramon Nou
Alberto Nuñez
John O'Donnell
Satoshi Ohshima
Ariel Oleksiak
Stephen Olivier
Ana-Maria Oprescu
Anne-Cecile Orgerie
Salvatore Orlando
Per-Olov Ostberg
Herbert Owen
Sergio Pacheco Sanchez
Gianluca Palermo
George Pallis
Nicholas Palmer
Jairo Panetta
Alexander Papaspyrou
Michael Parkin
Davide Pasetto
George Pau
Christian Perez
Maria Perez-Hernandez
Francesca Perla
Jean-Jacques Pesqué
Franck Petit
Fabrizio Petrini
Frédéric Pétrot
Guillaume Pierre
Jean-Francois Pineau
Luis Piñuel
Jelena Pjesivac-Grbovic
Kassian Plankensteiner
Oscar Plata
Sabri Pllana
Leo Porter
Carlos Prada-Rojas
Manuel Prieto Matias
Radu Prodan
Christophe Prud'homme
Vivien Quema
Enrique Quintana-Ortí
Rajmohan Rajaraman
Lavanya Ramakrishnan
Pierre Ramet
Praveen Rao
Vinod Rebello
Pablo Reble
Sasank Reddy
Veronika Rehn-Sonigo

Giuseppe Reina
Olivier Richard
Etienne Riviere
Victor Robles
Thomas Röblitz
Jean-Louis Roch
David Rodenas
Ivan Rodero
Mathilde Romberg
Arnold Rosenberg
Diego Rossinelli
Philip Roth
Atanas Rountev
Francois-Xavier Roux
Jan Sacha
Ponnuswamy Sadayappan
Rizos Sakellariou
Tetsuya Sakurai
Alberto Sanchez
Jesus Sanchez
Frode Sandnes
Idafen Santana
Mitsuhisa Sato
Erik Saule
Robert Sauter
Bruno Schulze
Michael Schwind
Mina Sedaghat
Frank Seinstra
Pierre Sens
Damian Serrano
Patricia Serrano-Alvarado
Javier Setoain
Muhammad Shafique
Abhishek Sharma
Rémi Sharrock
Hemant Shukla
Christian Siebert
Juergen Sienel
Frederique Silber-Chaussumier
Claudio Silvestri
Luca Silvestri
Oliver Sinnen
Raül Sirvent
João Sobral
Fengguang Song
Siang Song
Borja Sotomayor
Leonel Sousa
Daniel Spooner
Anton Stefanek
Manuel Stein
Mark Stillwell
Corina Stratan
Frederic Suter
Petter Svärd
Guillermo Taboada
Daisuke Takahashi
Domenico Talia
Jie Tao
Issam Tarrass
Osamu Tatebe
Shirish Tatikonda
Andrei Tchernykh
Marc Tchiboukdjian
Cedric Tedeschi
Enric Tejedor Saavedra
Christian Tenllado
Radu Teodorescu
Alexandre Termier
Dan Terpstra
Samuel Thibault
Jeyarajan Thiyagalingam
Gaël Thomas
Rollin Thomas
Christopher Thraves
Yuanyuan Tian
Mustafa Tikir
Sebastien Tixeuil
Stanimire Tomov
Nicola Tonellotto
Johan Tordsson
Juan Touriño
Jesper Traff
Damien Tromeur-Dervout
Paolo Trunfio
Hong-Linh Truong
Bora Ucar
Osman Unsal
Timo van Kessel

Rob van Nieuwpoort
Hans Vandierendonck
Ana Lucia Varbanescu
Sebastien Varrette
Mariano Vazquez
Tino Vázquez
Jose Luis Vazquez-Poletti
Luís Veiga
Rossano Venturini
Javier Verdu
Jerome Vienne
Frederic Vivien
Pierre-André Wacrenier
Frédéric Wagner
Stephan Wagner
Oliver Wäldrich
Matthäus Wander
Takumi Washio
Vince Weaver
Jan-Philipp Weiß
Torben Weis
Philipp Wieder
Jeremiah Willcock
Samuel Williams
Martin Wimmer
Xingfu Wu
Brian Wylie
Changyou Xing
Lei Xu
Yingxiang Xu
Ramin Yahyapour
Edwin Yaqub
Asim YarKhan
Srikant YN
Elad Yom-Tov
Kazuki Yoshizoe
Haihang You
Fa Zhang
Hui Zhang
Li Zhao
Wolfgang Ziegler
Wolf Zimmermann
Jaroslaw Zola
Marco Zuniga

Table of Contents – Part I

Topic 1: Support Tools and Environments

Topic 2: Performance Prediction and Evaluation

Topic 3: Scheduling and Load Balancing

Topic 4: High-Performance Architecture and Compilers

Topic 5: Parallel and Distributed Data Management

Topic 6: Grid Cluster and Cloud Computing

Topic 7: Peer to Peer Computing

Topic 8: Distributed Systems and Algorithms

Table of Contents – Part II

Topic 9: Parallel and Distributed Programming

Topic 10: Parallel Numerical Algorithms

Topic 11: Multicore and Manycore Programming

Topic 12: Theory and Algorithms for Parallel Computation

Topic 16: GPU and Accelerators Computing

Introduction

Rosa M. Badia, Fabrice Huet, Rob van Nieuwpoort, and Rainer Keller

Topic chairs

The increasing trend to distribute computing over large-scale parallel and distributed platforms, such as clouds, grids and large clusters, often combined with the use of multicore processors and hardware accelerators, overlaps with an increasing pressure to make computing more dependable. Indeed, parallel programming for these type of platforms remains a complex task due to the numerous components (hardware and software) that can both affect correctness and performance. Therefore, the parallel and distributed computing community continuously requires better tools and environments to design, program, debug, test, tune, and monitor parallel programs. This topic aims to bring together tool designers, developers, and users to share their concerns, ideas, solutions, and products covering a wide range of platforms, including homogeneous and heterogeneous multicore architectures. The chairs of this topic sought contributions with solid theoretical foundations and experimental validations on production-level parallel and distributed systems. Submissions proposing new program development tools and environments that help copying with the expected high complexity of forthcoming exascale parallel systems were encouraged. The accepted papers cover several of these topics with outstanding contributions:

- The paper "Run-Time Automatic Performance Tuning for Multicore Applications" addresses the challenge that the programmers face when several applications run in parallel and influence each other indirectly. Their solution is based on Perpetuum, a novel operating-system-based auto-tuner that is capable of tuning parallel applications while they are running. The approach is a fully functional auto-tuner that extends the Linux kernel, and the application tuning process does not require any user involvement. General multicore applications are automatically re-tuned on new platforms while they are executing, which makes portability easy.
- There are two papers that deal with data races in cache memories. The paper "Exploiting Cache Traffic Monitoring for Run-Time Race Detection" tackles the problem of finding and fixing data races in an automatic fashion. The approach monitors the cache coherency bus traffic for parallel accesses to unprotected shared resources. This technique has low overhead and requires just minor extensions to standard multicore hardware and software to make measurements more accurate. The paper "Accelerating Data Race Detection with Minimal Hardware Support" proposes a high performance hybrid hardware/software solution to race detection that uses minimal hardware support. This hardware extension consists of a single extra instruction,

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 1–2, 2011.

StateChk, that simply returns the coherence state of a cache block without requiring any complex traps to handlers. To leverage this support, a new algorithm for race detection is proposed and a new scheduling manipulation heuristic to achieve high coverage rapidly.

- The paper "Quantifying the potential task-based dataflow parallelism in MPI applications" proposes an approach to automatically estimate how much of an application can benefit from dataflow parallelism and how to find the best strategy to expose dataflow parallelism of the application. The framework presented in the paper automatically detects data dependencies among tasks in order to estimate the potential parallelism in the application. Furthermore, based on the framework, an interactive approach to find the optimal partitioning of code is developed.
- The paper "Event log mining tool for large scale HPC systems" proposes an approach to analyse the log files automatically generated by systems like supercomputers. These log files are so large and complex that human analysis is difficult and error prone. The paper presents a novel methodology for creating event clusters and extracting cluster templates from large datasets presenting an intuitive output to system administrators. The algorithm is able to keep up with the rapidly changing environments by adapting the clusters to the incoming stream of events.
- The paper "Reducing the overhead of direct application instrumentation using prior static analysis" presents an approach to reduce the instrumentation overhead of parallel programs. When using direct instrumentation, the measurement overhead increases with the rate at which these functions are visited. If applied indiscriminately, the measurement dilation can even be prohibitive. The paper shows how static code analysis in combination with binary rewriting can help eliminate unnecessary instrumentation points based on configurable filter rules that can be applied and modified without re-compilation.

Run-Time Automatic Performance Tuning for Multicore Applications

Thomas Karcher and Victor Pankratius

Karlsruhe Institute of Technology, IPD
76128 Karlsruhe, Germany
{thomas.karcher,victor.pankratius}@kit.edu

Abstract. Multicore hardware and system software have become complex and differ from platform to platform. Parallel application performance optimization and portability are now a real challenge. In practice, the effects of tuning parameters are hard to predict. Programmers face even more difficulties when several applications run in parallel and influence each other indirectly. We tackle these problems with Perpetuum, a novel operating-system-based auto-tuner that is capable of tuning applications while they are running. We go beyond tuning one application in isolation and are the first to employ OS-based auto-tuning to improve system-wide application performance. Our fully functional auto-tuner extends the Linux kernel, and the application tuning process does not require any user involvement. General multicore applications are automatically re-tuned on new platforms while they are executing, which makes portability easy. Extensive case studies with real applications demonstrate the feasibility and efficiency of our approach. Perpetuum realizes a first milestone in our vision to make every performance-critical multicore application auto-tuned by default.

1 Introduction

Software developers need to create parallel applications to exploit the multicore hardware potential. Intuitive performance tuning, however, has become difficult for several reasons: (1) Different multicore platform characteristics may cause application optimizations to work on one platform, but not on others. (2) The behavior of complex parallel applications is hard to predict. (3) Applications may have several tuning parameters that impact performance. Potential parameter interdependencies can be difficult to understand. (4) The typical search space spanned by tuning parameters is large. (5) Good performance configurations that work for each application in isolation might not work when several applications run simultaneously on the same system.

In practice, programmers resort to manual and often unsystematic experiments to find program parameter configurations that lead to good performance. Auto-tuning has shown great potential to automate this process and make the search more intelligent using a feedback loop. Offline tuning approaches execute an application until it terminates, gather run-time feedback, and calculate new

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 3–14, 2011.

tuning parameter values that are likely to improve performance in the next run. Most of the existing solutions, however, have drawbacks. For example, [7,18,24] target domain-specific numerical programs (e.g. matrix multiply or Fourier transform). They generate on every platform a new set of executable programs and pick the best-performing one. Unfortunately, this principle does not work for general parallel programs that do not perform any of these numerical analyses. Another issue is that isolated application tuning is inappropriate in today's scenarios. A typical multicore system environment changes all the time due to dynamic resource allocation and applications that run in parallel. This requires long-running applications to be tuned at run-time.

Our paper makes several novel contributions to tackle the aforementioned problems. We introduce Perpetuum, the first auto-tuner for shared-memory multicore applications that integrates into the Linux operating system. Perpetuum's design offers unique opportunities to tune several applications simultaneously and hide the complexity of the tuning process from users and developers. Perpetuum optimizes the performance of applications while they are running, assuming that applications expose their performance-relevant tuning parameters and the associated value ranges to the operating system. Perpetuum's OS-integration reduces tuning overhead and eases portability; an application ported to a new computer that runs Perpetuum will be automatically re-tuned. Moreover, our approach is applicable beyond numerical scientific programs. Two extensive case studies demonstrate feasibility. We achieve respectable performance improvements for compression and multimedia applications in single-process and multi-process scenarios.

The paper is organized as follows. Section 2 introduces the Perpetuum auto-tuner. Section 3 discusses how to prepare applications for online tuning. Section 4 presents case studies with a reengineered parallel compression application. Several scenarios demonstrate Perpetuum's effectiveness in single-process and multi-process contexts. Section 5 presents evaluation studies for an on-line tunable parallel video-processing application written from scratch. Section 6 discusses related work. Section 7 provides a conclusion.

2 The Perpetuum Run-Time Application Tuner

Figure 1 shows the overall system architecture of Perpetuum and how it is integrated into Linux. All tunable applications run in user space. An exclusive part of each tunable application's address space is reserved for a dedicated tuning parameter address space; this space is used by Perpetuum to store, read, and modify the values of all tuning parameters associated with an application.

The auto-tuner is an independent component within the Linux kernel. A tunable application communicates with the auto-tuner using the system call interface. There are three new system calls: (1) The `sys_optAddParam()` call registers a new tunable parameter; (2) `sys_optStartMeasure()` starts a wall clock time counter to measure execution time; (3) `sys_optStopMeasure()` stops the clock counter. Values in the tuning parameter address space can only be changed

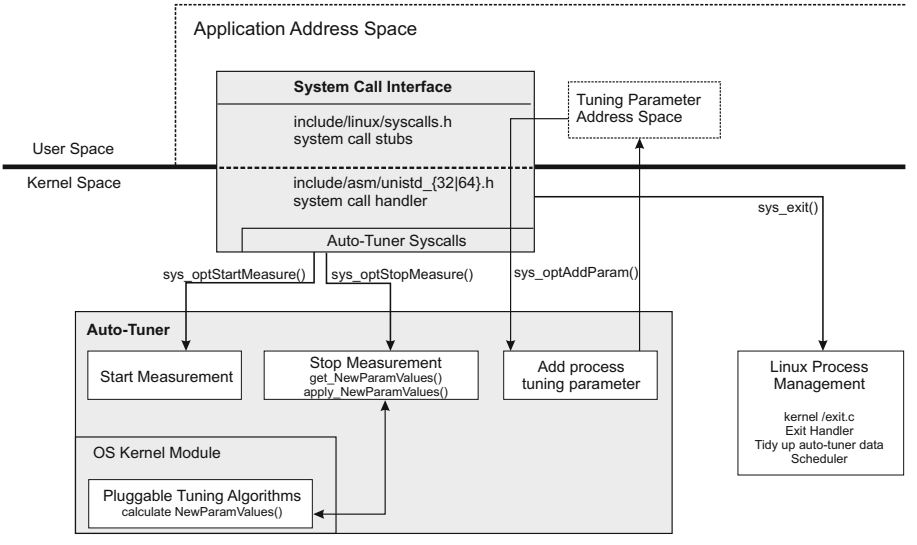


Fig. 1. Overview of Perpetuum's system architecture

during the `sys_optStopMeasure()` call, which blocks the calling thread until all parameters are updated.

Perpetuum uses one system-wide, application-independent tuning algorithm. However, this algorithm can be easily exchanged by system administrators (e.g., a simplex-based algorithm [16] can be replaced by another optimization algorithm). The algorithm is implemented in a plugin style as a Linux kernel module. In contrast to other auto-tuners (see Section 6), our architecture allows plugins to access operating system data, e.g., on workloads and system state. The tuning algorithm is called in a loop by every executing program. The tuning parameters of each application are updated (in its tuning parameter address space) with values that the tuner considers promising for the next iteration.

Perpetuum's current tuning algorithm is based on an adapted version of [16] that works on a discrete integer space and in a scenario with multiple applications that are tuned simultaneously. The number n of application parameters to optimize spans our n-dimensional search space. We generate a simplex with $n+1$ points. Our simplex consists of a starting configuration point and n more points that are obtained by adding a constant displacement in each dimension. When searching for better configurations, we move simplex points based on application execution time feedback and the rules defined in [16].

Perpetuum does not make any modifications to the Linux scheduler, which is part of the Linux process management module. This design decision is based on the fact that Perpetuum influences application tuning knobs that are on a higher abstraction level [19]. By contrast, the scheduler influences low-level resource management decisions, e.g., on which core to execute a certain thread. Perpetuum influences, however, the scheduler in an indirect way: Applications

reacting to a change caused by Perpetuum may increase or reduce the number of threads that the scheduler controls.

We remark that even though Perpetuum has been developed for shared-memory multicore machines, it could also be run on every node of a cluster to automatically improve single-node multithreaded performance of work assigned to the node. Fine-granular performance tuning in clusters thus becomes easier as well.

3 Preparing Applications for Online Tuning

We assume that every application to be tuned at run-time has one compute-intensive "hot-spot", i.e., a modular part of code that is executed in a repetitive manner. Applications should have a longer run-time so that the auto-tuner gets a chance to execute several iterations, adapt parameter values, and observe the effects. The programmer is responsible for developing an application with such a hot-spot or identify one in existing code. To establish an auto-tuning feedback loop, the programmer inserts measurement probes that determine the execution time of the parameterized hot-spot, as shown in the C code example below:

```
int threadCount = 1;
addParam(&threadCount, 1, 16); //tunable degree of parallelism
while (calculationRunning) {
  startMeasurement();          //tuning feedback probe
  doCalculation(threadCount);  //hot-spot
  stopMeasurement();           //tuning feedback probe
}
```

The auto-tuner will automatically set `threadCount`'s values to a number between 1 and 16. It is the responsibility of the programmer that such changes produce consistent results. Our case studies further illustrate in more complex examples that the adaptation of applications for online tuning is not difficult to do.

After each iteration of the application's tuning hot-spot, the auto-tuner collects feedback information. Based on the elapsed execution time, it calculates new values for all tuning parameters before the next iteration begins. The optimization cycle repeats until the application terminates.

Note that the auto-tuner algorithm can adjust the tuning parameters according to the overall system workload that indirectly influences the run-time of the hot-spot. When two applications compete for example for cache or memory I/O, the auto-tuner aims for a cross-process optimum, which is obtained based on the objective function of the tuning algorithm. If an application terminates and releases its resources, another application can be assigned the newly available resources. We now show in two case studies how Perpetuum adapts application parameters in an automated fashion.

4 Perpetuum in Action: Automated Online-Tuning in Parallel Compression

This case study exemplifies how to reengineer an existing parallel application and make it tunable at run-time. We illustrate Perpetuum's online tuning in three scenarios.

4.1 Environment

The sequential Bzip2 file compressor divides a file stream into independent blocks and passes them through a pipeline of algorithms [20]. At the end of the pipeline, compressed blocks are concatenated in their original order and stored in an output file.

We employ the parallel Bzip2 version of [17] that has two command line parameters: the number of compression threads t and the block size b in hundred kilobytes. In our scenarios, we use $t \in \{3, 4, \ldots, 64\}$ and $b \in \{1, 2, \ldots, 9\}$. The tuning hot-spot is the compression code that is applied to each file, located in Bzip's `handle_compress()` function. We reengineered the application and added two system calls to measure the wall-clock time of the hot-spot. In addition, we added two system calls to make t and b tunable. When a directory of files is compressed, the hot-spot is executed in a repetitive fashion. Our implementation can update t and b with new values after finishing the compression of the current file.

We conducted the experiments in a controlled environment on the following machine: Intel Core 2 Quad Q6600 machine, 2.40GHz, running Linux 2.6.34 with Perpetuum. We deactivated the graphical user interface and all other interfering applications. All scenarios use the same collection of 50 files (each with a size of 2 MB), so the compression hot-spot executes 50 times. The fact that all files have the same size is not a constraint of the auto-tuner; this setup was chosen to make results comparable and identify sources of bias more easily.

4.2 Scenario 1: Tuning a Single Process

This scenario shows that Perpetuum successfully tunes one application while that application is running. Perpetuum controls the t and b parameters and aims to reduce the run-time of the hot-spot.

To evaluate tuning effectiveness, we exhaustively benchmarked all parameter configurations for a single Bzip2 process without auto-tuning, for the total of $9 \times 61 = 558$ configurations. The execution time for each configuration was measured 3 times to avoid bias. These results allow us to compare Perpetuum's results with the real optimum.

The exhaustive measurements show that if $b \in \{1, 2\}$ and $t \geq 5$, the execution time is within the best 20%. We thus expect Perpetuum to reduce the block size (ideally to $b = 1$) and increase thread count to $t \geq 5$. With the best configuration, the entire program executes in 6.5 seconds, whereas the worst configuration takes 22.9 seconds. This is the range in which Perpetuum can be expected to optimize the application's run time.

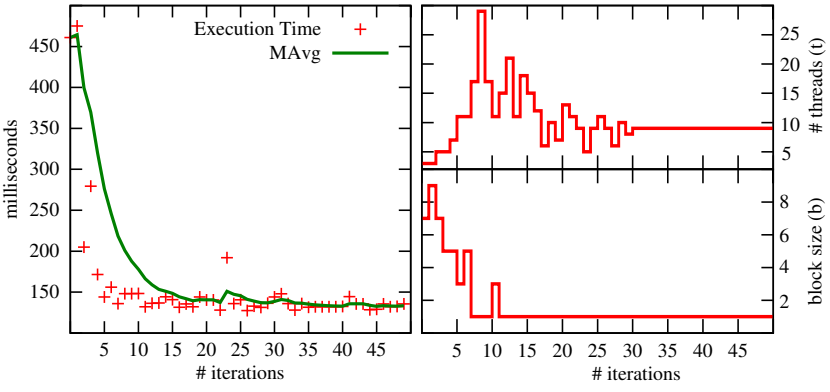


Fig. 2. Online tuning of parallel Bzip2. Left graph: hot-spot execution times after each iteration. Right graphs: values of the tunable parameters.

Figure 2 shows the execution time of the hot-spot (which accounts for almost the entire program execution time) for the block size and thread count chosen by Perpetuum in each iteration. We also plot an exponential moving average (MAvg) of execution times using $a_0 = x_0$ and $a_i = 0.75a_{i-1} + (1-0.75)x_i$, where a_i is the moving average value and x_i the execution time measured at the end of iteration i. Optimization starts at the worst-case configuration $b = 7$ and $t = 3$ where compression needs $458ms$. Without tuning, the entire application would have taken $458ms \times 50 = 22.9$ seconds to finish. Perpetuum reduces the average execution time to a total of 8 seconds, which is 2.9x faster. Note that this is not the classical speedup measure in comparison to the sequential program, but a performance boost in comparison to the parallel program. Speedup compared to the sequential time of 24.5 seconds is even higher, namely 3.1, which is not bad considering the quadcore machine and almost no programming effort. The final tuning result is just 23% worse than the best attainable execution time.

The graphs for thread count t and block size b illustrate how Perpetuum works. Both values increase at first. The auto-tuner then realizes that increasing thread count alone is not too effective and that a smaller block size reduces execution times more significantly. The block size quickly converges to 1, while other thread counts are tried out. The step for t is doubled until iteration #8, and t finally converges to 9 threads after some oscillation. Our exhaustive measurements exploring the search space show that the finally obtained configuration of $b = 1$ and $t = 9$ is within the best 1% of all performance configurations.

We remark that a starting configuration can be randomly generated. If optimization had started, for example, with another configuration (e.g., $b = 5, t = 20$), finishing all iterations without tuning would have taken 8.6 seconds, and with tuning 7.2 seconds (which is still 1.2x faster). In general, if a starting configuration already has good values, Perpetuum tries to tune the application but will not be able to significantly improve performance, so it will stop tuning after some time.

4.3 Scenario 2: Simultaneously Auto-tuning Two Processes

This scenario evaluates how Perpetuum simultaneously tunes two processes that are started at the same time (see Figure 3 (a)). We execute two instances of the parallel Bzip2 application that work on individual copies of the file benchmark from in scenario 1. Each instance starts with the same configuration $b = 5$ and $t = 3$ which is within the worst 10% of execution times. The execution time variance with two processes is higher than in a single-process scenario, due to increased CPU, RAM, and hard disk activity.

Without auto-tuning, starting both instances at the same time and waiting for the last one to finish takes 26.5 seconds. With auto-tuning, it takes just 13.5 seconds. This boosts performance of the parallel application by a factor of 1.96. Fig. 3 shows how Perpetuum adapted block size and thread count for each process. First, the auto-tuner reduces the block size for both processes. Process 2 reaches $b = 1$, which was the optimum in scenario 1. Process 1 is also assigned $b = 1$ for a few iterations, but the auto-tuner finds out that it can reduce execution time by increasing block size to 3, which differs from the single-process scenario. The sum of moving averages (MAvg Sum) of the two processes decreases, which shows that Perpetuum globally improves performance.

Perpetuum automatically finds the critical point around $t = 5$ after 10 iterations, which we manually identified ourselves in the exhaustive exploration of the search space in the single-process scenario (the single process was significantly slower when $t \leq 4$). As a result, Perpetuum increases the thread counts for both processes. Process 1 converges to $t = 5$ and process 2 to $t = 24$.

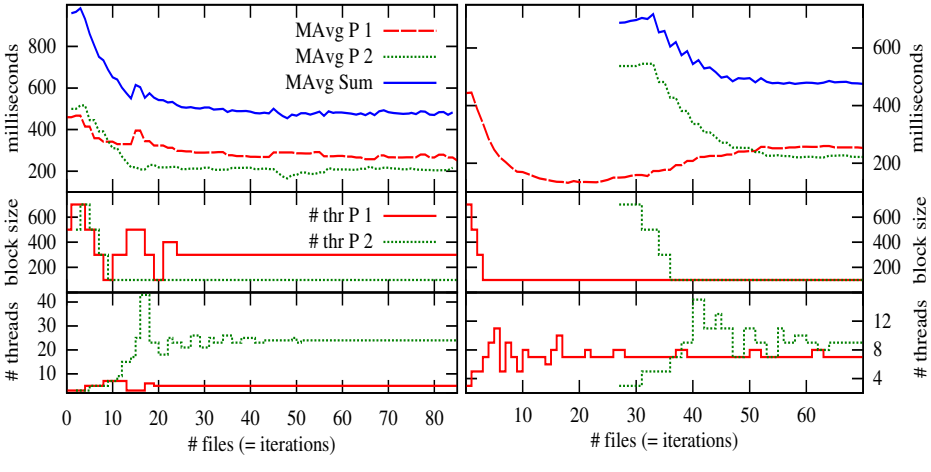


Fig. 3. Online tuning of parallel Bzip2. Graphs in first row show hot-spot moving average execution times; graphs in other rows show tuning parameter values. Scenario 2 (left): two instances are started at the same time and tuned simultaneously. Scenario 3 (right): two instances are started with a time lag.

4.4 Scenario 3: Simultaneously Auto-tuning Two Processes Starting with a Time Lag

This scenario is similar to scenario 2, except that the second process is started 4 seconds after the first one. Figure 3 (b) shows the timeline: The first process starts off solo, as in scenario 1. The block size converges again to $b = 1$ while the thread count roughly converges to $t = 7$. Then, the second process starts. While the tuning parameters of process 2 are modified as expected, the execution time of process 1 increases due to interference with process 2. The auto-tuner does not change the block size in both processes, but assigns process 2 more threads, which likely has the effect of hiding latency. This strategy improves overall performance, as demonstrated by the decreasing moving average sum of execution times.

4.5 Summary

The auto-tuner significantly improves performance in all of our scenarios. Perpetuum adjusts parameters that have more impact on performance variance (e.g., block size) earlier than others. Another insight is that the common programmer intuition to set the number of threads to the number of cores would fail here: Configurations with 4 threads and an arbitrary block size had a performance within the worst 10% of all configurations. Perpetuum could not be fooled into this false assumption and quickly converged to better values within the first 10 iterations.

5 Automated Online-Tuning in Parallel Video Processing

This Section presents a study with an online-tunable multimedia application written from scratch [1]. The application performs parallel edge detection on a video stream. The output video stream consists of images that show the edges of objects. In computer vision, edge detection is an important basis for other algorithms, e.g., to track or identify objects in robotics, security applications, or human-computer interaction.

The application has five multithreaded filters organized in a pipeline. Each filter works in parallel within a pipeline stage on one frame of the video stream. **Stage S1 (Gauss)** performs a Gaussian blur by applying a convolution mask. **Stage S2 (Gradient)** applies a Sobel mask to compute the gradient strength and direction for each pixel. **Stage S3 (Trace)** traces the edges based on the gradients computed in the previous stage. **Stage S4 (Suppress)** suppresses pixels that are not on an edge. **Stage S5 (Non-Max)** performs some clean-ups in the picture by eliminating weaker edges that are parallel to stronger ones.

Parallelism is introduced using Intel's Threading Building Blocks [10] and assigning a tunable number of threads to each pipeline stage. For each stage, thread count can be set from 1 to 64. The tunable hot-spot measures the execution time of every 10 frames passing the entire pipeline. The experiments are conducted on the same machine as in the Bzip2 case study. As an input data

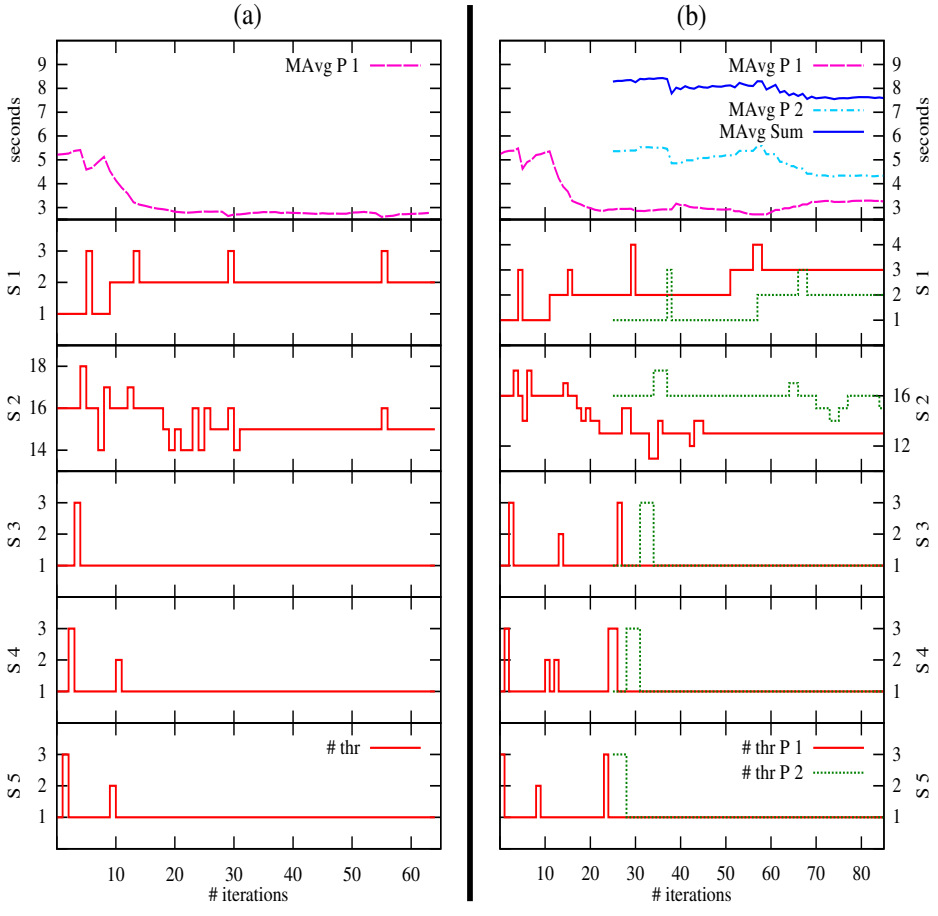


Fig. 4. Online tuning of a video processing application. The graphs in the first row illustrate hot-spot execution times, the others the tunable parameter values. (a): single-process scenario 1; (b): two-process scenario 2.

set, we use the first 720 frames of an open source movie [8]. Our input file has an AVI format with MPEG-4 compression, 854x480 pixels resolution, 24 frames per second, and a total size of roughly 12 MB.

5.1 Scenario 1: Tuning a Single Process

The total search space with our five parameters consists of $64^5 = 1,073,741,824$ configurations, which can hardly be benchmarked exhaustively, so we did explorative studies. The first two stages have more impact on the overall application run-time than the last three stages. We thus focused on exploring the first two stages with thread counts between 1 and 16 for each stage. All measurements are repeated 3 times. The best-performing configuration has 11 threads for Gauss

(S1) and 5 threads for Gradient (S2), with a total run-time of 116.3 seconds. Intuitive configurations assigning 1 thread per stage end up within the worst 10% performance. Configurations with one thread for S1 are in the worst 5% of all configurations. The worst configuration has threads assigned to stages as follows: 1-16-1-1-1. The average run-time without auto-tuner is 384.4 seconds. S1's thread count parameter has high sensitivity; increasing it from 1 to 2 causes performance to surge, but performance improvements diminish if more than 2 threads are assigned to this stage.

Figure 4 (a) exemplifies how Perpetuum performs in the worst case with start configuration 1-16-1-1-1. In the first iteration, hot-spot execution time is about 5.2 seconds, but the auto-tuner is able to finally reduce it to 2.7 seconds, which is a 1.9x improvement. It is also remarkable that the auto-tuner tries tuning the thread count of the last stages but quickly realizes that they don't have much effect, so the values remain constant. By contrast, the thread counts in the first stages are tuned more often, and the tuner automatically detects that increasing thread count for S1 above 1 significantly improves performance.

Perpetuum finally converges to the non-intuitive configuration of 2-15-1-1-1, showing that this application needs a total of 20 threads on our 4-core machine. This empirical result proves programmers wrong who assume that the number of threads must equal the number of cores.

5.2 Scenario 2: Simultaneously Tuning Two Processes Starting with a Time Lag

Similarly to the Bzip2 online tuning case study, we start two processes of the video processing application, both with the configuration 1-16-1-1-1. Figure 4 (b) shows that this tuning scenario is more difficult. The first process is tuned similarly to scenario 1; if S1 receives more than one thread, performance improves significantly. At iteration 25, the second process starts interfering. The performance of process two finally improves after the auto-tuner finds that increasing S1's thread count is good. Note that even though process one's run-time increases until it terminates in iteration 88, the overall system performance represented by the moving average sum still improves. Finally, each application's hot-spot execution time is lower than before tuning.

6 Related Work

The advantages of integrating auto-tuners into operating systems haven been acknowledged by the operating systems community [11]. Other details on experiments with Perpetuum are summarized in a technical report [12]. Most of the related work covers online tuning with a different focus and with other techniques. *Orio* [9] focuses mostly low-level operation performance optimizations on a particular code fragment annotated with specific structured comments. *MATE* [15] provides dynamic tuning for MPI applications and is designed for distributed architectures. An adaptive task scheduler for multi-tasked data-parallel jobs is introduced in [3], however, assuming job granularity and a distributed system

environment. The work of [4] uses hardware performance counters and aims to minimize cache contention by clustering threads and assign dedicated cache regions to threads. The *CAER* environment [14] provides a run-time solution that targets a reduction of cross-core interference due to contention. *Active Harmony* [6,21,22,23] tunes one parallel program at a time in a heterogeneous, distributed environment. Each application has to obey a dedicated API to send performance feedback to a dedicated optimization server and receive new configurations via message passing. By contrast, Perpetuum is targeted at interactive use on shared-memory multicore desktops and server machines, so we have other assumptions about application characteristics and the acceptable computation and communication overhead. For example, Perpetuum is 15 seconds (30%) faster than our adapted version of *Active Harmony* running compression scenario one on our hardware. Several other approaches work on a lower abstraction levels than Perpetuum. The work of [13] combines static and dynamic binary compiler optimizations to select the best-performing variant of a program function out of multiple versions. A compiler framework that detects at run-time which code optimizations to apply is shown in [5]. Machine learning is applied in [2] to iteratively learn about program features and adapt compiler optimizations.

7 Conclusion

Perpetuum's infrastructure presented in this paper is the first OS-based approach to allow automatic performance tuning at runtime for simultaneously executing multicore applications. Our approach works well beyond scientific numerical programs and can be used in standard desktop PCs and servers. We are also the first to integrate such an auto-tuner into the Linux operating system, which has several advantages: (1) The performance optimization algorithm can access system information to compute the global performance optimum for all active applications in a cooperative way; (2) OS integration allows fast response times for online tuning; (3) Auto-tuning as a standard service in the OS allows programmers to outsource tuning logic from their code and make their code base easier to maintain; (4) Tedious and intuitive manual tuning (which might not even find optimum performance) becomes obsolete; (5) Parallel application portability is improved, as applications are automatically re-tuned on each platform. Overall, Perpetuum paves the way towards making auto-tuning a standard approach in multicore application development.

Acknowledgements. We thank the Excellence Inititative and the Landesstiftung Baden-Württemberg for their support.

References

1. Abudiab, I.: Online-tunable parallel edge detection in video streams. Student project thesis. Karlsruhe Institute of Technology (2010)
2. Agakov, F., et al.: Using machine learning to focus iterative optimization. In: CGO 2006, p. 11 (2006)

3. Agrawal, K., et al.: Adaptive scheduling with parallelism feedback. In: PPoPP 2006, p. 1 (2006)
4. Azimi, R., et al.: Enhancing operating system support for multicore processors by using hardware performance monitoring. SIGOPS Oper. Syst. Rev. 43(2), 56 (2009)
5. Cavazos, J., Moss, J.E.B., O'Boyle, M.: Hybrid optimizations: Which optimization algorithm to use? In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 124–138. Springer, Heidelberg (2006)
6. Ţăpuş, C., et al.: Active harmony: towards automated performance tuning. In: SC 2002, p. 44 (2002)
7. Frigo, M., Johnson, S.: FFTW: an adaptive software architecture for the FFT. In: Proc. IEEE ICASSP 1998, vol. 3, p. 1381 (1998)
8. Goedegebure, S., et al.: Big buck bunny. An open source movie (April 2008), http://www.bigbuckbunny.org (last accessed May 2011)
9. Hartono, A., Ponnuswamy, S.: Annotation-based empirical performance tuning using Orio. In: IPDPS 2009, p. 1 (2009)
10. Intel: Threading building blocks (August 2006), http://www.threadingbuildingblocks.org
11. Karcher, T., et al.: Auto-tuning support for manycore applications: perspectives for operating systems and compilers. SIGOPS Oper. Syst. Rev. 43(2), 96 (2009); Special Iss. on the Interaction among the OS, Compilers, and Multicore Processors
12. Karcher, T., Pankratius, V.: Auto-Tuning Multicore Applications at Run-Time with a Cooperative Tuner. Karlsruhe Reports in Informatics 2011-4 (February 2011)
13. Mars, J., Hundt, R.: Scenario based optimization: A framework for statically enabling online optimizations. In: Proc. CGO 2009, p. 169 (2009)
14. Mars, J., et al.: Contention aware execution: online contention detection and response. In: Proc. CGO 2010, p. 257 (2010)
15. Morajko, A., et al.: Mate: Monitoring, analysis and tuning environment for parallel & distributed applications: Research articles. Concurr. Comput.: Pract. Exper. 19(11), 1517 (2007)
16. Nelder, J.A., Mead, R.: A simplex method for function minimization. The Computer Journal 7(4), 308 (1965)
17. Pankratius, V., et al.: Parallelizing bzip2: A case study in multicore software engineering. IEEE Software 26(6), 70 (2009)
18. Puschel, M., et al.: Spiral: code generation for dsp transforms. Proceedings of the IEEE 93(2), 232 (2005)
19. Schwedes, S.: Operating system integration of an automatic performance optimizer for parallel applications. Master's thesis, Karlsruhe Institute of Technology (2009)
20. Seward, J.: Bzip2 (2011), http://www.bzip.org
21. Tabatabaee, V., Hollingsworth, J.K.: Automatic software interference detection in parallel applications. In: SC 2007, vol. 1, p. 14 (2007)
22. Tabatabaee, V., et al.: Parallel parameter tuning for applications with performance variability. In: SC 2005, p. 57 (2005)
23. Tiwari, A., et al.: Tuning parallel applications in parallel. Parallel Comput. 35(8-9), 475 (2009)
24. Whaley, C.R., et al.: Automated empirical optimizations of software and the atlas project. Parallel Computing 27(1-2), 3 (2001)

Exploiting Cache Traffic Monitoring for Run-Time Race Detection

Jochen Schimmel and Victor Pankratius

Karlsruhe Institute of Technology, IPD
76128 Karlsruhe, Germany
{schimmel,pankratius}@kit.edu

Abstract. Finding and fixing data races is a difficult parallel programming problem, even for experienced programmers. Despite the usage of race detectors at application development time, programmers might not be able to detect all races. Severe damage can be caused after application deployment at clients due to crashes and corrupted data. Run-time race detectors can tackle this problem, but current approaches either slow down application execution by orders of magnitude or require complex hardware. In this paper, we present a new approach to detect and repair races at application run-time. Our approach monitors cache coherency bus traffic for parallel accesses to unprotected shared resources. The technique has low overhead and requires just minor extensions to standard multicore hardware and software to make measurements more accurate. In particular, we exploit synergy effects between data needed for debugging and data made available by standard performance analysis hardware. We demonstrate feasibility and effectiveness using a controlled environment with a fully implemented software-based detector that executes real C/C++ applications. Our evaluations include the Helgrind and SPLASH2 benchmarks, as well as 29 representative parallel bug patterns derived from real-world programs. Experiments show that our technique successfully detects and automatically heals common race patterns, while the cache message overhead increases on average by just 0.2%.

1 Introduction

Race conditions are frequent parallel programming errors that are difficult to detect even for experts. Unfortunately, a solution to the problem of finding all races in arbitrary parallel programs is equivalent to the halting problem [5]. Race detection tools thus have no other choice than use heuristics and accept trade-offs, e.g., in accuracy, false alarm reports, or analysis speed.

A large body of work presents race detectors that are employed during program development [7,8,26], which introduce significant analysis overhead. Application bug reports, however, show very often that racy code might still be present after deployment at clients, which can cause severe damage when crashes corrupt data. This paper tackles this problem and introduces an approach for run-time race detection and automated race healing for production environments. Our

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 15–26, 2011.

detection heuristics focus on speed and on detecting the most common racy patterns due to wrong locking.

Current proposals for run-time race detection [4,11,12,16,20,21,28] typically require specialized hardware. Most standard hardware, however, does not have such costly extensions. The novel extensions proposed in this paper aim to lower the entry barrier and make run-time race detection available in many systems. Our key idea exploits synergy effects between hardware used for performance monitoring and hardware needed for run-time race detection. Moreover, our extensions can be used for more accurate performance monitoring if run-time race detection is not required.

We introduce TachoRace, a novel light-weight race detector that leverages data from hardware performance counters in multicore processors for data race detection. We track down events in the first-level cache of each core and automatically heal races with a new cache protocol extension. We validate the proposed hardware extensions in a controlled environment based on a simulator using PIN [13]. TachoRace executes real binary programs, simulates caches, cache protocols, and performance counters. This infrastructure allows us to precisely quantify TachoRace's effectiveness for race detection as well as performance overhead.

The paper is organized as follows. Section 2 introduces our assumptions and requirements. Section 3 presents the principles of cache traffic monitoring for race detection and healing. Section 5 shows detailed evaluations. Section 6 discusses related work. Section 7 provides a conclusion.

2 Assumptions and Requirements

2.1 Software

A data race occurs when two threads simultaneously access the same memory location without synchronization, and at least one of them performs a write operation. This work focuses on locks as a means for synchronization and on errors resulting from incorrect lock usage. As a race detection in general is equivalent to solving the halting problem [5], our approach specializes on finding races caused by wrong locking, i.e., patterns (b) and (c) in Figure 1, and a generalization of these patterns (e.g., with more than two threads or several locks). Previous work [22] shows that these error patterns are representative for frequent errors in practice.

Run-time race detection can be made more accurate by annotating which variable a lock should protect. Our approach introduces the `lock_annotate` language extension to let programmers specify the relationship between a lock and a locked element. The `lock_annotate` construct registers the address of the lock, the address of the locked element and the locked element's size. The locked element can be composed of other elements that are contiguously stored in memory. Here is an example in C:

```
int account = 0; Lock acc_lock;  /*acc_lock protects account*/
lock_annotate(&acc_lock, &account, sizeof(account));
```

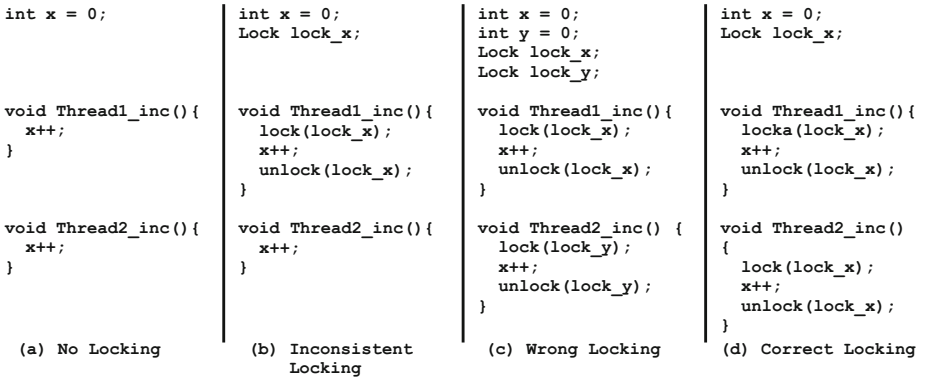


Fig. 1. Examples for lock usage patterns; (a)–(c) are incorrect programs

Earlier studies [18,19] have shown that parallel programs typically have only a few lines of code containing synchronization constructs, so the expected number of annotations is small. We remark that even languages like Java with block-oriented synchronization keywords provide explicit locks for performance reasons [10], so the aforementioned locking error patterns can also occur in Java.

2.2 Hardware

We detect data races by observing cache bus traffic while applications are running. This can already be done on existing processors, but unfortunately the lack of measurement precision requires hardware extensions. Our specific intention was to envisage extensions that don't require a radically different hardware infrastructure, so they can eventually be available in standard processors. We thus build on cache coherency protocol information that we gather from state-of-the-art hardware performance counters. Our extensions can be used for more accurate performance monitoring if race detection is not needed.

Reading cache coherency protocol data through performance counters incurs almost no run-time overhead compared to the overhead introduced by other race detectors [8,26]. Among others, we employ the `CMP_SNOOP` performance counter [6] to monitor Modified/Exclusive/Shared/Invalid (MESI) messages and count the number of cache lines requested by processor cores.

Current processor performance counters don't provide yet all necessary functionality for race detection. For example, counters on Intel processors don't provide the memory addresses of accesses causing cache events, or filters for events on a range of memory addresses. We thus implemented a software simulator using PIN [13] to validate our technique. Our approach introduces additional debug registers attached to each core, which each consist of one memory address field and one size field (in bytes) for a shared data element. The registers are used to configure a performance counter to only count events with accesses to a specified memory location; the data size – if greater than zero – expands a

filter to a contiguous memory region. For now, we assume that threads are not migrated among cores. As a proof of concept, we evaluate a hardware configuration in which each core has one additional debug register, and assume that the number of registers suffices to supervise programs with a reasonable number of locks. The debug register's address field contains the starting address of a lock-protected data element; the size field can be used to monitor accesses to contiguous data structures such as objects or arrays. Debug registers can be initialized in a transparent way by extending *lock* and *unlock* constructs in libraries such as Pthreads.

TachoRace effectively identifies and corrects races occurring due to wrong locking. It is not designed for situations in which locking is incorrectly not done at all; it also does not correct code with races that actually never occur.

3 Monitoring Cache Traffic to Detect and Heal Races

3.1 Race Detection

TachoRace's principle for run-time race detection is based on inference from observed cache traffic. As an example of how it works, let's assume a dual core machine on which thread T1 executes on core C1 and thread T2 on core C2. Suppose that a programmer forgot to acquire a lock and protect variable x, as in thread 2 in Figure 1 (b). T1 enters the critical section to increment x. At this point, the address of x is stored in the debug register of C1 and a corresponding performance counter is initialized on C1 to count all MESI events accessing this address. T1 loads x from main memory into its local cache, and increments it. Now if T2 attempts to increment x simultaneously, it has to fetch x in a similar way and issue MESI messages on the bus. These messages are registered at C1, which increments the performance counter for access to x address. The new counter value greater than one indicates an incorrect usage of locks, because no other thread should have been allowed to access x.

TachoRace detects and heals data races only when they occur. Our conflict detection scheme targets inconsistent lock usage as in patterns like Figure 1 (b) and (c). However, we also handle situations in which multiple read accesses to a locked element occur inconsistently; for example, one thread acquires a lock before accessing variable x (for read access only), while another thread simultaneously reads x without acquiring the lock. Technically, this is not a race, but points to a potential error, and TachoRace reports a warning.

3.2 Race Healing

TachoRace heals races by modifying conflicting thread access schedules in real-time. Messages that delay the execution of other cores that cause a conflict are then issued on the bus. We extend the MESI protocol by five Inter-Processor Interrupt (IPI) messages: "RaceWait", "RaceContinue", "DeadlockCheck", "NoDeadlock", and "DeadlockFound".

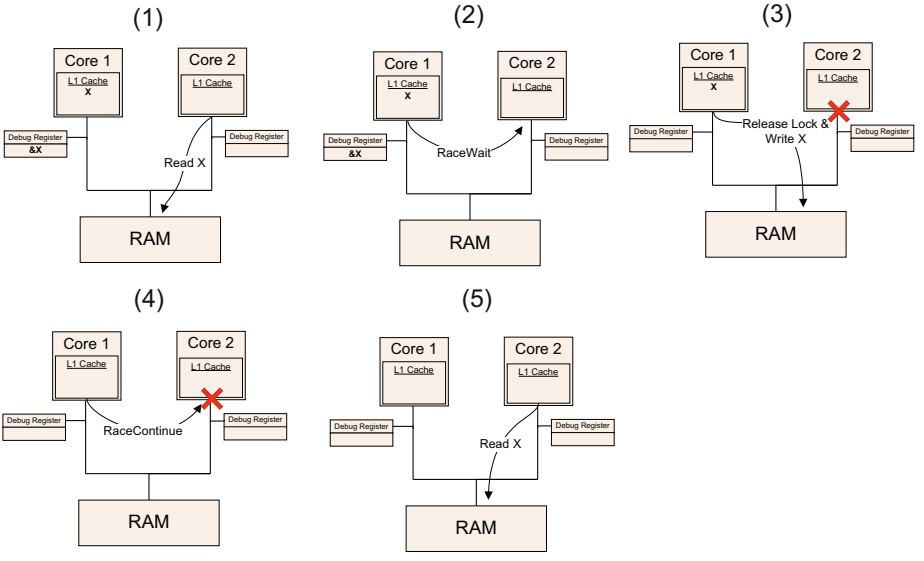


Fig. 2. Illustration of the race prevention strategy issuing RaceWait and RaceContinue messages

Figure 2 illustrates our race healing protocol. In step (1), core one acquired a lock on x and manipulated x's value, so x is available in core one's cache. TachoRace stores the address of the locked data element in core one's debug register. When core two attempts to read x from main memory, the new requests are visible on the bus. In step (2), core one detects the potentially conflicting request by listening to bus traffic and issues a "RaceWait" message. In step (3), core two receives "RaceWait" and blocks the execution of its thread until it receives "RaceContinue" in step (4). The "RaceContinue" message is issued when core one's thread executes its *unlock* operation. Finally, core two's thread resumes execution in step (5) and re-issues the reading operation on x.

Due to space limitations, we have to omit details on correctness checking of the protocol and describe how TachoRace handles some special cases. A special case is trapped when automatic race repair avoids a race, but leads to a deadlock. This can happen when two data races overlap; for example, one thread acquires lock A, securing access to variable X, while thread two acquires lock B, securing access to variable Y. If both threads concurrently write on the variable that is locked by the other thread, TachoRace would send a "RaceWait" message to each thread, but no one will issue a "RaceContinue" message. The inherent problem is that TachoRace does not know the programmer's intention in a program that is simply wrong. We thus check for deadlocks whenever a core waiting for a "RaceContinue" message emits a "RaceWait" message.

In brief, such rare deadlock situations are handled as follows. If a thread issues "RaceWait" and pauses, it broadcasts on the bus a "DeadlockCheck" message identifying the thread that has sent the initial "RaceWait" message. Snooping

the bus, each active thread checks if the message matches itself. In a match, the respective thread broadcasts a "DeadlockCheck" message if it has been paused by "RaceWait", again identifying the thread that sent "RaceWait". A thread will eventually either send a "NoDeadlock" message or the "RaceWait" chain exploration will reach the first thread, the one that initiated deadlock checking. This thread issues a "DeadlockFound" message, so TachoRace can report the incident. We remark that additional messages to handle deadlocks hardly influence average application performance. The reason is that the described chain of events rarely occurs.

4 The Detector

Current multicore processors do not have a debug register like the one proposed in this paper, but it is required to implement our run-time race detection. To validate TachoRace, we thus developed a hardware and cache simulator based on PIN [13], which is capable of executing real, multithreaded binary applications. TachoRace runs on Windows and Linux. TachoRace can be configured to use a wide range of cache architectures that may have a different number of cores and different cache levels. Every cache level can be individually configured to be shared among certain cores. For example, it is easy to model Intel's Core 2 Quad Q6600 processor, where each pair of cores share a common L2 cache, whereas every single core has a private L1 cache. TachoRace can even use a different cache coherence protocol on each cache level. We implemented the widely-used MESI protocol [6], but our simulator can be easily extended to use MSI or MOESI [1]. We also support the Least-Recently-Used replacement strategy and an adjustable cache line size. Each cache level can be configured to be fully associative, set associative, or n-way associative. TachoRace does not consider prefetching. All caches contain data only, as in most architectures instruction data is read-only; instruction caches are not modeled.

As a proof of concept, the implementation is based on the following model: The processor contains $n \geq 1$ processing cores, each of which has its own level one cache. Higher cache levels and main memory are shared among x cores (e.g., with $x = 2$ for Intel's Q6600). A program has a maximum of n threads, each of which is attached to one distinct core, and threads are not migrated from one core to another core. If there are more cores available than threads, the redundant cores remain idle. Only one parallel program is running at a time. Another program only starts when the previous program has finished, excluding scheduling overlaps. Threads can be deliberately paused and resumed to achieve different thread interleavings. We don't simulate the operating system, interrupts, or traps, so we can ensure that the currently executing program is the only one to cause caching activity.

We remark that the restrictions in the simulation environment were chosen to create a controlled environment that cleanly demonstrates that the results are due to the race detection approach, and not due to other factors or noise. As TachoRace uses concrete hardware memory addresses, it also works when threads from different processes incorrectly access a shared resource.

5 Evaluation

5.1 Setup

We evaluate the effectiveness of TachoRace's on-the-fly race detection at run-time. As it is the first detector of its kind, we compare its results with Helgrind [26] (an open-source detector) and Intel's commercial Thread Checker [7]. We use two well-known benchmarks: The Helgrind race detection unit tests [25] and SPLASH2 [27].

The Helgrind unit tests consist of more than 50 parallel programs. We select the tests designed for race detection, i.e., a subset of 29 executable programs. All lock declarations are annotated as described earlier. Each test creates several threads and executes a small piece of code that either has a data race or implements correct code that might look like a race. Table 1 shows an overview: out of 29 cases, 4 have a racy pattern with no locking at all (which TachoRace cannot detect by design), 10 are synchronized correctly, and the other 15 use locks incorrectly. In addition, we include the following applications from SPLASH2: "cholesky" (a numerical application), "water-nsquared" (a physical simulation), and "raytrace" (a parallel raytracer). We seeded 13 data races into these applications (see Table 2), using the patterns shown in Figure 1 (b) and (c) by randomly deleting pairs of lock acquisitions and releases. Some of the races influenced the progress of the applications and even crashed them when the race occurred. TachoRace proved to heal the races and prevent the crashes in all of these cases.

5.2 Results

Table 1 shows effectiveness results. TachoRace finds all races in all fourteen test cases that use locks. It is successful on all of the test cases where it should find a race. TachoRace did not report false positives and resolved all detected conflicts at run-time by delaying the execution of the malicious threads, so the programs were able to produce correct outputs. Races in the five remaining racy programs are not detected, as TachoRace was not designed to find races in programs that do not use any locks at all. In an additional stress-test, we inserted sleep statements at key positions in the parallel program's code to cause different thread schedule interleavings, and TachoRace's was able to detect the same races. We encountered no situation where TachoRace's race healing produced a deadlock.

By contrast, Helgrind incorrectly reported 5 race-free test cases to contain races. Intel's Thread Checker reported just one false positive, but its overhead lead to application execution time slowdowns of up to 3324x (!). Such huge slowdowns are not unusual for dynamic detectors that check for races at each memory access.

Table 3 shows efficiency results. The message overhead introduced by RaceWait/ RaceContinue messages and by counter accesses is low, compared to regular MESI messages that would have been on the bus without TachoRace. On

average, TachoRace introduces just 0.2% more messages. The introduced slowdowns for application execution are minor; the exact slowdown depends on the specific hardware, however, assuming $10ns$ for message handling (see [24] for clock cycle estimations) and counter access, yields for all 29 test cases a median application slowdown of $50ns$. Even the maximum slowdown of $10ms$ (e.g., for test case 20) corresponds to an application run-time slowdown of 0.002%. By contrast, measured overhead with Intel's Thread Checker is significantly higher for all test cases. On an Intel Quadcore machine running Ubuntu Linux 9.1, we obtained a median slowdown of 77 times the application execution time, and a maximum slowdown 3324 times (e.g., for test case 20).

Table 1. Detection results for general bug patterns for Helgrind, Intel Thread Checker, and TachoRace

Test case no.	Helgrind test case ID	Error Class acc. Fig. 1	Description	Has Race?	Helgrind Found?	Helgrind False Alarm	Intel TC Found?	Intel TC False Alarm	Tacho-Race Found & Healed
1	1	a	write vs. write, no locking	1	1	0	1	0	0
2	3	d	correct synchronization with locks and signals	0	0	0	0	0	0
3	4	d	correct sync., producer/consumer-pattern	0	0	0	0	0	0
4	6	d	correct sync. with locks and signals	0	0	0	0	0	0
5	8	d	correct sync. with thread-joining	0	0	0	0	0	0
6	9	a	read vs. write without locking	1	1	0	1	0	0
7	11	d	two worker threads, sync. with locks and signals	0	1	1	0	0	0
8	12	d	producer/consumer-pattern with mutexes	0	1	1	0	0	0
9	13	d	mutex-synchronization	0	1	1	0	0	0
10	15	d	mutex-synchronization, three threads	0	0	0	0	0	0
11	20	a	wrong sync. using timeouts	1	1	0	1	0	0
12	32	d	sync. with thread-joining and mutex	0	1	1	0	0	0
13	47	b	read vs. write with incorrectly used mutex	1	1	0	1	0	1
14	50	b	read vs. write with incorrectly used mutex	1	1	0	1	0	1
15	52	b	wrong signal-based synchronization	1	1	0	1	0	1
16	55	d	correct synchronization with locks	0	1	1	1	1	0
17	56	a	four threads, no sync. on global variable	1	1	0	1	0	0
18	64	a	producer/consumer-pattern with unsync. thread	1	1	0	0	0	0
19	65	c	producer/consumer-pattern with wrong locking	1	1	0	1	0	1
20	68	b	correct write, unlocked read on glob. var.	1	1	0	1	0	1
21	69	c	1 reader, 3 writer, incorrect mutex usage	1	1	0	1	0	1
22	128	c	incrementing using wrong mutex	1	1	0	1	0	1
23	146	c	3 workers, 4 global variables, wrong mutex	1	1	0	1	0	1
24	301	c	2 mutexes used incorrectly	1	1	0	1	0	1
25	302	c	2 workers, using wrong mutex	1	1	0	1	0	1
26	305	b	4 workers, inconsistent locking	1	1	0	1	0	1
27	306	b	3 workers, third without sync.	1	1	0	1	0	1
28	310	c	3 workers, one uses wrong mutex	1	1	0	1	0	1
29	311	c	4 threads, thread 4 uses wrong mutex	1	1	0	1	0	1
			Number of races detected	19	24		19		14
			Total number of false positive alarms			5		1	0

Table 2. Errors seeded in the SPLASH2 benchmark

Test	SPLASH2 App	File	Code Lines	Effect	Testtype	TachoRace
1	cholesky	malloc.C	141 - 145	program crashes	true-positive	detected
2	cholesky	malloc.C	150 - 188	not visible	true-positive	detected
3	cholesky	malloc.C	198 - 204	not visible	true-positive	detected
4	cholesky	malloc.C	277 - 279	not visible	true-positive	detected
5	cholesky	mf.C	109 - 126	not visible	true-positive	detected
6	cholesky	mf.C	148 - 162	not visible	true-positive	detected
7	cholesky	solve.C	329 - 332	wrong PIDs	true-positive	detected
8	cholesky	solve.C	349 - 360	not visible	true-positive	detected
9	cholesky	solve.C	372 - 382	not visible	true-positive	detected
10	water-nsquared	interf.C	145 - 151	not visible	true-positive	detected
11	water-nsquared	intraf.C	133 - 137	not visible	true-positive	detected
12	raytrace	shade.C	200 - 205	not visible	true-positive	detected
13	raytrace	shade.C	279 - 283	not visible	true-positive	detected

Table 3. The message traffic overhead introduced by TachoRace is low, compared to the regular MESI traffic

Test Case No.	TachoRace Overhead					Regular Messages							
	RaceContinue		RaceWait		Counter Accesses	MESI_Invalidate		MESI_Shared		MESI_Retry		Total Messages	
	#msgs	%	#msgs	%		#msgs	%	#msgs	%	#msgs	%	#msgs	%
1	0	0	0	0	0	5070	90.01	529	9.39	34	0.60	5633	100
2	0	0	0	0	0	4078	92.45	275	6.23	58	1.31	4411	100
3	0	0	0	0	0	4356	91.76	339	7.14	52	1.10	4747	100
4	0	0	0	0	0	6326	95.19	264	3.97	56	0.84	6646	100
5	0	0	0	0	0	5243	95.34	232	4.22	24	0.44	5499	100
6	0	0	0	0	0	24096	98.29	364	1.48	54	0.22	24514	100
7	0	0	0	0	0	4679	92.14	331	6.52	68	1.34	5078	100
8	0	0	0	0	0	7041	94.08	383	5.12	60	0.80	7484	100
9	0	0	0	0	0	5206	92.29	378	6.70	57	1.01	5641	100
10	0	0	0	0	0	8337	92.94	552	6.15	81	0.90	8970	100
11	1	0.01	1	0.01	5	6580	91.22	577	8.00	54	0.75	7213	100
12	0	0	0	0	0	5722	91.14	497	7.92	59	0.94	6278	100
13	1	0.02	1	0.02	4	4660	88.29	568	10.76	48	0.91	5278	100
14	1	0.02	1	0.02	81	6117	94.72	288	4.46	51	0.79	6458	100
15	1	0.01	1	0.01	83	9067	95.73	346	3.65	56	0.59	9471	100
16	0	0	0	0	0	6529	94.00	361	5.20	56	0.81	6946	100
17	0	0	0	0	0	22226	95.19	1054	4.51	69	0.30	23349	100
18	0	0	0	0	0	5760	89.16	611	9.46	89	1.38	6460	100
19	2	0.04	2	0.04	131	4172	84.27	699	14.12	76	1.54	4951	100
20	300	4.02	300	4.02	464	5045	67.57	777	10.41	1044	13.98	7466	100
21	9	0.13	9	0.13	195	5998	86.25	771	11.09	167	2.40	6954	100
22	1	0.02	1	0.02	65	5298	95.00	230	4.12	47	0.84	5577	100
23	1	0.02	1	0.02	322	4971	84.80	799	13.63	90	1.54	5862	100
24	1	0.02	1	0.02	3	4058	88.41	487	10.61	43	0.94	4590	100
25	20	0.20	20	0.20	327	8644	85.61	525	5.20	888	8.79	10097	100
26	1	0.02	1	0.02	201	5086	91.85	364	6.57	85	1.54	5537	100
27	1	0.02	1	0.02	118	4517	92.77	284	5.83	66	1.36	4869	100
28	1	0.02	1	0.02	129	4493	87.38	576	11.20	71	1.38	5142	100
29	2	0.02	2	0.02	112	8179	92.00	627	7.05	80	0.90	8890	100

6 Related Work

We refer to [23] for details on our alternative approach that does not require lock annotations, but which is less accurate.

On-the-fly race detection schemes typically require specialized hardware [29]. TachoRace is the first approach alleviate this problem by exploiting synergies

between hardware required for debugging and hardware required for performance monitoring.

Some Transactional Memory approaches have been extended to detect races. For example [4] require additional registers at the granularity level of cache lines; by contrast, TachoRace works at the granularity level of memory addresses and avoids false sharing problems. *ToleRace* [22] detects patterns as shown in Figure 1, but operates on copies of shared variables and introduces additional overhead.

BugNet [15] works as an application-level debugging aid and introduces hardware extensions for event capturing. The *FastTrack* [3] dynamic detector has lightweight vector clocks but still incurs average program execution slowdown of 8.5x, which is inappropriate for online race detection detection. [21] works at the granularity of memory pages and requires lock annotations. Differing from our approach [21] require page copies containing the locked data elements as soon as a critical section is entered, which leads to high overhead and more memory consumption. *Light64* [16] introduces one additional register per core and requires each program to be executed several times, so the tool can compare data changes to detect races; such repetitions are not required for TachoRace. The detector of [20] has a lazy release consistency memory model and a theoretically exponential overhead. Programs are slowed down by a factor of 200%, and the approach has been demonstrated to work just on two out of four tested programs. *Isolator* [21] dynamically ensures isolation for programs in which some parts correctly obey a locking discipline, while others don't. In contrast to TachoRace, Isolator has a different goal and ensures that incorrectly synchronized threads do not interfere in correctly synchronized parts of the program. Isolator does not provide a detailed solution for cases in which Isolator's repair attempts would introduce deadlocks.

Contest [9] introduces sleep statements into multithreaded programs to alter buggy schedules. Healing has been demonstrated just for one bug pattern (load-store bugs), and slowdowns can be up to 3.75x. *AVIO* [11] proposes cache coherence hardware extensions to detect atomicity violations, but requires multiple program runs (some of which need to be correct) so the tool can infer invariants. *Colorama* [2] proposes hardware extensions to automatically infer critical sections, but creates additional memory overhead and even introduces races if the inference mechanism does not make correct predictions; this cannot happen with TachoRace. *Atom-Aid* [12] dynamically reduces the probability that atomicity violations can manifest; by contrast, TachoRace repairs races when they occur. *Autolocker* [14] employs program analysis to find a locking policy that does not lead to race conditions and uses lock annotations similar to TachoRace. However, resource-intensive pointer analysis would have been necessary to detect all accesses to a particular variable. In contrast to TachoRace, Autolocker may refuse to execute certain programs.

Hard [28] introduces a hardware implementation of the lock-set algorithm, but in contrast to TachoRace, it does not heal races. The hybrid dynamic race detection approach in [17] combines lock-set and happens-before-based detection to improve accuracy, but has slowdowns by orders of magnitude.

7 Conclusion

Online race detectors serve as the last safety net to prevent parallel programming bugs in applications deployed at clients from causing greater damage. The required hardware, however, is typically specialized and expensive, which makes it unlikely that it will become available in everyone's multicore system. The tradeoff approach proposed in this paper alleviates this problem by exploiting synergy effects between hardware needed for performance monitoring and hardware needed for online race detection. TachoRace not only detects common race patterns, but also automatically fixes races while programs execute. In the long run, the ideas presented in this paper can bring us closer to making on-the-fly race detection available on every multicore desktop.

Acknowledgements. We thank the Excellence Inititative and the Landesstiftung Baden-Württemberg for their support. Many thanks also to Sebastian Crüger for his support during implementation.

References

1. AMD. Amd64 architecture programmer's manual (September 2007), http://www.amd.com
2. Ceze, L., et al.: Colorama: Architectural support for data-centric synchronization. In: Proc. IEEE HPCA 2007, pp. 133–144 (2007)
3. Flanagan, C., Freund, S.N.: Fasttrack: efficient and precise dynamic race detection. In: Proc. PLDI 2009, pp. 121–133. ACM, New York (2009)
4. Gupta, S., et al.: Using hardware transactional memory for data race detection. In: Proc. IEEE IPDPS 2009, pp. 1–11 (2009)
5. Helmbold, D.P., McDowell, C.E.: A taxonomy of race detection algorithms. Technical report, University of California at Santa Cruz, UCSC-CRL-94-35, Santa Cruz, CA, USA, September 28 (1994)
6. Intel. Intel 64 and IA-32 architectures software developer's manual (December 2009), www.intel.com
7. Intel. Intel thread checker v.3.1 (2011), http://software.intel.com
8. Jannesari, A., et al.: Helgrind+: An efficient dynamic race detector. In: Proc. IEEE IPDPS 2009 (2009)
9. Krena, B., et al.: Healing data races on-the-fly. In: Proc. ACM PADTAD 2007, pp. 54–64 (2007)
10. Lea, D.: The java.util. concurrent synchronizer framework. Sci. Comp. Prog 58(3) (2005)
11. Lu, S., et al.: Avio: detecting atomicity violations via access interleaving invariants. In: Proc. ASPLOS-XII, pp. 37–48. ACM, New York (2006)
12. Lucia, B., et al.: Atom-aid: Detecting and surviving atomicity violations. In: Proc. ISCA 2008, pp. 277–288. ACM, New York (2008)
13. Luk, C.-K., et al.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proc. PLDI 2005, pp. 190–200. ACM, New York (2005)
14. McCloskey, B., et al.: Autolocker: synchronization inference for atomic sections. In: Proc. POPL 2006. ACM, New York (2006)

15. Narayanasamy, S., et al.: Bugnet: Continuously recording program execution for deterministic replay debugging. In: Proc. ISCA 2005, pp. 284–295. ACM, New York (2005)
16. Nistor, A., et al.: Light64: Lightweight hardware support for data race detection during systematic testing of parallel programs. In: Proc. IEEE MICRO 2009 (2009)
17. O'Callahan, R., Choi, J.-D.: Hybrid dynamic data race detection. In: Proc. PPoPP 2003. ACM, New York (2003)
18. Pankratius, V., Jannesari, A., Tichy, W.: Parallelizing bzip2: A case study in multicore software engineering. IEEE Software 26(6), 70–77 (2009)
19. Pankratius, V., Adl-Tabatabai, A.-R.: A Study of Transactional Memory vs. Locks in Practice. In: Proc. SPAA 2011. ACM, New York (2011)
20. Perkovic, D., Keleher, P.J.: Online data-race detection via coherency guarantees. In: Proc. OSDI 1996. USENIX (1996)
21. Rajamani, S., et al.: Isolator: dynamically ensuring isolation in concurrent programs. In: Proc. ASPLOS 2009. ACM, New York (2009)
22. Ratanaworabhan, P., et al.: Detecting and tolerating asymmetric races. In: Proc. PPoPP 2009. ACM, New York (2009)
23. Schimmel, J., Pankratius, V.: TachoRace: Exploiting Performance Counters for Run-Time Race Detection. Technical Report 2010-01, Karlsruhe Institute of Technology, Germany (April 2010)
24. Slater, R., Tibrewala, N.: Optimizing the mesi cache coherence protocol for multithreaded applications on small symmetric multiprocessor systems (1998), http://tibrewala.net/papers/mesi98
25. Valgrind-project. Data-race-test:test suite for helgrind, a data race detector (2008)
26. Valgrind-project. Helgrind: a data-race detector (2011), http://valgrind.org
27. Woo, S., et al.: The SPLASH-2 programs: characterization and methodological considerations. In: Proc. ISCA 1995. ACM, New York (1995)
28. Zhou, P., et al.: Hard: Hardware-assisted lockset-based race detection. In: Proc. IEEE HPCA 2007, pp. 121–132 (2007)
29. Zhou, Y., Torrellas, J.: Deploying architectural support for software defect detection in future processors. In: Workshop on the Evaluation of Software Defect Detection Tools (2005)

Accelerating Data Race Detection with Minimal Hardware Support

Rodrigo Gonzalez-Alberquilla[1], Karin Strauss[2,3], Luis Ceze[3], and Luis Piñuel[1]

[1] Univ. Complutense de Madrid, Madrid, Spain
{rogonzal,lpinuel}@pdi.ucm.es
[2] Microsoft Research, Redmond WA, USA
kstrauss@microsoft.com
[3] University of Washington, Seattle WA, USA
luisceze@cs.washington.edu

Abstract. We propose a high performance hybrid hardware/software solution to race detection that uses minimal hardware support. This hardware extension consists of a single extra instruction, StateChk, that simply returns the coherence state of a cache block without requiring any complex traps to handlers. To leverage this support, we propose a new algorithm for race detection. This detection algorithm uses StateChk to eliminate many expensive operations. We also propose a new execution schedule manipulation heuristic to achieve high coverage rapidly. This approach is capable of detecting virtually all data races detected by a traditional happened-before data race detection approach, but at significantly lower space and performance overhead.

1 Introduction

Writing much-needed multithreaded programs often requires dealing with concurrency bugs that result from subtle interaction between threads. Among these bugs, data races are the most common. Unsynchronized accesses to shared data could lead to crashes or silent data corruption, so current languages including Java [6] and the new C++ standard [11] disallow or discourage data races.

Researchers have proposed a variety of mechanisms to detect and avoid data races, including many hardware-only [8,9,12,13,18] and software-only [14,15,17] solutions. Hardware-only solutions are typically complex. They require extensive hardware support, like changes to the cache hierarchy, extending cache coherence messages with additional information, and modifying the cache coherence protocol state machine to check for events of interest. The storage requirements, many times close to key processor structures, are also quite prohibitive. Software-only solutions, on the other hand, require no modification to the architecture, but are typically too slow to be an always-on feature. The analysis operations performed in software are slower, and these algorithms require a significant amount of metadata and frequent inter-thread communication.

We propose a hybrid solution: hardware support is boiled down to the bare minimum, reducing complexity, and making detection of inter-thread communication much faster than prior approaches. We augment the ISA with one simple instruction that takes an address as input and returns the coherence state of the cache block containing that

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 27–38, 2011.

address. We also propose a new algorithm that uses this support to effectively detect data races. Our solution leverages two key insights: (1) the dynamic information we need can be extracted from coherence state already tracked by the hardware; (2) there is a well-defined category of dynamic data races that are much cheaper to detect and yet can be proven to include all static data races given sufficient executions. We also show how to perturb execution schedules to speed up the exposure of data races to the detection mechanism, achieving high accuracy compared to traditional happened-before data race detection, but at significantly lower space and time overheads.

Sections 2, 3 and 4 explain the proposed hybrid system, Sections 5, 6 and 7 evaluate it and compare it to previous work, and Section 8 concludes.

2 Background

Terminology. A *data race* exists if there is no synchronization order between any two accesses to the same address by different threads, at least one of them being a write access. A *static race* is a pair of static instructions that, when executed, may be involved in a data race, and a *dynamic race* is one manifestation of a data race at execution time. An *epoch* is the set of dynamic instructions in a thread executed between two consecutive synchronization operations. To simplify our discussion, we assume a static direct mapping of threads to cores. We discuss how to relax this assumption in Sections 4.1 and 4.4.

Data race detection. The basic approaches to data race detection are happened-before-based [13,14] and lockset-based algorithms [15,18]. We focus on the former, which leverage Lamport's happened-before relation [2] (to partially order memory accesses based on observed synchronization operations) and program order to determine if conflicting accesses are logically concurrent. Due to space constraints, we omit an explanation of happens-before race detection (HapB) and FastTrack (FastT), a state-of-the-art software implementation of HapB for Java, but we expect the reader to be familiar with them [1,2,14].

Cache coherence. Without loss of generality, we assume an invalidate-based MESI protocol. The coherence state of a cache line implicitly carries valuable information about recent accesses, *e.g.*, if the block is in M state in a cache, the line was last written by the local processor; the E state indicates the cache has read that block before; the S state indicates the cache has read or written that block before, and then another cache may have requested that block; the I state indicates that a remote write happened. We leverage this implicit information for lightweight memory access monitoring.

3 Minimal Hardware Support for Data Race Detection

Our proposed minimal hardware support for data race detection consists of simply exposing the coherence state of a cache block to software via one additional instruction. A software layer then records and uses the state information to detect data races. To leverage this support, we propose a new race detection algorithm, "AccessedBefore", or simply AccB. Its key idea is to use a software-managed address-indexed table to

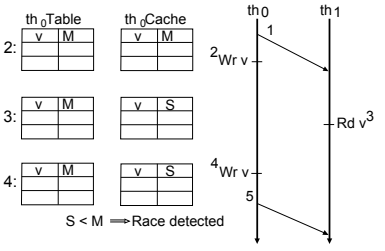


Fig. 1. Race detection with coherence state

Table 1. Types of downgrades and races

Transition at local thread	Access at local thread	Access at remote thread	Race type
$M \rightarrow S$	write	read	W→R
$M \rightarrow I$	write	write	W→W
$M \rightarrow I$	read	write	R→W
$E \rightarrow I$			
$S \rightarrow I$			

track the last observed state of cache blocks and detect if they have been downgraded within the boundaries of an epoch. A downgraded block within an epoch indicates a potential data race: a remote cache has issued an upgrade request to the block. Note that all state necessary to the analysis is local to a thread, so no inter-thread communication is required; HapB, in contrast, requires substantial inter-thread communication.

3.1 AccessedBefore (AccB) Algorithm

Figure 1 illustrates how AccB works. Thread 0 performs a synchronization operation and starts an epoch (1). When thread 0 performs a write to variable v (2), the corresponding cache block transitions to M state and the software layer records the pair of $address$ and $state < v, M >$ in its local table. When thread 1 subsequently reads variable v (3), the block cached by thread 0 is downgraded to S state. At this point, the software layer is unaware of the downgrade. Finally, when thread 0 is about to write v again (4), the software layer reexamines the current state of v's block (S) and the state recorded in its local table (M), observes a downgrade has happened and detects the race. If this last write never happens, the downgrade check is performed when the epoch ends (5).

Table 1 shows the different types of downgrade and the races they indicate. For example, the first row corresponds to the example in Figure 1. Table 2 summarizes AccB's operation by showing the actions taken by the software layer on each relevant event. Again, note that all analysis actions are local to a thread: the only communication between threads happens through the cache coherence protocol (which would be present even in the absence of AccB). Also, the information collected into the local table only pertains to a *single epoch*, as we are not interested in downgrades across synchronization operations. In addition, AccB epochs can be redefined to the instructions executed between two source synchronization operations because AccB does not require any notion of ordering with previous epochs from other threads, while HapB does. These are three important advantages of our approach when compared to HapB, which has additional storage and communication requirements.

In essence, AccB looks for access conflicts between concurrently running epochs, which must be the result of a race (or false sharing as described below). Thus, a race will be detected if the epochs with racy accesses overlap in time.

Interestingly, for every race in a program there must exist an execution in which the epochs of the racy instructions overlap in time (we formally proved this statement but leave it out due to space constraints). We propose an optimization to quickly expose

Table 2. Events of interest and related algorithm actions

Event of interest	Algorithm action
Beginning of epoch	Clear local table.
Before memory access	Check the current state of the corresponding cache block against the entry in its local table (if any) to detect downgrades.
After memory access	Record the state of the corresponding cache block in its local table.
End of epoch	Check every entry in its local table and their corresponding state in the cache to detect remaining downgrades.

races to AccB: carefully perturbing the execution schedule to increase the probability of overlapping racy epochs.

3.2 Sources of Inaccuracy

We now discuss the two sources of inaccuracy in AccB: (1) false sharing, and (2) block evictions. The next section discusses optimizations to mitigate them.

False sharing. To keep the hardware support required by our proposal to a minimum, we do not extend the memory access information to granularity finer than what is already provided by coherence protocols: a cache block. False sharing of the block may result in false positives. Other race detection approaches at the same granularity would have the same limitation (*e.g.*, HapB). Moreover, our approach can be easily extended to finer granularity if necessary (at extra cost) and is orthogonal to software techniques to mitigate false positives.

Cache block eviction. On eviction, a block loses its state thus the cache loses its ability to detect downgrades, so races may be missed (false negatives).

4 Implementation

4.1 Hardware Support

We extend the ISA of an off-the-shelf multiprocessor with a `StateChk` (`StChk off(base),reg`) instruction, which returns the state of `off(base)`'s cache block in register `reg`. If the block is not present in the cache, `StateChk` returns a special NotPresent (NP) state to distinguish from a block in Invalid state. The last valid state is returned if the cache block is in a transient state.

Implementing the `StateChk` instruction requires minor changes to (1) cache data paths, and (2) cache controllers. A new multiplexer creates a path for coherence state into the processor via the existing cache data path. Cache controllers suffer one modification: if the requested block is not currently cached, the cache controller returns the NP state without triggering a miss request.

We assume L1 caches to be the point of coherence, but other configurations are possible. They belong to one of two categories: (1) coherence is maintained among caches private to a hardware thread (*e.g.*, private non-inclusive L1 and L2 caches), and (2) coherence is maintained in caches shared by more than one hardware thread (*e.g.*, SMT processor with a single L1 data cache). In the first case, the proposed mechanism

works seamlessly: the state is obtained from the private cache where a hit happens (NP in case of a miss in all private caches). In the second case, accesses and resulting changes of state by different threads need to be distinguished by replicating the state for each thread.

4.2 Software Layer

Data structures. A thread-local hash table records information about accesses performed during an epoch. This table is indexed by data address and stored in main memory. Each entry contains the expected state (based on the type of the last access to the cache block) for the corresponding block and the address of the instruction that performed the last local access to the address.

Instrumentation points. We use dynamic binary rewriting to instrument every source synchronization operation (thread creation, mutex and conditional variable creation, lock release, and waiting) and every memory operation not involved in a synchronization operation. The epoch ending instrumentation is inserted right before source synchronization operations. It searches the local table for any downgraded variables in the ending epoch and subsequently clears the table in preparation for the next epoch.

The memory access instrumentation checks the state of the corresponding address in the cache via a `StateChk` instruction, and compares it with the state recorded in the table. If it detects a downgrade, it reports the race with the corresponding address and the instruction address of the previous access. It then updates the state in the table with the maximum (following the order $M > E = S > I$) of the recorded state and the current state. Using the maximum is safer than executing `StateChk` again after the instrumented access executes because downgrades could be missed in the window between the instrumented access executes and the second `StateChk` instruction executes.

4.3 Optimizations

These optimizations improve accuracy and reduce instrumentation overhead.

Coverage improvement with schedule perturbation. AccB only detects races between epochs that overlap in time. We perturb executions to encourage an increased variety of overlapping epoch sets. When an epoch starts, the thread randomly chooses an action: (1) to continue executing normally, or (2) to join its thread to a rescheduling barrier. The thread waits at this barrier until a bounded random timeout occurs. At this point, all threads that joined this first barrier start executing their epochs. Once a thread finishes its epoch, it joins a checking barrier. When all threads that joined the first barrier join this second barrier, or it times out, epoch checks are done and all threads continue.

Further reducing overheads with extra hardware support (AccB++). We can further accelerate AccB with very simple modifications: we add a small number of metadata bits to caches and use them to reduce the number of accesses to the local table. The coherence state of each cache block is augmented with two extra bits, namely, locally read bit (lrd) and locally written bit (lwr), and caches are augmented with a single

downgraded bit (dgd). These bits record the nature of the local accesses within the last epoch (lrd or lwr) to a particular cache block, and downgrades (dgd) to any cache block touched by the local thread within that epoch. An additional instruction gang-clears these bits in the local cache and is used by the software layer in the beginning of every epoch. The cache controller is modified to set lrd or lwr on a local read or local write access, respectively, and to set dgd on downgrades due to remote requests (but only if either lrd or lwr for that block is set). Finally, the `StateChk` instruction returns these three bits together with the regular coherence state.

The software layer uses these additional bits to detect accesses followed by downgrades within an epoch. A `StateChk` instruction is inserted immediately before each memory access and the dgd bit is checked. If the dgd bit is set, a data race is detected. These bits optimize how the local table is used: the lrd and lwr bits reduce the number of accesses to the table, since only information about the first read and write accesses to a variable in each epoch need to be recorded (this is sufficient to report one data race – others may be detected once the first is eliminated). On every memory access, instead of checking if the address is present in the table, the lrd and lwr bits are checked. If none are set, this is the first access to this variable within the current epoch, so the address and the corresponding instruction address are added to the table. If only the lrd bit is set and the access being instrumented is a write, this is the first write access to the variable, so the instruction address of the table entry is updated. If the lwr bit is already set, no new updates are needed. Note this does not completely eliminate the use of the local table because it is still necessary for end-of-epoch checks and for recording the instruction address of accesses.

A small victim cache next to the data cache reduces the impact of cache block evictions. Whenever a block that has its lrd and/or lwr bit set is evicted from the data cache, it is cached in the victim cache. This allows reporting a race even if the block involved in the race has been evicted from the data cache.

4.4 System Issues

Thread migration. Thread migration can lead to changes to the coherence states observed by AccB, affecting its accuracy. To mitigate this potential problem, the software layer may check via an instruction like x86's CPUID in which core the thread is running at every epoch end and compare it with the core identification number recorded in the previous epoch. If they are different, a migration has taken place and the instrumentation ignores any races detected for that epoch. Note that to preserve locality, the OS typically keeps the mapping between threads and cores as stable as possible. Finally, epochs typically run much faster than context switch time scales. Therefore, thread migration is unlikely to lead to major accuracy degradation in AccB.

Speculation. Speculative execution can cause additional false positives in a few scenarios: (1) `StateChk` is executed speculatively in the local core, (2) load is executed speculatively in a remote core, and (3) prefetch request is issued in a remote core. The simplest solution is to allow false positives, which are likely to be low. Other solutions to the first problem are to either reuse mechanisms traditionally used for load speculation (*e.g.*, replay or snoop) or to only set the access bit when the load retires. Solutions to (2) consist of limiting speculation to when it is safe. For example, allowing a speculative

load to proceed only when it reaches the point-of-no-return in designs like CHERRY [7] or if the cache block is in the local cache in a valid state. (3) can be easily mitigated by turning prefetching off during debugging runs; an alternative is marking prefetches until later access confirmation, at the cost of extra complexity.

5 Experimental Setup

We evaluate AccB using the PIN [5] dynamic binary instrumentation framework with a tool that includes the software layer from Section 4 and a detailed memory hierarchy: 8 32KB 8-way set associative LRU DL1s with 64-byte blocks and MESI coherence. The latency of `StateChk` is the same as a cache hit.

We compare AccB with an implementation of HapB and FastT using the same instrumentation framework. All algorithms are exposed to the same memory interleavings for accuracy comparisons. HapB is complete, so we verified that every race found by AccB has also been found by its HapB counterpart.

HapB Implementation. We have carefully optimized HapB by using hash-sets for read- and write-sets and Bloom-filters to speed up intersections of hash-sets. Same-thread epochs are stored in an ordered linked list and pruned as soon as an old epoch is ordered before all current epochs (space-optimal implementation). Vector clocks are implemented as regular arrays.

FastT Implementation. We implemented FastT for C++. Unlike the original implementation for Java, which embeds metadata in the object, the implementation for C++ stores metadata in a global table because C++ is not type safe.

Benchmarks. We use the SPLASH-2 benchmarks [16] and commercial workloads (Apache httpd server, MySQL database, AGet, PBZip) compiled with gcc's standard *-O2* optimization flag and run with 8 threads. We do not report performance for AGet and PBZip because they are non-deterministic. We verify that AccB detects races reported in the literature for Apache and MySQL [3].

6 Evaluation

6.1 AccB versus HapB

Table 3 compares AccB and HapB in terms of performance, space overhead and accuracy. The first group of rows in Table 3 show the speedup of an application instrumented with AccB, and with extra hardware support (AccB++), compared to HapB. For example, barnes instrumented with AccB runs 11% faster than when instrumented with HapB. The speedup grows to almost $2\times$ with AccB++. Overall, AccB++ achieves speedups of up to almost $6\times$. A few benchmarks (lu, radix, and water spatial) experience modest slowdowns with AccB, caused by the type of synchronization used in these benchmarks: most synchronization is based on barriers, which allow HapB to clean up all information about old epochs and significantly reduce its checking overheads. AccB

Table 3. Performance, space overhead and accuracy comparison of AccB and HapB

	brns	*chlsk*	*fft*	*fmm*	*lu*		*ocean*		*radx*	*rayt*	*vrnd*	*water*		*aget*	*pbzip*
					cnt	*ncnt*	*cnt*	*ncnt*				*nsqr*	*spt*		
Speedup (× — HapB/AccB)															
AccB	1.11	1.03	1.02	1.16	0.98	0.95	1.19	1.21	0.98	1.08	5.71	1.23	0.99	–	–
AccB++	1.99	1.31	1.27	1.55	1.54	1.23	1.19	1.21	1.04	1.23	5.90	1.71	1.30	–	–
FastT †	0.03	0.01	0.07	0.01	0.08	0.08	0.33	0.33	0.18	0.21	0.29	0.08	0.06	–	–
Space overhead (%)															
AccB avg	0.8	9.4	31.8	3.3	19.9	20.2	25.6	26.1	32.9	41.0	0.2	15.8	30.6	0.1	5.6
AccB max	77.1	13.0	87.5	29.6	33.3	33.2	110.6	113.3	116.5	40.4	0.4	25.7	80.1	0.1	32.1
Accuracy (%)															
AccB	97.8	–	–	95.4	–	–	100.0	100.0	–	100.0	100.0	–	–	100.0	100.0

incurs extra overheads because it performs table checks on every memory access in addition to end-of-epoch checks. Note that AccB++ always shows speedups[1].

The second group shows average and maximum space overheads for AccB over HapB. For example, AccB uses on average 0.8% and at most 77% of the storage used by HapB for barnes. For most benchmarks, AccB uses significantly less space than HapB. In some cases (ocean, radix), AccB incurs a higher maximum space overhead compared to HapB (but the average is still lower). This is due to uncommon program behavior: frequent barriers and large accessed sets.

The last row shows accuracy, *i.e.*, how many races AccB detects compared to HapB for 500 runs. AccB detects all races for most benchmarks. Section 6.3 provides more insight into those very few races not detected by AccB.

6.2 Overheads Characterization

Performance. Table 4 characterizes the performance overheads of AccB and AccB++ compared to HapB, aggregated for all benchmarks. This study is data structure independent: it counts high level operations to each algorithm's data structures, *i.e., lookups* and *updates*. The numbers show the relative frequency of events for AccB and AccB++, normalized to HapB. Lookups (row 2) and updates (row 3) are direct accesses to AccB's local table and to HapB's sets. Branches (row 4) refer to branches taken while manipulating these data structures. AccB incurs many more lookups than HapB because AccB performs lookups at every memory access, while HapB performs them only at epoch ends. Even though AccB's lookups are more frequent, AccB is still faster than HapB because there is high locality in AccB's table accesses and most are cache hits (besides being thread-local). Also, HapB is very control flow intensive, as demonstrated by the large number of branches. HapB's data structures are larger (due to multiple epochs, not just the current), which results in worse cache behavior. Finally, HapB requires transferring vector clocks and epoch information, which implies additional communication among threads, *i.e.*, costly misses.

[1] † The results show that FastT is much slower than HapB. The reason is twofold: first, FastT experiences additional overheads compared to its original Java implementation due to the global table required by C++; second, HapB performs intersections at the end of each epoch, FastT performs checks at every access.

Table 4. Number of operations executed by AccB and AccB++ compared to HapB

	AccB	AccB++
Lookups	3326.7%	15.6%
Updates	100.0%	29.8%
Branches	4.2%	5.1%

Table 5. Overheads, storage requirements of HapB and AccB

	HapB	AccB
Avg. entries per epoch	360.3	623.2
Avg. epochs in history	16.5	0
Avg. simultaneous entries	71.6k	9.1k
Size (MB)	2.15	0.28

Table 6. Relative percentage of false positives in AccB compared to HapB

brns	*chlsk*	*fft*	*fmm*	*lu*		*ocean*		*radx*	*rayt*	*vrnd*	*water*		*aget*	*pbzip*
				cnt	*ncnt*	*cnt*	*ncnt*				*nsqr*	*spt*		
99.1	98.3	100.0	81.3	100.0	100.0	100.0	98.6	100.0	100.0	100.0	100.0	100.0	51.8	75.0

With simple additional support, AccB++ has lower overheads than AccB. AccB++ reduces lookups by two orders of magnitude and updates by more than 60%. AccB++ has higher number of branches, but still much lower than HapB.

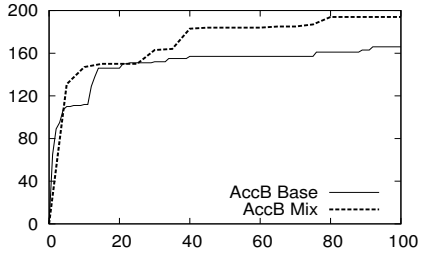
Space. Table 5 shows the space overhead of HapB and AccB averaged across all benchmarks. It reports the number of entries per epoch (row 2), overall number of epochs kept in history (row 3), total number of entries used by all epochs in all threads simultaneously (row 4) and overall storage requirements (row 5).

AccB records more entries per epoch than HapB (row 2). This is due to AccB only ending epochs at synchronization sources, which makes AccB epochs longer. AccB keeps no history while HapB keeps history on 16.5 epochs on average (row 3). AccB requires a much lower total number of entries (over 7× fewer). Overall, AccB reduces space overhead by more than 7×. Storage requirements for AccB++ are similar to AccB. In addition to being larger, the storage HapB requires is *shared* and accessed by all threads when their epochs end. Conversely, the storage AccB requires is much smaller and *purely local*.

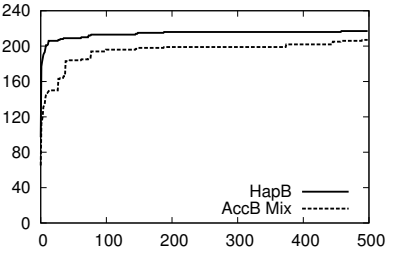
6.3 Accuracy Characterization

False positives. Table 6 shows the false positives detected by AccB relative to HapB at the same tracking granularity. AccB never has more false positives than HapB. False positives are inherent to the tracking granularity (cache blocks) for both AccB and HapB and can be reduced with additional software support (*e.g.*, by changing the data layout to avoid false sharing), but this is beyond the scope of this paper.

False negatives. As explained in Section 3.2, AccB has two sources of false negatives (*i.e.*, missed races). The first is due to limited cache capacity, which causes cache blocks to be evicted and the access information to be lost (*CBE* – cache block evictions). The second is due to epochs with races not overlapping on AccB executions. We separate the two effects by modifying our simulator with unbounded space to store evicted cache blocks, such that the CBE problem is completely eliminated. No new races were found, so all races missed by AccB for these benchmarks are due to non-overlapping epochs, a problem that can be addressed with scheduling perturbations and/or multiple runs.



(a) Aggregate number of static races found as the number of executions increases for AccB and AccB with scheduling perturbations (AccB Mix).



(b) Aggregate number of static races found as the number of executions increases for AccB Mix, compared to HapB.

Fig. 2. Sensitivity to scheduling perturbations and number of runs

Other benchmarks with larger epochs could cause the CBE problem. However, architectures with private L2 caches are common today, so there is much more space than the DL1s used in this evaluation. Alternatively, a victim cache that only stores evicted downgraded lines may be sufficient to mitigate the problem.

Sensitivity to scheduling perturbations and number of runs. Figure 2(a) shows how the aggregate number of static races detected by AccB, with and without scheduling perturbations (AccB Mix and AccB Base), grows with the number of executions for fmm. After about 25 runs, AccB Mix clearly shows new races while AccB Base does not. This happens when the scheduling perturbations start exposing more diverse epoch overlaps. These results also show that scheduling perturbations indeed help AccB find races faster.

Figure 2(b) shows how fast AccB Mix approximates the number of static races detected by HapB over 500 runs. AccB detects most races in the first few executions (about 2/3 are detected within the first 10 runs). The number of races AccB Mix detects continues growing after that, although increasingly more slowly. We manually inspected a few of the races that AccB had not detected after 500 runs and found that for each undetected race there was another race that originated at the same programming mistake (*e.g.*, missing critical section) and that was successfully detected by AccB.

7 Related Work

Conflict exceptions [4] (CE) relates to our work in the type of bugs it detects. CE detects when a synchronization-free region (epoch) conflicts with another concurrent synchronization-free region. Such conflicts can only happen when a data race exists. This is in essence the same type of event AccB detects. However, CE detects these events in a fully precise way, in order to throw an exception. This requires significantly more hardware (50% cache overhead for access bits). We sacrifice some precision in order to keep hardware at a minimum. AVIO [3] is an atomicity violation detector that also augments and leverages coherence state. However, atomicity violations do not necessarily imply data races.

The works most related to ours are by Min and Choi [8], and Nagarajan and Gupta [10]. Both propose using traps to expose certain cache coherence events to enable analysis of parallel program behavior. Nagarajan and Gupta [10] showcased their mechanism with deterministic replay and barrier speculation. Min and Choi [8] developed a limited form of happened-before detection for a subclass of programs (structured parallelism only). In contrast, our hardware proposal does not rely on software traps; it is essentially a *load operation that returns coherence state*. Software traps are arguably more flexible, but are also much more costly to implement. Importantly, these proposals focus on other applications of tracking coherence events. We propose a *new race detection algorithm* that uses our novel hardware support to reduce performance overheads, and also significantly reduce space overhead compared to happened-before.

8 Conclusions

In this paper, we propose a data race detection solution that requires minimal hardware support. This solution captures many of the same races a more traditional mechanism based on happened-before captures, but at much lower overheads. We expect the overhead reductions and the hardware simplicity to make this solution sufficiently compelling for multicore designers to include support in their designs.

Acknowledgements. This work was supported in part by the National Science Foundation under grant CCF-1016495, the Spanish government under CICYT-TIN 2008/00508 and an FPU grant, a Microsoft Faculty Fellowship, and gifts from Intel.

References

1. Flanagan, C., Freund, S.N.: FastTrack: Efficient and Precise Dynamic Race Detection. In: Conference on Programming Language Design and Implementation (2009)
2. Lamport, L.: Time, Clocks and the Ordering of Events in a Distributed System. Communications of the ACM (1978)
3. Lu, S., et al.: AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants. In: International Conference on Architectural Support for Programming Languages and Operating Systems (2006)
4. Lucia, B., et al.: Conflict Exceptions: Providing Simple Concurrent Language Semantics with Precise Hardware Exceptions. In: International Symposium on Computer Architecture (2010)
5. Luk, C.-K., et al.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: Conference on Programming Language Design and Implementation (2005)
6. Manson, J., Pugh, W., Adve, S.: The Java Memory Model. In: Symposium on Principles of Programming Languages (2005)
7. Martinez, J., et al.: Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. In: International Symposium on Microarchitecture (2002)
8. Min, S.L., Choi, J.-D.: An Efficient Cache-based Access Anomaly Detection Scheme. In: International Conference on Architectural Support for Programming Languages and Operating Systems (1991)
9. Muzahid, A., et al.: SigRace: Signature-Based Data Race Detection. In: International Symposium on Computer Architecture (2009)

10. Nagarajan, V., Gupta, R.: ECMon: Exposing Cache Events for Monitoring. In: International Symposium on Computer Architecture (2009)
11. Nelson, C., Boehm, H.-J.: Concurrency Memory Model. C++ standards committee paper (October 2007)
12. Prvulovic, M.: CORD: Cost-effective (and Nearly Overhead-free) Order-recording and Data Race Detection. In: International Symposium on High Performance Computer Architecture (2006)
13. Prvulovic, M., Torrellas, J.: ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In: International Symposium on Computer Architecture (2003)
14. Ronsse, M., De Bosschere, K.: RecPlay: a fully integrated practical record/replay system. ACM Transactions on Computer Systems 17, 2 (1999)
15. Savage, S., et al.: Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Transactions on Computer Systems 15, 4 (1997)
16. Woo, S., et al.: The SPLASH-2 Programs: Characterization and Methodological Considerations. In: International Symposium on Computer Architecture (1995)
17. Yu, Y., Rodeheffer, T., Chen, W.: RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In: Symposium on Operating Systems Principles (2005)
18. Zhou, P., et al.: HARD: Hardware-Assisted Lockset-based Race Detection. In: International Symposium on High Performance Computer Architecture (2007)

Quantifying the Potential Task-Based Dataflow Parallelism in MPI Applications

Vladimir Subotic, Roger Ferrer, Jose Carlos Sancho,
Jesús Labarta, and Mateo Valero

Barcelona Supercomputing Center
Universitat Politecnica de Catalunya
{vladimir.subotic,roger.ferrer,jsancho,jesus.labarta,mateo.valero}@bsc.es

Abstract. Task-based parallel programming languages require the programmer to partition the traditional sequential code into smaller tasks in order to take advantage of the existing dataflow parallelism inherent in the applications. However, obtaining the partitioning that achieves optimal parallelism is not trivial because it depends on many parameters such as the underlying data dependencies and global problem partitioning. In order to help the process of finding a partitioning that achieves high parallelism, this paper introduces a framework that a programmer can use to: 1) estimate how much his application could benefit from dataflow parallelism; and 2) find the best strategy to expose dataflow parallelism in his application. Our framework automatically detects data dependencies among tasks in order to estimate the potential parallelism in the application. Furthermore, based on the framework, we develop an interactive approach to find the optimal partitioning of code. To illustrate this approach, we present a case study of porting High Performance Linpack from MPI to MPI/SMPSs. The presented approach requires only superficial knowledge of the studied code and iteratively leads to the optimal partitioning strategy. Finally, the environment provides visualization of the simulated MPI/SMPSs execution, thus allowing the developer to qualitatively inspect potential parallelization bottlenecks.

1 Introduction

New proposals for large-scale programming models are persistently spawned, but most of these initiatives fail because they attract little interest of the community. It takes a giant leap of faith for a programmer to take his already working parallel application and to port it to a novel programming model. This is especially problematic because the programmer cannot anticipate how would his application perform if it was ported to the new programming model, so he may doubt whether the porting is worth the effort. Moreover, the programmer usually lacks developing tools that would make the process of porting easier.

MPI/SMPSs is a new hybrid dataflow programming model that showed to be efficient for numerous applications. In a manner similar to MPI/OpenMP, MPI/SMPSs parallelizes computation of the distributed-memory nodes using

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 39–51, 2011.

MPI [13], while it parallelizes computation of the shared-memory cores using SMPSs [10], a task-based dataflow programming model. This integration of message-passing paradigm and dataflow execution potentially extracts distant parallelism (parallelism of code sections that are mutually "far" from each other). Finally, MPI/SMPSs outperforms MPI in numerous codes [8], among which is the High Performance Linpack (HPL), the application that is used to rank the parallel machines on the top 500 supercomputers lists [1].

To continue its progress, MPI/SMPSs must get wider community involved by encouraging MPI programmers to port their applications to MPI/SMPSs. This encouragement is strictly related to assuring the programmer that he can benefit from this porting and that the porting would be easy. Therefore, our goal in this study is to develop a framework that provides support to:

- help an MPI programmer estimate how much parallelism MPI/SMPSs can achieve in his MPI application, so he can decide whether the porting is worth the effort.
- help an MPI programmer find the optimal strategy to port his MPI application to MPI/SMPSs.

2 SMPSs Programming Model

SMPSs [10] is a new shared-memory task-based parallel programming model that uses dataflow to exploit parallelism. SMPSs slightly extends C, C++ and Fortran, offering semantics to declare some part of a code as a task, and to specify memory regions on which that task operates. In porting a sequential code to SMPSs, the programmer has to specify the following: *taskification* – to mark with pragma statements the functions that should be executed as tasks; and *directionality of parameters* – to mark inside pragmas how are the passed arguments used within these function. The specified directionality can be: *input*, *output* and *inout*. Figure 1 illustrates the annotations needed to port a sequential C code to SMPSs.

Given the annotations, the runtime is free to schedule all tasks out-of-order, as long as the data dependencies are satisfied. The main thread starts and when it reaches a taskified function, it instantiates it as a task and proceeds with the execution. Based on the parameters' directionality, the runtime places the task

```
#pragma css task   input(A[SizeA]) output (B[SizeB])
void compute(float *A, float *B) {
    …
}
```

```
int main () {
    …
    compute(a,b);
    …
}
```

Note: The code in black presents the unchanged code of the legacy C application. Conversely, the code in dark gray presents the annotations needed to mark the *taskification choice*, while the code in light gray presents the annotations needed to declare the *directionality of parameters*.

Fig. 1. Annotations needed to port a code from sequential C to SMPSs

instance in the dependency graph of all tasks. Then, considering the dependency graph, the runtime is free to dynamically schedule the execution of tasks to achieve high parallelism. To further increase dataflow parallelism, the runtime automatically renames data objects to avoid all false dependencies (dependencies caused by buffer reuse).

Integrated with MPI, SMPSs allows to taskify functions with MPI transfers and thus potentially extract very distant parallelism. The idea is to encapsulate functions with MPI transfers inside tasks, and thus relate the messaging events to dataflow dependencies. For example, a task with *MPI_Send* of some buffer locally reads (*input* directionality) that buffer from the memory and passes it to the network, while a task with *MPI_Recv* of some buffer gets that buffer from the network and locally stores (*output* directionality) it to the memory. Taskification of transfers overcomes strong synchronization points of pure MPI execution and potentially exploits distant parallelisms, providing much better messaging behavior than fork-join based MPI/OpenMP. Marjanovic *at. el.* [8] showed that apart from better peak GFlops/s performance, compared to MPI, MPI/SMPSs delivers better tolerance to bandwidth reduction and external perturbations (such as OS noise).

3 Motivation

Finding the best taskification strategy is far from trivial. Figure 2 shows a simple sequential application composed of four computational parts (A, B, C and D), the data dependencies among those parts, and some of the possible taskification strategies. Although the application is very simple, it allows many possible taskifications that expose different amount of parallelism. $T0$ puts all code in one task and, in fact, presents non-SMPSs code. $T1$ and $T2$ both break the application into two tasks but fail to expose any parallelism. On the other hand, $T3$ and $T4$ both break the application into 3 tasks, but while $T3$ achieves no parallelism, $T4$ exposes parallelism between C and D. Finally, $T5$ breaks the application into 4 tasks but achieves the same amount of parallelism as $T4$. Considering that increasing the number of tasks increases the runtime overhead of instantiating and scheduling tasks, one can conclude that the optimal taskification is $T4$, because it gives the highest speedup with the lowest cost of the increased number of tasks. On the other hand, for a complex MPI application, the number of possible taskifications could be huge, so finding the optimal taskification can

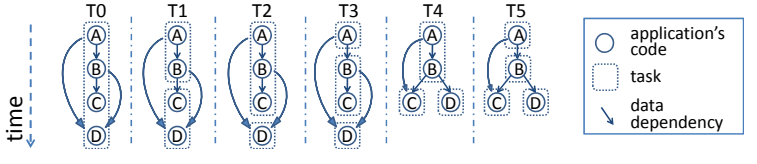


Fig. 2. Execution of different possible taskifications for a code composed of four parts

be both hard and time consuming. As a result, most likely, a programmer ends up with a sub-optimal taskification of his code.

We believe that it would be very useful to have an environment that quickly anticipates the potential parallelism of a particular taskification. We design such environment and we show how it should be used to find the optimal taskification. In this paper, as a case study we present a black-box approach to port the High Performance Linpack (HPL) from MPI to MPI/SMPSs. First, the environment instruments the studied application and generates quantitative profile of the execution. Then, considering the obtained profile the interactive trial-and-error process can start following this method: 1) the programmer proposes a coarse-grained taskification for the code; 2) given the taskification, the environment estimates potential parallelism and offers the visualization of the resulting MPI/SMPSs execution. 3) based on the output, the programmer proposes a finer-grained taskification and returns to step 2. This interactive algorithm converges into the optimal taskification.

4 Framework

The idea of the framework is to: 1) run an MPI/SMPSs code by executing tasks in the order of their instantiation; 2) dynamically detect memory usage of all tasks; 3) identify dependencies among all task instances; and 4) simulate the execution of the tasks in parallel. First, the framework forces sequential execution of all tasks, in other words it executes tasks in the order of their instantiation. That way, the instrumentation can keep the shadow data of all memory references and thus identify data dependencies among tasks. Considering the detected dependencies, the framework creates the dependency graph of all task, and finally, simulates the MPI/SMPSs execution. Moreover, the framework can visualize the simulated time-behavior and offer deeper insight into the MPI/SMPSs execution.

The framework (Figure 3) takes the **input code** and passes it through the tool chain that consists of Mercurium based **code translator**, Valgrind based

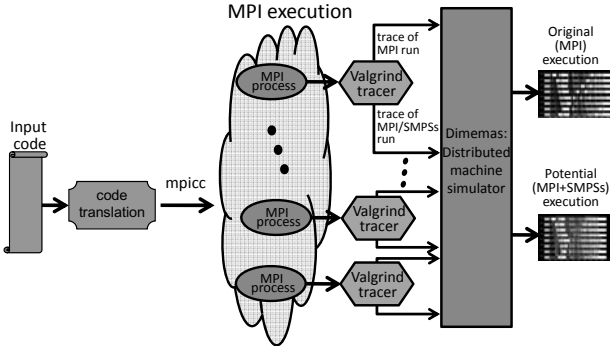


Fig. 3. The framework integrates Mercurium code translator, Valgrind tracer, Dimemas simulator and Paraver visualization tool

tracer, Dimemas **replay simulator** and Paraver **visualization tool**. Input code is a complete MPI/SMPSs code or an MPI code with only light annotations specifying the proposed *taskification*. A Mercurium based tool translates the input code in the pure MPI code with inserted functions annotating entries and exits from tasks. Then the obtained code is compiled and executed in pure MPI fashion. Each MPI process runs on top of one instance of Valgrind virtual machine that implements a designed tracer. The tracer makes the trace of the (actually executed) MPI execution, while at the same time, it reconstructs what would be the traces of the (potential) MPI/SMPSs execution. Dimemas simulator merges the obtained traces and reconstructs time-behavior of these traces on a parallel platform. Finally, Paraver can visualize the simulated time-behaviors and allow to profoundly study the differences between the (instrumented) MPI and the (corresponding simulated) MPI/SMPSs execution. In our prior work [14], we used a similar idea to estimate the potential benefits of overlapping communication and computation in pure MPI applications.

4.1 Input Code

The input code can be MPI/SMPSs code or an MPI code with light annotations. The input code has to specify which functions (parts of code) should be executed as tasks, but not the directionality of the function parameters. Thus, the input code can be an MPI code, only with annotations specifying which functions should be executed as tasks. Figure 4 on the left shows an example of an MPI code with annotated *taskification* choice.

4.2 Code Translator

Our Mercurium based tool translates the input code into the code with forced serialization of tasks. The obtained code is a pure MPI code with empty functions (*hooks*) annotating when the execution enters and exits from a task (Figure 4). The translated code is then compiled with *mpicc*, and the binary of the MPI execution is passed for further instrumentation. It is important to note that the

Input code	Translated code

`#pragma css task`
`void compute(float *A, float *B) {`
`…`
`}`
	`void compute(float *A, float *B) {`
	`…`
	`}`

`int main () {`
`  …`
`  compute(a,b);`
`  …`
`}`
	`int main () {`
	`  …`
	`  start_task_valgrind("compute");`
	`  compute(a,b);`
	`  end_task_valgrind("compute");`
	`  …`
	`}`

Note: The input code does not have to be a complete MPI/SMPSs code, because the instrumented code only needs to mark all entries/exits from each task. Thus, as shown, the input code can be an MPI application only with a specified proposed taskification.

Fig. 4. Translation of the input code required by the framework

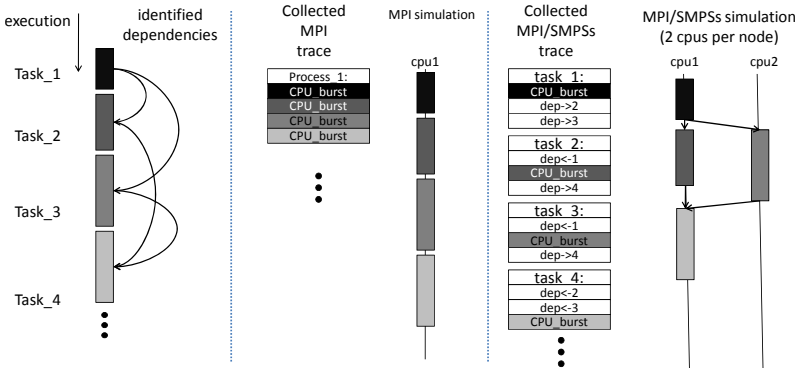
hooks can be inserted directly in the input code, allowing to declare as task any part of the application's code. This way, the framework overcomes the limitation of SMPSs runtime that only complete functions may be treated as tasks, and further eases the process of proposing taskifications.

4.3 Tracer

Valgrind [9] is a virtual machine that uses just-in-time (JIT) compilation techniques. The original code of an application never runs directly on the host processor. Instead, the code is first translated into a temporary, simpler, processor-neutral form called Intermediate Representation (IR). Then, the developer is free to do any translation of the IR, before Valgrind translates the IR back into machine code and lets the host processor run it.

Leveraging Valgrind functionalities, the tracer instruments the execution and makes two Dimemas traces: one describing the instrumented MPI execution; and the other describing the potential MPI/SMPSs execution. The tracer uses the following Valgrind functionalities: 1) intercepting the inserted *hooks* in order to track which task is currently being executed; 2) intercepting all memory allocations in order to maintain the pool of data objects in the memory; 3) intercepting memory accesses in order to identify data dependencies among tasks; and 4) intercepting all MPI calls in order to track MPI activity of the execution. Using the obtained information, the tracer generates the trace of the original (actually executed) MPI execution, while at the same time, it reconstructs what would be the trace of the potential (not executed) MPI/SMPSs execution.

The tool instruments accesses to all memory objects and derives data dependencies among tasks. By intercepting all dynamic allocations and releases of the memory (*allocs* and *frees*), the tool maintains the pool of all dynamic memory objects. Similarly, by intercepting all static allocations and releases of the



Note: The tracer describes the MPI traces by emitting two types of records: 1) computation record defining the length of computation burst; and 2) communication record specifying the parameters of MPI transfers. Conversely, it describes the MPI/SMPSs trace by breaking the original computation bursts into tasks and synchronizing the created tasks according to the identified data dependencies.

Fig. 5. Collecting trace of the original MPI and the potential MPI/SMPSs execution

memory (*mmaps* and *munmaps*), and reading the debugging information of the executable, the tool maintains the pool of all the static memory objects. The tracer tracks all memory objects, intercepting and recoding accesses to them at the granularity of one byte. Based on these records, and knowing in which task the execution is at every moment, the tracer detects all read-after-write dependencies and interpret them as dependencies among tasks.

The tool creates the trace of the executed MPI run, and at the same time, considering identified task dependencies, it creates what would be the trace of the potential MPI/SMPSs run (Figure 5). When generating the original trace, the tool describes the actually executed run by putting in the trace two types of records: 1) computation record stating the length of computation burst in terms of the number of instructions 2) communication record specifying the parameters of the executed MPI transfer. Additionally, when reconstructing the trace of the potential MPI/SMPSs run, the tracer breaks the original computation bursts into tasks, and then synchronizes the created tasks according to the identified data dependencies.

4.4 Replay Simulator

Dimemas is an open-source tracefile-based simulator for analysis of message-passing applications on a configurable parallel platform. The communication model, validated in [4], consists of a linear model and nonlinear effects, such as network congestion. The interconnect is parametrized by bandwidth, latency, and the number of global buses (denoting how many messages can concurrently travel throughout the network). Also, each processor is characterized by the number of input/output ports that determine its injection rate to the network. Finally, the simulated output of Dimemas can be visualized in Paraver.

We extended Dimemas to support synchronization of tasks in a way that allows Paraver to visualize all data dependencies. We implemented a task synchronization using an intra-node instantaneous MPI transfer that specifies the source and the destination tasks. This way, Paraver can visualize the simulated time-behavior showing both MPI communications among processes and data dependencies among tasks. Using this feature, the developer can visually detect each execution bottleneck and further inspect its causes.

5 Experiments

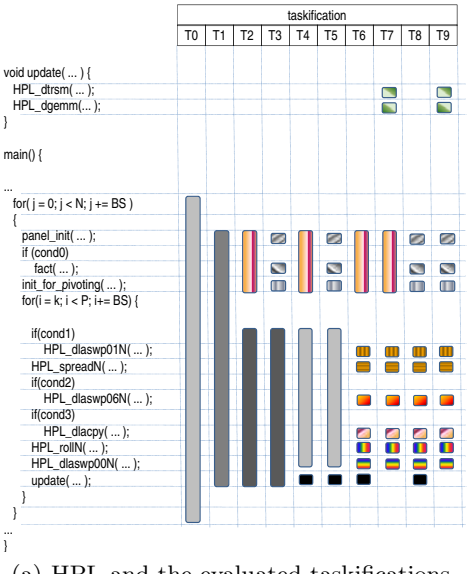
Our experiments explore MPI/SMPSs execution of HP Linpack on a cluster of many-core nodes. We used HPL with the problem size of 8192 and with 2x2 (PxQ) data decomposition. Also, we test various granularities of execution by running HPL with block sizes (BS) of 32, 64, 128, and 256. Our target machine consists of four many-core nodes, with one MPI process running on each node. We are primarily interested in the MPI/SMPSs potential parallelism inherent in the code, so we make most of the measurements for unlimited resources on the target machine – infinite number of cores per node and ideal interconnect

between the nodes. These results represent the upper bound of achievable parallelism. Finally, we show how this potential parallelism inherent in the application results in speedup when the application executes on a realistic target machine.

The major part of our experiment consist of exploring the potential MPI/SMPSs taskifications of HPL. In a case study with HPL, we present a top-to-bottom approach that uses a trial-and-error method, requires no knowledge of the studied code, and finally leads to exposing dataflow parallelism in the code. The approach uses the following method: 1) we propose a coarse-grained taskification for the code; 2) given the taskification, the environment estimates potential speedup and offers visualization of the resulting MPI/SMPSs execution. 3) based on the output, we choose a finer-grained taskification and return to step 2. We start from the most coarse-grain taskification ($T0$) that puts whole MPI process into one task and actually presents the traditional MPI execution (Figure 6(a)). Then using $T0$ as the baseline, we determine the *potential parallelism of* $Ti(1 \leq i \leq 9)$ *normalized to* $T0$ as the speedup of Ti over $T0$ when both these taskifications execute on a machine with unlimited number of cores per node and unlimited network performance (Figure 7(b)).

5.1 Results

First, the framework instruments the application to obtain the profile that guides the taskification process. Table 6(b) shows the accumulated time spent in each



(a) HPL and the evaluated taskifications.

		granularity			
		BS-32	BS- 64	BS-128	BS-256
task name	outer	panel_init	0.0003	0.0002	0.0001	0.0000
		fact	0.7525	1.2071	1.8795	3.2077
		init_for_pivoting	0.0246	0.0487	0.0925	0.1795
	inner	HPL_dlaswp01N	0.2583	0.2917	0.2906	0.2815
		HPL_spreadN	0.1599	0.0800	0.0378	0.0181
		HPL_dlaswp06N	0.1222	0.1359	0.1367	0.1274
		HPL_rollN	0.3267	0.1619	0.0762	0.0363
		HPL_dlacpy	0.0857	0.0932	0.0929	0.0912
		HPL_dlaswp00N	0.3706	0.4485	0.4736	0.4837
	update	HPL_dtrsm	0.8269	1.6674	2.8347	5.0772
		HPL_dgemm	97.0683	95.8614	94.0813	90.4935

(b) Distribution of total execution time spent in tasks (%).

		granularity			
		BS-32	BS- 64	BS-128	BS-256
task name	outer	panel_init	0.0003	0.0003	0.0003	0.0003
		fact	1.3468	3.7670	11.3572	37.8463
		init_for_pivoting	0.0221	0.0761	0.2797	1.0594
	inner	HPL_dlaswp01N	0.0073	0.0289	0.1130	0.4392
		HPL_spreadN	0.0022	0.0040	0.0073	0.0141
		HPL_dlaswp06N	0.0034	0.0135	0.0532	0.1987
		HPL_rollN	0.0046	0.0080	0.0148	0.0283
		HPL_dlacpy	0.0024	0.0092	0.0361	0.1423
		HPL_dlaswp00N	0.0052	0.0222	0.0921	0.3773
	update	HPL_dtrsm	0.0117	0.0826	0.5514	3.9605
		HPL_dgemm	1.3677	4.7492	18.3016	70.5900

(c) Average function duration (ms).

Note: In Tables 6(b) and 6(c), apart from statistic for each function of the code, we present the statistics for two logical sections: **outer** – consisting of *panel_init*, *fact* and *init_for_pivoting*; and **inner** consisting of *HPL_dlaswp01N*, *HPL_spreadN*, *HPL_dlaswp06N*, *HPL_rollN*, *HPL_dlacpy* and *HPL_dlaswp00N*.

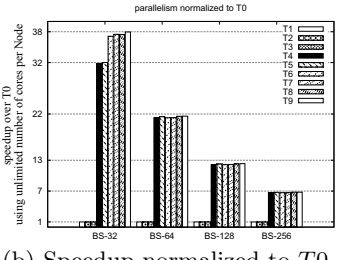
Fig. 6. Taskifications evaluated for HPL and duration and time spent in each function

function of the application. This information identifies instances of which functions need to execute concurrently in order to achieve significant parallelism. In this example, those are instances of functions *update*, because the application spends in that function from 95.57% (for $BS = 256$) to 97.83% (for $BS = 32$). On the other hand, Figure 6(c) shows the average duration of each function. This information identifies which function is a good candidate to be broken down into smaller tasks. In this example, function *panel_init* is very short so breaking it into smaller tasks makes little sense. Also, it is important to note that decreasing BS reduces execution time of most of the functions, so this could also be a way to make finer-grained execution.

Considering the data showed on previous tables, we start the process of exposing parallelism by: 1) proposing a taskification ($T1$ - $T9$ in Figure 6(a)); 2) testing how many tasks we created (Figure 7(a)); and 3) testing the potential speedup of the taskification (Figure 7(b)). $T0$ is the baseline taskification that makes only one task per MPI process. $T1$ puts each iteration of the outer loop in one task, but this strategy gives no additional parallelism compared to $T0$. Furthermore, $T2$ breaks down the code into section *outer* and separate iterations of the inner loop, still giving no improvement in speedup. $T3$ additionally breaks

		granularity			
		BS-32	BS-64	BS-128	BS-256
taskification	T1	1024	516	260	132
	T2	66.314	16.778	4.298	1.130
	T3	69.898	18.570	5.194	1.578
	T4	131.600	33.040	8.336	2.128
	T5	135.184	34.832	9.232	2.576
	T6	327.458	81.826	20.450	5.122
	T7	425.382	106.216	26.502	6.614
	T8	331.042	83.618	21.346	5.570
	T9	428.966	108.006	27.398	7.062

(a) Total number of tasks created.



(b) Speedup normalized to $T0$.

Note: In Figure 7(b), all taskifications ($T0$-$T9$) execute in MPI/SMPSs fashion on an ideal target machine. Then, the speedup of taskification Ti over taskification $T0$ represents the parallelism of taskification Ti normalized to taskification $T0$.

Fig. 7. Number of task instances and the potential parallelism of each taskification

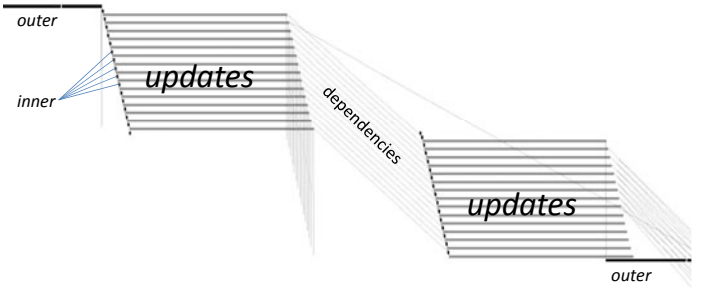
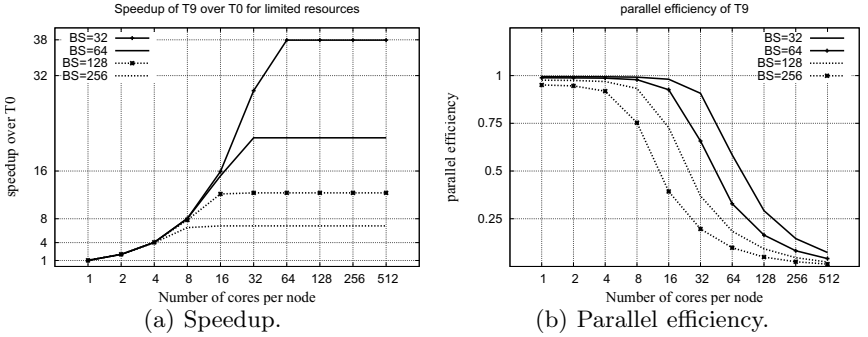


Fig. 8. Paraver visualization of the first 63 tasks and the dependencies among them (taskification T4, BS=256)

down section *outer*, but with no increases in speedup. This can be explained by Paraver visualization (results not presented in the paper) that shows that in $T2$ and $T3$, each iteration of the inner loop depends on the previous iteration, and thus impedes parallelism. Finally, $T4$ compared to $T2$ separates section *inner* from function *update* and releases the significant amount of parallelism. Namely, it achieves the speedup of 6.76, 12.28, 21.48 and 32.02 for block sizes of 256, 128, 64 and 32, respectively (Figure 7(b)). Also, $T4$ significantly increases the number of tasks in the application to 2.128, 8.336, 33.040 and 131.600 for block sizes of 256, 128, 64 and 32, respectively (Figure 7(a)). Now, Paraver visualization reveals that in $T4$: 1) each section *inner* depends on the section *inner* in the previous iteration of the inner loop; and 2) each *update* depends on the section *inner* in the same iteration of the inner loop. Thus, because section *inner* is much shorter than *update*, all dependent sections *inner* can execute quickly, and then independent instances of *update* can execute concurrently (Figure 8).

Further breaking down of *outer*, *inner* and *update* contributes little to the potential speedup (Figure 7(b)). Breaking of *outer*, for block sizes of 256, 128 and 64, causes slightly higher parallelism of $T5$, $T8$ and $T9$, compared to $T4$, $T6$ and $T7$. On the other hand, breaking of *inner*, for block size of 32, causes significantly higher parallelism of $T6$, $T7$, $T8$ and $T9$, compared to $T4$, $T5$. This effect happens because for very high concurrency of *update* (speedup is higher than 30), the critical path of the execution moves and starts passing through section *inner*. In these circumstances breaking of *inner* significantly increases parallelism by allowing concurrency of functions *HPL_dlaswp00N*, *HPL_dlaswp01N* and *HPL_dlaswp06N*. Finally, breaking of *update*, for block size 32, causes slightly higher parallelism of $T9$ compared to $T8$.

Figure 9 shows the speedup and parallel efficiency of $T9$ for different number of cores per node. The results show that high parallelism in the application is useful not to achieve high speedup on a small parallel machine, but rather to deploy efficiently a large parallel machine. Figure 9(a) shows that for a machine



(a) Speedup. (b) Parallel efficiency.

Note: Parallel efficiency denotes the ratio between the application's speedup achieved on some parallel machine and the number of cores of that parallel machine. Infact, the metric presents the overall average core utilization in the whole machine.

Fig. 9. Speedup and parallel efficiency for T9 for various number of cores

with 4 cores per node, $T9$ with all block sizes achieve a speedup of around 4, with difference between the highest and the lowest of less than 2%. However, for a machine with 32 cores per node, $T9$ with block sizes of 256, 128, 64 and 32, achieves the speedup of 6.80, 12.34, 21.57 and 29.47, respectively. Furthermore, Figure 9(b) shows parallel efficiency (core utilization) – the ratio between the application's speedup achieved on some parallel machine and the number of cores in that machine. Adopting that an application efficiently utilizes a machine if the parallel efficiency is higher than 75%, the results show that $T9$ with block sizes of 256, 128, 64 and 32, can efficiently utilize the machine of 8, 15, 26 and 47 cores per node, respectively. Therefore, to efficiently employ many-core machine with hundreds of cores per node, HPL has to expose even more parallelism, for instance, by making finer-grain taskification with further reduction of block size.

6 Related Work

Back in 1991 the community started claiming that instruction-level parallelism is dead [15], and consequently in the following 20 years appeared many programming models that exploit task-level parallelism. OpenMP [11] is the most popular programming model for shared memory that was founded with the idea of parallelizing loops, but from version 3.0 provides support for task parallelism. Cilk [2] implements a model of spawning various tasks and specifying a synchronization point where these tasks are waited for. MPI tasklets [5] parallelize SMP tasks by incorporating dynamic scheduling strategy into current MPI implementations. There are also proposals that originated from the industry, such as: TBB [12] from Intel and TPL [6] from Microsoft. Still, all these proposals suffer from the limitations of fork-join based programming models. On the other hand, SMPSs [10] is a programming model in which the programmer specifies dependencies among tasks, rather than specifying synchronization points. Then, based on the specified dependencies, the runtime schedules tasks in dataflow manner, potentially extracting very distant parallelism. Furthermore, SMPSs can be integrated with MPI, allowing better messaging behavior. Marjanovic *at. el.* [8] demonstrate that compared to MPI, MPI/SMPSs provides superior performance as well as higher tolerance to network reduction and external noise.

However, there is little development support for these programming models. Alchemist tool [16] identifies parts of code that are suitable for thread-level speculation. Embla [7] estimates the potential speed-up of fork-join based parallelization. Starsscheck [3] checks correctness of pragma annotations for STARSs family of programming models. Our work adds up to these efforts by designing a framework that estimates the potential parallelism of MPI/SMPSs. Furthermore, our work goes beyond the state-of-the-art tools because: 1) it deals with complex execution model that integrates MPI with task-based dataflow execution; 2) it allows to study MPI/SMPSs execution before the original MPI application is ported to MPI/SMPSs; 3) it provides an estimation of the parallelism on the configurable target platform; and 4) it provides visualization of the simulated execution.

7 Conclusion

Tasks-based parallel programming languages are promising in exploiting additional parallelism inherent in MPI parallel programs. However, the complexity of this type of execution impedes an MPI programmer from anticipating how much dataflow parallelism he can obtain in his application. Moreover, it is nontrivial to determine which parts of code should be encapsulated into tasks in order to expose the parallelism and still avoid creating unnecessary tasks that increase runtime overhead. To address this issue, we have developed a framework that automatically estimates the potential dataflow parallelization in applications. We show how, using the framework, one can find optimal taskification choice for any application through a trial-and-error iterative approach that requires no knowledge of the studied code. We prove the effectiveness of this approach on a case study in which we explore the taskification of High Performance Linpack (HPL). The results show that HPL expresses substantial amount of potential dataflow parallelism that allows the application to efficiently utilize cluster of nodes with up to 47 cores per node. Moreover, we show that the global partitioning significantly impacts parallel efficiency, and thus, in order to efficiently utilize higher number of cores, finer-granularity of execution should be used.

Acknowledgements. We thankfully acknowledge the support of the European Commission through the HiPEAC-2 Network of Excellence (FP7/ICT 217068), the TEXT project (IST-2007-261580), and the support of the Spanish Ministry of Education (TIN2007-60625, and CSD2007-00050), and the Generalitat de Catalunya (2009-SGR-980).

References

1. Top500 List: List of top 500 supercomputers, http://www.top500.org/
2. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System. J. Parallel Distrib. Comput. 37, 55–69 (1996)
3. Carpenter, P.M., Ramirez, A., Ayguade, E.: Starsscheck: A tool to find errors in task-based parallel programs. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6271, pp. 2–13. Springer, Heidelberg (2010)
4. Girona, S., Labarta, J., Badia, R.M.: Validation of dimemas communication model for mpi collective operations. In: PVM/MPI, pp. 39–46 (2000)
5. Kale, V., Gropp, W.: Load Balancing for Regular Meshes on SMPs with MPI. In: Keller, R., Gabriel, E., Resch, M., Dongarra, J. (eds.) EuroMPI 2010. LNCS, vol. 6305, pp. 229–238. Springer, Heidelberg (2010)
6. Leijen, D., Hall, J.: Parallel performance: Optimize managed code for multi-core machines. MSDN Magazine (2007)
7. Mak, J., Faxén, K.-F., Janson, S., Mycroft, A.: Estimating and Exploiting Potential Parallelism by Source-Level Dependence Profiling. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6271, pp. 26–37. Springer, Heidelberg (2010)

8. Marjanovic, V., Labarta, J., Ayguadé, E., Valero, M.: Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In: ICS, pp. 5–16 (2010)
9. Nethercote, N., Seward, J.: Valgrind, http://valgrind.org/
10. Pérez, J.M., Badia, R.M., Labarta, J.: A dependency-aware task-based programming environment for multi-core architectures. In: CLUSTER, pp. 142–151 (2008)
11. Proposed Industry Standard. Openmp: A proposed industry standard api for shared memory programming
12. Reinders, J.: Intel threading building blocks: outfitting C++ for multi-core processor parallelism. O'Reilly Media, Inc., Sebastopol (2007)
13. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI: The Complete Reference. The MIT Press, Cambridge (1998)
14. Subotic, V., Sancho, J.C., Labarta, J., Valero, M.: A Simulation Framework to Automatically Analyze the Communication-Computation Overlap in Scientific Applications. In: CLUSTER 2010 (2010)
15. Wall, D.W.: Limits of Instruction-Level Parallelism. In: ASPLOS (1991)
16. Zhang, X., Navabi, A., Jagannathan, S.: Alchemist: A transparent dependence distance profiling infrastructure. In: CGO 2009 (2009)

Event Log Mining Tool for Large Scale HPC Systems

Ana Gainaru[1,3], Franck Cappello[1,2], Stefan Trausan-Matu[3], and Bill Kramer[1]

[1] University of Illinois at Urbana-Champaign, IL USA
[2] INRIA, France
[3] University Politehnica of Bucharest, Romania

Abstract. Event log files are the most common source of information for the characterization of events in large scale systems. However the large size of these files makes the task of manual analysing log messages to be difficult and error prone. This is the reason why recent research has been focusing on creating algorithms for automatically analysing these log files. In this paper we present a novel methodology for extracting templates that describe event formats from large datasets presenting an intuitive and user-friendly output to system administrators. Our algorithm is able to keep up with the rapidly changing environments by adapting the clusters to the incoming stream of events. For testing our tool, we have chosen 5 log files that have different formats and that challenge different aspects in the clustering task. The experiments show that our tool outperforms all other algorithms in all tested scenarios achieving an average precision and recall of 0.9, increasing the correct number of groups by a factor of 1.5 and decreasing the number of false positives and negatives by an average factor of 4.

1 Introduction

Event logs are a rich source of information for analysing the cause of failures in cluster systems. However the size of these files has continued to increase with the ever growing size of supercomputers, making the task of analysing log files a hard and error prone process when handled manually. The current way used by system administrators for searching through the log data is pattern matching, by comparing numerical thresholds or doing regular expression matching on vast numbers of log entries looking for each pattern of interest. However by using this method, only those faults that are already previously known to the domain expert can be detected. As a consequence, data mining algorithms have recently been explored for extracting interesting information from log data without the control of a human supervisor [15, 21, 14]. However the algorithms must be adapted since it has been found that traditional clustering methods are not working well when they are applied to high dimensional data [11].

As mentioned in [2], log files will change during the course of a system's lifetime due to many reasons, from software upgrades to minor configuration changes and so it is normal to encounter novel events as time passes by. This makes it difficult for the algorithms to learn patterns or models. The learned patterns may not be applicable for a long time so all analyzing techniques must be able to detect phase shifts in behavior. Current data mining algorithms [17, 19, 10, 4, 15, 14] have difficulties in dynamically updating the groups to cope with an incoming stream of novel events.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 52–64, 2011.

In this paper we present HELO (Hierarchical Event Log Organizer) a novel unsupervised clustering engine that aims to accurately mine event type patterns from log files generated by large supercomputers. Our algorithm adapts data mining techniques to cope with the structure format of log files making the process computational efficient and accurate. Existing event pattern mining tools are not taking advantage of the characteristics that log files share or are unable to classify messages in an online manner. Our algorithm requires no prior knowledge or expectations as events are defined by their existence. This is critical when dealing with leading edge systems or with environments that change from system to system.

We have made experiments in order to compare our tool with two Apriori tools (Loghound [17], SLCT [19]), two other pattern extractors (IPLOM [10], MTE [4]), and an affinity propagation technique (StrAp [23]). HELO outperforms all other algorithms providing a better precision without an overhead in the computational cost. We will show that our tool increases the corrected classified messages by a factor of 1.5 and decreases the number of false positives and false negatives by an average factor of 4.

The rest of the paper is organized as follows: Section 2 provides related work and describes other mining algorithms that will be used as a comparison for the results obtained by HELO. In section 3 we present our classifier tool, highlighting its properties and characteristics. Section 4 presents the log files used for the experiment scenarios and section 5 shows several performance results being obtained in order to validate the proposed mining tool. Finally, in section 6 we provide conclusions and present future work.

2 Related Work

Indexing the information found in log files is an important task since analysing groups of related messages can find problems better than by looking at individual events [15, 21]. For example there are many anomalies that are indicated by incomplete message sequences. In general a change in the normal behaviour of the system is usually an indicator of a problem. Extracting templates and shaping this behaviour can greatly help systems in detecting or even predicting faults [16].

There is a considerable amount of papers that deal with message clustering: some use supervised learning and some unsupervised data mining techniques. All the supervised methods need a training phase that is quite expensive since it requires manually annotated events [7, 20]. Unsupervised techniques require only a few input parameters, the rest being done automatically by the algorithm. The most used unsupervised methods for extracting information from log files are the Apriori algorithm for frequent itemsets [17, 19, 18], Latent Semantic Indexing [9], event pattern [10, 4] and k-nearest neighbours [23]. Also, there are some studies that use the source code to extract error description format [6]. However there are systems that don't give access to the code that generates log files so these algorithms can not be used regardless of the system.

HELO differs from the other event pattern mining tools for several reasons. First, there are very few methods that classify events in an online matter and most of them have a major limitation: they are unable to dynamically update the clusters for novel events. StrAp [23] is able to create new groups for the outliers. However the tool was

design to cope with numerical data, having limitations in clustering log files. HELO was implemented so that it is able to adapt the initial templates in order to cope with any changes in the incoming stream of events. Also HELO uses an efficient splitting process that considers different priorities for different words according to their semantic meaning. All other tools partition the dataset only according to the syntactic form of message description. This a limitation when dealing with log files since symbols used in the message description indicate different types of components or registers.

In the next paragraphs we present 5 different algorithms that mine log files and cluster events based on the similarity between their descriptions. We discuss their methodology and limitations. For computing the accuracy of all algorithms the log files were manually labelled and classified after discussions with our faculty's tech support group.

Loghound [17] and SLCT [19] are Apriori-based tools designed for automatically discovering event cluster formats from log files by considering log messages as data points and then clustering them according to different density values. The size of log files generated by todays supercomputers has continue to grow, so since the set management part in the Apriori algorithm is costly for even a smaller number of patterns [8], analysing these logs is becoming a problem to these methods.

Iterative Partitioning Log Mining (IPLoM) [10] is an algorithm for mining clusters from event logs. The authors use 2 splitting steps by token count and token position and one step where the algorithm searches for bijections between tokens from different messages and splits the data accordingly. One limitation of this algorithm is the fact that all messages in one cluster must have the same length. Also another limitation in IPLoMs analysis is the syntactic depth of the mining process.

Streaming Affinity Propagation (StrAp) [23] is a clustering algorithm that extends Affinity Propagation to data streaming. In the first step, the tool finds the number of clusters that can be formed with the offline training set and then divides it by retaining the best items that represent each cluster. In the second step the rest of the input messages are treated as stream of data and the tool achieves online clustering by making new groups for the outliers and occasionally updating the exemplars from existing groups. The algorithm was implemented to cope with numerical input data, for example the duration of execution for each job. We implemented in StrAp, a Hamming distance metric for log messages [3] that is able to work with non-numeric values.

Message Template Extractor (MTE) is a component contained by the FDiag tool [4] that adds structure to the logs by extracting the messages template. The main idea used by the authors is that tokens in the English dictionary show the same patterns in different messages from the same type and that alpha-numerical values do not have the same property. The MTE extracts two template sets, one for constants and one for variables. The tool considers variables to be alpha-numerical tokens, i.e. words that contain letters, numbers in decimal or hex and symbols. However there are some cases when variables are also English normal words. For example, this is the case of filenames.

3 Methodology

Table 1 presents different event examples that illustrate the usual form of a log message. The first part of the description is defined by header information and the rest is

Table 1. Log message examples

Header	Message
[02:32:47][c1-0c1s5n0]	Added 8 subnets and 4 addresses to DB
[02:32:51][c3-0c0s2n2]	address parity check..0
[02:32:52][c3-0c0s2n2]	address parity check..1
[02:32:57][c1-0c1s5n0]	Added 10 subnets and 8 addresses to DB
[02:34:21][c2-0c1s4n1]	data TLB error interrupt

represented by the error message. In this study we only use the message description for classifying events for all the tools. However HELO can group messages after different criteria according to how much out of the header is included in the algorithm's input.

A message description can be seen as constructed by variables and constants. Constant are words that keep their value in a group template and are represented by strings like "address" or "subnets" from our examples. These words carry crucial information since they describe the message type. Message variables like 8 from our message identify manipulated objects or states for the program. HELO is a hierarchical process that finds representations for all message types that exist in a log file by extracting constants and variables from message descriptions. The tool uses in the splitting process the fact that words formed by letters and not numbers or punctuation marks, have more chances of being constants in the final templates.

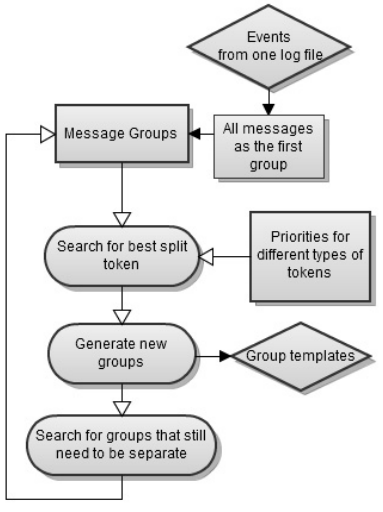
The methodology used by the tool has two different steps: an offline classification part where events found in log files are used to create the first template set by dividing them according to their description patterns and an online clustering part that classifies each new event and dynamically reshapes the previous found groups according to them.

3.1 Offline Clustering

The offline component of HELO deals with mining group patterns from log file messages. Basically the algorithm groups events considering their description in a 2 step hierarchical process. In the first step the algorithm searches for the best split column for each cluster and in the second step the clusters are divided correspondingly. A split column represents a word position in the message description that is used to divide the cluster into different groups. The methodology is described in Figure 1(a).

HELO starts with the whole unclustered log file as the first group and recursively partitions it until all groups have the cluster goodness over a specified threshold. The cluster goodness characterizes how similar all messages in one group are and is define as the percentage of common words in all events description over the average message length. This threshold can be provided by the user but is not mandatory for the execution of the tool. For all our experiments, we use the default value without trying to gain performance by tuning this parameter. This value has been set to 40% and was chosen after observing empirically that, for most of the log files, this value gives the best results.

The cluster goodness threshold is used to establish the generality of the final groups. If the threshold is low then there will be more words considered variables so the group generality increases. For example, with a lower threshold the tool generates a group



(a) Diagram

Algorithm 1 Find Split Position

Input: Partition M[] of messages
Output: Word position that can be used for the split

```
1:  for every message Maux in M do
2:      for every word_position in Maux do
3:          if Maux[word_position] is a hybrid word then
4:              Wrd = extract_hybrid(Maux[word_position])
5:          else if Maux[word_position] is a number then
6:              Wrd = number_value()
7:          else
8:              Wrd = Maux[word_position]
9:          end if
10:         if Wrd is unique for this word_position then
11:             appearance for this word = 1
12:         else
13:             increase appearance for this word with one
14:         end if
15:     end for
16: end for
17: get word_position where mean(words appearance) is
    max
18: return word_position
```

(b) Splitting process pseudocode

Fig. 1. Offline clustering methodology

with messages describing L2 cache errors, and with a higher threshold there will be two groups, one for L2 read errors and one for write.

We could not compare the real execution times obtained for all algorithms since they are implemented in different languages. We analysed only the theoretical complexity time between HELO and the other tool that obtained a good accuracy and precision for both online and offline test cases, StrAp. For the offline classification, StrAp measures the distance between any two messages; this means a time complexity of $O(N^2)$. HELO is an iterative process so uses different number of cycles for different log files. From all experiments, the clustering process is finished after 5 steps, but the worst case theoretical scenario takes $NlogN$ steps (if each message represents a unique template but their similarity is just under the threshold).

3.2 Splitting Process

In the first step of the process, the algorithm searches for the best split column for all clusters. Traditional data mining algorithms compute the information gain for each variable that could be used for the split and choose the one with the highest gain in accuracy. However, in our case, the best splitting position is the one that contains the maximum number of constant words. We consider that words with a high number of appearances on one position has more chances of being a constant, so HELO searches for the column where most unique words have a high appearance rate. This position corresponds to the column where the mean number of appearances for every unique word is maximum while still having enough words in order to be relevant to the analysed event dataset.

Table 2. Log template example

machine check interrupt (bit=0x1d): L2 dcache unit write read parity error
machine check interrupt (bit=0x10): L2 DCU read error
machine check interrupt (bit=d+): L2 * * * n+

For a better understanding we will consider the events presented in Table 1. From the fifth column the values are no longer relevant since the total number of messages that have words on that position is too small. The reason that this position is not considered is that the splitting process tries to obtain balanced groups in each step.

HELO considers that different type of words have different priorities dependent on their semantics. There are three types of considered words: English words, numeric values and hybrid tokens (words that are composed of letters, numbers and symbols of any kind). The lowest priority is for all-numeric values since the algorithm considers that these words have the most chances of becoming variables in the clusters. In the example from above, 8 and 10 from column 2 are decreasing the appearance mean for all unique words, so the split will not be done here. Hybrid values are represented by tokens like check..0 from our example. The algorithm extracts and considers only the English words incorporated in the hybrid token. For our example both check..0 and check..1 are considered as word check. If we reanalyze the example presented above, the first and the third columns could be chosen by the algorithm for the splitting position.

The creation of new groups has the exact semantic rules as the previous step. Since the numeric values have the least priority, all messages that contain any numeric value on the splitting position will be gathered in the same group. For our example, no matter which of the first three columns is chosen for the split, three groups will be generated, one with the first and the fourth message, one with the second and the third and one with the last message. For each created group, HELO computes the cluster goodness. If the value is under the chosen threshold, the group is sent to be divided again.

There are some cases where the partition step splits the groups by a column that is a good choice for the majority of the future groups but that divides some messages that should be together. After the splitting process is over and the final templates are created, HELO reanalyzes the clusters and merges group templates that are very similar. The default value for the threshold has been set to 80% since the group templates should be very similar in order to be considered one. In this last step templates are compared to one another so the time complexity is O(G^2). However, this is not a problem since the total number of templates is very small comparing to the whole dimension of the logs.

3.3 Output

When all clusters are stable, the algorithm identifies cluster description for each partition. A group template represents a line of text where variables are represented by different wildcards. HELO uses three types of wildcards: d+ represents numeric values, * represents any other single words, and n+ represents all columns of words that have a value for some of the messages and do not exist for others. In the example in Table 2 all three types of wildcards are illustrated.

The group templates describe all type of events that the system generates, in an intuitive way. The user-friendly group description generated by the tool could ease the work of system administrators to follow and understand errors from log files.

3.4 Online Clustering

The online clustering process deals with grouping messages as they are being generated by the system. Clustering tools must be able to change the group templates in order to manage novel messages that could appear. The input is given by the groups obtained with the offline process on the initial dataset. In HELO, each cluster needs to be represented by a description in the format described in the previous section and some statistics about the group.

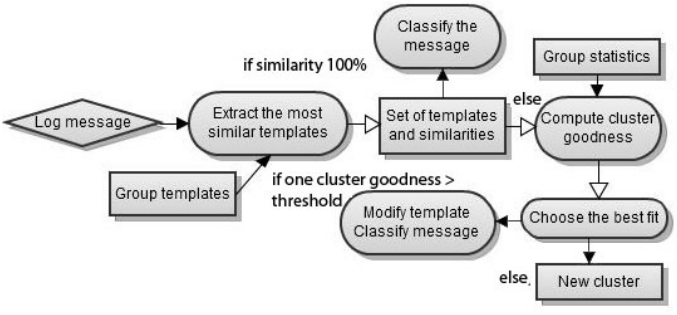


Fig. 2. Online clustering diagram

For each new message, the online component checks the description of the messages and retrieves the most appropriate group templates. If a message fits the exact description of a group (this means the group template does not need to be modified) then the search is over and we stamp the message with the template's group id. If the message does not have an exact match with any of the groups then we compute the cluster goodness for all the clusters retrieved before, after including the new message in each of them. For computing the goodness of the group if the message is inserted, we retrieve the average length of all messages in this cluster from the group statistics file. This information decides if including the new message decreases the cluster goodness under the threshold or not. If no cluster has the goodness over a specific threshold then a new group is formed. Else the group with the best cluster goodness will be chosen and the group template will be modified to accommodate the new message. The methodology for the online component of HELO is presented in figure 2.

4 Log Files

The logs we use for validating our tool are generated by five different supercomputers. Table 3 gives statistic information about the traces. All files except Mercury are downloaded from two websites: [5, 1]. Mercury logs are owned by the NCSA [13] and are not available to the public for privacy issues.

Table 3. Log data statistics

System	Messages	Time	Log type
BlueGene/L	4,747,963	6 months	event logs, login logs
Mercury	>10 million	3 months	event logs
PNNL	4,750	4 years	event logs
Cray XT4	3,170,514	3 months	event, syslog, console
LANL	433,490	9 years	cluster node outages

Table 4. Parameter values

Tool	Parameter	Value
IPLoM	File Support	0-0.5
Loghound	Support Th	0.01-0.1
SLCT	Support Th	0.01-0.1
StrAp	Offline support	$N/10^2$-$N/10^3$

We chose these 5 datasets because their diversity makes the analysis process more reliable: LANL [2] has a friendly format for all the tools under study; Cray XT4 has a very large amount of event patterns making the online classification less precise; Mercury has a very large amount of total messages, a few hundred thousand events per day, making it a good scalability testing scenario ; PNNL [22] has a large number of groups but most of them include a small amount of messages making it difficult for algorithms that take the frequencies of events into consideration to classify them; also BlueGene/L, Cray and Mercury put a lot of semantic problems.

We identified groups from each log file manually after discussions with people from the tech support group, and used this information to compute the performance of all the tools under study as an information retrieval task.

5 Results

We have made experiments in order to compare our tool with the other 5 algorithms using the traces presented in the previous section. The measures used are the classic evaluation units from information retrieval field: precision, recall and f-measure [12]. We define the main parameters used: A true positive is represented by a group template that is found by one tool and that it's also one of the annotated clusters; False negatives represent group templates that are not found by the tool when they should be and false positives are the templates that are found by the tool but are not part of the annotated clusters. Precision can be seen as a measure of exactness or fidelity and it represents the proportion of correct found templates to all the generated templates. Recall is computed as the proportion of true positives to all the messages from the manually annotated clusters and it represents a measure of completeness. F-measure is another information retrieval measuring unit that evenly weights precision and recall into a single value.

The output format is different for different tools. Even though Loghound, SLCT and IPLoM all compute the same type of groups with the one used by HELO but by only considering one type of wildcard, the rest of the tools have their own private cluster description. StrAp groups messages based on the computational distance between event description so the output is represented by an array of the same dimension as the log file with each line represented by a group id. In order to have a fair comparison between the output of all tools, we use the output format from IPLoM, but also compute the array of classification ids, like the one generated by Strap, for all other tools.

Each tool (except MTE) has different parameters that guide the output result. Parameters chosen to run each tool are the ones that give the best result and are shown in Table 4. HELO's parameters were left at their default values.

Due to the diversity of output formats we performed two types of offline comparisons. For the first sets of experiments, we computed precision and recall for the groups of messages found by all tools except StrAp. Since MTE eliminates all variables from the clusters description without placing some wildcard symbols in their place, we use for MTEs analysis a special manually computed group file that eliminates all variables from the group description. For the second sets of experiments we compute the percentage of corrected classified messages from the log file. These two scenarios shows how well the tools classify historic log files in an offline manner.

In the last set of experiments we determine the percentage of correctly classified events in an online manner. Experiments are done only for HELO and StrAp since those are the only tools that can cluster messages in a data streaming scenario. The analysis will show how well the algorithms adapt to the overall changes in the system.

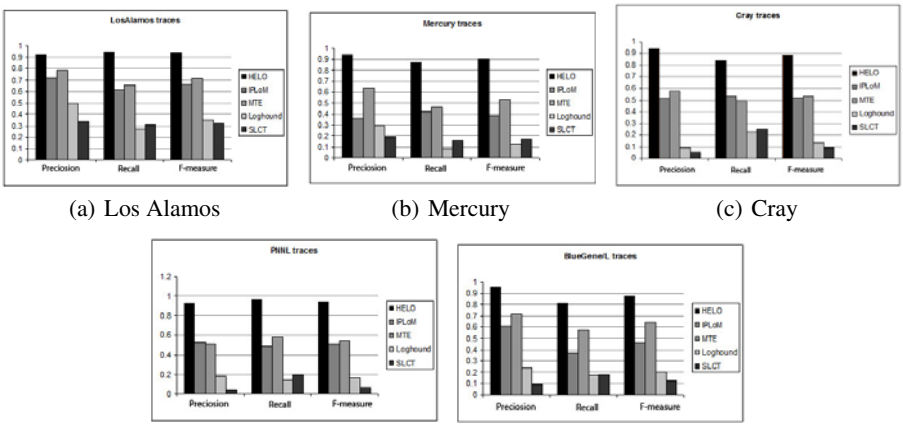
5.1 Offline

Figure 3 is showing the performance results obtained with all 5 tools for the input datasets. LosAlamos traces have the most user friendly format. Most of the generated messages are composed of English words, without having any messages represented by lists of registers. All tools obtained their best result for this dataset. Mercury and Cray generate many messages that represents the continuation of a previous message event (lists of memory locations or registers for example). This type of messages that have nothing to do with other event description, drastically decreases the performance of the mining tools. PNNL traces have the most consistency and syntactic problems for the event messages, so we can see the highest difference in performance (in both precision and recall) between our tool and the rest of the algorithm. One example of line descriptions that contain the same message but are written in different forms:
Corrective Measures SDE / DS2100 (upper) need to be replaced
Corrective Measures Upper DS2100 in need of Replacement

In general a decrease in the precision value indicates a high number of false positives. This usually means that the tool is generating more group templates than necessary, group templates that are not in the manual annotated file. One case, for example, is when the manual templates contain a lot of groups with messages of different length. For all log files, HELO keeps its precision value constant (HELO at around 0.92-0.93 with the next higher tool at around 0.67). The messages generated by all algorithms except HELO contain groups for each possible ending of the manual template. This results in an increase of the generated groups that are not manual templates.

A decrease in the recall value is influenced by a high number of false negatives. This means that there are many groups in the manual template file that are not generated by the tool. For all tools precision is mostly affected for PNNL and Mercury. The number of correct generated templates is decreasing for this log files basically because the systems produce more words with semantic problems than the rest.

The second sets of experiments computes the percentage of corrected classified messages from the log file. Figure 4 shows all results for all the analysed tools and for



(a) Los Alamos (b) Mercury (c) Cray

(d) PNNL (e) Blue Gene/L

Fig. 3. Comparing performance of HELO with the other 5 tools

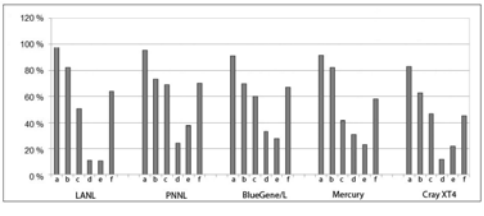


Fig. 4. Corrected classified messages a) HELO b) StrAp c) IPLoM d) Loghound e) SLCT f) MTE

all log files. The differences in results are due to the fact that log messages are not distributed uniformly over the group of messages. As expected LosAlamosl classifies messages with a higher confidence. On the other hand, the worst classification is obtained for Cray. The results for Cray can also be explained by the unfriendly form of messages generated by this supercomputer. The following message shows an example of an event description from which is hard to extract the relevant English word: $< ffffffff834c270 > : ptlrpc : lustre\_conn\_cnt + 80$

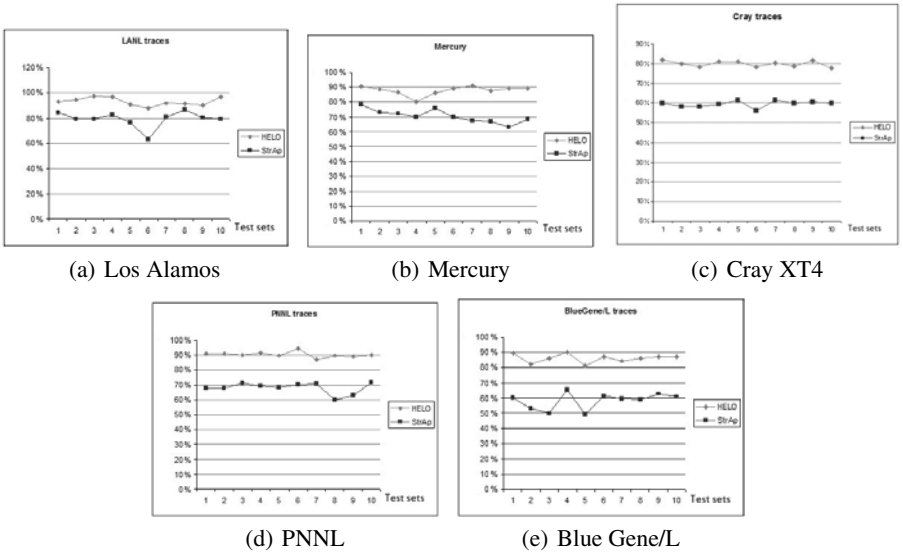
5.2 Online

In the last set of experiments, we compare the results of HELOs online component with the ones obtained by StrAp. For this set of experiments we use the 10-fold cross validation. We divide each log into 10 equal sets and then use one part for a training process of offline classification and the rest of 9 sets for online clustering using the groups found as input. We use the same method 10 times switching each time the set for the training phase. The same manually annotated files as for the offline process are used and we compute for both algorithms the number of corrected classified messages. Figure 5 shows results for both tools for all ten training sets and for each log files. In most of the cases, the two graphs follow the same curves. The different values for the

percentage of correct classified messages by one tool are given by the characteristics of events from each training set. If the training set has many new and different events from the ones found in the training set, it is likely that the value will decrease and if the training set contains all events that are in other sets than the tool will obtain the best classification, very close to the clustering obtained by the offline component.

In general, the performance follows the shape of the offline one. The shifts in the two graphs can be explained by the different methodology used by the tools. If the training set has a lot of semantic problems the distance between the two graphs will be higher. On the other hand, StrAp regroups the clusters when the number of messages that do not belong to existing clusters exceeds a threshold so StrAps performance will increase in the case of many clustered messages with different lengths.

The overall values are lower than for the offline components because usually the online classification algorithms focus on finding the best local solution for each message and not the overall best clustering result. However the results are still very good.



(a) Los Alamos (b) Mercury (c) Cray XT4

(d) PNNL (e) Blue Gene/L

Fig. 5. Comparing performance of the online component of HELO with StrAp

6 Conclusion and Future Work

In this paper we introduced HELO, a message pattern mining tool for log files generated by large scale supercomputers. We developed two components: 1) an offline clustering process that finds group templates with a high precision for events gathered in log files for long periods of time and 2) an online classification algorithm that groups new events adapting the templates to the changes in the underlying distribution. Current approaches like pattern matching are no longer efficient due to the shear mass of data to go into further analysis such as correlative analysis and forecasting.

We tested HELO against the performance of other tools used for the same task, that are using different methods. In our experiments, we used five different logs generated by different systems that have been prior manually annotated. Results clearly show that HELO outperforms other algorithms for all offline and online tasks having a precision and recall average of over 0.9 without having an overhead in the execution time.

The extracted group templates are used to describe events generated by the supercomputers and to future characterize the overall behaviour of fault and failures in the system. It is important to have a high precision for this mining step since in the future we intend to use the groups to analyse temporal and spatial characteristics as well as correlations between events. HELO not only has a high accuracy but also presents to system administrators the description of each group making it easier for a human interaction in the process of cluster reorganization before the analyser step.

References

[1] Archive, F.T., http://fta.inria.fr (accessed on 2010)
[2] Schroeder, G.G.B.: A large-scale study of failures in high-performance computing systems. In: IEEE DSN 2006, pp. 249–258 (June 2006)
[3] Bookstein, A., all: Generalized hamming distance. Information Retrieval Journal 5(4), 353–375 (2002)
[4] Chuah, E., et al.: Diagnosing the root-cause of failures from cluster log files (2010)
[5] T. computer failure data repository, http://cfdr.usenix.org (accessed on 2010)
[6] Fu, Q.: all. Execution anomaly detection in distributed systems through unstructured log analysis. In: ICDM, pp. 149–158 (December 2009)
[7] Fu, S., Xu, C.-Z.: Exploring event correlation for failure prediction in coalitions of clusters. In: Proceedings of the ACM/IEEE Conference on Supercomputing (November 2007)
[8] Han, J., et al.: Mining frequent patterns without candidate generation. In: ACM SIGMOD, pp. 1–12 (May 2000)
[9] Lan, Z., all: Toward automated anomaly identification in large-scale systems. IEEE Trans. on Parallel and Distributed Systems 21(2), 174–187 (2010)
[10] Makanju, A., et al: Clustering event logs using iterative partitioning. In: 15th ACM SIGKDD, pp. 1255–1264 (2009)
[11] McCallum, A., all: Efficient clustering of high-dimensional data sets with application to reference matching. In: ACM SIGKDD, pp. 169–178 (August 2000)
[12] Mitra, M., Chaudhuri, B.: Information retrieval from documents: A survey. Information Retrieval Journal 2(2-3), 141–163 (2000)
[13] NCSA, http://www.ncsa.illinois.edu (accessed on 2010)
[14] Pang, W., et al.: Mining logs files for data-driven system management. ACM SIGKDD 7, 44–51 (2005)
[15] Park, Geist, A.: System log pre-processing to improve failure prediction. In: DSN 2009, pp. 572–577 (2009)
[16] Salfner, F., et al.: A survey of online failure prediction methods. ACM Computing Surveys 42(3) (March 2010)
[17] Stearley, J.: Towards informatic analysis of syslogs. In: IEEE Conference on Cluster Computing (September 2004)
[18] Stearley, J.: Towards informatic analysis of syslogs. In: IEEE International Conference on Cluster Computing, vol. 5, pp. 309–318 (2004)
[19] Vaarandi, R.: Mining event logs with slct and loghound. In: IEEE NOMS 2008, pp. 1071–1074 (April 2008)

[20] Wei Peng, S.M., Li, T.: Mining logs files for data driven system management. ACM SIGKDD 7, 44–51 (2005)
[21] Xue, Z., et al.: A survey on failure prediction of large-scale server clusters. In: ACIS SNPD 2007, pp. 733–738 (June 2007)
[22] Zarza, G., et al.: Fault-tolerant routing for multiple permanent and non-permanent faults in hpc systems. In: PDPTA 2010 (July 2010)
[23] Zhang, X., Furtlehner, C., Sebag, M.: Data streaming with affinity propagation. In: Daelemans, W., Goethals, B., Morik, K. (eds.) ECML PKDD 2008, Part II. LNCS (LNAI), vol. 5212, pp. 628–643. Springer, Heidelberg (2008)

Reducing the Overhead of Direct Application Instrumentation Using Prior Static Analysis*

Jan Mußler[1], Daniel Lorenz[1], and Felix Wolf[1,2,3]

[1] Jülich Supercomputing Centre, 52425 Jülich, Germany
[2] German Research School for Simulation Sciences, 52062 Aachen, Germany
[3] RWTH Aachen University, 52056 Aachen, Germany

Abstract. Preparing performance measurements of HPC applications is usually a tradeoff between accuracy and granularity of the measured data. When using direct instrumentation, that is, the insertion of extra code around performance-relevant functions, the measurement overhead increases with the rate at which these functions are visited. If applied indiscriminately, the measurement dilation can even be prohibitive. In this paper, we show how static code analysis in combination with binary rewriting can help eliminate unnecessary instrumentation points based on configurable filter rules. In contrast to earlier approaches, our technique does not rely on dynamic information, making extra runs prior to the actual measurement dispensable. Moreover, the rules can be applied and modified without re-compilation. We evaluate filter rules designed for the analysis of computation and communication performance and show that in most cases the measurement dilation can be reduced to a few percent while still retaining significant detail.

1 Introduction

The complexity of high-performance computing applications is rising to new levels. In the wake of this trend, not only the extent of their code base but also their demand for computing power is rapidly expanding. System manufacturers are creating more powerful systems to deliver the necessary compute performance. Software tools are being developed to assist application scientists in harnessing these resources efficiently and to cope with program complexity. To optimize an application for a given architecture, different performance-analysis tools are available, utilizing a wide range of performance-measurement methodologies [17,13,21,18,7]. Many performance tools used in practice today rely on *direct instrumentation* to record relevant events, from which performance-data structures such as profiles or traces are generated. In contrast to statistical sampling, direct instrumentation installs calls to measurement routines, so-called hooks, at function entry and exit points or around call sites. This can be done on multiple levels ranging from the source code to the binary file or even the

* This material is based upon work supported by the US Department of Energy under Award Number DE-SC0001621.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 65–76, 2011.

memory image [20]. Often the compiler can inject these hooks automatically using a profiling interface specifically designed for this purpose.

Of course, instrumentation causes measurement intrusion – not only dilating the overall runtime and prolonging resource usage but also obscuring measurement results – especially, if the measurement overhead is substantial. If applied indiscriminately, the measurement dilation can render the results even useless. This happens in particular in the presence of short but frequently-called functions prevalent in C++ codes. In general, the measurement overhead increases with the rate at which instrumentation points are visited. However, depending on the analysis objective, not all functions are of equal interest and some may even be excluded from measurement without loosing relevant detail. For example, since the analysis of message volumes primarily focuses on MPI routines and their callers, purely local computations may be dispensable. Unfortunately, manually identifying and instrumenting only relevant functions is no satisfactory option for large programs. Although some automatic instrumentation tools [6,21] offer the option of explicitly excluding or including certain functions to narrow the measurement focus, the specification of black and white lists usually comes at the expense of extra measurements to determine suitable candidates.

To facilitate low-overhead measurements of relevant functions without the need for additional measurement runs, we employ static analysis to automatically identify suitable instrumentation candidates based on structural properties of the program. The identification process, which is accomplished via binary inspection using the the Dyninst library [3], follows filter rules that can be configured by refining and combining several base criteria suited for complementary analysis objectives. The resulting instrumentation specification is then immediately applied to the executable via binary re-writing [22], eliminating the need for re-compilation. Our methodology is available in the form of a flexible standalone instrumentation tool that can be configured to meet the needs of various applications and performance analyzers. Our approach significantly reduces the time-consuming work of filter creation and improves the measurement accuracy by lowering intrusion to a minimum. An evaluation of different filter criteria shows that in most cases the overhead can be reduced to only a few percent.

Our paper is structured as follows: After reviewing related work in Section 2, we present the design of the configurable instrumentation tool in Section 3. Then, in Section 4, we discuss the base filter criteria and the heuristics involved in their implementation. A comprehensive experimental evaluation of these criteria in terms the number of instrumented functions and the resulting measurement dilation is given in Section 5. Finally, we draw conclusions and outline future work in Section 6.

2 Related Work

To generally avoid the overhead of direct instrumentation, some tools such as HPCToolkit [13] resort to sampling. Although researchers recently also started combining sampling with direct instrumentation [19], the choice between the two

options is usually a trade-off between the desired expressiveness of the performance data and unwanted measurement dilation. Whereas sampling allows the latter to be controlled with ease, just by adjusting the sampling frequency, it delivers only an incomplete picture, potentially missing critical events or providing inaccurate estimates. Moreover, accessing details of the program state during the timer interrupt, such as arguments of the currently executed function, is technically challenging. Both disadvantages together make direct instrumentation the favorite method for capturing certain communication metrics such as the size of message payloads. This insight is also reflected in the current design of the MPI profiling interface [14], whose interposition-wrapper concept leverages direct instrumentation. However, to avoid excessive runtime overhead, the number of direct instrumentation points need to be selected with care, a task for which our approach now offers a convenient solution. If only the frequency of call-path visits is of interest, also optimizations such as those used by Ball and Larus for path profiling can be chosen [2].

Among the tools that rely on direct instrumentation, the provision of black lists to exclude functions from instrumentation (or white lists to include only a specific subset) is the standard practice of overhead minimization. In Scalasca [7] and TAU [21], the specification of such lists is supported through utilities that examine performance data from previous runs taken under full instrumentation. Selection criteria include the ratio between a function's execution time and its number of invocations or whether the function calls MPI – directly or indirectly. Yet, in malign cases where the overhead of full instrumentation is excessive, the required extra run may be hard or even impossible to complete in the first place. The selection lists are applied either statically or dynamically. The latter is the preferred method in combination with compiler instrumentation, which can be configured only at the granularity of entire files. In addition to user-supplied filter lists, TAU provides a runtime mechanism called *throttling* to automatically disable the instrumentation of short but frequently executed functions. A general disadvantage of runtime selection, whether via filter lists or automatically, however, is the residual overhead caused by the dynamic inspection of function identifiers upon each function call. Our solution, in contrast, neither requires any extra runs nor performs any dynamic checks.

Another generic instrumenter was designed by Geimer et al. [6]. Like ours, it can be configured to support arbitrary measurement APIs. Whereas we analyze and modify the binary, their instrumenter identifies potential instrumentation points in the source code. While allowing the restriction of target locations according to file and function names, the lack of static source-code analysis functionality prevents it from providing suggestions as to which functions should be instrumented. Moreover, changing the instrumentation always entails an expensive re-compilation.

An early automatic filter mechanism was developed as an extension of the OpenUH compiler's profiling interface [8]. Here, the compiler scores functions according to their estimated number of executions and their estimated duration, which are derived from the location of their call sites and their number

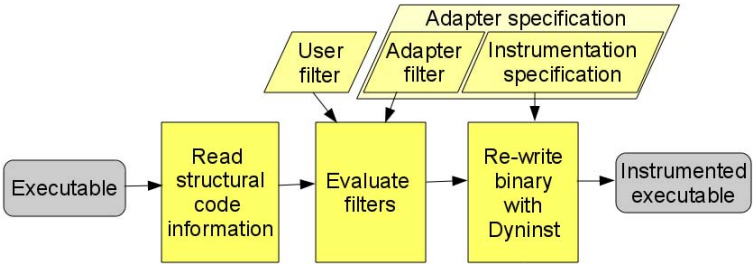


Fig. 1. The basic instrumentation workflow

of instructions, respectively. Based on this assessment, the compiler skips the instrumentation of those functions that are either short or called within nested loops. However, generally not instrumenting small functions was criticized by Adhianto et al. [1]. They argue that small functions often play a significant role, for example, if they include synchronization calls important to parallel performance. In our approach, providing rules that explicitly define exceptions, for example, by forcing the instrumenter to include all functions that call a certain synchronization primitive, can avoid or mitigate the risk of missing critical events.

If measurement overhead cannot be avoided without sacrificing analysis objectives, overhead compensation offers an instrument to retroactively improve the accuracy of the measured data. Initially developed for serial applications [10], it was later extended to account for overhead propagation in parallel applications [11]. The approach is based on the idea that every call to the measurement system incurs a roughly constant and measurable overhead with a deterministic and reversible effect on the performance data. However, variations in memory or cache utilization may invalidate this assumption to some degree.

3 A Configurable Instrumenter

Figure 1 illustrates the different steps involved in instrumenting an application and highlights the functional components of our instrumenter. As input serves a potentially optimized application executable, which is transformed into a ready-to-run instrumented executable, following the instructions embodied by user filters and adapter specifications. The instrumentation process starts with the extraction of structural information from the binary program, a feature supported by the Dyninst API. Although the inclusion of debug information into the executable during compilation is not mandatory, it tends to enrich the available structural information and can help formulate more sophisticated filter rules. As a next step, the instrumenter parses the provided filter specifications and determines the functions to be instrumented. Optionally, the instrumenter can print the names of instrumentable functions to simplify the creation of filter lists. The instrumentation itself is applied using Dyninst's binary rewriting capabilities.

The raw instrumenter can be configured in two different ways: First, developers of performance tools can provide an adapter specification (top right in Figure 1) to customize the instrumenter to their tool's needs. This customization includes the specification of code snippets such as calls to a proprietary measurement API to be inserted at instrumentation points. In addition, the adapter may include a predefined filter that reflects the tool's specific focus. Second, application developers can augment this predefined filter by specifying a user filter (top center in Figure 1) to satisfy application- or analysis-specific requirements. Below, we explain these two configuration options in detail.

3.1 Adapter Specification

The adapter specification is intended to be shipped together with a performance tool. It consists of a single XML document, which is both human readable (and editable) and at the same time accessible to automatic processing through our instrumenter. The format provides four different element types:

- The description of additional dependencies, for example, to measurement libraries that must be linked to the binary.
- The definition of optional adapter filter rules. These adhere to the same syntax as the user filter rules, which are introduced further below. The filter rules allow the exclusion of certain functions such as those belonging to the measurement library itself or those known to result in erroneous behavior. For example, when using Scalasca, which requires the measurement library to be statically linked to the application, the adapter filter would prevent the library itself from being instrumented.
- The definition of global variables.
- The definition of a set of code snippets to be inserted at instrumentation points.

The instrumenter supports instrumentation on three different levels: (i) function, (ii) loop, and (iii) call site. There are up to four instrumentation points associated with each level, plus two for initialization and finalization:

- *before*: Immediately before a call site (i.e., function call) or loop.
- *enter*: At function entry or at the start of a loop iteration.
- *exit*: At function exit or at the end of a loop iteration.
- *after*: Immediately after a call site (i.e., function call) or loop.
- *initialize*: Initialization code which is executed once for each instrumented function, call site, or loop.
- *finalize*: Finalization code which is executed once for each instrumented function, call site and loop.

The initialization and finalization code is needed, for example, to register a function with the measurement library or to release any associated resources once they are no longer in service.

To access an instrumentation point's context from within the inserted code, such as the name of the enclosing function or the name of the function's source

```
<adapter>
  <functions>
    <variables><var name="i" size="4" /></variables>
    <init>i = 0;</init>
    <enter>i = i + 1;
           printf("entering %s for the %d time\n",@ROUTINE@,i);
    </enter>
  </functions>
</adapter>
```

Fig. 2. Example adapter specification that counts the number of visits to an instrumented function and during each invocation prints a message which contains the function name and the accumulated number of visits

file, the instrumenter features a set of variables in analogy to [6]. These variables are enclosed by `@` and include, among others, the following items: `ROUTINE`, `FILE`, and `LINE`. To concatenate strings, we further added the `.` operator. At instrumentation time, a single `const char*` string will be generated from the combined string. In addition to specifying default code snippets to be inserted at the six locations listed above, an adapter specification may define uniquely named alternatives, which can be referenced in filter rules to tailor the instrumentation to the needs of specific groups of functions. An example adapter specification is shown in Figure 2. So far, we created adapter specifications for Scalasca, TAU and the Score-P measurement API [15]. The latter is a new measurement infrastructure intended to replace the proprietary infrastructures of several production performance tools.

3.2 Filter Specification

While the adapter customizes the instrumenter for a specific tool, the user filter allows the instrumenter to be configured for a specific application and/or analysis objective such as communication or computation. It does so by restricting the set of active instrumentation points of the target application.

The filter is composed of *include* and *exclude* elements, which are evaluated in the specified order. The *exclude* elements remove functions from the set, whereas *include* elements add functions to the set. A filter element consist of a particular set of properties a function must satisfy. The properties can be combined through the use of the logical operators *and*, *or*, and *not*. The properties are instances of base filter criteria, which are described in Section 4. For each instrumentable function, every rule is evaluated to decide whether the function matches the rule or not. An example for a filter definition is given in Figure 3. In addition to defining whether a function is instrumented or not, the user can also change the default code to be inserted by selecting alternative code snippets from the adapter specification, referencing them by the unique name that has been assigned there. Separate snippets can be chosen depending on whether the instrumentation occurs around functions, call sites, or loops.

```
<filter name="mpicallpath" instrument="functions=function" start="none">
<include>
   <property name="path">
       <functionnames match="prefix">MPI mpi</functionnames>
   </property>
</include>
</filter>
```

Fig. 3. Example for a filter definition that instruments all functions that appear on a call path leading to an MPI function

4 Filter Criteria

The purpose of our filter mechanism is to exclude functions that either lie outside our analysis objectives or whose excessive overhead would obscure measurement results. To avoid instrumenting any undesired functions, the instrumenter supports selection criteria (i) based on string matching and/or (ii) based on the program structure. String matching criteria demand that a string (e.g., the function name) has a certain prefix or suffix, contains a certain substring, or matches another string completely. String matching can be applied to function names, class names, namespaces, or file names. It is a convenient method, for example, to prevent the instrumentation of certain library routines that all start with the same prefix. In contrast, structural criteria take structural properties of a given function into account.

The first group of structural properties considers a function's position in the call tree, that is, its external relationships to other functions. This is useful to identify functions that belong to the context of functions in the center of our interest. For example, if the focus of the analysis are MPI messaging statistics, the user might want to know from where messaging routines are called but at the same time may afford to skip purely local computations in the interest of improved measurement accuracy.

Call path: Checks whether a function may appear on the call path to a specified set of functions, for example, whether the function issues MPI calls – either directly or indirectly. Unfortunately, since the decision is based on prior static analysis of the code, virtual functions or function pointers are ignored.

Depth: Checks whether a function can be called within a certain call-chain depth from a given set of functions. Relying on static call-graph analysis, as well, this property suffers from the same restrictions as the call-path property.

The second group of structural properties considers indicators of a function's internal complexity. This is motivated by short but frequently called functions that often contribute little to the overall execution time but cause over-proportional overhead.

Lines of code: Checks whether the number of source lines of a function falls within a given range. Using available debug information, the instrumenter

computes the number of source lines between the first entry point and the last exit point of a function. Note that the number of source lines may depend on the length of comments or the coding style. Moreover, inlining of macros or compiler optimizations may enlarge the binary function compared to its source representation.

Cyclomatic complexity: Checks whether the cyclomatic complexity [12] of a function falls within a given range. The cyclomatic complexity is the number of linearly independent paths through the function. We chose the variant that takes also the number of exit points into account. It is defined as $E - N + P$, with N representing the number of nodes in the control flow graph (i.e., the number of basic blocks), E the number of edges between these blocks, and P the number of connected components in the graph, which is 1 for a function. Again, inlining and compiler optimizations may increase the cyclomatic complexity in comparison to what a programmer would expect.

Number of instructions: Checks whether the number of instructions falls within a given range. Since the number of instructions is highly architecture and compiler dependent, it is challenging to formulate reasonable expectations.

Number of call sites: Checks whether the function contains at least a given number of function calls. Note that the mere occurrence of a call site does neither imply that the function is actually called nor does it tell how often it is called.

Has loops: Checks whether a function contains loops. Here, similar restrictions apply as with the number of call sites.

Of course, it is also possible to combine these criteria, for example, to instrument functions that either exceed a certain cyclomatic complexity threshold or appear on a call path to an MPI function.

5 Evaluation

In this section we evaluate the effectiveness of selected filter rules in terms of the overhead reduction achieved and the loss of information suffered (i.e., the number of functions not instrumented). The latter, however, has to be interpreted with care, as not all functions may equally contribute to the analysis goals. Since all of our filter criteria are parameterized, the space of filter rules that could be evaluated is infinitely large. Due to space and time constraints, we therefore concentrated on those instances that according to our experiences are the most useful ones. They are are listed below. Criteria not considered here will be the subject of future studies.

- MPI Path: Instrument only functions that appear on a call path to an MPI function. This filter allows the costs of MPI communication to be broken down by call path, often a prerequisite for effective communication tuning.
- CC 2+: Instrument all functions that have at least a cyclomatic complexity of two. This filter removes all functions that have only one possible path of execution.

Table 1. Number of functions instrumented under full instrumentation and percentage of functions instrumented after applying different filter rules

Application	Language	Full	CC 2+	CC 3+	LoC 5+	MPI Path
104.milc	C	261	51%	38%	71%	43%
107.leslie3d	Fortran	32	78%	66%	75%	28%
113.GemsFDTD	Fortran	197	62%	58%	81%	12%
115.fds4	C/Fortran	238	80%	74%	88%	0.4%
121.pop2	C/Fortran	982	65%	53%	77%	16%
122.tachyon	C	342	35%	27%	61%	5%
125.RAxML	C	313	77%	65%	85%	25%
126.lammps	C++	1378	56%	43%	64%	39%
128.GAPgeomfem	C/Fortran	36	61%	53%	72%	31%
130.socorro	C/Fortran	1331	50%	41%	74%	24%
132.zeusmp2	C/Fortran	155	84%	80%	89%	46%
137.lu	Fortran	35	66%	60%	77%	43%
Cactus Carpet	C++	3539	35%	29%	50%	6%
Gadget	C	402	62%	52%	26%	21%

- CC 3+: Instruments all functions that have a cyclomatic complexity of three or higher. Compared to the CC 2+ filter, functions need an additional loop or branch not to be removed.
- LoC 5+: Instrument all functions with five or more lines of code. This filter is expected to remove wrapper functions as well as getters, setters, or other one liners.

As test cases, we selected the SPEC MPI2007 benchmark suite [16], a collection of twelve MPI applications from various fields with very different characteristics; Gadget [5], a simulation that computes the collision of two star clusters; and Cactus Carpet [4], an adaptive mesh refinement and multi-patch driver for the Cactus framework. A full list of all applications can be found in Table 1 together with information on their programming language. We built all applications using the GNU compiler with optimization level `O2` enabled. All measurements were performed on Juropa [9], an Intel cluster at the Jülich Supercomputing Centre. Our instrumenter was linked to version 6.1 of Dyninst. To improve interoperability with the GNU compiler, we added GCC exception handling functions to the list of non-returning functions in Dyninst. We ran our test cases using the Scalasca measurement library in profiling mode, which instruments all MPI function by default through interposition wrappers.

Table 1 lists the number of instrumented functions when applying different filter rules including full instrumentation. The numbers do not include MPI functions, which are always instrumented. Also, the instrumenter was configured not to instrument the Scalasca measurement system itself. Otherwise, all functions Dyninst identified in the binary, which do not include dynamically linked libraries, were potential instrumentation candidates. The number of functions varies greatly among the fourteen applications, with *107.leslie3d* having only 32 compared to the two C++ codes with 1378 and 3539 functions, respectively. Judging by the

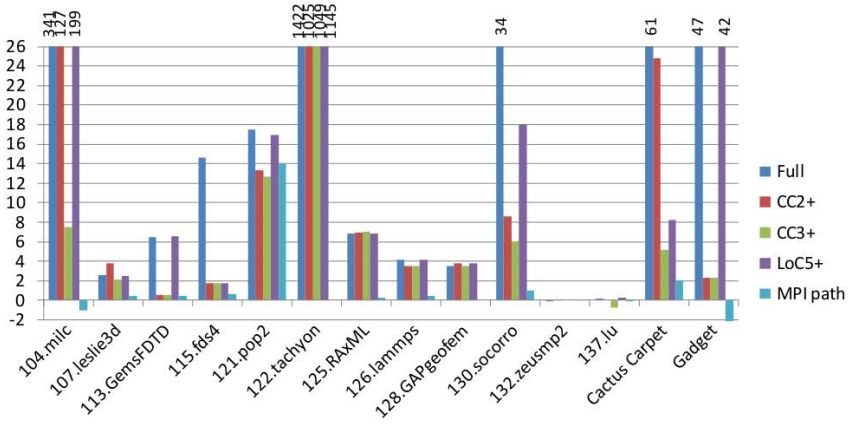


Fig. 4. Runtime overhead of the fully instrumented binary and after applying different filters. The values are given in percent compared to an uninstrumented run. Values exceeding 26% are clipped. Missing bars imply zero overhead. In general, measurement inaccuracies prevented a precise representation of values around zero, sometimes resulting in negative overhead figures.

fraction of eliminated functions, the MPI Path filter seems to be the most aggressive one, in one instance (*115.fds4*) leaving only the main function instrumented. The LoC 5+ filter, by contrast, leads only to relatively mild eliminations, with most codes loosing less then 40%. Finally, the difference between the two cyclomatic filters, which together occupy a solid middle ground in terms of their aggressiveness, is significant but not too pronounced.

The measurement overheads observed for the individual combinations of applications and filters are presented in Figure 4. Seven of the fourteen applications show less than 8% overhead even under full instrumentation, indicating that full instrumentation is not generally impracticable. Among the remaining applications, three including the C++ code Cactus Carpet exhibit extreme overheads above 50% without filters. The worst case is clearly *122.tachyon* with more than 1,000% overhead, which, however, contains functions with only two binary instructions. In almost all cases, with the exception of *121.pop2*, at least one filter exists that was able to reduce the overhead to 2% or less – within the limits of our measurement accuracy. For *121.pop2*, we achieved only a moderate reduction, although with 13% the lowest overhead of *121.pop2* was not alarming. As a general trend, the MPI Path filter resulted in the lowest overhead. Again, the only exception is 121.pop2, where many functions contain error handling code that may call MPI functions such as MPI_Finalize and MPI_Barrier. Thus, many functions were instrumented that actually do not call MPI during a normal run. Whereas the LoC 5+ filter did often enough fail to yield the desired overhead decrease, CC 3+ can be seen as a good compromise, often with higher although still acceptable overhead below 10% – but on the other hand with less functions removed from instrumentation and, thus, with less loss of information.

Finally, the ratio between the fraction of filtered functions and the overhead reduction can serve as an indicator of how effective a filter is in selecting functions

that introduce large overhead. Ignoring the codes with initial overheads below 5%, for which this measure might turn out to be unreliable, LoC 5+ shows varying behavior: In the cases of *104.milc*, *115.fds4*, *130.socorro*, and *Cactus Carpet* very few functions are removed compared to the achieved overhead reduction. For the other applications, the filter is largely ineffective. The cyclomatic filters, by contrast, yield high returns on their removal candidates in the majority of cases. Exceptions are *121.pop2*, *122.tachyon*, and *125.RAxML*. Finally, the effectiveness of MPI Path roughly correlates with the number of functions still instrumented. However, although it removes many functions, it still retains critical information. For example, the detection of MPI call paths that incur waiting time is not affected because all functions on such paths remain instrumented.

6 Conclusion and Future Work

In this paper, we presented an effective approach to reducing the overhead of direct instrumentation for the purpose of parallel performance analysis. Based on structural properties of the program, including both a function's internal structure and/or its external calling relationships, we are able to identify the most significant sources of overhead and remove them from instrumentation. Our solution, which was implemented as a generic and configurable binary instrumenter, requires neither any expensive extra runs nor re-compilation of the target code. We demonstrated that, depending on the analysis objective, in almost all of our test cases the overhead could be reduced to only a few percent. Overall, the MPI Path filter was most effective, allowing low-overhead measurements of the communication behavior across a wide range of applications – except for one malign case with MPI calls in rarely executed error handlers. Moreover, if the focus lies on computation, CC 3+ offers a good balance between the number of excluded functions and the overhead reduction achieved. Finally, the union of MPI Path and CC 3+ seems also promising and should be tried if investigating correlations between the computational load and the communication time is an analysis goal. Whereas this study only considered parallelism via MPI, future work will concentrate on filter rules also suitable for OpenMP applications. A particular challenge to be addressed will be the non-portable representation of OpenMP constructs in the binary.

Acknowledgment. We would like to thank the developer team of the Dyninst library, especially Madhavi Krishnan and Andrew Bernat, for their continuous support.

References

1. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: HPCToolkit: Tools for performance analysis of optimized parallel programs. Concurrency and Computation: Practice and Experience 22(6), 685–701 (2009)
2. Ball, T., Larus, J.R.: Efficient path profiling. In: Proc. of the 29th ACM/IEEE International Symposium on Microarchitecture, pp. 46–57. IEEE Computer Society, Washington, DC, USA (1996)

3. Buck, B., Hollingsworth, J.: An API for runtime code patching. Journal of High Performance Computing Applications 14(4), 317–329 (2000)
4. Cactus code (2010), http://www.cactuscode.org
5. Gadget 2 (2010), http://www.mpa-garching.mpg.de/gadget
6. Geimer, M., Shende, S.S., Malony, A.D., Wolf, F.: A generic and configurable source-code instrumentation component. In: Allen, G., Nabrzyski, J., Seidel, E., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2009. LNCS, vol. 5545, pp. 696–705. Springer, Heidelberg (2009)
7. Geimer, M., Wolf, F., Wylie, B., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience 22(6), 702–719 (2010)
8. Hernandez, O., Jin, H., Chapman, B.: Compiler support for efficient instrumentation. In: Proc. of the ParCo 2007 Conference. Advances in Parallel Computing, vol. 15, pp. 661–668 (2008)
9. JuRoPA (2010), http://www.fz-juelich.de/jsc/juropa
10. Malony, A.D., Shende, S.S.: Overhead compensation in performance profiling. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 119–132. Springer, Heidelberg (2004)
11. Malony, A.D., Shende, S.S., Morris, A., Wolf, F.: Compensation of measurement overhead in parallel performance profiling. International Journal of High Performance Computing Applications 21(2), 174–194 (2007)
12. McCabe, T.: A complexity measure. IEEE Transactions on Software Engineering 2, 308–320 (1976)
13. Mellor-Crummey, J., Fowler, R., Marin, G., Tallent, N.: HPCView: A tool for top-down analysis of node performance. The Journal of Supercomputing 23(1), 81–104 (2002)
14. Message Passing Interface Forum: MPI: A message-passing interface standard, version 2.2 (September 2009), ch. 14: Profiling Interface
15. an Mey, D., et al.: Score-P – A unified performance measurement system for petascale applications. In: Proc. of Competence in High Performance Computing, Schloss Schwetzingen, Germany (2010), (to appear)
16. Müller, M., van Waveren, M., Lieberman, R., Whitney, B., Saito, H., Kumaran, K., Baron, J., Brantley, W., Parrott, C., Elken, T., Feng, H., Ponder, C.: SPEC MPI2007 – An application benchmark suite for parallel systems using MPI. Concurrency and Computation: Practice and Experience 22(2), 191 (2010)
17. Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.-C., Solchenbach, K.: VAMPIR: Visualization and analysis of MPI resources. Supercomputer 12(1), 69–80 (1996)
18. Schulz, M., Galarowicz, J., Maghrak, D., Hachfeld, W., Montoya, D., Cranford, S.: Open|SpeedShop: An open source infrastructure for parallel performance analysis. Scientific Programming 16(2-3), 105–121 (2008)
19. Servat, H., Llort, G., Giménez, J., Labarta, J.: Detailed performance analysis using coarse grain sampling. In: Lin, H.-X., Alexander, M., Forsell, M., Knüpfer, A., Prodan, R., Sousa, L., Streit, A. (eds.) Euro-Par 2009. LNCS, vol. 6043, pp. 185–198. Springer, Heidelberg (2010)
20. Shende, S.S.: The role of instrumentation and mapping in performance measurement. Ph.D. thesis, University of Oregon (August 2001)
21. Shende, S.S., Malony, A.D.: The TAU parallel performance system. International Journal of High Performance Computing Applications 20(2), 287–311 (2006)
22. Williams, C.C., Hollingsworth, J.K.: Interactive binary instrumentation. IEEE Seminar Digests 915, 25–28 (2004)

Introduction

Shirley Moore, Derrick Kondo, Brian Wylie, and Giuliano Casale

Topic chairs

Research on performance evaluation over the past several years has resulted in a range of techniques and tools for modeling, analyzing, and optimizing performance of applications on parallel and distributed computing systems. With the emergence of extreme-scale computing architectures, the need for tools and methodologies to predict and improve application performance and to adapt to evolving architectures will become even greater, due to increased complexity and heterogeneity of the systems. Furthermore, the scope of the term performance has expanded to include reliability, energy efficiency, scalability, and system-level context. This years conference topic Performance Prediction and Evaluation aims to bring together researchers involved with the various aspects of this broader scope of application and system performance modeling and evaluation on large-scale parallel and distributed systems.

The six papers accepted for this topic reflect a growing interest in end-to-end performance issues related to network performance, multicore architectures, resource allocation, scheduling, and energy usage. The first three papers focus on different aspects of communication modeling and performance. The fourth and fifth papers address cache partitioning and job scheduling, respectively. The sixth paper focuses on an application-level methodology for minimizing system energy consumption.

- The paper A contention-aware performance model for HPC-based networks: A case study of the Infiniband network presents a methodology for dynamically predicting communication times in congested networks and applies the methodology to an Infiniband network. The paper Using the last-mile model as a distributed scheme for available bandwidth prediction proposes decentralized heuristics for estimating the available bandwidth between nodes in a large-scale distributed system. The heuristics are based on the last-mile model, which characterizes each node by its incoming and outgoing capacity and uses this last-mile (end-host) bandwidth to predict overall performance of the end-to-end paths. The last paper in the network performance area, Self-stabilization versus robust self-stabilization for clustering in ad-hoc network, is an experimental comparison of the performance of four clustering protocols for maintaining a scalable hierarchical network routing scheme in the presence of topological changes due to failures and node motion in a mobile ad-hoc network. Two of the protocols are self-stabilizing, meaning that they converge in finite time to a state that provided optimum service, and two are robust self-stabilizing, meaning that they not only converge to

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 77–78, 2011.

optimum service but also maintain minimal useful service during the stabilization period.
- The paper Multilayer cache partitioning for multiprogram workloads presents a coordinated cache partitioning scheme for multiprogram workloads on multicore systems that considers multiple levels of the cache hierarchy simultaneously. The scheme attempts to satisfy specified quality of service (QoS) values for all applications by partitioning the shared cache hierarchy across them and then distributes the remaining excess cache capacity (if any) across applications such that a global performance metric is maximized. In Backfilling with guarantees granted upon job submission, the authors present two scheduling algorithms based on conservative backfilling by adding prioritized compression and delayed prioritized compression. They use traces of actual workload data from the Parallel Workloads Archive to show that their algorithms generally perform better than normal conservative backfilling.
- The final paper addresses the growing challenge of reducing the energy consumption of high performance computing systems. Entitled Reducing energy usage with memory and computation-aware dynamic frequency scaling, it introduces a methodology that chooses fine-grained dynamic voltage frequency scaling (DVFS) settings, with potentially different setting for different parts of applications, with the goal of minimizing system-wide energy usage. The authors provide a set of automated tools that capture relevant application characteristics at the loop level, match these features using a database of benchmark results to determine the DVFS strategy, and insert the DVFS commands into the application using binary instrumentation.

We would like to take this opportunity to thank all authors who submitted a paper to this topic area, thank the reviewers for their careful evaluations, and finally thank the Euro-Par Organizing Committee for their outstanding management of this years conference.

Reducing Energy Usage with Memory and Computation-Aware Dynamic Frequency Scaling

Michael A. Laurenzano[1], Mitesh Meswani[1], Laura Carrington[1], Allan Snavely[1], Mustafa M. Tikir[2,⋆], and Stephen Poole[3]

[1] San Diego Supercomputer Center, La Jolla, CA, United States of America
{michaell,mitesh,lcarring,allans}@sdsc.edu
[2] Google, Inc, Mountain View, CA, United States of America
mustafa.m.tikir@gmail.com
[3] Oak Ridge National Laboratory, Oak Ridge, TN, United States of America
spoole@ornl.gov

Abstract. Over the life of a modern supercomputer, the energy cost of running the system can exceed the cost of the original hardware purchase. This has driven the community to attempt to understand and minimize energy costs wherever possible. Towards these ends, we present an automated, fine-grained approach to selecting per-loop processor clock frequencies. The clock frequency selection criteria is established through a combination of lightweight static analysis and runtime tracing that automatically acquires *application signatures* - characterizations of the patterns of execution of each loop in an application. This application characterization is matched with one of a series of benchmark loops, which have been run on the target system and probe it in various ways. These benchmarks form a covering set, a *machine characterization* of the expected power consumption and performance traits of the machine over the space of execution patterns and clock frequencies. The frequency that confers the optimal behavior in terms of power-delay product for the benchmark that most closely resembles each application loop is the one chosen for that loop. The set of tools that implement this scheme is fully automated, built on top of freely available open source software, and uses an inexpensive power measurement apparatus. We use these tools to show a measured, system-wide energy savings of up to 7.6% on an 8-core Intel Xeon E5530 and 10.6% on a 32-core AMD Opteron 8380 (a Sun X4600 Node) across a range of workloads.

Keywords: High Performance Computing, Dynamic Voltage Frequency Scaling, Benchmarking, Memory Latency, Energy Optimization.

1 Introduction

Energy costs have become a significant portion of the costs involved in the operational lifetime of largescale systems. These costs have impacts that manifest

⋆ This work was done as an active employee of the San Diego Supercomputer Center.

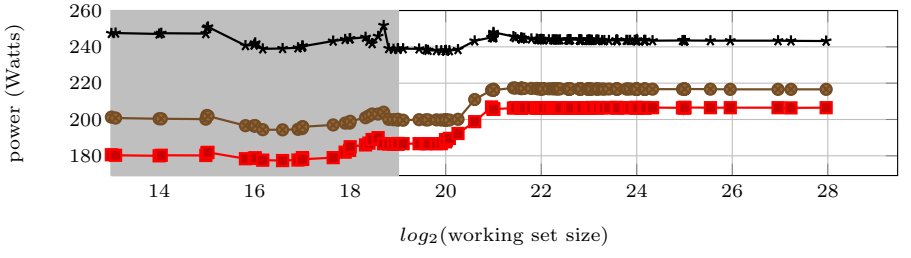
E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 79–90, 2011.

themselves in economic, social and environmental terms. It is therefore prudent to understand and minimize these costs where possible. With that goal in mind, in this work we introduce a methodology that facilitates dynamic voltage frequency scaling (DVFS) based on the expected impact that operating at some frequency will have on HPC application performance and power consumption. This methodology is then leveraged in order to choose fine-grained clock frequency settings, potentially a different frequency for each loop, for the application that minimizes system-wide energy use. Along with this methodology, we present a set of open source tools that automates the entire process.

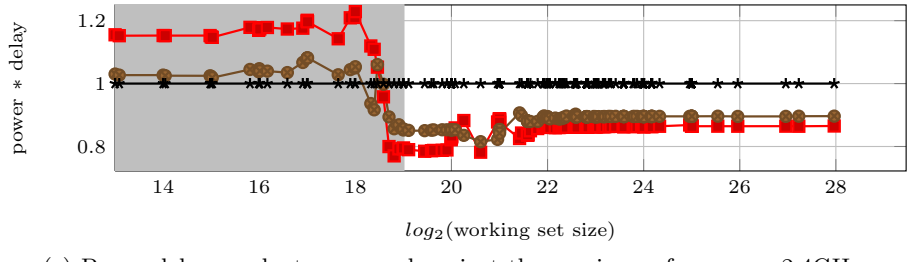
Certain classes of scientific problems and subproblems exhibit memory bound behavior, i.e., the time to solution for the problem is decided primarily by the proximity, size and speed of available memory. Historically, the amount of memory available to computer hardware has increased at an exponential rate[1]. Nevertheless, many applications can, and will continue to, use all of the memory available to them. This means that it is important to consider the impact of physically distant data on performance and power consumption. To facilitate processor frequency scaling as a means to reducing power consumption, many modern processors have been designed to operate at a different clock frequency than certain parts of the memory subsystem[2]. This observation, along with the notion that some applications spend much of their time waiting on data that is physically distant, implies that the execution of such applications may suffer only small or acceptable performance losses when operating at lower clock frequencies, which in turn yields lower power consumption rates. Clock frequency management policies that are in use today, however, generally do not take full advantage of this opportunity. They tend to rely on very broad and coarse-grained measures to determine when it is prudent to lower clock frequency based on perceived inactivity[3] [4] [5]. Our methodology seeks a more refined clock management policy that can exploit the opportunity to down-clock the processor in cases where overall system activity is high, but where the processor is stalled on high-latency memory events.

The opportunity to decrease power consumption by down-clocking the processor as it waits for physically distant data is demonstrated in Figure 1, which shows the performance (Figure 1(a)), power consumption (Figure 1(b)), and power-delay product (Figure 1(c)) for a series of Stream-derived[6] stride 8 memory load tests being run at different working set sizes and clock frequencies on an 8-core Intel Xeon E5530. The results in Figure 1(a), which shows the measured memory bandwidth for this series of tests, suggest that performance is independent of processor clock frequency when the working set size is larger than 512KB. This size coincides with the size of the L2 cache, or equivalently, when the working set size is large enough that the data resides in a memory level farther than L2 cache. Figure 1(b) shows the average power consumption levels during these same memory load tests. It is important to note that power consumption is dependent on clock frequency even for working set sizes where performance is not. Taken together, we can view the results of Figure 1 as an opportunity to reduce power consumption while minimally impacting performance

(a) Performance, expressed as memory bandwidth (MB/s).



(b) Power consumption (Watts) of the entire system (y-axis minimum is 170 Watts).



(c) Power-delay product compared against the maximum frequency, 2.4GHz.

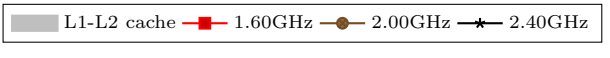


Fig. 1. Performance, power, and power-delay product of a series of Stream-derived stride 8 load tests for several clock frequencies on an Intel Xeon E5530

for certain working set sizes. The power-delay product for the Stream tests is given in Figure 1(c), which shows that power-delay product can be significantly reduced for certain working set sizes by lowering the processor clock frequency. Power-delay product is a metric that combines power usage and performance, and is simply the product of delay and normalized power usage of an application run with some clock frequency management policy when compared to the baseline clock frequency management policy. Note that power-delay product is equivalent to energy usage normalized to the baseline clock frequency mode, so these terms can be used interchangeably.

Though useful as a proof of concept, it is rarely the case that application behavior is as simple as the tests shown in Figure 1. Unlike with the simple Stream benchmark, the processing unit usually has some amount of computation that can be performed while it is stalled on memory accesses, leading to varying degrees of performance degradation when the processor is down-clocked. As such, it is necessary to understand the complex effects that memory, computational behavior and clock frequency have on performance, power and their interesting combinations (such as energy). Our approach to advancing this understanding uses a benchmark to cover the space of some possible behavioral parameters (memory size, memory access pattern, computation amount, computation type, ILP, clock frequency) to measure the effect that these factors have on performance, power and energy. For applications, we can then measure the parameters over which we have limited control (memory and computation related parameters) in order to make informed decisions about the parameter we can control (clock frequency) in order to control the power, performance, and energy characteristics of the application. When applied to selecting for energy-optimal clock frequency, our experiments show that this approach yielded measured, system-wide energy reductions of up to 10.6%.

2 Methodology

To measure the power consumption of a system we employ a WattsUp? power meter [7] to act as an intermediary between the power source and the system power supply. In order to automate the insertion of power measurement interface and clock frequency management calls, we implemented a binary instrumentation tool and library based on the PEBIL instrumentation toolkit [8]. The clock frequency management mechanism is built on top of the `cpufreq-utils` package[1] [3] that is available with many Linux distributions. This instrumentation tool and library provide a powerful and low-overhead way of automatically providing a frequency management strategy to the application without requiring any build-time steps or system support. The power measurement apparatus, at the time of this writing, costs less than $150. Since a data center the size of SDSC [9] has a 2 million dollar annual electricity bill, using this kind of tool within a large data center could save a lot of money without a lot of effort.

2.1 Benchmarking for Power and Performance

In an effort to better understand how a system behaves in the presence of certain types of computational and memory demands, we have developed a benchmarking framework called `pcubed` (**P**MaC's **P**erformance and **P**ower benchmark) that allows us to generate a series of loops while retaining control over the working set size and memory address stream behavior, floating point (FP) operation counts, and data dependence features of each. The first two parameters relate to the

[1] The `cpufreq-utils` frequency switching mechanism currently requires superuser privileges, but we plan to implement a userspace tool that supports our methodology.

behavior of the memory subsystem, while the latter two are related to how effectively the processor can hide memory access latency by performing other useful operations. `pcubed` generates a series of loops, each composed of a series of strided memory references and double-precision FP operations performed on an array. The individual test permutations vary on working set size (*arrsize*), stride length (*stride*), number of memory operations (*memops*), number of FP operations (*fltops*), and number of independent FP operation sequences (*parops*).

Running a set of tests encompassing wide ranges and combinations of these parameters at all available clock frequencies for a target system yields a set of results that describes how that system behaves with respect to performance and power consumption in the presence of a wide range of demands for its computational and memory resources at every clock frequency. The results can then be used as the foundation for forming hypotheses about how an application with a certain set of features in common with the benchmark instances will operate in terms of both performance and power usage at every clock frequency on the target system.

2.2 Application Characterization

In order to determine how an application's characteristics relate to the sets of `pcubed` test characteristics, it is necessary to recognize those features in the application. These collected features are a set of observable characteristics that are related to the input parameters that can be supplied to `pcubed`. These observables are the level 1, 2 and 3 cache hit rates (derived from *arrsize* and *stride*), the ratio of the number of FP operations to the number of memory operations (derived from *fltops* and *memops*) and the lookahead distances for FP and integer computation respectively (derived from *parops* and the loop structure derived from static analysis on the binary), expressed as the average lookahead distance divided by the number of instructions in the loop.

Feature characterization is done at the loop level since loops are the vehicle through which most computation is performed in HPC applications, though it would also be possible to do this at the function level. Every loop and inner loop is examined in order to quantify certain features about its memory behavior, FP intensity and data dependency information. This step consists of a static analysis pass and a runtime trace of memory and control flow behavior that is performed by a binary instrumentation tool implemented with the PEBIL toolkit.

In order to make determinations about the expected behavior of an application loop, we first map it to one of the `pcubed` test loops that is collected as part of the system characterization. For this, we use the geometrically nearest loop in the 6-dimensional space whose members are the set of observable characteristics derived from the `pcubed` input parameters: level 1, 2 and 3 cache hit rates, the ratio of FP operations to memory operations and the average lookahead distance for FP and integer computation. As we will show later, the use of geometric distance between loop feature sets as a basis for comparison seems to work well in practice, but understanding whether geometric distance is the best measure of similarity for two loops is an open research problem.

3 Experimental Results

The technique we have proposed in this work can be used to evaluate an application and its matching `pcubed` loops on any metric involving performance and power. Here we evaluate only power-delay product (energy = E), though metrics such as energy-delay product ($E * D$) or metrics that further emphasize performance such as $E * D^2$ could be evaluated. In order to develop a DVFS strategy for an application whose purpose is to minimize energy usage, we use the set of results gathered from the `pcubed` loop that is geometrically closest to each of the application's loops. We choose the clock frequency for each `pcubed` loop which minimized energy to be the frequency at which we will run the matching application loop. We used two systems and a series of benchmark applications to evaluate this frequency selection scheme.

The first of these systems is an Intel Xeon E5530[10] workstation. The E5530 has 2 quad-core processors. Each core has its own 32KB L1 cache and 256KB L2 cache. Each of the quad-core processors has a shared 8MB L3 cache (for a total of 16MB of L3 for the 8 cores). Each of the 8 cores can be independently clocked at 1.60GHz, 1.73GHz, 1.86GHz, 2.00GHz, 2.13GHz, 2.26GHz, 2.39GHz or 2.40GHz. The second system is a Sun X4600[11] node that is a part of the Triton Resource[12] at the San Diego Supercomputer Center. This Sun X4600 node contains 8 quad-core AMD Opteron 8380[13] processors. Each core has its own 64KB L1 cache and 512KB L2 cache, and each processor shares 6MB of L3 cache (for a total of 48MB of L3 for the 32 cores). Each of the 32 cores can be independently clocked at 800MHz, 1.30GHz, 1.80GHz or 2.5GHz.

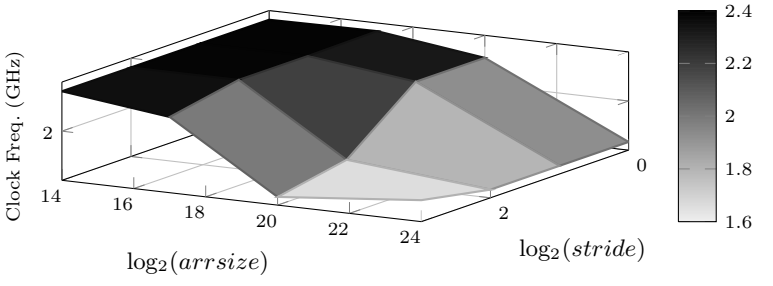
Both of these systems were probed for every available clock frequency by running `pcubed` on a set of 2320 benchmark instances[2] covering a wide range of loop characteristics for every clock frequency exposed by each system, which is 8 frequencies for the Intel Xeon E5530 and 4 frequencies for the AMD Opteron 8380. The runtime of `pcubed` is roughly 6 hours per clock frequency, though this depends on the actual set of tests being run and the clock frequencies involved.

3.1 Drawing Conclusions about System Behavior

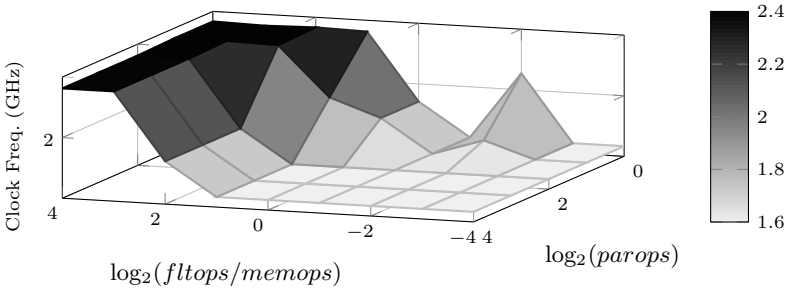
Running `pcubed` on a target system can allow us to draw some conclusions about that system. For instance, if it were found that the energy-optimal frequency for a large number of tests was at the lower end of the available frequencies, it would be possible to argue that lowering the range of available clock frequencies could result in a more energy-efficient system. As a slight variation on this, if a system were being planned that had a very narrow workload which demonstrated similarities to a class of `pcubed` loops that was most energy efficient at lower frequencies, this would suggest that a similar, cheaper architecture that offered a lower maximum frequency would be sufficient for that workload. Similarly

[2] The number of combinations and the exact parameters required depends on the features of the underlying architecture such as number of cache levels, cache line sizes, number of computational units, and number of registers.

if most tests were found to be energy-optimal at the higher clock frequencies, this could indicate that raising the range of available frequencies might have an impact on a system's energy efficiency. Neither of these phenomena were found for either the Intel Xeon E5530 or the AMD Opteron 8380, but it remains to be seen whether such systems exist. By examining the `pcubed` results alone, we can also get an understanding of what feature thresholds delineate energy-optimal frequency domains for the target system. For example, Figures 2(a) and 2(b) show maps of which clock frequency is the most energy efficient for `pcubed` tests as a function of memory behavior and computational behavior respectively. The data in these maps meets our expectations in that the energy-optimal clock frequency generally declines as the amount of time spent stalled on memory increases or as the availability of computation to the processor decreases.



(a) Energy-optimal clock frequency as a function of memory behavior. These tests have fixed $memops = 1$, $fltops = 2$ and $parops$=1.



(b) Energy-optimal clock frequency as a function of the availability of computation. These tests have fixed $arrsize = 16$MB and $stride = 1$.

Fig. 2. `pcubed`-measured energy-optimal clock frequencies on an Intel Xeon E5530

3.2 Energy-Optimal Clock Frequency Selection

For each application benchmark, we make an instrumentation pass over the executable, run the instrumented executable, then run a post-processing step on the results in order to extract the features described in Section 2.2. The overhead of the runtime application analysis depends on the application and its behavior, but for all of the benchmarks studied the maximum overhead was a

13x slowdown (average of 4.0x slowdown) on application runtime, but this step only needs to be run once per application. This post-processing step combines the static and dynamic application analysis, locates the geometrically closely `pcubed` benchmark loop, uses the results from that `pcubed` loop to make a determination about which clock frequency will result in energy-minimal execution for the application's loop, then makes a second instrumentation pass on the executable in order to embed the DVFS strategy into the application.

The set of applications used for the Intel Xeon E5530 is the NAS Parallel benchmarks[14], compiled with both the pgi and gnu compiler, as well as GUPS[15], SSCA#2[16], S3D[17] and HYCOM[18] compiled with the pgi compiler. BT and SP of the NAS Parallel Benchmarks were run on all 4 cores of a single socket. All other benchmarks were run on all of the 8 available cores. The power-delay product (or energy) for each of these benchmarks run with our DVFS scheme is computed against a benchmark run without our scheme, which is to say that it is computed against the default or peak clock rate of the system. Figure 3 shows these power-delay products. The average amount of energy saved for this set of benchmarks is 2.6%, but is as high as 7.6% for CG compiled with the gnu compiler. The set of applications used on the AMD Opteron 8380 is the NAS Parallel benchmarks, GUPS, SSCA#2 and HYCOM, all compiled with the pgi compiler. BT and SP of the NAS Parallel Benchmarks were run on 16 cores on 4 of the 8 sockets available. All other benchmarks were run on all of the 32 available cores. Shown in Figure 3, the average energy saved on the Opteron is 2.1% with a maximum savings of 10.6% on CG.

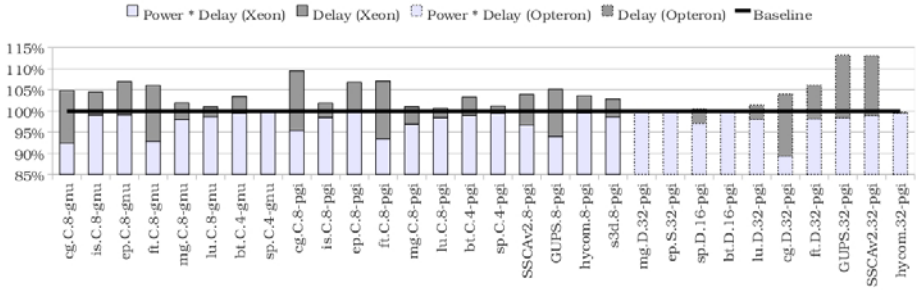


Fig. 3. Application energy usage (Power∗Delay) and slowdown (Delay) when run with a DVFS management scheme, normalized to the default frequency management scheme

In addition to power-delay product, Figure 3 also shows the raw delay to give an account of the slowdown incurred by the tests shown there. The delay is non-trivial and averages 3.8% for both the Intel Xeon E5530 and the AMD Opteron 8380. This highlights the idea that if performance is of enough importance, it is unwise to optimize purely in terms of energy. Rather, in that case it would be prudent to use a higher order function such as energy-delay product that further emphasizes performance. With modifications to the few (less than 10) lines of

source code that currently perform the evaluation of the `pcubed` tests based on energy, one could easily perform evaluations based on energy-delay or any other function of performance and power.

3.3 Technique Validation

It would be time consuming to run every loop of an application at every clock frequency to determine which of those clock frequencies resulted in energy-optimal execution. A simple approach that used this strategy would require a number of runs that is on the order of the product of the number of loops and the number of available clock frequencies. Alternatively, our approach uses a set of benchmark runs (that only have to be run once in the lifetime of a system) in addition to a single instrumented application run in order to gather a heuristic to this effect. But how good is this heuristic? In order to begin to answer this we exhaustively verified that the selected clock frequency on the Intel Xeon E5530 were energy-optimal for a pair of benchmarks codes that have the property that their runtime is dominated by a single loop and therefore an exhaustive search on this loop requires relatively little effort.

For CG, the loop that is responsible for 95% of the dynamic instruction count was found to be geometrically closest, using the metrics described in Section 2.2, to the `pcubed` loop that has $arrsize$ = 1MB, $stride$ = 1, $fltops$ = 8, $memops$ = 16 and $parops$ = 1 (meaning that every FP operation is dependent on the result of its predecessor), which was found by our technique to be energy-optimal when run at 2.13GHz. By subjecting the dominant loop in CG to every available clock frequency and measuring the energy required to complete each run we found that 2.13GHz is the *actual* energy-optimal operating frequency for this loop, confirming that our selection is correct. A similar methodology was applied to the dominant loop in GUPS, which was found by our technique to be energy optimal at 1.60GHz. The `pcubed` instance found to be geometrically closest to the dominant GUPS loop has $arrsize$ = 16MB, $stride$ = 8, $fltops$ = 4, $memops$ = 64 and $parops$ = 4 (meaning that the FP operations carry only an inter-iteration dependence onto themselves). This loop was also found to run energy-optimally at 1.60GHz. This does not serve as conclusive proof that the frequencies selected by our methodology are energy-optimal in all cases nor does it indicate that every interesting aspect of program behavior is encapsulated by the space of possible loops that can be generated by `pcubed`. It does, however, serve to provide some validation for a scenario where exhaustively validating the frequency quality selection would be extremely time consuming.

4 Related Work

Dynamic voltage frequency scaling is a well known technique that has been used to reduce power and energy usage in the context of various application domains [19] [20] [21] [22] [23]. The DVFS research in HPC tends to follow one of two approaches. The first approach is to identify and exploit MPI inter-task load imbalance. The work done in [24] and [25] focuses on locating these imbalances and

applying reduced frequencies to computation regions that are not on the application's critical path. By reducing the energy used on a non-critical path, overall energy can be reduced since power consumption is decreased with negligible performance loss. Since these approaches seek to exploit inter-task imbalance for energy gains instead of intra-task imbalance, they are complementary to ours.

The second approach, which our work falls into, seeks to find a way to exploit performance-clock independence that occurs within a task as a result of memory access stalls. Ge et. al. show in [26] that it is possible to reduce energy or energy-delay by running some memory bound applications either at a fixed frequency for the entire run or by hand-selecting the dynamic frequency settings for the application. Our technique goes further and demonstrates how to automatically select and use a set of such frequency settings.

In [27], the application is run to collect profiling information, then is divided by hand into phases that consist of regions that are either of similar memory behavior or are split by MPI calls. The application is then augmented to give it the capability to perform frequency scaling at its phase boundaries, and then sets of phase/frequency combinations are run in order to determine how particular frequency selections affect power and performance. This work differs from ours in two major ways. First, their methodology differs from ours in terms of the how the application is broken down for analysis. Our methodology currently looks at loop boundaries as the only possible scaling locations; theirs incorporates other possible frequency scaling points. The other major difference between their methodology and ours is that they *search* for the best frequency for the phases in the application by running it with different frequency scaling strategies, while our approach attempts to probe for the capabilities of the system then determines the frequency for the application's loops analytically.

5 Future Work

Going forward, we plan to develop the frequency selection tools as a user-level package so that it does not require root privileges. We also intend to further develop the tools and ideas in order to determine whether they extend to more architectures and compilers to determine the applicability of our methodology to other architectures. Instead of limiting the application analysis to loops only, we also would like to expand the scope of the analysis to include functions.

The extent to which `pcubed` and the parameters used to motivate its development (cache hit/miss behavior, amount of computational work available, and ILP) cover a sufficiently large portion of HPC workloads is unclear. It is likely that more thorough treatment of certain aspects of program behavior is needed. Cache coherence behavior, memory access pattern type, type and width of FP operations, and high latency off-chip events such as I/O or MPI Communications events are obvious candidates for further exploration along these lines. Another way of approaching the same problem would be to leverage some of the existing body of work relating hardware counter-supplied information to processor clock frequency selection. This will facilitate a better understanding of

how application-level performance relevant features such as memory access pattern translate to the underlying conditions, observed from hardware counters, that have an effect on clock frequency selection. Doing this effectively should minimize the set of benchmarks needed to form a covering set of the behaviors that can be expected for HPC applications, which may widen the applicability of the techniques proposed in this work. It could change or narrow the scope of the information that must be gathered from the application in order to perform a mapping of application regions to benchmark instances.

6 Conclusions

This work has shown a benchmark-based approach to selecting processor clock frequency in a way that takes advantage of unnecessarily high clock rates that are maintained during memory bound computations. This methodology is implemented on top of open source software and uses a system-specific performance and power characterization that is automatically derived from the results of a set of benchmark loops, generated by the `pcubed` benchmarking framework, that are run at each clock frequency on the system. A set of tools that capture static and runtime information for an application executable is then used to analyze the application's loops in order to find the benchmark loops whose features match them most closely. From this matching, we are able to select a DVFS strategy for the application that is expected to result in minimizing energy usage during execution. `pcubed` was run, and DVFS strategies were employed on a series of benchmarks on both an Intel Xeon E5530 and an AMD Opteron 8380, where we realized energy savings of up to 7.6% and 10.6% respectively.

Acknowledgements. This work was funded in part by the Department of Defense and used elements at the Extreme Scale Systems Center, located at Oak Ridge National Laboratory and funded by the Department of Defense. This work also used resources from Dash and from the Triton Resource at the San Diego Supercomputer Center. Special thanks to Phil Papadopoulos, Jim Hayes and Jeffrey Filliez at SDSC for their help related to gathering measurements on the Triton Resource.

References

1. Moore, G.E.: Cramming More Components onto Integrated Circuits. Electronics Magazine (1965)
2. Pallipadi, V.: Enhanced Intel SpeedStep Technology and Demand-Based Switching on Linux, `http://software.intel.com/en-us/articles/enhanced-intel-speedstepr-technology-and-demand-based-switching-on-linux/`
3. CPU Frequency Scaling, `https://wiki.archlinux.org/index.php/Cpufrequtils`
4. CPUSpeed, `http://www.carlthompson.net/Software/CPUSpeed`
5. AMD PowerNow! Technology, `http://support.amd.com/us/Embedded_TechDocs/24404a.pdf`

6. McCalpin, J.D.: STREAM: Sustainable Memory Bandwidth in High Performance Computers. Technical Report, University of Virginia (2000)
7. WattsUp? Meters, https://www.wattsupmeters.com/secure/products.php?pn=0
8. Laurenzano, M.A., et al.: PEBIL: Efficient Static Binary Instrumentation for Linux. In: International Symposium on Performance Analysis of Systems and Software (2010)
9. SDSC San Diego Supercomputer Center, http://www.sdsc.edu/
10. Intel Xeon Processor E5530, http://ark.intel.com/Product.aspx?id=37103&processor=E5530&spec-codes=SLBF7
11. Sun Fire X4600 M2 Server Architecture, http://www.sun.com/servers/x64/x4600/arch-wp.pdf
12. Triton Resource, http://tritonresource.sdsc.edu/pdaf.php
13. Keltcher, C.N., et al.: The AMD Opteron Processor for Multiprocessor Servers. In: International Symposium on Microarchitecture (2003)
14. Bailey, D.H., et al.: The NAS Parallel Benchmarks. International Journal of High Performance Computing Applications (1991)
15. Luszczek, P., et al.: Introduction to the HPC Challenge Benchmark Suite. In: International Conference on Supercomputing (2005)
16. Bader, D.A., et al.: Designing Scalable Synthetic Compact Applications for Benchmarking High Productivity Computing Systems. Cyberinfrastructure Technology Watch (2006)
17. Hawkes, E.R., et al.: Direct Numerical Simulation of Turbulent Combustion: Fundamental Insights Towards Predictive Models. Journal of Physics: Conference Series (2005)
18. Halliwell, G.R.: Evaluation of Vertical Coordinate and Vertical Mixing Algorithms in the HYbrid-Coordinate Ocean Model (HYCOM). Ocean Modelling (2004)
19. Poellabauer, C., et al. Feedback-based Dynamic Voltage and Frequency Scaling for Memory-bound Real-time Applications. In: Real Time and Embedded Technology and Applications Symposium (2005)
20. Choi, K., et al.: Dynamic Voltage and Frequency Scaling Based on Workload Decomposition. In: International Symposium on Low Power Electronics and Design (2004)
21. Rajamani, K., et al.: Application-aware Power Management. In: Symposium on Workload Characterization (2007)
22. Isci, C., et al.: An Analysis of Efficient Multi-core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In: International Symposium on Microarchitecture (2006)
23. Lorch, J.R., Smith, A.J.: Operating System Modifications for Task-Based Speed and Voltage. In: International Conference on Mobile Systems, Applications and Services (2003)
24. Freeh, V.W., et al.: Just-in-time Dynamic Voltage Scaling: Exploiting Inter-node Slack to Save Energy in MPI Programs. Journal of Parallel and Distributed Computing (2008)
25. Kimura, H., et al.: Emprical Study on Reducing Energy of Parallel Programs using Slack Reclamation by DVFS in a Power-scalable High Performance Cluster. In: International Conference on Cluster Computing (2007)
26. Ge, R., et al.: Improvement of Power-performance Efficiency for High-end Computing. In: Parallel and Distributed Processing Symposium (2005)
27. Freeh, V.W., Lowenthal, D.K.: Using Multiple Energy Gears in MPI programs on a Power-scalable Cluster. In: Symposium on Principles and Practice of Parallel Programming (2005)

A Contention-Aware Performance Model for HPC-Based Networks: A Case Study of the InfiniBand Network

Maxime Martinasso and Jean-François Méhaut

University of Grenoble, Computer Science Laboratory LIG, Grenoble, France
{maxime.martinasso,jean-francois.mehaut}@imag.fr

Abstract. Multi-core clusters are cost-effective clusters largely used in high-performance computing. Parallel applications using message passing as a communication mechanism may introduce complex communication behaviours on such clusters. By sending and receiving data simultaneously from and to several nodes, parallel applications create concurrent accesses to the resources of the network. In this paper, we present a general model that expresses network resource sharing characterised by a dynamic contention graph. The model is based on a linear system weighted by bandwidth distribution factors called penalty coefficients that are specific to a network technology. We propose a method to solve the linear system and present an analysis to determine penalty coefficients on InfiniBand technology. We use complex network conflicts to assess the ability of the model to predict with low errors.

Keywords: Contention model, performance prediction, InfiniBand.

1 Introduction

High-Performance platforms are dedicated to execute complex scientific applications with the focus on the highest possible performance achievable. In High-Perfomance Computing (HPC) industrial sector software performance has become a keyword for developing such platforms. Reaching optimum performance for end-user applications on clusters is a difficult task that requires the use of a wide range of techniques from specialised algorithms to tuned runtime libraries. This task can be helped by several tools [8] [6] that trace application events that access hardware or software resources of the platform. Nevertheless, strong expertise is required to understand and to analyse the relationship between these events and application performance. The analysis of performance-loss may be simplified if techniques exist to predict application performance and the gain obtained by using such hardware or software. Performance prediction is not only a topic of high interest for the HPC community but is also very challenging.

Clusters of multi-core computers can be seen as platforms providing resources to an application such as computation power, memory, storage and network. The network resource is an important element in performance analysis as it is a

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 91–102, 2011.

slow component of the platform. High performance networks or system-area networks, such as InfiniBand, have been developed to reduce this gap. They are an important architectural element during the design of a cluster. Even for highly specialised network hardware operated by multi-core computers, concurrent accesses are inevitable and degrade network performance. Modelling concurrent accesses and their performance degradations is a step forward to enhance performance improvements of scientific applications.

Each HPC-based network executes its own flow control mechanism when concurrent accesses occur. For instance, InfiniBand provides a credit-based flow-control mechanism on the buffer availabilities to ensure the integrity of the connection. The diversity of flow control mechanisms increases the complexity in identifying models that predict communication times. Models are either too simplistic or too tailored for a particular technology.

In this article we present a method to predict elapsed times of communications that take place concurrently on high performance networks. Our approach generalises the concept of concurrent accesses by introducing the notion of dynamic contention graphs. By creating artificial contention graphs on a network we demonstrate the feasibility of solving a linear system which calculates the elapsed times of every communication. Moreover, our method can be applied on different networks resulting in accurate models. We apply it to a widely popular HPC network: InfiniBand.

The remainder of this paper is organised as follows: in Section 2, we present the background of our study. In Section 3, we introduce our methodology based on dynamic contention graph and a sequence of linear equations. Section 4 presents a model dedicated to InfiniBand network. In Section 5, we validate its accuracy. In Section 6, we conclude and present our future directions.

2 Background

Performance prediction of communication over networks is an extensively researched topic. Contention-free models are generally based on a linear equation [10]. Such models predict the communication delay by multiplying the inverse of the bandwidth with the message size to transfer and by adding a constant value (the network latency). More refined models [3][1] decompose a similar linear equation into more parameters that express the characteristics of a communication. Such parameters can be measured and their values can depend on the message size [14]. These models do not consider contention and resource sharing.

In the aforementioned models, parameter values used to solve a linear equation are taken independently of the resource availability. In [2] previous models are enhanced by introducing a queueing system to represent contention effects. A vast collection of new parameters were introduced for this model. These parameters are difficult to evaluate and limit the applicability of such models.

In [11] contention effects are modelled by a coefficient that divides the bandwidth availability by the number of communications sharing the same link. The presented experiments are limited to mono-processor computers. Therefore, on

a node, concurrent requests of network resources originating from different cores are not considered. The model avoids de facto a large set of contention cases on multi-core technology.

2.1 Elements Influencing Network Contention

Our objective is to propose a methodology to identify models that can accurately predict communication times in a congested network. Many elements should be considered to achieve this goal:

- The first element that we consider is the dynamic of contention behaviours over the network. Contention behaviours are directly linked to the application that is executed on the cluster. The application is responsible for triggering communications that may interfere with each other to access the network resources leading to time delays.
- The second element is the network technology. Each network has its own flow control that regulates the concurrent accesses to its resource. They directly affect and characterise delays caused by contention behaviours.
- The third element is the topology of the network. Many cluster topologies exist, for instance, nodes may be linked to a single switch that may be linked to other switches. Depending on how many network components a communication passes through, its delay depends on the availability of each of the network components.

In our methodology we will focus on the first two elements. The third element, i.e. the topology of the network, will be restricted to one switch connecting several nodes. In [9] we were focused on HPC clusters with dual-core computers and contention graphs based on a mesh of communications. In this paper, we generalise this work to any contention graph and any network technology.

3 Methodology

To address the dynamic of the contention behaviour we introduce the notion of a contention graph that characterises the dynamic usage of network resources by an application. In addition, we add weights named penalty coefficients to a contention graph. The penalty coefficients are factors mirroring the effects on performance of network control flow mechanisms that distribute network bandwidth. We present a technique to determine penalty coefficients by only referring to contention graphs. Finally, our model solves a sequence of linear equations for each communication. Each linear equation represents the communication as part of the dynamic of a contention graph. Linear equations are weighted with penalty coefficients.

3.1 Dynamic Contention Graph

Network contention is characterised by the quantity of data to transfer. Communication delays may change due to other communications present on the network. This variation depends on the sources and destinations, as well as the

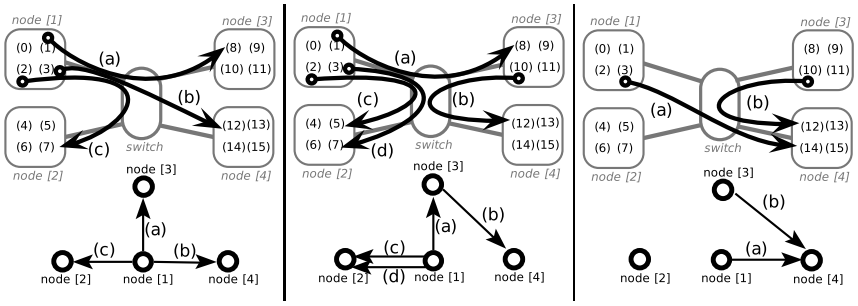


Fig. 1. Examples of an MPI application with 16 ranks spawn on 4 nodes with 4 cores each. During the run of the application, communications start and end creating several contention graphs that are modelled over a dynamic directed graph K.

message size of the communications. For instance, two communications sending data from the same node or to an identical node may create different contention effects on the network by an overlapping demand of network resources. Since the contention effects are related to the communications that are being processed, we introduce the notion of a dynamic contention graph.

A dynamic contention graph is a dynamic directed graph, denoted K, on which the nodes are the cluster nodes and the edges are the communications between the nodes. A contention graph may have parallel edges, however, as our analysis does not consider internal communications, a contention graph does not have any self-loop edges. A communication can start or end at any time modifying the current contention graph. In the same manner, K is modified when an edge is added or removed. In order to reflect the fact that K evolves in the course of different steps, we will denote a contention graph at step s as K_s. Therefore, a step of a contention graph is a static directed graph. Examples of contention graphs and their representation with K are presented in Figure 1.

On multi-core computers and also often on HPC platforms, one MPI task is spawned on one core. Therefore, we bind the maximum degree of K, i.e. the maximum number of simultaneous outgoing and ingoing communications from/into a compute node, to the maximum number of cores per node.

3.2 Sequence of Linear Models

During the execution of an application, communications are creating contention graphs, which may be divided into several steps. While transferring data, one communication can be part of several contention graph steps. We modelled the elapsed time of a specific communication during a specific contention graph step by a single linear equation. In the event that this communication is part of more than one step, its total elapsed time will be equal to the sum of all linear equations modelled in each step. The elapsed time T_i of a communication i of K sending m_i bytes can be approximated by the following formula:

$$T_i = \sum_{j \in K_s} t_{i,j} = \alpha \sum_{j \in K_s} \rho_{i,j} * m_{i,j} \tag{1}$$

$$\sum_{j \in K_s} m_{i,j} = m_i \tag{2}$$

with K_s is a contention graph at step s, $t_{i,j}$ the time spent by the communication i in K_s with $t_{i,j} = \alpha * \rho_{i,j} * m_{i,j}$, α is the inverse of the network bandwidth, $m_{i,j}$ the amount of bytes transferred during $t_{i,j}$ and $\rho_{i,j}$ ($\rho_{i,j} \geq 1$) a coefficient called penalty, which acts as the delay caused by other communications in K_s. To reflect the progress of communications over steps of K, we introduce the following relation between $t_{i,j}$:

$$t_{1,1} = T_1 \text{ and for } \forall j \text{ such that } i \geq j > 1, \;\; t_{i,j} = T_i - T_{i-1} \tag{3}$$

The problem of the evaluation of T_i resides in the number of unknown variables present in the model. If a communication is part of s contention graph steps, the model has $2 * s$ unknowns to be determined: s penalty coefficients and s sizes of the data transferred during each step. However, if the penalty coefficients are known for every K_s and only the initial message size m_i of each communication i is known, then it is possible to calculate T_i for all i. By solving T_i in an increasing order of i, one knows from (3) the value of $t_{i,j}$ with $j < i$ and thus the value of $m_{i,j}$. From (2) one can deduce the size $m_{i,i}$ and then $t_{i,i}$ leading to the result T_i by (1). In the next subsection we introduce a method to approximate the penalty coefficients.

3.3 Approximation of Penalty Coefficients

Penalty coefficients are factors which divide bandwidth and thus increase communication delays. Their values depend on the underlying flow control that manages the over-subscription of a resource. Therefore, an identical contention graph step generates different penalty values depending on the network technology.

Penalty coefficients are directly related to the shape of a contention graph. In our model we consider that one contention graph step creates a unique set of penalties. Since dedicated HPC networks are highly deterministic, this consideration does not appear to be a strong restriction. However, if we consider a network topology with several switches, this condition should be revised.

To determine penalty coefficients of a contention graph we use real experiences on a cluster. We have developed a simple MPI benchmark that creates contention graphs. This benchmark spawns as much MPI processes as available cores. An MPI process is either only sending data or receiving data. By selecting a subset of processes that send or receive data and by binding them to an MPI communicator we can create a contention graph. We configure our benchmark to create a specific kind of contention graph that we call static-synchronous contention graph. In a static-synchronous contention graph communications start at the same time and have a same amount of data to transfer. This benchmark gathers statistical data over elapsed times of every communication.

To approximate penalty coefficients, we use the reverse approach as presented in the previous subsection 3.2 to solve T_i. By knowing an approximation of every T_i of K and the initial size $m_i = m$ of every communication we are able to approximate $\rho_{i,j}$ for any static-synchronous contention graph. The maximum number of steps of such graph is equal to the number of its communications.

A static-synchronous contention graph of s steps has a number of communications that decreases while the number of steps increases. Static-synchronous graphs with only one communication have a penalty coefficient equal to 1. A static-synchronous graph of two communications implies a linear system which is directly solvable. Therefore, it is possible to calculate penalty coefficients of any static-synchronous graphs with two communications. Static-synchronous graphs of three communications have, in the general case, three steps. We are interested in calculating penalty coefficients of the first step: $\rho_{1,1}$, $\rho_{2,1}$, $\rho_{3,1}$. To compute these penalty coefficients we need to know the penalty coefficients of its two other steps: $\rho_{2,2}$, $\rho_{2,3}$ and $\rho_{3,3}$. The last step has only one communication remaining, therefore $\rho_{3,3} = 1$. For the values of $\rho_{2,2}$ and $\rho_{2,3}$, we can use the penalty coefficients of a static-synchronous graph of two communications that represents the graph at this step. Therefore, it becomes possible to compute $\rho_{1,1}$, $\rho_{2,1}$, $\rho_{3,1}$ at the first step by solving the system below:

$$\begin{cases} \rho_{1,1} = T_1/(\alpha * m) \\ \rho_{2,1} = T_1/(\alpha * m_{2,1}) \text{ with } \begin{cases} m_{2,1} = m - m_{2,2} \\ m_{2,2} = (T_3 - T_2)/(\alpha * \rho_{2,2}) \end{cases} \\ \rho_{3,1} = T_1/(\alpha * m_{3,1}) \text{ with } \begin{cases} m_{3,1} = m - m_{3,2} - m_{3,3} \\ m_{3,2} = (T_3 - T_2)/(\alpha * \rho_{3,2}) \\ m_{3,3} = (T_2 - T_1)/(\alpha * \rho_{3,3}) \end{cases} \end{cases}$$

By generalising this method, we can recursively determine penalties for a n-communication static-synchronous graph. For its $(n-1)$ last steps we can apply the penalty coefficients of $(n-1)$ static-synchronous graphs having respectively $(n-1)$ to 1 communications. The shape of each of these static-synchronous graphs should correspond to the shapes of the respective steps in the n-communication graph to which their penalties are applied to.

In our model we assume that a step of a contention graph is comparable to a static-synchronous graph, in other words, that the transition between steps, i.e. the network reconfiguration of its contention behaviour, is immediate. Within this assumption, when a new contention graph step starts, every communication enters the new step at the same time (synchronous) and their contention behaviour (i.e. penalty coefficients) will not change (static) until the step ends and a following step starts. Therefore, we are able to approximate every penalty coefficient of each step of a contention graph.

By analysing a set of hundreds of contention graphs and their penalty coefficients, we present a model that approximates penalties based on the shape of a contention graph. This model is applicable for InfiniBand network.

4 Modelling Penalty Coefficients over InfiniBand

In the top 500 list, about 35% of the clusters were using InfiniBand[5] in 2009, which has increased to about 42% in 2010. Research on InfiniBand is mainly covering routing strategies in order to increase application performance.

In [13] the impact of static routing in multistage InfiniBand networks is presented. The study focuses on evaluating the available bisection-bandwidth between nodes obtained through different switches. Similarly to [11], it considers concurrent communications between different pairs of nodes and not concurrent communications that are going to or are initiated from different cores of one node. In our study, however, we demonstrate that such concurrent communications create significant contention delays even with a single switch.

An analysis based on a LogP model for small message size communication performance over InfiniBand was also discussed in [12].

4.1 InfiniBand Network Testbed

Our experiments were carried out on a cluster based on 34 nodes with 8 cores per node. Each node is connected with a InfiniBand Mellanox Technologies MT26418 card to 3 Voltaire Grid Director 9024 switches. Our MPI implementation is OpenMPI 1.2.7 using the OpenFabrics low-level library.

Routing paths in InfiniBand network are statically fixed. Taking into account the routing strategy leads to complex modelling. Our intention is to explore the possibilities in approximating penalty coefficients. In order to limit the influence of static routing strategy in our analysis we select a set of nine nodes that are connected to a unique switch. Each switch has an internal fat-tree topology that guarantees an efficient distribution of the bandwidth between the links. Thus the variance of experimented results remains low, as well as the distortion in the approximation of the penalty coefficients.

InfiniBand flow control uses a credit-based mechanism. A network card that receives data sends messages called *link credit* to inform the sender about the available buffer size remaining. If a receiver buffer is full then the destination network card stops sending *link credit* messages and the source network card will hold in a *back pressure* state delaying the entire set of its outgoing communications. The communication is resumed when sufficient buffer size is available.

4.2 Penalty Coefficients and Model for InfiniBand

Upon analysing the penalty coefficients of several hundreds of static-synchronous contention graphs we propose the model below. A contention graph step is represented by a connected directed graph $K(V, E)$ where V is the set of nodes, i.e. the cluster compute nodes, and E is the set of edges, i.e. the communications involved in the contention graph step. As usual, we use $\delta^+(v)$ to denote the outdegree of a vertex v of K and, respectively, $\delta^-(v)$ to denote the indegree of a vertex v of K. We use $V^+(v) \subset V$ as the out-neighbourhood of vertex v (set of vertices being a destination vertex of an edge outgoing from v). We extend the notion of $V^+(v)$ to a set of out-neighbourhood edges of an edge $e = (s, d)$

as $E^+(e) = \{(s, d') : d' \in V^+(s)\} \subset E$. We also consider a specific set of edges being the edges ingoing to the destination vertex d of an edge (s, d). This subset is defined as follow: $E^{\bowtie}(e) = \{(s', d) : s' \neq s\} \subset E$. The penalty coefficient $\rho(e)$ of an edge $e = (s, d) \in E$, being oriented from s to d, is calculated as follows:

$$\rho(e) = \max_{E^+(e)} \{\delta^+(s) + k(e)\} \tag{4}$$

with $k(e)$ defined as follow:

(a) $k(e) = 0$ if $(\delta^-(d) \leq \delta^+(s)$ and $\delta^+(s) = \delta^+(s'))$ or $\delta^-(d') = 0$ such that $e' = (s', d) \in E^{\bowtie}(e)$ and $e'' = (s, d') \in E^+(e)$ respectively;

(b) $k(e) = \frac{1}{max(\rho(e')-1)}$ if $\delta^+(s) = 1$ with $e' = (s', d) \in E^{\bowtie}(e)$;

(c) $k(e) = \sum_{\forall e'=(s,d') \in E^+(e) \wedge \forall e''=(s'',d') \in E^{\bowtie}(e')} \frac{1}{\delta^+(s'')}$ otherwise.

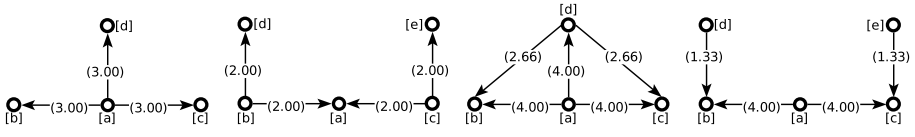


Fig. 2. Penalty coefficients of four contention graphs

In Figure 2 we apply the model above on four contention graphs. In the first graph, the outgoing degree of node [a] is 3, $\delta^+(a) = 3$. Nodes [b], [c] and [d] have an ingoing degree of 1, $\delta^-(b) = \delta^-(c) = \delta^-(d) = 1$ therefore from item (a) k is nullified for all edges. From (4) each edge has a penalty of 3. In the second graph k is also nullified, because for both edges (b, a) and (c, a): $\delta^-(a) = \delta^+(b) = \delta^+(c)$. For the third graph neither item (a) or item (b) is applicable. The penalty coefficients of edges (a, b) (a, c) (a, d) are then $max(\delta^+(a) + \frac{2}{\delta^+(d)}, \delta^+(a) + \frac{2}{\delta^+(d)}, \delta^+(a)) = 4$. Similarly, edges (d, b) and (d, c) have each a penalty coefficient equal to $max(\delta^+(d) + \frac{2}{\delta^+(a)}, \delta^+(d) + \frac{2}{\delta^+(a)}) = 2.66$. Finally, the last example combines the item (b) and item (c). Edges (a, b) and (a, c) have a penalty of 4 from item (c). Item (b) is applicable for edges (d, b) and (e, c) as $\delta^+(d) = \delta^+(e) = 1$. Their penalty coefficients are equal to $1 + \frac{1}{max(\rho((a,b)),\rho((a,c))-1} = 1.33$.

Discussion: In our model, the contention factor depends mainly on the number of outgoing communications from a node. Inside a node, the network card is the first network component that is shared. The network card allocates a first distribution of bandwidth capacity. We model this distribution as a fair distribution among the communications (4). In addition, bandwidth may also be reduced

when communications share the same destination node. On a receiving card of a node, the buffer allocation depends on the bandwidth requested by each ingoing communication to the node. Therefore, in our model we characterise this value by the function k that adds a factor proportional to the bandwidth requests made by others communications having the same destination node (item (c)). However, k can be nullified if n communications that are sending data to the same destination also share their source node with n outgoing communications. Alternatively, k can also be nullified if a communication does not suffer contention on the receive node (item (a)). Finally, if a network card is in a *back pressure* mode (the receiver blocks the sender) then its outgoing communications are delayed. This effect is modelled by choosing the maximum delay over all communications outgoing from the same node. Furthermore, if a communication with a full bandwidth allocation reaches a node, we model its allocated bandwidth on the receive node as the maximum bandwidth available (item (b)).

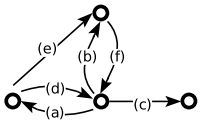
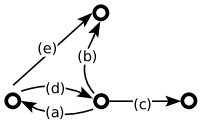
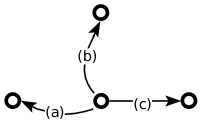
dynamic graph steps	com	penalty	data remaining [o]	time spent [s]
	a	3.5	11983700	0.0160590
	b	3.5	11983700	0.0160590
	c	3.5	11983700	0.0160590
	d	3.33	11534300	0.0160590
	e	3.33	11534300	0.0160590
	f	1.5	0	0.0160590
	a	3.5	4294170	0.0297984
	b	3.5	4294170	0.0297984
	c	3.5	4294170	0.0297984
	d	2.33	0	0.0297984
	e	2.33	0	0.0297984
	f	-	0	0.0160590
	a	3.0	0	0.0363749
	b	3.0	0	0.0363749
	c	3.0	0	0.0363749
	d	-	0	0.0297984
	e	-	0	0.0297984
	f	-	0	0.0160590

Fig. 3. Example to determine communication times of a dynamic contention graph. All communications have an identical size of 20MB and start at the same time.

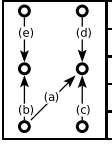
5 Examples and Validation

Before evaluating the accuracy of our model, we will introduce one example to calculate the elapsed time of one arbitrary chosen contention graph. We consider the dynamic contention graph presented in Figure 3. This dynamic contention graph has six communications and for simplicity all communications start at the same time and have a data transfer size of 20MB. By executing our benchmark

(presented in 3.3), we measure an effective bandwidth of 1.82 GB/s on our testbed evaluating $\alpha = 5.105 \times 10^{-10}$. A single communication of 20MB under no contention lasts for 0.010706 seconds.

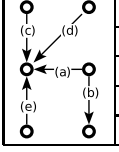
By solving the linear system step by step we deduce the time of each communication. The first communication to end is communication f after a time of 0.016059 seconds. During that time the communications a, b and c have sent 8987820 bytes. Then the contention graph goes to a new step (without f), in which the remaining data size per communication is deducted and new penalty coefficients are applied. The second step ends when communications d and e terminate at time 0.0297984. The final step has only three communications each of them having 4294170 bytes remaining. At this step the penalty coefficient per communication is 3, i.e. each communication accesses one third of the available bandwidth. The last step ends at time 0.0363749. By executing our benchmark and creating this arbitrary contention graph, we measure the time of each communication on our testbed. We compare the measured times to the predicted times leading to an absolute error of 3% for communications a, b and c, an absolute error of 1% for d, e and 2% for f.

Graph A



	$\rho(e)$	T_p	T_m	err.
a	5, 4, 2	0.036132	0.036328	0%
b	5, 4, 2	0.036132	0.036326	0%
c	2.5, 2.5	0.026765	0.027653	-3%
d	2.5, 2.5	0.026765	0.027651	-3%
e	1.5	0.013382	0.013413	0%

Graph B



	$\rho(e)$	T_p	T_m	err.
a	5, 2	0.043894	0.045236	-2%
b	5, 2	0.043894	0.045228	-2%
c	3.5	0.037471	0.040073	-6%
d	3.5	0.037471	0.040072	-6%
e	3.5	0.037471	0.040071	-6%

Fig. 4. Examples of model validation: $\rho(e)$ is the list of penalty coefficients, T_p is the predicted time, T_m is the measured time and err. is the relative error.

In Figure 4, we present the measured times and predicted times of two other dynamic contention graphs with identical communication properties as in aforementioned example. Graph A shows communications a and b starting from the same node having different conflicts on their receive nodes. Communications of graph B are mainly congested in an ingoing conflict of four communications. For these two graphs, our model accurately predicts elapsed times.

The previous examples display a set of heterogeneous communication elapsed times. These times are not simple to determine and their variation shows the complexity in accurately predicting them. In a last validation analysis of our model, we investigate the model prediction for a set of dynamic contention graphs that are derived from random directed graphs. These dynamic contention graphs are categorised in groups following their number of edges. Figure 5 displays, per group, the measured times of their communications. For instance, we measured 60 contention graphs of 5 edges leading to evaluating 300 communication times. Figure 5 also displays, per group, the maximum value of the absolute average error per graph and the maximum value of the absolute error over all communications. The absolute average error for a graph is the average of the absolute

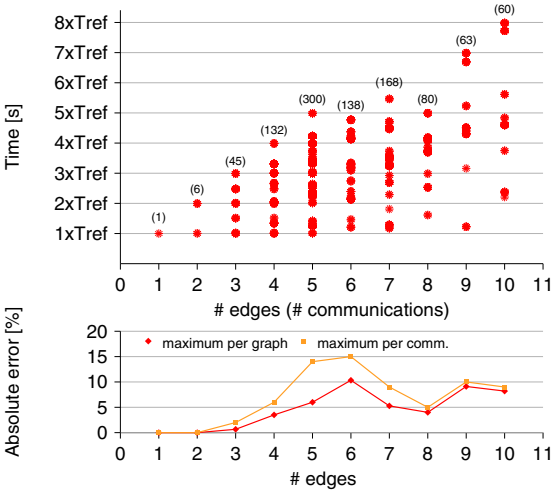


Fig. 5. Validation on several contention graphs. $T_{ref} = 0.010706$ is the reference time of a contention-free communication. For graphs with 1, 2 and 3 edges, times are rational factor of T_{ref}. However, for a higher number of edges, communication times are more largely spread. Our predictions are within an acceptable range of errors.

error among its communications. The figure shows that we accurately predicted communication times with an acceptable error below 15% over a total of nearly 1000 measured communications.

6 Conclusion and Future Work

In this paper we introduce a new methodology for assessing contention over HPC-based networks. This method models congestion over a network by a contention graph and a linear system weighted by delay factors called penalty coefficients. We propose a technique that determines penalty coefficients from experimented contention graphs. We applied this technique on an InfiniBand network. By analysing the penalty coefficients of contention graphs we approximate their values only by referring to the contention graph. Finally, we accurately predict the communication times of nearly a thousand communications requiring only one parameter: the effective bandwidth of the network.

We applied our methodology on a network topology restricted to one switch. Our future work will focus on extending our model to consider network topology and to identify models for large-scale applications running on hundreds of cores. In addition, we will incorporate our model into an existing simulator [4], which replays events of an application and determines dynamic contention graphs. Besides, all-to-all collective operations generate congested communications [7]. It will be interesting to model their performance following our methodology.

We consider applying this methodology on other network technologies. By modelling several networks, it will be possible to compare application performance on those networks.

Acknowledgement. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

1. Alexandrov, A., Ionescu, M., Schauser, K., Scheiman, C.: LogGP: Incorporating Long Messages into the LogP model for Parallel Computation. Journal of Parallel and Distributed Computing 44(1), 71–79 (1997)
2. Moritz, C.A., Frank, M.I.: LoGPC: Modeling Network Contention in Message-Passing Programs. IEEE Transactions on Parallel and Distributed Systems 12(4), 404–415 (2001)
3. Culler, D., Karp, R., Patterson, D., Sahay, A., Santos, E., Schauser, K., Subramonian, R., von Eicken, T.: LogP: a practical model of parallel computation. Commun. ACM 39(11), 78–85 (1996)
4. Casanova, H., Legrand, A., Quinson, M.: SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In: 10th IEEE International Conference on Computer Modeling and Simulation (2008)
5. InfiniBand Trade Association: InfiniBand Architecture Specification, Release 1.2.1
6. Intel Corporation: Intel Trace Analyzer and Collector 8.0 Reference Guide
7. Steffenel, L.A., Martinasso, M., Trystram, D.: Assessing Contention Effects on MPI_Alltoall Communications. In: Cérin, C., Li, K.-C. (eds.) GPC 2007. LNCS, vol. 4459, pp. 424–435. Springer, Heidelberg (2007)
8. Geimer, M., Felix, W., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience 22(6), 702–719 (2010)
9. Martinasso, M., Méhaut, J.-F.: Model of concurrent MPI communications over SMP clusters. Tech. Rep. 00071352, HAL-INRIA (2006)
10. Hockney, R.W.: The Communication Challenge for MPP: Intel Paragon and Meiko CS-2. In: Parallel Computing, vol. 20, pp. 389–398. North-Holland, Amsterdam (1994)
11. Kim, S.C., Lee, S.: Measurement and Prediction of Communication Delays in Myrinet Networks. Journal of Parallel and Distributed Computing 61(11), 1692–1704 (2001)
12. Hoefler, T., Mehlan, T., Mietke, F., Rehm, W.: LogfP - A Model for small Messages in InfiniBand. In: Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium, IPDPS (2006)
13. Hoefler, T., Schneider, T., Lumsdaine, A.: Multistage Switches are not Crossbars: Effects of Static Routing in High-Performance Networks. In: Proceedings of the 2008 IEEE International Conference on Cluster Computing, pp. 116–125 (2008)
14. Kielmann, T., Bal, H.E., Verstoep, K.: Fast Measurement of LogP Parameters for Message Passing Platforms. In: IPDPS 2000: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing, pp. 1176–1183 (2000)

Using the Last-Mile Model as a Distributed Scheme for Available Bandwidth Prediction

Olivier Beaumont[1], Lionel Eyraud-Dubois[1], and Young J. Won[1,2]

[1] INRIA Bordeaux Sud-Ouest
LaBRI, University of Bordeaux 1, Talence, France
{Olivier.Beaumont,Lionel.Eyraud-Dubois}@labri.fr
[2] IIJ Research Lab., Tokyo, Japan
young@iijlab.net

Abstract. Several Network Coordinate Systems have been proposed to predict unknown network distances between a large number of Internet nodes by using only a small number of measurements. These systems focus on predicting latency, and they are not adapted to the prediction of available bandwidth. But end-to-end path available bandwidth is an important metric for the performance optimisation in many high throughput distributed applications, such as video streaming and file sharing networks. In this paper, we propose to perform available bandwidth prediction with the last-mile model, in which each node is characterised by its incoming and outgoing capacities. This model has been used in several theoretical works for distributed applications. We design decentralised heuristics to compute the capacities of each node so as to minimise the prediction error. We show that our algorithms can achieve a competitive accuracy even with asymmetric and erroneous end-to-end measurement datasets. A comparison with existing models (Vivaldi, Sequoia, PathGuru, DMF) is provided. Simulation results also show that our heuristics can provide good quality predictions even when using a very small number of measurements.

Keywords: Network Coordinate System, Last-Mile, Network Measurement, Available Bandwidth Prediction, Labeling Scheme.

1 Introduction

Predicting network performance (latency or available bandwidth) is important for many Internet applications. For video on demand [18] and peer-assisted streaming [14] for example, estimations of available bandwidth allow the construction of an efficient overlay topology.

A number of measurement tools have been developed [9] which measure the available bandwidth on the path between two given Internet nodes. However, in a large scale system, performing measurements between all pairs of nodes would incur too large of an overhead. Thus, there is a need for the possibility to infer

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 103–116, 2011.

(in this paper we also use the term predict[1]) the unmeasured bandwidth values from a limited number of actual available measurements.

For latency estimation, several solutions have been successfully proposed, under the global terminology of Network Coordinate Systems. Most of these solutions embed network nodes into a metric space (not necessarily Euclidean) and approximate the latency between nodes by the distance between their embeddings, which can easily be computed from their coordinates. GNP [16] is an example of such a system, in which each node is positioned in an Euclidean space with respect to a number of landmarks whose positions are already established. Vivaldi [6] is a decentralised extension of GNP, which avoids the need for landmark nodes. However, the efficient counterparts of these coordinate systems for available bandwidth prediction are still to be proposed.

Recent literature about overlay design for peer-to-peer data dissemination [3] has generalised the use of a "last-mile" approximation, in which the rates of simultaneous communications are only limited by the upload and download capacities of each node. This simplifying assumption is quite natural in the context of the Internet, and allows to derive provably efficient overlay designs. In this paper, we analyse the validity of this approximation, and how it can be used to develop a technique for predicting available bandwidth from a limited number of measurements.

More precisely, we propose a decentralised heuristic to compute the capacities of each node from a relatively small number of measurements. We analyse this heuristic by using a dataset of available bandwidth measurements performed on PlanetLab [5]. The accuracy of the predicted values with our solution compares favourably with existing solutions, while requiring significantly fewer measurements.

The organisation of the paper is as follows. We first describe the related works in Section 2. Section 3 presents the rationale behind the last-mile model and our proposed heuristic to compute the capacities of the nodes. In Section 4, we present the evaluation of our solution and compare it to other existing solutions. Concluding remarks and future works are given in Section 5.

2 Related Works

2.1 Latency Estimation

In the context of latency estimation, a number of network coordinate systems have been proposed, based on the following idea: embedding the nodes of the network into a multi-dimensional space and using the distance between two

[1] The design of techniques for efficient and reliable available bandwidth measurements is an interesting research question, but it is not in the scope of this paper. Instead, we assume in this work that a (limited) number of measurements are available, and our goal is to use these measurements to provide estimations for the unmeasured bandwidth values.

points in this space as an estimation of the latency between the corresponding nodes. We can make a distinction [11] between landmark-based and decentralised approaches. In the landmark-based approach (*e.g.* GNP [16], PIC, etc.), a fixed number of landmark nodes are selected and positioned in the space. Non-landmark nodes measure their latency to these landmark nodes and compute their coordinates so as to minimise the resulting prediction error. On the other hand, in the decentralised approach (*e.g.* Vivaldi [6], Big Bang Simulation, etc.), all participating nodes have the same role, and the coordinates of the nodes are computed in a decentralised way by direct measurements between participating hosts.

Vivaldi [6] is one of the most well-known decentralised coordinate system. It relies on the simulation of a system of springs, in which the interaction between two nodes is modeled by a spring whose force represents the estimation error. This simulation procedure allows to adapt the computed coordinates to changing network conditions.

Recent works have also studied embedding into a hyperbolic structure [7]. An example of such a system is the Sequoia algorithm [17], which embeds the nodes as the leaves of a weighted tree, and approximates the distance between two nodes by the length of the path between their respective positions in the tree. The Sequoia algorithm comes with a theoretically proven performance guarantee, and can be applied to both latency and bandwidth estimation. However, the algorithm is quite sensitive to violations of the triangular inequalities, and there is for the moment no decentralised version of Sequoia: computing the embedding requires the measurements between all pairs of nodes.

An important problem with metric-based embeddings comes from the violations of triangle inequalities, which are often observed in Internet measurements. Several studies have thus considered non-metric embeddings. IDES [15] is based on matrix factorisation, which consists of approximating a large matrix by the product of two smaller matrices. Each node is thus assigned two vectors (an incoming and outgoing vector), which correspond respectively to one row and one column of the two smaller matrices. The distance between two nodes A and B is computed as the scalar product of the outgoing vector of A and the incoming vector of B. The IDES system is based on a set of landmark nodes, and recently a decentralised version has been proposed, called DMF [13] for decentralised matrix factorisation. DMF is an iterative procedure in which each node locally minimizes the prediction error by solving a least square problem.

2.2 Bandwidth Estimation

There is a relatively small number of studies focusing on bandwidth estimation. The authors of Sequoia [17] studied the applicability of their algorithm to available bandwidth, and the works based on matrix factorisation [13] can be applied to bandwidth estimation as well. PathGuru [19] is a landmark-based system specifically designed for available bandwidth estimation, which relies on the observation that, in certain circumstances, Internet available bandwidth forms an ultra metric space. In PathGuru, each node measures the available bandwidth to

and from every landmark, and the estimation of bandwidth between two given nodes A and B is performed using the pair of landmarks which most closely forms an ultra metric space with A and B.

BRoute [10] is another system for available bandwidth estimation, which is based on the observations that most bottleneck links are on the path edges, and that relatively few routes exist near the source and destination. Unlike the previous solutions which only require end-to-end measurements, BRoute uses landmarks and network management tools (such as traceroute and BGP routing information) to identify the bottleneck links near each source and destination, and to infer which links are used by packets between A and B.

A number of works in the literature of communication optimization in large scale systems assume that each participating node is characterised by its upload and/or download bandwidth. This applies to a variety of topics, such as video-on-demand [18,4], peer-assisted streaming [14,3,2] or multi-port divisible load scheduling [1]. Thanks to its simplicity, this rather natural assumption allows to derive provably efficient algorithms. In this paper, we analyse how well this model can approximate the actual available bandwidth in a large scale distributed platform.

3 Last-Mile Bandwidth Prediction Model

3.1 Last-Mile Model

Throughout the paper, we denote by $\mathcal{M}_{A,B}$ the measured available bandwidth from node A to node B, and by $\mathcal{P}_{A,B}$ the corresponding predicted value.

The previous research on the properties of the Internet indicate that the bandwidth at the edge of the network, the so called last-mile (end-host) bandwidth, reflects the overall performance of the complete end-to-end path. Hu et al. [12] show that 60% of wide-area Internet paths between end-hosts have their bottleneck in the first or second hop. A recent study also shows a similar property in broadband access networks [8]. As an insight, we have observed the dataset obtained from measurements on the PlanetLab platform [5] which is described more precisely in Section 4 where it is used to evaluate our heuristics.

As a representative example, Figure 1 is the plot of the outgoing bandwidth measurements from hosts `planetlab3.hiit.fi` and `planet-lab7.millenium.berkeley.edu`, which will be denoted `hiit` and `berkeley` in what follows, to all the other hosts in the platform. The bandwidth values are sorted for increased readability. The plot for `berkeley` (Figure 1(b)) shows what can be expected for a host with a low outgoing capacity: the bandwidth to the first 50 hosts is limited by their respective ingoing capacity, and then the bandwidth to the rest of the nodes is limited by the outgoing capacity of `berkeley`, and thus the plot remains quite flat. On the other hand, the plot for `hiit` (Figure 1(a)) shows the result for a host with a large outgoing capacity: the bandwidth increases quite smoothly. In both cases however, a small number of larger bandwidth measurements can be noticed by a sharp increase around node 300, which can be interpreted as bogus measurements and will be discussed later.

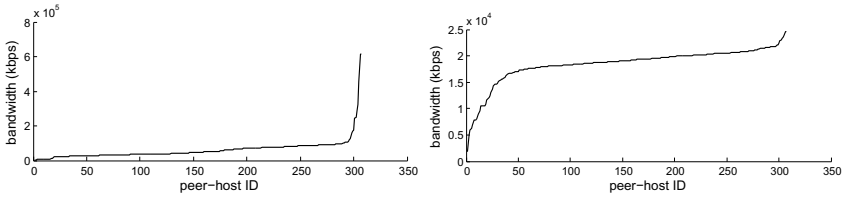


Fig. 1. Outgoing bandwidth distributions for two PlanetLab hosts; the values are sorted in increasing order: (a) planetlab3.hiit.fi; (b) planetlab7.millennium.berkeley.edu

This observation has led to propose a last-mile modelisation of available bandwidth [14]. In this model, each participating node x is represented by two different bandwidth capacities, one for upload (β_x^{out}), and one for download (β_x^{in}). Based on these values, and assuming that they are the only limiting factors for the end-to-end performance, the predicted bandwidth $\mathcal{P}_{x,y}^{\text{LM}}$ between two nodes x and y is given by $\min(\beta_x^{\text{out}}, \beta_y^{\text{in}})$.

In this paper, in order to analyse the validity of this model, we propose heuristics to compute values for β_x^{out} and β_x^{in} so as to minimise the prediction error. We first propose a simple way to compute reasonable initial values, and then describe an iteration procedure following the one used in the context of DMF [13].

3.2 Initial Values

The initial observation may identify that the upload capacity of a node β_x^{out} has to be at least as large as the measured value between node x and any other node y (otherwise it would have been impossible to measure such a high value for this particular node y). However, setting $\beta_x^{\text{out}} = \max_y \mathcal{M}_{x,y}$ is potentially dangerous: only one bogus measurement is enough to obtain wrong predictions for x. Furthermore, the last-mile assumption is not always satisfied in practice, and it may happen that some nodes share a bottleneck link. A typical example is the case of two nodes A and B on a common local area network, which is connected to the Internet through a DSL connection. The bandwidth from A to any other node X on the Internet is then limited by this DSL connection, while the bandwidth from A to B is not. Hence, setting $\beta_A^{\text{out}} = \max_y \mathcal{M}_{A,y} = \mathcal{M}_{A,B}$ would result in largely over-estimated predictions for bandwidth from A to any X on the Internet, since it would ignore the limiting DSL connection.

This situation can be observed on our dataset. For instance, on Figure 1(a), most values lie below 2.10^5 kbps, except for a couple of outliers with very high measured bandwidth, which can be either erroneous measurements or hosts with a local, direct high-capacity connection to `hiit`.

This observation motivates the removal of a few outliers hosts before computing β^{in} and β^{out} values. The solution we propose in this paper is to define β_A^{out} as a given percentile $1-\alpha$ of all measured values $\mathcal{M}_{A,y}$. It is a generalisation of the previous straightforward answer of taking the maximum value, which corresponds to $\alpha = 0$; larger choices of α ignore more and more measurement values.

This solution yields a very simple way of computing β^{out} and β^{in} values, and is very resilient to missing and erroneous measurements and "too high" bandwidths due to nodes in the same local network, since corresponding measurements are ignored.

Algorithm 1. Computing initial values for the last-mile

Input: $\mathcal{M}_{,:}$ measurement matrix
k: number of neighbours for each node
α: percentile parameter
Output: β^{out} and β^{in}
for all node A **do**
select a random set S of k neighbours
sort $up = (\mathcal{M}_{A,y})_{y \in S}$ and $down = (\mathcal{M}_{y,A})_{y \in S}$
$\beta_A^{\text{out}} = (1-\alpha) -$ percentile of up
$\beta_A^{\text{in}} = (1-\alpha) -$ percentile of $down$
end for

With this procedure, each host can compute its own β^{out} and β^{in} values independently, assuming that it has access to the measurements of all pairs it is involved in. Furthermore, a standard technique to reduce the measurement overhead (at the cost of accuracy) is the *random sampling* of the hosts, in which each host selects a random subset of neighbours and performs available bandwidth measurements to and from this subset. By computing the $(1-\alpha)$-th percentile of these measurements to use as β^{out} and β^{in}, the result is expected to be a reasonable approximation of the real β^{out} and β^{in} values obtained if all measurements were available. This results in Algorithm 1.

In practice, many overlay networks provide the ability of choosing a random node, either by construction (*e.g.* Distributed Hash Tables) or using gossiping algorithms, so that random sampling can easily be implemented in a distributed way.

3.3 Iterative Procedure

In order to improve on this initial calculation, following Vivaldi [6] and DMF [13], we propose a procedure in which nodes iteratively update their β^{in} and β^{out} values. To update its β^{out} value, each node A obtains the values of β^{in} from its neighbours, and sets β_A^{out} to the value x that minimises the prediction error:

$$E(x) = \sum_{y \in S} \left(\mathcal{M}_{A,y} - \min(x, \beta_y^{\text{in}})\right)^2 \tag{1}$$

The value of β_A^{out} which minimises this quantity can be easily computed, since each y such that $\beta_y^{\text{in}} \leq \beta_A^{\text{out}}$ contributes a constant factor to the error. Hence for x between two consecutive values $\beta_{y_1}^{\text{in}}$ and $\beta_{y_2}^{\text{in}}$, the error can be rewritten as:

$$\sum_{\beta_y^{\text{in}} \leq \beta_{y_1}^{\text{in}}} \left(\mathcal{M}_{A,y} - \beta_y^{\text{in}}\right)^2 + \sum_{\beta_y^{\text{in}} > \beta_{y_1}^{\text{in}}} \left(\mathcal{M}_{A,y} - x\right)^2 \tag{2}$$

And this expression is minimised for x equal to the average of the $\mathcal{M}_{A,y}$ values for $\beta_y^{\text{in}} > \beta_{y_1}^{\text{in}}$. Of course, if this value is above or below the prescribed interval, x is set to the corresponding bound of the interval. By sorting the β_y^{in} values and testing all the k possible intervals, it is possible to compute the value of x that minimises equation (1). The resulting iterative procedure is described in algorithm 2.

Algorithm 2. Iterative procedure.

Input: $\mathcal{M}_{,}$: measurement matrix
k: number of neighbours for each node
α: percentile parameter
i: number of iterations
Output: β^{out} and β^{in}
Initialise β^{out} and β^{in} with Algorithm 1
for i iterations **do**
  for all node A **do**
    Sort $(\beta_y^{\text{in}})_{y \in S_A}$
    for all interval l $(\beta_{y_l}^{\text{in}} \leq \beta_{y_{l+1}}^{\text{in}})$ **do**
      Compute x_l which minimises eq (2)
    end for
    Select l so that $E(x_l)$ is smallest (eq (1))
    Update $\beta_A^{\text{out}} = x_l$
    Update β_A^{in} similarly
  end for
end for

4 Evaluation

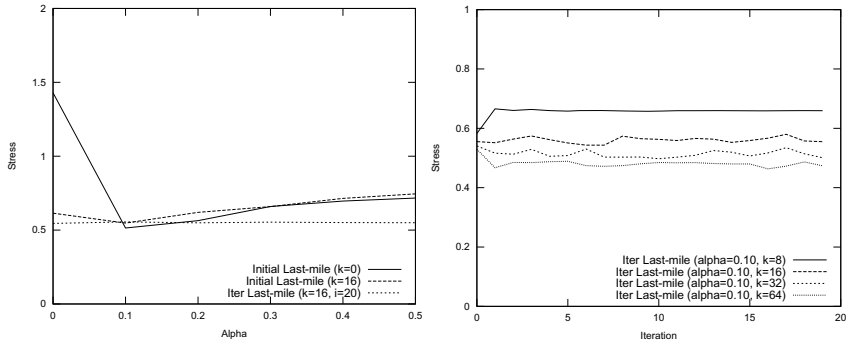
4.1 Methodology

The experimental results described in this paper are based on a dataset from the S-cube project [20]. This project aims at monitoring the large scale distributed platform PlanetLab [5]. Available bandwidth is measured between almost all pairs of nodes of PlanetLab, and results are made available as regular snapshots of the platform. For space reasons, we only present here results obtained from the snapshot of April 20th, 2010; however other snapshots yield the same conclusions. This snapshot contains 426 hosts, with some missing measurements, and we extracted a set of 308 hosts for which the complete measurement matrix is available[2].

The quality of the prediction algorithms is given by the precision of the predictions compared to the original values. In this paper, we use the *modified relative error* as defined by the authors of IDES [15]:

$$e_{x,y} = \frac{|\mathcal{M}_{x,y} - \mathcal{P}_{x,y}|}{\min(\mathcal{M}_{x,y}, \mathcal{P}_{x,y})}$$

[2] The code and dataset used to obtain the results of this section will be made publicly available upon acceptance of the paper.



(a) (b)

Fig. 2. (a) Stress of the last mile embeddings for different values of α; (b) Stress of the last mile embeddings after each iteration

where the min-operation serves to increase the penalty for underestimated values. Most plots in this paper depict the cumulative distribution function (CDF) of the modified relative error for all pairs of hosts. Therefore, if algorithm $\mathcal{A}$ provides better estimations than algorithm $\mathcal{B}$, then the plot corresponding to $\mathcal{A}$ will be above the plot corresponding to $\mathcal{B}$ in the graph.

Sometimes it is more convenient to represent the fitness of the embedding with a single value. In that case we will consider the 80-th percentile of the modified error ratios, *i.e.* the error e such that 80% of the node pairs have their available bandwidth estimated with error at most e. We also measure the stress, to represent the global error of the prediction, defined by:

$$stress = \sqrt{\frac{\sum_{x,y}(\mathcal{M}_{x,y} - \mathcal{P}_{x,y})^2}{\sum_{x,y}\mathcal{M}_{x,y}^2}}$$

4.2 Parameter Tuning

We first analyse the effect of the parameter α on the accuracy of the predictions, both for the initial values obtained with algorithm 1 and for the result of the iterative procedure after 20 iterations (our observations show 20 iterations are enough to reach convergence, see figure 2(b)). The number of neighbours k is fixed to $k = 16$, and we also compare to the special case in which there is no random neighbour selection (all measurements are used), which we denote as $k = 0$. The resulting stress values are depicted in Figure 2(a). For the cases which involve random selection, we report average, minimum and maximum values over 10 runs.

The first observation is that using a non zero value of α is very important when considering all measurements, which is expected as discussed in section 3: a small number of invalid measurements have a very bad impact on the accuracy of the predictions. With random selection ($k = 16$), the effect of the parameter α is not

as big, and it is even lower for the iterative procedure, which effectively improves the fitness of the embedding and gives a result which does not depend on this parameter. This hints that the result of the iterative procedure is independent of the initial values. In the rest of the evaluation, we will use the value $\alpha = 0.1$.

The effect of the parameter k (number of neighbours for each node) is studied in details in the next section, together with the comparison with other prediction heuristics.

We also study the convergence of the iterative procedure by measuring the stress of the fitness obtained after each iteration. The result is shown on Figure 2(b) and shows that the total stress remains stable across the iterations, and that the convergence is fast. We can also see that the initial values computed by algorithm 1 are actually quite precise.

4.3 Comparison Methods

We compare our results with several other solutions from the literature:

- The Vivaldi [6] algorithm provides a basis for comparison even though it was originally designed for latency estimation.
- The Sequoia [17] algorithm, based on tree embeddings, is advertised as being usable for both latency and bandwidth estimation.
- PathGuru [19] is a landmark-based solution explicitly designed for bandwidth estimation.
- DMF [13] is an algorithm which was proposed in the context of latency estimation, but it can be used for bandwidth estimation as well since it does not make any assumption on the structure of the input measurement matrix.

Public implementations of Vivaldi[3] and DMF[4] are available and have been used for this evaluation. However, no implementation seems to be available for PathGuru and Sequoia, so we implemented them based on their description in the corresponding paper.

This dataset is used as input to different prediction algorithms. However, Sequoia and Vivaldi are originally designed for latency prediction, for which smaller values mean that nodes are closer. Hence, these algorithms are fed with the inverse of the available bandwidth measurements[5], and their resulting distance predictions are inversed as well before comparing to the original measurements.

For all algorithms, we used the default values of the parameters as they are described in the corresponding paper (15 prediction trees for Sequoia and $l = 10$ dimensions for DMF). However, we changed the number of neighbours in DMF and landmarks in PathGuru to explore the compromise between accuracy and number of measurements used.

[3] http://www.eecs.harvard.edu/~syrah/nc/

[4] http://www.run.montefiore.ulg.ac.be/~liao/DMF

[5] This choice is different from the one made in the evaluation of Sequoia [17], in which the authors subtract the bandwidth values from a large constant. Using the inverse as we are doing actually yields better results for Sequoia.

4.4 Evaluation Results

We first analyse the variability of the results with respects to the random choices involved: choice of the levers for Sequoia, of the landmarks for PathGuru, and of the neighbours of each node for DMF and last-mile. We provide in Table 4.4 the average and standard deviation of the stress and of the 80-th percentile of the modified error ratio for 30 runs for each heuristic. For visual comparison, the CDFs of modified relative error for a selection of parameters are given on Figure 3. For a given heuristic and parameter value, the CDFs corresponding to the 30 runs are depicted together on the plot to visualise the variability.

In addition, Figure 4 provides a direct comparison of the most relevant heuristics. On this figure the CDFs of one run for each heuristic are plotted together. The low variability exhibited by table 4.4 ensures that these particular plots are relevant enough.

Table 1. Average and standard deviation of stress and 80-th percentile error

Algorithm	80-th perc. error		stress	
	avg	std	avg	std
Vivaldi ($k = 32$)	3.93	0.98	9800	$7.6\text{x}10^7$
Vivaldi ($k = 128$)	4.68	2.6	7400	$1.3\text{x}10^8$
Sequoia	1.5	0.097	0.73	0.0012
PathGuru ($k = 32$)	2.00	0.55	0.77	0.0013
PathGuru ($k = 64$)	2.58	0.33	0.78	0.00069
PathGuru ($k = 128$)	2.54	0.099	0.79	0.00081
LM ($k = 8$)	0.76	0.0027	0.64	0.00012
LM ($k = 16$)	0.64	0.0012	0.56	0.00024
LM ($k = 32$)	0.64	0.00083	0.51	0.00017
LM ($k = 64$)	0.65	0.0007	0.47	0.00016
LM ($k = 128$)	0.65	0.00019	0.42	0.000043
DMF ($k = 8$)	2.12	0.0079	3.16	2.5
DMF ($k = 16$)	1.33	0.0019	1.14	0.024
DMF ($k = 32$)	0.64	0.00025	0.51	0.00055
DMF ($k = 64$)	0.47	0.000073	0.35	0.00011
DMF ($k = 128$)	0.39	0.000043	0.26	0.000074

The results for Vivaldi show as expected that this algorithm is not appropriate for bandwidth estimation. We can also see that the prediction of last-mile and DMF (for large enough values of k) are much more accurate and stable than the predictions of PathGuru and Sequoia. PathGuru in particular is very sensitive to the choice of the landmarks, and its performance does not really increase with the number of landmarks (however it gets more stable). The predictions of Sequoia are better than those of PathGuru, but remember that Sequoia needs to access the measurements between all pairs of nodes. Sequoia is also (together with Vivaldi) the only heuristic which produces symmetric estimations, and this is a big disadvantage because available bandwidth between two nodes is often asymmetric.

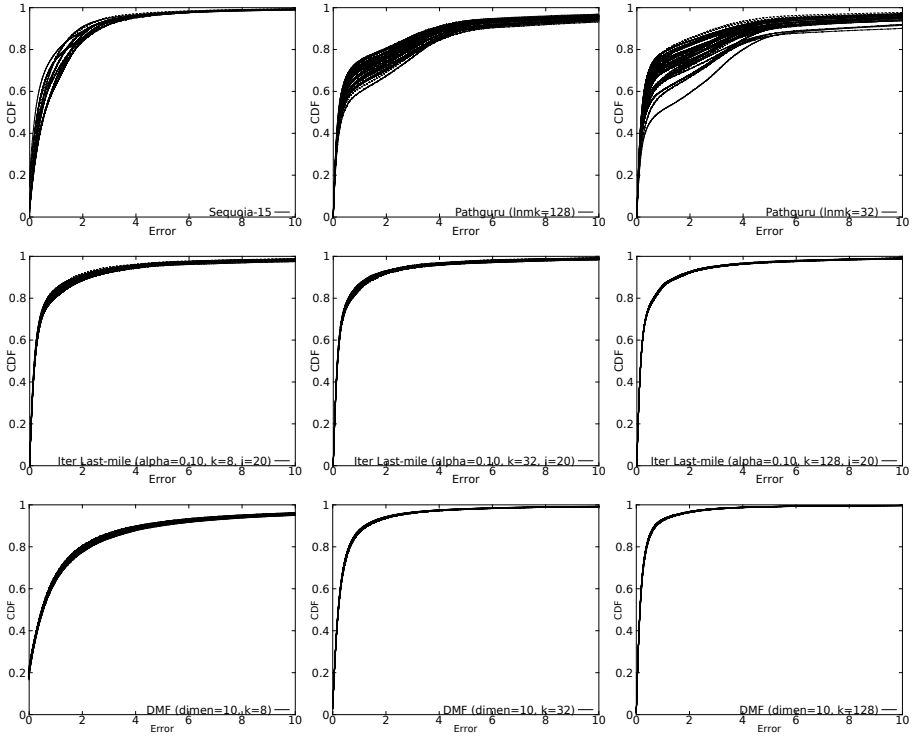
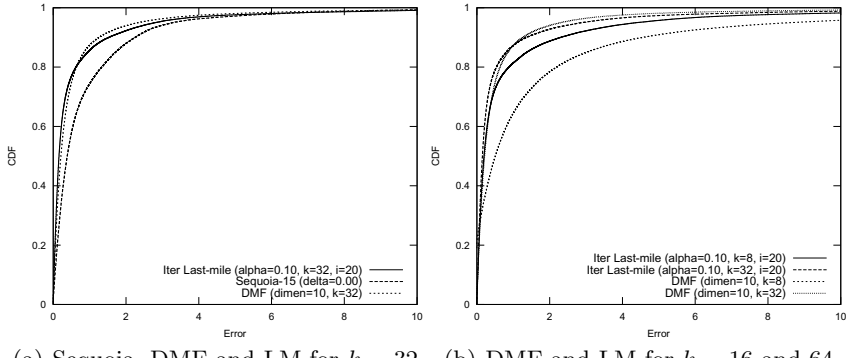


Fig. 3. CDFs of modified relative error: 30 runs of Sequoia and PathGuru; last-mile for different values of k; DMF for different values of k



(a) Sequoia, DMF and LM for $k = 32$ (b) DMF and LM for $k = 16$ and 64

Fig. 4. Direct comparison of modified relative error

We can also see that while DMF is able to make a better use of a larger number of measurements, last-mile achieves a reasonably good accuracy even for low values of k. Actually, increasing k does not increase much the accuracy of the predictions of last-mile, but it makes them more stable. In particular, last-mile with 16 neighbours per node is about as accurate as DMF with 32 neighbours per node, which is the default value proposed by the authors of DMF [13] for latency estimation. It is worth pointing out that measuring available bandwidth incurs a larger overhead than measuring latency; hence, being able to use a smaller number of measurements is an attractive feature.

These results show that the last-mile model is able to explain a large part of the structure of the available bandwidth on the Internet, with a very low number of parameters (each node is characterised by only 2 values, to be compared with 20 for DMF with 10 dimensions) and accessing a small number of measurements. The last-mile model is thus a promising approach for the prediction of available bandwidth on the Internet.

5 Concluding Remarks

Estimating the available bandwidth between nodes in a large scale distributed platform is a crucial issue in many distributed applications. On the other hand, it is impossible to rely on complete measurement sets, because of the intrinsic cost of these measurements, and because many measures may be inaccurate due to varying external conditions. Therefore, as it has been done successfully for latency estimations, several labeling schemes have been proposed, such as Sequoia and PathGuru, that enable to predict at low cost the bandwidth between any pair of hosts.

In this paper, we propose simple decentralised heuristics to use the last-mile model as a prediction mechanism for available bandwidth, by characterising each node by an incoming and an outgoing capacity. Based on real-world PlanetLab bandwidth measurements, we show that this model, although simple, achieves better prediction accuracy than the current available solutions, in particular when the number of available measurements is low. The prediction results of PathGuru depend heavily on the choice of landmarks, and Sequoia suffers from its inability to provide asymmetric predictions. When more measurements are available, decentralised matrix factorisation provides more precise predictions than our last-mile heuristic, probably because each node is described with a larger number of parameters.

In the future work, we are planning to investigate the possibility to increase the number of parameters in the last-mile model for a better accuracy, and also to make a combined use of latency and available bandwidth measurements in order to improve the predictions of the model.

References

1. Beaumont, O., Bonichon, N., Eyraud-Dubois, L.: Scheduling divisibleworkloads on heterogeneous platforms under bounded multi-port model. In: IEEE IPDPS 2008, pp. 1–7 (April 2008)
2. Beaumont, O., Eyraud-Dubois, L., Agrawal, S.K.: Broadcasting on large scale heterogeneous platforms under the bounded multi-port model. In: IEEE IPDPS 2010, pp. 1–10 (April 2010)
3. Bonald, T., Massoulié, L., Mathieu, F., Perino, D., Twigg, A.: Epidemic live streaming: optimal performance trade-offs. ACM SIGMETRICS Perform. Eval. Rev. 36, 325–336 (2008)
4. Boufkhad, Y., Mathieu, F., de Montgolfier, F., Perino, D., Viennot, L.: An upload bandwidth threshold for peer-to-peer video-on-demand scalability. In: IEEE IPDPS 2009, pp. 1–10 (May 2009)
5. Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., Bowman, M.: Planetlab: an overlay testbed for broad-coverage services. ACM SIGCOMM Comput. Commun. Rev. 33, 3–12 (2003)
6. Dabek, F., Cox, R., Kaashoek, F., Morris, R.: Vivaldi: a decentralized network coordinate system. In: ACM SIGCOMM 2004, Portland, OR, USA, pp. 15–26 (September 2004)
7. Dinitz, M.: Online, dynamic, and distributed embeddings of approximate ultrametrics. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 152–166. Springer, Heidelberg (2008)
8. Dischinger, M., Haeberlen, A., Gummadi, K.P., Saroiu, S.: Characterizing residential broadband networks. In: IMC 2007, San Diego, CA, USA, pp. 43–56 (October 2007)
9. Goldoni, E., Schivi, M.: End-to-end available bandwidth estimation tools, an experimental comparison. In: Ricciato, F., Mellia, M., Biersack, E. (eds.) TMA 2010. LNCS, vol. 6003, pp. 171–182. Springer, Heidelberg (2010)
10. Hu, N., Steenkiste, P.: Exploiting internet route sharing for large scale available bandwidth estimation. In: IMC 2005, pp. 16–16 (October 2005)
11. Ledlie, J., Gardner, P., Seltzer, M.: Network coordinates in the wild. In: USENIX NSDI 2007, pp. 299–311 (April 2007)
12. Li, N.H., Li, L.E., Mao, Z.M., Steenkiste, P., Wang, J.: A measurement study of internet bottlenecks. In: IEEE INFOCOM 2005 (March 2005)
13. Liao, Y., Geurts, P., Leduc, G.: Network distance prediction based on decentralized matrix factorization. In: IFIP NETWORKING 2010, pp. 15–26 (May 2010)
14. Liu, S., Zhang-Shen, R., Jiang, W., Rexford, J., Chiang, M.: Performance bounds for peer-assisted live streaming. ACM SIGMETRICS Perform. Eval. Rev. 36, 313–324 (2008)
15. Mao, L.K.Y., Saul, Smith, J.M.: Ides: An internet distance estimation service for large networks. IEEE JSAC 24(12), 2273 (2006)
16. E., T.S., Ng, H.Z.: Predicting internet network distance with coordinates-based approaches. In: IEEE INFOCOM 2002, pp. 170–179 (June 2002)
17. Ramasubramanian, V., Malkhi, D., Kuhn, F., Balakrishnan, M., Gupta, A., Akella, A.: On the treeness of internet latency and bandwidth. In: ACM SIGMETRICS 2009, Seattle, WA, USA, pp. 61–72 (June 2009)

18. Suh, K., Diot, C., Kurose, J., Massoulie, L., Neumann, C., Towsley, D., Varvello, M.: Push-to-peer video-on-demand system: Design and evaluation. IEEE JSAC 25(9), 1706–1716 (2007)
19. Xing, C., Chen, M., Yan, L.: Predicting available bandwidth of internet path with ultra metric space-based approaches. In: IEEE GLOBECOM 2009 (December2009)
20. Yalagandula, P., Sharma, P., Banerjee, S., Basu, S., Lee, S.-J.: S3: a scalable sensing service for monitoring large networked systems. In: ACM SIGCOMM Workshop on Internet Network Management, Pisa, Italy, pp. 71–76 (September 2006)

Self-stabilization versus Robust Self-stabilization for Clustering in Ad-Hoc Network

Colette Johnen[1] and Fouzi Mekhaldi[2]

[1] LaBRI, Univ. Bordeaux, CNRS. F-33405 Talence Cedex, France
[2] LRI, Univ. Paris-Sud XI, CNRS. F-91405 Orsay Cedex, France

Abstract. In this paper, we compare the two fault tolerant approaches: self-stabilization and robust self-stabilization, and we investigate their performances in dynamic networks. We study the behavior of four clustering protocols; two self-stabilizing `GDMAC` and `BSC`, and their robust self-stabilizing version `R-GDMAC` and `R-BSC`. The performances of protocols are compared in terms of their cluster-heads number, availability of both minimal and optimum services and the stabilization time.

Keywords: Ad-hoc networks, clustering, self-Stabilization, Robust self-stabilization.

1 Introduction

A mobile ad-hoc network is a multi-hop wireless communication network, supporting mobile users, realised without any existing infrastructure. In a flat architecture of ad-hoc network, all nodes are considered equal and they take the same part in the network management, like routing and forwarding tasks. To achieve the routing in flat architecture, each node maintains a routing table with entries for all nodes in the network. Moreover, owing to the lack of infrastructure, each node must relay data packets of all its neighbors. Hence, flat routing protocols in ad-hoc networks are not scalable, due to the communication cost, size of routing tables and, energy consumption. Therefore, clustering was introduced in ad-hoc networks to improve the scalability by allowing hierarchical routing.

Clustering is a hierarchical network organization which consists in partitioning the network into clusters, such that nodes within a closed proximity form a cluster. Each cluster is composed of a single cluster-head and some ordinary nodes. As nodes are mobile, the clustering protocol must maintain the clustering structure in spite of topological changes like nodes arrival/departure and links creation/failure.

Self-stabilization and Robust self-stabilization. One of the most wanted properties of distributed systems is the fault tolerance and adaptivity to topological changes, which consist of the system's ability to react to faults and perturbations in a well-defined manner. Self-stabilization is an approach to design fault-tolerant and adaptive to topological changes distributed systems.

A self-stabilizing protocol, regardless of its initial state, converges in finite time (called stabilization period) to a legitimate state where the intended behavior

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 117–129, 2011.

is exhibited, without any external intervention. Thus, self-stabilizing protocols are attractive because they do not require any correct initialization (as any state can be the initial one), they can recover from any transient failure, and they are adaptive to dynamic topology reconfigurations. Whatever is the current configuration, the system converges to a legitimate configuration according to the current network topology.

Despite such advantages, self-stabilization has a major drawback. During all stabilization periods, a self-stabilizing protocol does not guarantee any property. Thus, self-stabilization is suited for distributed systems with intermittent disruptions, where the delay between two successive disruptions is so large that the system can reach a legitimate state and provide the full (optimum) service for some time. Whereas in large scale mobile ad-hoc networks where the network topology changes very often, the paradigm of self-stabilization is no more satisfying. Indeed, as the delay between two successive disruptions is very small, the system is continuously disrupted and it may never provide its optimum service. As a consequence, the availability and reliability of self-stabilizing systems is compromised. To overcome these drawbacks, the robust self-stabilization approach has been developed [1,2].

A protocol is robust self-stabilizing if (1) it is self-stabilizing; (2) it quickly reaches a safe configuration where a minimal useful service is provided; (3) the minimal useful service holds during progress of the protocol toward the optimum service (i.e., during convergence to a legitimate configuration); (4) and it is also maintained despite multiple occurrences of some specific disruptions, called *highly tolerated disruptions*. Whatever the occurrence of highly tolerated disruptions, the useful minimal service still provided. Whereas the occurrence of other disruptions is handled by the self-stabilization mechanism, i.e., after their occurrence, the system may behave arbitrarily, but it will quickly provide the minimal useful service. Therefore, the robustness as defined in [1,2] may be seen as a service guarantee, which is provided by both: the fast recovering to a desired system characteristic (minimal useful service), and its preservation in spite of highly tolerated disruptions.

Contribution. Self-stabilizing protocols are almost evaluated only in terms of worst-case time and space complexities. In this context, theoretical studies of the robust self-stabilization approach has been done in [1,2]. However, clustering protocols presented in these two papers are written in the shared memory state model (a non realistic model). Furthermore, no experimental study has been made to compare the two approaches (i.e., self-stabilization and robust self-stabilization), and to investigate their performances in dynamic networks. In this paper, we compare the two approaches, through an experimental study. We study the behavior of four clustering protocols; two self-stabilizing protocols `GDMAC` [3] (Generalized Distributed Mobility-Adaptive Clustering) and `BSC` [4] (Bounded Size Clustering), and their robust self-stabilizing version `R-GDMAC` [1] and `R-BSC` [2]. The performances of protocols are compared in terms of their number of cluster-heads, availability of minimal and optimum services, and the stabilization time.

For our study, we use the *standard* simulation environment in the research community: Network Simulator 2 (NS2) [5]. Obviously to achieve this study, protocols were adapted to the message passing model.

Related Works. The problem of clustering is well studied in the literature, and several clustering protocols have been proposed in the context of multi-hop wireless networks. A large number of them are self-stabilizing [6,7,8,9,10,11,12,13]. However, only [1,2] are robust self-stabilizing. A survey on clustering protocols can be found in [14].

GDMAC protocol is evaluated in [15] with respect to its convergence time and message complexity. Whereas, according to our knowledge, the three other protocols (R-GDMAC, BSC and R-BSC) have never been evaluated.

The remainder of the paper is organized as follows. In Section 2, an overview of the studied clustering protocols is given. In Section 3, the simulation model and some important remarks are discussed. The observed metrics are described in section 4, and the performance evaluation results with the analysis remarks are detailed in Section 5. Finally, we conclude our study in Section 6.

2 Overview of the Studied Clustering Protocols

A clustering protocol consists of partitioning the network into non-overlapping groups of nodes called clusters. Each cluster has a single head (called cluster-head), and eventually a set of ordinary nodes. Each cluster-head acts as local coordinator of its cluster, and may participate to the management of the global network. So, cluster-heads have more tasks to perform than ordinary nodes. As consequence, cluster-heads must be more suitable than ordinary nodes.

Protocols GDMAC, R-GDMAC, BSC and R-BSC consider weight-based networks, i.e., a weight W_v is assigned to each node v of the network. In ad-hoc or sensor networks, amount of bandwidth, memory space, processing capacity or battery power of a node could be used to determine weight values. The choice of cluster-heads is based on the weight associated to each node: the higher the weight of a node is, the better this node is appropriate for the role of cluster-head.

The studied protocols build 1-hop clusters, where the ordinary nodes are neighbor of their cluster-head, i.e., they can directly communicate with it. Note that both GDMAC and R-GDMAC (resp. BSC and R-BSC) provide the same final clustering structure.

• GDMAC [3] is a self-stabilizing protocol building clusters having the following *ad-hoc clustering properties*:

- The cluster-head has the highest weight in its cluster.
- A cluster-head cannot have more than k neighbor cluster-heads.
- For every ordinary node v, there is no a v's neighbor cluster-head Y such that $W_Y > W_X + h$ where X is the current cluster-head of v. Otherwise, v changes the affiliation and it chooses Y as new cluster-head.

k and h are protocol parameters, and their value may be different from a node to another one. The parameter k allows to bound the number of cluster-heads

that can be neighbors. Whereas h is used to reduce the switching overhead of an ordinary node (i.e., the number of moves from its current cluster to a new neighbor one due to cluster-head's weight change).

• `R-GDMAC` [1] is a robust self-stabilizing version of the `GDMAC` protocol.

• `BSC` [4] is a self-stabilizing protocol building bounded size clusters. The built clusters respect the following *well balanced clustering properties*:

- The cluster-head has the highest weight in its cluster.
- The leader of a cluster is not overburden by the management workload of its cluster. Thus, a cluster can have at most $SizeBound$ ordinary nodes ($SizeBound$ is a parameter of the protocol).
- A node stays cluster-head only if it cannot join a neighbor cluster: all neighbor clusters are full. This property limits the number of cluster-heads locally. Therefore, if a leader v has a neighbor leader u such that $W_v > W_u$, then the cluster of v is full, i.e., it contains exactly $SizeBound$ members.

• `R-BSC` [2] is a robust self-stabilizing version of `BSC` protocol.

The main idea of `GDMAC` and `R-GDMAC` protocols is that an ordinary node v becomes cluster-head if in its neighborhood there is not a cluster-head having a weight greater than v's weight.

Each cluster-head should have less than k neighbor cluster-heads. Hence, if several (more than k) cluster-heads become neighbor, at least a cluster-head has to resign its status. To implement this property, each cluster-head v checks the number of its neighbors that are cluster-heads. If they exceed k, then it determines the $(k+1)^{th}$ highest weight among neighbor cluster-heads. All v's neighbor cluster-heads having a weight less than this value have to become ordinary.

Similarly, in `BSC` and `R-BSC` protocols, a node v becomes cluster-head if there is not cluster-head in v's neighborhood. Furthermore, a cluster-head v stays in this status only when it cannot join one of its neighbor clusters without violating the *well-balanced* clustering properties.

`BSC` and `R-BSC` build bounded size clusters. So, in order to prevent the violation of the size condition, a node u cannot freely join a cluster: u needs the permission of its potential new cluster-head. Therefore, each cluster-head v maintains a list of nodes who are authorized to join its cluster.

Robustness property. The robustness in `R-GDMAC` and `R-BSC` ensures that a minimal useful service is quickly provided. Once the minimal service is available, each node belongs to a cluster, and each cluster has a cluster-head. Furthermore, for `R-BSC` protocol, no cluster should have more than $SizeBound$ ordinary nodes. Preserving the minimal useful service ensures that the hierarchical structure is continuously provided throughout the network even during the its reorganization. In order to maintain the hierarchical structure over the network during reconstruction of clusters, `R-GDMAC` and `R-BSC` protocols use the following resignation process.

Resignation process. A cluster-head v that wants to become ordinary, does not take the ordinary status: v becomes a nearly ordinary node (i.e., it takes the

nearly ordinary status). In this state, v performs correctly its task of cluster-head, but no node can join v's cluster. The members of v's cluster has to quit their cluster. Moreover, v can become ordinary only once its cluster is empty. These conditions guarantee that during construction/maintenance of clusters, no cluster-head abandons its leadership.

The robustness property in clustering protocols is very useful. It ensure a high availability of hierarchical organization; and it allows the continuity functioning of upper-layer hierarchical protocols, as hierarchical routing protocols.

The set of highly tolerated disruptions handled by robust protocols are:

- the change of node's weight,
- the crash of ordinary nodes,
- the creation of new communication links without the emergence of new nodes,
- the failure of communication links between (1) two ordinary nodes, (2) two nodes behaving as cluster-heads (i.e., cluster-head or nearly ordinary node)
- the emergence of networks correctly partitionned (i.e., where the minimal service is already provided).

3 Model and Simulation Remarks

The simulation experiments are carried out thanks to the `NS2.34` simulator [5]. Our network is composed of mobile nodes with a propagation radio range of 250m randomly placed within a 1200m*1200m area. The density of a node (i.e., the number of neighbors per node) is at most 15. The parameters value used during simulation are presented in Figure 1.

Mobility model. Each node moves randomly according to the *Random Waypoint model* [16]. Initially, network nodes are randomly placed in the network area. At the beginning of the simulation, each node selects a random destination and moves toward it with a randomly chosen speed (uniformly distributed between 0 and `Speed` m/s). Upon reaching this destination, another random speed and destination are targeted after a pause time. The process is repeated until the simulation ends.

Weight variation model. The four studied protocols assume that each node has a weight, that can change during time.

Parameter	Value
Simulation time	100s
Number of nodes	70
Transmission range	250m
Network area	1200m*1200m
Density	15
Speed	0m/s - 12m/s
Pause time	0.5s
Wmin	50
Wmax	80
Δ	2
SizeBound	10
k	2
h	3
freq	2

Fig. 1. Parameters value

Initialy, each node randomly chooses its weight w between two values $Wmin$ and $Wmax$. The weight of a node changes according to a frequency $freq$, which is the number of changes per second. For example, if $freq = 0.2c/s$, then the node's weight changes once every 5 seconds. According to the frequency value, the time when a node undergoes the weight change is chosen randomly. The new weight of a node is chosen randomly between $W - \Delta$ and $W + \Delta$.

In order to study the influence of network size, mobility of nodes, and node's weight variation, 3 different types of simulations have been conducted:

1. *Network size variation*: nodes are not mobile, and their number varies between 10 and 70. The frequency of weight variation is set to $2c/s$.
2. *Weight variation*: in order to see how protocols behave when reconstruction of clusters is high, we increase the frequency of the weight variation in a static network of 70 nodes. The values of frequency considered are: 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 1, 2, 3, 4 and 5.
3. *Mobility variation*: the speed of nodes is varied between $0m/s$ and $12m/s$ to see how protocols behave in presence of mobile nodes. The network contains 70 nodes and the weight changes twice per second.

For all protocols, identical mobility and weight variation scenarios are used in order to gather fair results. Furthermore, to get accurate results, each simulation is driven with ten different runs. The presented metrics are then averaged on these different runs. Furthermore, in order to show how these average values are confident, a confidence interval is computed using the confidence level 95%.

4 Observed Metrics

To analyze the performance of clustering protocols and to compare robust self-stabilization with self-stabilization, the following metrics are studied:

• *The average number of cluster-heads*: as small as it is the number of cluster-heads, the protocol is far from being trivial; because in a trivial solution, all nodes are cluster-head. Thus, one goal of clustering protocols is to provide a hierarchical structure with a small number of cluster-heads.

• *The availability of minimum service*: it represents the percentage of time where the minimum service is available. A configuration where the minimum service is available is a configuration where the hierarchical structure is provided. More specifically, it is a configuration where:

- Each ordinary node belongs to a cluster.
- Each ordinary node is a neighbor of its leader (within its transmission range).
- Moreover, for protocols `BSC` and `R-BSC`, each cluster must have at most $SizeBound$ members.

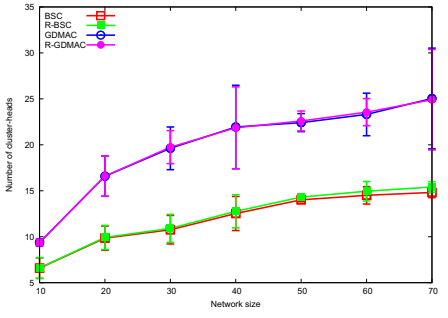
• *The availability of optimum service*: it represents the percentage of time where the optimum service is available. The optimum service is available in a configuration (called legitimate) if the built clusters verify the *ad-hoc* or *well balanced clustering properties* defined in Section 2.

As these metrics vary over time according to weight change and nodes mobility, measurements are collected every 0.02 seconds to obtain the average values.

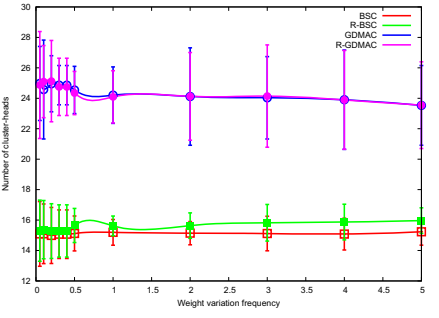
5 Simulation Results and Performances Analysis

5.1 Average Number of Cluster-Heads

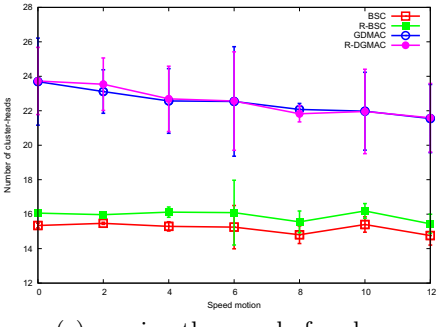
The variation of cluster-heads number in function of the network size, weight variation frequency and nodes speed are presented respectively in Figures 2(a), 2(b) and 2(c).



(a) varying the network size



(b) varying the frequency of weight changes



(c) varying the speed of nodes

Fig. 2. Average number of cluster-heads

Protocols `GDMAC` and `R-GDMAC` (resp. `BSC` and `R-BSC`) have the same behavior, because they use the same cluster-heads selection policy: weight based criteria.

For clustering protocols based on dominating sets, the number of cluster-heads is intrinsically related to the variant of dominating set computed. In fact, we distinguish a classification in two groups. Protocols `BSC` and `R-BSC` generate a smaller number of cluster-heads than protocols `GDMAC` and `R-GDMAC`. The structure used by `BSC` and `R-BSC` is the capacited dominating sets, where a cluster-head can have a neighbor cluster-head only if its cluster is full. So, two cluster-heads are

rarely neighbor. While, the structure used by GDMAC and R-GDMAC is a k-fold dominating set where at most $k+1$ cluster-heads can be neighbor. If due to the mobility or weight change, $k+1$ cluster-heads become neighbor, no one needs resign its status. This feature leads to a higher total number of cluster-heads.

In a large scale network, a robust self-stabilizing clustering protocol generates a slightly higher number of cluster-heads than its self-stabilizing version (see BSC and R-BSC). Recall that during resignation process, robust self-stabilizing protocols use an intermediate hierarchical status, called nearly ordinary. A cluster-head wanting to resign, it takes the nearly ordinary status. A nearly ordinary node may become ordinary only once its cluster is empty; and during all this period it behaves and it is considered as a cluster-head. This is why the average number of cluster-heads is higher in a robust self-stabilizing protocol compared to its self-stabilizing version. However, the difference in cluster-heads number depends on the resignation overhead: how many cluster-heads resign their status to be ordinary?

In BSC and R-BSC protocols, as soon as two cluster-heads become neighbors, one of them must resign expect if one of clusters is full. Whereas in GDMAC and R-GDMAC protocols, if the number of neighbor cluster-heads does not exceed $k+1$, no resignation is required. As the resignation process is more frequent in protocols BSC and R-BSC than GDMAC and R-GDMAC. Thus, the difference in cluster-heads number between R-BSC and BSC is significant, but not between GDMAC and R-GDMAC.

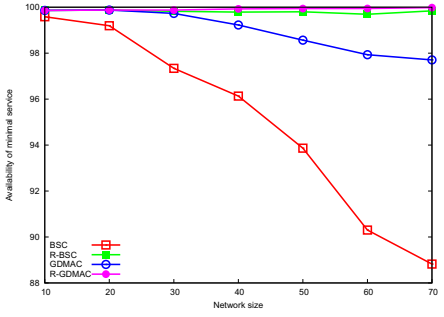
5.2 Availability of Minimum Service

The availability of minimal service in a static network according to the network size and the frequency of weight variation are illustrated respectively in Figures 3(a) and 3(b).

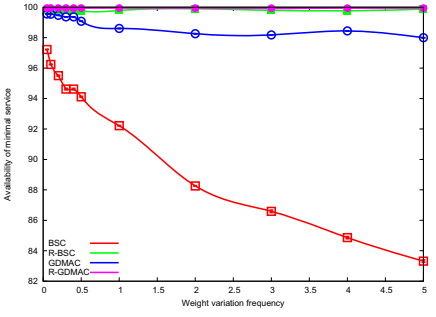
Robust self-stabilizing protocols R-GDMAC and R-BSC scale well to large networks, and they are more resistant to weight change. In fact, R-GDMAC and R-BSC maintain the minimal service, once provided, during almost all their execution time whatever the network size and the weight variation frequency.

This is not the case for self-stabilizing protocols. In GDMAC protocol, the minimum service is broken by increasing the network size and the weight variation frequency. Nevertheless, the rupture rate of the minimal service stays very small. Indeed, in a network of 70 nodes where the weight changes twice per second, the rupture rate is at most 3% (thus, 97% of time, the minimal service is available).

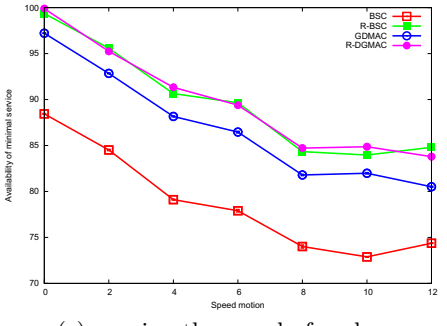
Whereas, BSC protocol has less guarantee of service than other protocols in a large scale network. Indeed, in a network of 70 nodes, the minimal service is unavailable during 12% of time. BSC is also the protocol which really suffers the most from unavailability of minimal service when the frequency of weight variation increases. Indeed, by changing the weight five time per second, the minimal service is unavailable almost 20% of time.



(a) varying the network size



(b) varying the frequency of weight changes



(c) varying the speed of nodes

Fig. 3. Availability of minimal service

The poor performance of BSC protocol reflects the interest of the robustness property in large scale networks, because R-BSC protocol maintains the minimal service without any rupture whatever the network size and the weight variation frequency.

Robust self-stabilizing protocols prevent the violation of the minimal service, by using the resignation process discussed in Section 2. This mechanism guarantees that during construction/maintenance of clusters, no clusterhead abandons its leadership, so the minimal service is continuously provided.

The rupture of minimal service in self-stabilizing protocols happens during reconstruction of clusters due to weight change. Nevertheless, it is more frequent in BSC than in GDMAC. In BSC, when two cluster-heads become neighbors, in most cases one of them must defer to the other. This feature can trigger cluster-head election/resignation that may propagate throughout the network, and generates a continuous disruption of minimal service. Such an effect is called *chain reaction*. In GDMAC, this chain reaction effect is minimized, and the minimal service is not

dramatically damaged. Furthermore, using the robust self-stabilization property improves the availability of minimal service even in the presence of chain reaction (R-BSC).

Robust protocols (R-GDMAC and R-BSC) are expected (theoretically proved in the sharded memory model) to guarantee the minimum service whatever the network size and frequency of weight change. The rupture observed (less than 0.3%) in large scale network or when the frequency is very high, is due to the following. In these protocols, when a node undergoes a change weight, it broadcasts a message to its neighbors indicating its new state (so, its new weight). When the weight variation is very high, the number of exchanged messages is important. So, the message loss and the unordered message reception happen more frequently. Owing to these disruptions, an ordinary node can affiliate with another node (by considering it as cluster-head), but which is not a cluster-head anymore (it already resigned).

Increasing the speed of nodes has a negative impact on the availability of minimal service (see Figure 3(c)). In a dynamic network, due to nodes motion, an ordinary node and its cluster-head may be outside the transmission range of each other, i.e., they are no longer neighbors. This situation breaks the minimal service. However, even in a dynamic network, the minimal service is preserved by robust self-stabilizing protocols better than self-stabilizing ones.

5.3 Availability of Optimum Service

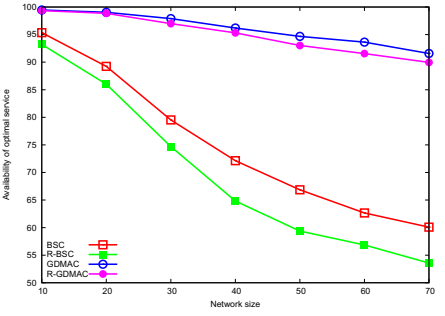
The availability of optimum service as a function of the network size, weight change frequency and nodes speed are presented in Figures 4(a), 4(b) and 4(c).

By increasing the network size, the weight variation frequency or the speed of nodes, the optimum service is less available in BSC and R-BSC protocols than GDMAC and R-GDMAC protocols.

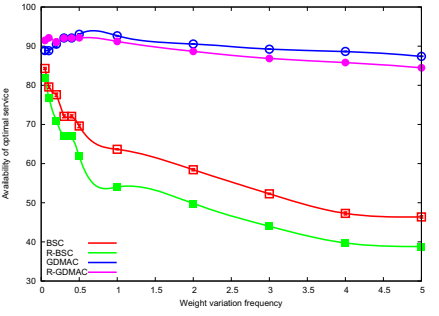
In BSC and R-BSC protocols, due to weight changes or nodes mobility, the hierarchical structure is continuously reconstructed in order to achieve the *well-balanced clustering properties*. As a result, the optimum service is often broken, so not highly available.

On the other hand, in GDMAC and R-GDMAC protocols, once the hierarchical structure respecting the *ad-hoc clustering properties* is built, it will rarely be modified due to the weight change. Furthermore, the mobility of nodes (especially of cluster-heads) rarely generates selection or resignation of cluster-heads. So, the optimum service is not affected.

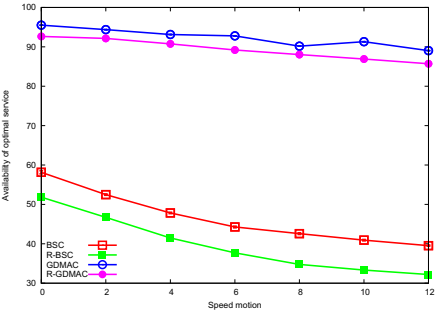
The optimum service is slightly less available in the case of robust self-stabilizing protocols compared to their self-stabilizing versions. This is caused by the convergence (i.e., stabilization) time towards the optimum service. In fact, Figure 5 shows that the time required by a robust self-stabilizing protocol to reach the optimum service is larger than the one required by its self-stabilizing version.



(a) varying the network size



(b) varying the frequency of weight changes



(c) varying the speed of nodes

Fig. 4. Availability of optimum service

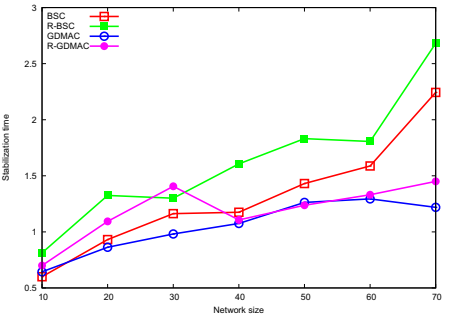


Fig. 5. Stabilization time in function of the network size

6 Concluding Remarks

This article presents the first experimental study results comparing robust self-stabilization approach with self-stabilization for the clustering problem. Thanks to this study, we extract the following remarks.

The property of robustness within clustering protocols induces an increase in the average number of cluster-heads, however really negligible.

Since the robustness property consists to slow-down the convergence process, in order to maintain the minimum service. This property leads to a slight increase in the stabilization time. However, this growth in the stabilization time depends on the size of the network, but not on the nodes speed nor the frequency of weight change.

The availability of optimum service is lower in a robust self-stabilizing protocol than its self-stabilizing version. Nevertheless, the minimum service is highly available in robust self-stabilizing protocols than self-stabilizing ones. As consequence, thanks to the robustness, when the optimum service is not provided, the minimum service is available and it will be preserved. Once the minimum service is provided, the network is completely partitionned, and each cluster has an effectual leader.

The minimum service is sufficient for the continuity of operation of upper-layer hierarchical protocols, as hierarchical routing protocols, since the hierarchical organization is available throughout the network. Therefore, robust self-stabilizing protocols are desirable, because they avoid disrupting upper-layer hierarchical protocols by maintaining the minimal service.

References

1. Johnen, C., Nguyen, L.H.: Robust self-stabilizing weight-based clustering algorithm. Theoretical Computer Science 410(6-7), 581–594 (2009)
2. Johnen, C., Mekhaldi, F.: Robust self-stabilizing construction of bounded size weight-based clusters. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6271, pp. 535–546. Springer, Heidelberg (2010)
3. Basagni, S.: Distributed and mobility-adaptive clustering for multimedia support in multi-hop wireless networks. In: VTC 1999, pp. 889–893 (1999)
4. Johnen, C., Nguyen, L.H.: Self-stabilizing construction of bounded size clusters. In: ISPA 2008, pp. 43–50 (2008)
5. The Network Simulator NS-2, http://www.isi.edu/nsnam/ns/
6. Bein, D., Datta, A.K., Jagganagari, C.R., Villain, V.: A self-stabilizing link-cluster algorithm in mobile ad hoc networks. In: ISPAN 2005, pp. 436–441 (2005)
7. Lin, C.R., Gerla, M.: Adaptive clustering for mobile wireless networks. IEEE Journal on Selected Areas in Communications 15, 1265–1275 (1997)
8. Chatterjee, M., Das, S.K., Turgut, D.: WCA: A weighted clustering algorithm for mobile ad hoc networks. Journal of Cluster Computing 5(2), 193–204 (2002)
9. Demirbas, M., Arora, A., Mittal, V., Kulathumani, V.: A fault-local self-stabilizing clustering service for wireless ad hoc networks. IEEE Transactions on Parallel and Distributed Systems 17, 912–922 (2006)

10. Drabkin, V., Friedman, R., Gradinariu, M.: Self-stabilizing wireless connected overlays. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 425–439. Springer, Heidelberg (2006)
11. Datta, A., Devismes, S., Larmore, L.: A self-stabilizing $o(n)$-round k-clustering algorithm. In: SRDS 2009 (2009)
12. Dolev, S., Tzachar, N.: Empire of colonies: Self-stabilizing and self-organizing distributed algorithm. Theoretical Computer Science 410, 514–532 (2009)
13. Mitton, N., Fleury, E., Guérin-Lassous, I., Tixeuil, S.: Self-stabilization in self-organized multihop wireless networks. In: WWAN 2005, pp. 909–915 (2005)
14. Abbasi, A.A., Younis, M.: A survey on clustering algorithms for wireless sensor networks. Computer Communications 30, 2826–2841 (2007)
15. Bettstetter, C., Friedrich, B.: Time and message complexities of the generalized distributed mobility-adaptive clustering (GDMAC) algorithm in wireless multihop networks. In: VTC 2003-Spring, pp. 176–180. IEEE, Los Alamitos (2003)
16. Camp, T., Boleng, J., Davies, V.: A survey of mobility models for ad hoc network research. Wireless communications & Mobile computing (WCMC): Special issue on mobile ad hoc networking: Research, trends and applications 2, 483–502 (2002)

Multilayer Cache Partitioning for Multiprogram Workloads*

Mahmut Kandemir[1], Ramya Prabhakar[1], Mustafa Karakoy[2], and Yuanrui Zhang[1]

[1] Pennsylvania State University, USA
[2] Imperial College, UK

Abstract. We present a fully-automated, model based, multilayer cache partitioning scheme for multiprogram workloads running on multicore machines. As opposed to prior efforts, this scheme partitions shared caches at multiple layers simultaneously in a coordinated fashion. This scheme tries to achieve two objectives. First, it tries to satisfy the specified quality of service (QoS) values for all applications by partitioning the shared cache hierarchy across them, and second, it distributes the remaining excess cache capacity (if any) across applications such that a global performance metric is maximized. Our experimental analysis shows that the proposed multilayer partitioning scheme generates, on average, 33.1% improvement (on the weighted speedup metric) over the next best-performing scheme and is very successful in satisfying the QoS requirements of applications. Also, we show that partitioning each layer in isolation cannot generate the benefits obtained through our coordinated partitioning scheme. In addition, we observed that the difference between our scheme and an optimal scheme (that derives best dynamic partitions) was less than 15% for all the workloads tested and 6.6% on average.

1 Introduction

To enable efficient and productive use of emerging multicore systems, there is an urgent need to design and implement robust on-chip memory systems. Current commercial multicore architectures employ *multilayer* on-chip cache hierarchies. As an example, Figure 1(a) shows a two-socket Intel Dunnington multicore architecture [1], which has three layers of on-chip caches. In each socket, while L1 caches are private and L3 is shared by all cores (in the socket), each L2 cache is shared by two cores. It is expected that in future on-chip cache hierarchies will be deeper [12] (see Figure 1(b) for a four-layer on-chip cache hierarchy) and probably contain a higher number of partially-shared caches (like L2 caches in Figure 1(a)).

While performance optimization is certainly important, another pressing issue for multicores is *quality of service* (QoS). In particular, when multiple applications use the same multicore architecture at the same time and share the same

* This research is supported in part by NSF grants 1017882, 0963839, CNS 0720645, CCF 0811687, CCF 0702519 and a grant from Microsoft Corporation.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 130–141, 2011.

set of on-chip resources, providing QoS to these applications (in addition to high performance) is becoming an increasingly important problem [6], [10], [11]. To guarantee QoS for applications, efficient and effective management of shared resources is critical. In the context of shared on-chip caches, capacity allocation and control has been a promising strategy for shared space management. This work makes three key contributions:

- We illustrate that coordinated multilayer cache partitioning is critical for extracting the maximum performance from on-chip caches of emerging multicores. Specifically, partitioning L2 alone or L3 alone may not be sufficient to satisfy specified QoS requirements for many workloads.
- Our multilayer cache partitioning scheme targets multiprogrammed workloads and tries to achieve two main objectives: (i) Satisfy specified QoS values for all applications, and ii) Distribute the remaining excess cache capacity (if any) across applications such that a global performance metric (weighted speedup [2] in most of our experiments) is maximized.
- We analyze the behavior of the proposed partitioning scheme using a large set of workloads and different multicore architectures. In our experimental evaluation, we also compare our approach against several alternate schemes as well as an optimal scheme which is not implementable in practice but guarantees the best workload performance.

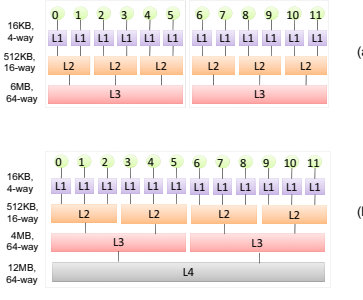


Fig. 1. Two different architectures with multilayer cache hierarchies

Several previous studies have investigated guaranteeing a certain level of performance in multicores by employing Quality-of-Service [11], [6], [2], [7], [17]. Iyer et al [11] proposed a memory hierarchy that allocates more cache and memory resources to higher priority jobs. Herdrich et al [4] proposed a rate-based technique to manage global power and also performance (or QoS) at the same time. Ko et al [5] proposed a scheme to allocate excess resources fairly by employing feedback controllers per-class. Qureshi and Patt [14] employed a utility model with cost-effective hardware support in partitioning of a shared cache. Guo et al [6] showed that providing strict QoS often causes resource fragmentation that reduces throughput significantly. Shekhar et al [17] employed formal control theory for dynamically partitioning the shared last level cache in multicores while providing QoS. These prior multicore works considered partitioning of single layer (usually the last layer of caches shared by all cores). This paper, in contrast, considers partitioning multiple layers of an on-chip cache hierarchy in a *coordinated fashion* across competing applications using a dynamic performance model.

Our experimental analysis shows that the proposed multi-level partitioning scheme outperforms all alternate schemes tested (except for the optimal scheme) for all workloads we have. Specifically, it generates, on average, 33.1% improvement (on the weighted speedup metric) over the next best-performing scheme

and is very successful in satisfying the QoS requirements of applications. In addition, we observed that the difference between our scheme and the optimal scheme is less than 15% for all workloads tested and 6.6% on average.

2 Motivational Example for Multilayer Partitioning

In this section, we present an example that motivates for multilayer cache partitioning. The multicore architecture considered in this example is illustrated in Figure 1(a), which is the same as Intel Dunnington (except for cache sizes). We use only a single socket of this architecture and run a workload consisting of six (single threaded) applications. There is a one-to-one mapping between applications and cores, i.e., each application is mapped to a core and executed there until it finishes. The QoS values (average data access latency values in this experiment) for all applications are set to 7 cycles. Our focus is on L2 and L3 caches as these are the ones shared by multiple cores. We perform experiments with six different schemes: *Default*, *Equal*, *L2 Only* and *L3 Only* are described in Section 5.1. The scheme referred to as *Ideal,* uses the best (possibly nonuniform) partitioning for each cache component, selected from a large set of possible partitioning through *exhaustive search.* The last scheme, called *Isolated,* runs each application stand-alone in the multicore architecture. Therefore, an application does not experience any contention under this scheme.

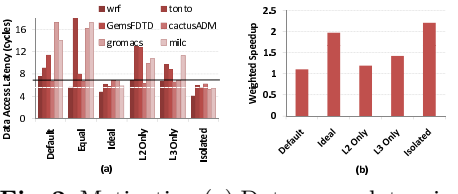


Fig. 2. Motivation (a) Data access latencies of the applications in a workload under different cache partitioning schemes. (b) Resulting weighted speedup values. Details of our experimental setup will be given later.

The results of our experiments with these six schemes are presented in Figure 2. The y-axis in (a) represents the average data access latency for individual applications in our workload, whereas that in (b) captures the value of the weighted speedup metric, our global (workload-wide) optimization metric which will be defined later in the paper. This metric is a measure of the overall workload performance. Our first observation from these results is that *Ideal* performs much better than *Default*, *Equal*, *L2 Only* and *L3 Only*, indicating that partitioning both L2 and L3 spaces can be critical for achieving high performance (see Figure 2(b)). More importantly, under *Ideal,* the QoS specifications are satisfied for all applications in the workload (see Figure 2(a)). In contrast, under *L2 Only* and *L3 Only*, only 2 and 3 applications, respectively, have their QoS requirements satisfied. In addition, as compared to Isolated, Ideal performs only 10.3% worse. Therefore, we can conclude that careful partitioning of on-chip caches can be very effective in practice.

3 Dynamic Performance Model

Since the workload mix can change during the course of execution (as applications terminate and start) and even the same workload (i.e., applications in

it) can exhibit different data access patterns and cache behavior (space requirements), a static partitioning strategy is probably not the best method. In order to perform runtime cache partitioning however, we need to be able to predict the impact of increasing and decreasing the cache space (number of ways) available to an application on its performance (data access latency). To achieve this, we propose a performance model which is parameterized using *cache space allocations at different layers* in the on-chip cache hierarchy. Specifically, our runtime model can be expressed using a *three-dimensional plot* where x and y axes denote, respectively, cache space allocations from L2 and L3 layers (note that a core has access to only one component from each layer). We build a separate model (three-dimensional plot) for each application in the workload. In the plot of application a, point $d_a(s_{L2}, s_{L3})$ indicates the observed average data access latency value for application a when allocated s_{L2} ways from L2 and s_{L3} ways from L3. When application a executes with allocation (s_{L2}, s_{L3}) for one epoch (enforcement interval), we record the observed d_a value in our three-dimensional plot. As a result, as an application is allocated different (s_{L2}, s_{L3}) values during the course of its execution, we can build a dynamic model for that application.

The most important use of this model (see Figure 3 for models of two applications) in our framework is to *predict* the performance of an application if allocated a certain number of L2 and L3 ways. That is, using the observed (s_{L2}, s_{L3}) values, we can fit a *surface* that represent the d_a values under various cache (way) allocations and use this surface to predict the performance of the application if allocated (s_{L2}, s_{L3}), i.e., s_{L2} L2 ways and s_{L3} L3 ways. This surface is dynamically updated, with newly-observed (s_{L2}, s_{L3}) values and the corresponding d_a values, to adapt to the dynamic modulations in application behavior. Consequently, if an application is allocated the same (s_{L2}, s_{L3}) ways at two different points during its execution, we update the surface with the most recently-observed d_a value (for that allocation pair). It is to be noted that, while this model captures the behavior of a single application under different cache allocations it experiences during execution, one needs a higher level approach to decide cache space allocations across concurrently-executing applications. This approach needs to consider the dynamic models built for all applications and make globally-optimal cache allocations. The next section gives the details of such an approach.

4 Proposed Partitioning Algorithm

We can divide the operation of our multilayer cache partitioning strategy into two main parts: *static part* and *dynamic part*. The static part includes profiling an application for enabling the dynamic part to start making predictions. The dynamic part on the other hand represents the steady-state execution of our approach. Figure 4 presents a high level view of our proposed approach. It is important to note that applications can enter and exit independently and our experimental evaluation considers different scenarios to test the robustness of our partitioning scheme.

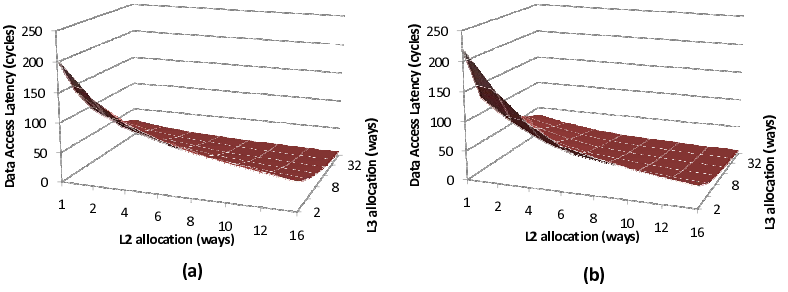


Fig. 3. Dynamic performance models for two different applications (a) milc and (b) gromacs

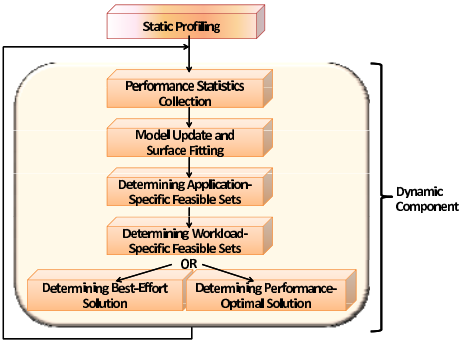


Fig. 4. High-level view of the proposed approach

Profiling: In the static part of our multilayer cache partitioning scheme, each application is profiled to obtain a certain number of performance values corresponding to pairs of (L2,L3) capacities (ways). While it is possible to perform this profiling step on-line, we use static profiling in our current implementation. However, it is important to note that we do not need to profile the entire application; instead, we observe that profiling just a couple of initial epochs is sufficient to start the dynamic model construction. We start building the 3D model dynamically after gathering few such data points in the profiling stage.

Dynamic Partitioning: The main task of the dynamic part of our approach is to decide the partitioning of the L2 and L3 cache spaces among simultaneously-executing applications in a workload at run-time. Therefore, we divide the execution of a workload into several *epochs* which are also called *enforcement intervals* in this paper. The data points observed at the end of the current epoch are recorded and used to update our 3D performance model, and the partitioning of the L2 and L3 cache spaces for the next epoch is decided based on this model. Therefore, our goal is to satisfy the QoS values at each enforcement interval. The accuracy of the model increases with the increase in the number of observed data points. The algorithm iteratively updates each application's latency model at every epoch to increase the accuracy of prediction. The dynamic partitioning portion of our scheme has the following major steps that repeat at each epoch (enforcement interval) boundary:

• *Collecting performance statistics.* For each application, the average data access latency observed in the current epoch is collected. This is done by (i) counting the number of data references made by the application (denoted K), (ii) measuring the time (in cycles) spent by the application in data accesses (denoted T), and (iii) calculating T/K. The details of our implementation to collect these statistics will be given later in Section 5.1. Assuming that, in this epoch, application a was run under allocations s_{L2} and s_{L3}, the calculated latency value ($d_a = T/K$) is used to update our performance model.
• *Surface fitting.* As stated above, for each application in the workload, we update its dynamic performance model using the values collected during the last epoch. We then fit a *surface* using statistical regression [3] that represents the performance model. Specifically, we use the method of *least squares* to determine the regression curve as it minimizes the sum of the squared errors in prediction. An important property of regression is that the best fit can be computed incrementally from a set of measured values. The more values we have (i.e., collected during the course of execution where application goes through several allocations), the more accurate is the prediction. However, we can start predicting with as few values as necessary. The learning module builds the per-application I/O latency model using the least squares method using a certain number of sampling points. Figure 3 illustrates the 3D performance models for two Spec2006 applications (milc on the left and gromacs on the right) in our experimental suite.
• *Identification of application-specific feasible sets.* Our goal in this step is to determine, for each application a in the workload, the set of (s_{L2}, s_{L3}) allocations whose corresponding latency values (d_a) are lower than the specified QoS value for that application (we use qos_a to denote the QoS value for application a). We want to emphasize that, in general, we can have multiple allocations that satisfy the specified QoS value. We use $\mathcal{E}_a$ to denote the set of feasible allocations for application a.
• *Determining the workload-specific feasible sets.* While the previous step determines, for each application, the set of cache allocations that satisfy its QoS metric, we need to select a unique cache allocation for each application such that the total allocation from any given cache component does not exceed the capacity (the number of ways) of that cache component. In mathematical terms, let us focus on a particular L2 component shared by n applications $(a_1, a_2, \cdots a_n)$, another L2 component shared by a different set of n applications $(a_{n+1}, a_{n+2}, \cdots a_{2n})$, and an L3 component shared by all these $2n$ applications $(a_1, a_2, \cdots a_{2n})$. That is, a set of n applications share an L2, a different set of n applications share another L2, and all these applications together share an L3. Let us assume that $\mathcal{E}_{a_i}$ is the application-specific feasible set for a_i, as determined by the previous step. To construct the workload-specific feasible set, we select for each application a_i an allocation (S^i_{L2}, S^i_{L3}) from its $\mathcal{E}_{a_i}$ such that all of the following constraints are satisfied:

$$S^1_{L2} + S^2_{L2} + S^3_{L2} + \cdots + S^n_{L2} \leq x_{L2}$$
$$S^{n+1}_{L2} + S^{n+2}_{L2} + S^{n+3}_{L2} + \cdots + S^{2n}_{L2} \leq x_{L2}$$

$$S^1_{L3} + S^2_{L3} + S^3_{L3} + \cdots + S^{2n}_{L3} \leq x_{L3},$$

where x_{L2} and x_{L3} denote the total number of ways for L2 and L3 caches, respectively. As can be seen, the goal here is to select an allocation for each application such that the total capacity of L2 or L3 is not exceeded. We use f to denote the set that contains these feasible (S^i_{L2}, S^i_{L3}) allocations. Note that f is a set of pairs and in general we have more than one f sets. In the rest of our discussion, we use $\mathcal{F}$ the set of workload-specific feasible partitions. Note that each element of $\mathcal{F}$ is an f set. It needs to be pointed out that the $\mathcal{F}$ may or may not be empty. The next two steps handle these two cases.

• *Determining the best-effort solution.* This step is executed only if $\mathcal{F}$ is empty. In this case, one can adopt several strategies, which include the following:

• *Minimum Loss.* In this strategy, we try to minimize the number of applications whose QoS could not be satisfied. This option can be used when we want to minimize the number of applications to be punished.

• *Weighted Loss.* In this strategy, applications are punished based on some weights assigned to them. When all weights are the same, applications are punished equally, resulting in some sort of fair punishment.

• *Determining the performance-optimal allocation set.* This step is executed only if $\mathcal{F}$ is not empty. If $\mathcal{F}$ contains only a single f, it is returned[1]; otherwise, this step proceeds as follows. Although in theory any f from the workload-specific feasibility set ($\mathcal{F}$) can be used for determining cache allocations in L2 and L3 components, the specific f chosen can make a significant difference in overall performance of the workload. This is because once an f is selected and the cache (way) allocations it indicates are made, there can be an excess cache space (number of ways) in both L2 and L3 layers. And, this *residual cache space* can be distributed across competing applications so that it optimizes the overall system performance. We now define two metrics that can be used to measure the overall system performance. We can use either of them to choose an f from the workload-specific feasibility set ($\mathcal{F}$). We choose f that utilizes the residual cache space to the maximum and that in turn maximizes our overall performance metric.

We use a metric, called *Weighted Speedup* metric (WS) [16], which is the sum of per application speedups (reduction in average data access latency) achieved using our partitioning scheme with respect to a baseline scheme. The baseline scheme can be either *Equal* where resources are equally shared among all the applications in the workload or *Default* which corresponds to full sharing of L2 and L3 caches by all cores that access them. That is, we have:

$$WS(our\_scheme) = \sum_{i=1}^{n} w_i * \frac{Latency_{a_i(baseline\_scheme)}}{Latency_{a_i(our\_scheme)}}, \quad (1)$$

where n is the number of applications in the workload and w_i is the weight assigned to application a_i. In all our experiments, the baseline scheme for our

[1] However, note that even in this case we may have residual cache space, which needs to be distributed to maximize a global metric.

weighted speedup metric is the performance of individual applications with equal partition. In practice, the weights assigned to the applications determine the proportion of residual cache space the applications are allocated. In most of the experimental results reported in this paper, we use the weighted speedup metric with *equal weights* assigned to all the applications in a workload.

5 Experimental Evaluation

5.1 Implementation and Setup

The default multicore configuration used has six cores; each core has a 16KB private cache; each pair of cores share an L2 component of 512KB; and all six cores share an L3 of 6MB. Our major simulation parameters and their default values are given in Table 1. Later in our experiments we change the values of some of these parameters and conduct a sensitivity study.

Table 1. Default multicore configuration

Parameter	Value
Number of Cores	6
ROB/Core	128 entry
Bandwidth/Core	4-fetch, 4-issue, 4-commit
Branch Predictor/Core	hybrid 8192-entry gshare / 2048-entry bimod / 8192-entry meta table
L1 Cache	6 × (16KB; 4ways; 32 byte line size; 2 cycles latency)
L2 Cache	3 × (512KB; 16 ways; 128 byte line size; 8 cycle latency)
L3 Cache	1 × (6MB; 64 ways; 256 byte line size; 20 cycle latency)
Off-Chip Access Latency	200 cycles
On-Chip Interconnect	point-to-point, 3 cycles per hop latency
Coherence Protocol	MOSI-based directory
QoS Specification	7 cycles for all applications
Epoch Length	100 million cycles

We implemented our partitioning scheme as a separate module with in Virtutech Simics [9]. In our implementation, a separate thread (implemented as a loadable module in Solaris 10) carries out the steps of the dynamic portion of our partitioning scheme. We used Simics under Solaris 10 to quantify performance of this implementation as well as the alternate partitioning schemes against which we compare our scheme. To have accurate timings, we also employed the GEMS module from University of Wisconsin [8]. Since Simics provides full-system simulation, the results presented below includes all the overheads incurred by our partitioning thread.

In this paper, we use the applications in the SPEC2006 benchmark suite [13], with the reference input sets. All benchmarks are fast forwarded by one billion instructions to bypass initialization steps, and then simulated for two billion cycles. Table 2 lists the ten workloads (w1 through w10) that we formed for this study using these applications. The last two columns of Table 2 gives the cumulative L2 and L3 cache misses for each workload when executed under the equal partitioning scheme (*Equal*). For each workload in our experimental suite, we performed experiments with eight different cache partitioning schemes:

• *Equal*. Under this scheme, each cache component in the multicore system is divided (way-wise) as evenly as possible among all applications that share it.

• *Default.* This scheme corresponds to unrestricted sharing, i.e., no cache component is partitioned and, as a result, applications that access a cache component can displace each other's data.
• *Coordinated.* This is our proposed coordinated inter-layer cache partitioning scheme described in detail in Section 4.
• *L2 Only.* In this scheme, we partition only L2 components, and L3 space is shared by all cores that access it. For partitioning L2, we use a dynamic performance model similar to the one used for *Coordinated.* Our initial experiments showed that this scheme generates competitive results to prior schemes such as [2], [14], and [15].

Table 2. Workloads used in the experimental analysis

	Applications						L2 Miss Rate	L3 Miss Rate
w1	bzip2	gcc	mcf	gobmk	hmmer	sjeng	27.4%	33.8%
w2	sjeng	libquantum	h264ref	omnetpp	astar	xalancbmk	21.7%	10.4%
w3	astar	perlbench	h264ref	hmmer	bzip2	libquantum	12.6%	18.2%
w4	milc	bwaves	zeusmp	gamess	namd	soplex	7.7%	19.3%
w5	dealll	povray	calculix	tonto	lbm	sphinx3	29.0%	13.4%
w6	wrf	tonto	GemsFDTD	cactusADM	gromacs	milc	18.6%	11.4%
w7	gobmk	hmmer	sjeng	leslie3d	calculix	tonto	24.2%	16.1%
w8	perlbench	h264ref	xalancbmk	GemsFTDT	soplex	dealll	16.7%	14.9%
w9	sjeng	libquantum	bzip2	gcc	gamess	sphinx3	15.8%	9.8%
w10	gobmk	astar	zeusmp	gamess	tonto	wrf	26.2%	17.9%

• *L3 Only.* In this scheme, we partition only L3 components, and each L2 component is shared by all cores that access it. For partitioning L3, we use a dynamic performance model similar to the one used for *Coordinated.*
• *L2+L3.* In this scheme, we partition L2 and L3 in an uncoordinated fashion using a separate performance model for each one of them. More specifically, under this scheme, performance models for L2 layer and L3 layer observe the data access latency in respective layers and adapt partitioning based on these dynamically updated performance models independently.
• *Static Best.* In this scheme, we select a static partitioning (for each cache component) that generates the best result (when the entire on-chip cache hierarchy considered) through exhaustive search of all possible static partitions. Consequently, this scheme represents the best partitioning of shared cache that can be achieved by any static scheme. Also note that in general the selected partitioning for a cache component will be nonuniform.
•*Dynamic Best.* This scheme represents the best dynamic strategy for partitioning the shared on-chip cache space. The workload execution in divided into epochs and, for each epoch, we run *Static Best* to determine the best partition of cache components for that epoch. Note that both *Dynamic Best* and *Static Best* return a result (partitioning) as output only if this partitioning satisfies all the specified QoS values.

5.2 Results

Unless stated otherwise, we use weighted speedup (with all weights being the same) as our global metric in distributing the residual cache space (if any) across

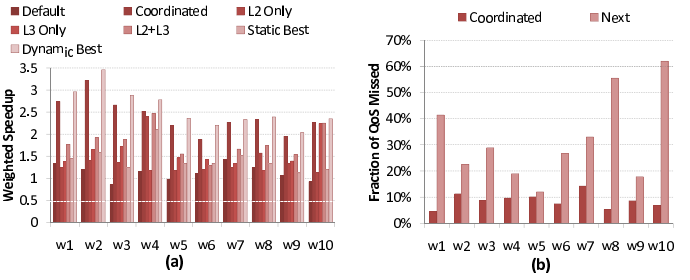


Fig. 5. (a) Weighted speedups with our default configuration. (b) Percentage of experiments where our approach and an alternate scheme could not satisfy the specified QoS, that was satisfied by *Dynamic Best*.

applications. Also, if not all the QoS specifications could be satisfied, we employ the *minimum loss* strategy for determining the application(s) to be punished.

Weighted Speedups. The bar-chart in Figure 5(a) gives the weighted speedup values for our ten workloads described above. The QoS value used for each application is 7 cycles in this experiment. One can make several observations from these results. First, excluding the ideal partitioning scheme (*Dynamic Best*), our proposed partitioning scheme (*Coordinated*) generates the best speedup values for all the workloads tested. Second, our scheme comes very close to the ideal partitioning, the average weighted speedup values being 2.40 and 2.57. Third, in w4, our scheme and L2 Only generate very similar results. This is because, in this workload, there are relatively small number of L2 misses (L3 accesses) and the main benefit comes from careful partitioning of the available L2 space. What is more interesting however is that our scheme generates much better results than *L2+L3* (which generates an average weighted speedup value of 1.80). The main reason for this is the lack of coordination (under this scheme) between L2 and L3 cache partitioning.

To show how successful our scheme is in satisfying QoS values, we present in Figure 5(b) the percentage of experiments where our approach could not satisfy the specified QoS but *Dynamic Best* was able to satisfy the same QoS (using exhaustive search). This data is collected over 400 experiments with QoS values randomly distributed between 3 cycles and 10 cycles, and spans all workloads we have. We see from the first bars in this plot that, on average, only in about 8.6% of our experiments, our approach could not satisfy a specified QoS that were satisfied by *Dynamic Best*. In comparison, the second bar, for each workload, gives similar results with the second-best scheme (*L2+L3* in most cases). We see that the average value for this scheme is around 43.8%.

QoS Values. We now focus on two workloads (w2 and w10) and present the data access latency values for these workloads (recall that the QoS values for all applications are set to 7 cycles in this experiment). The bar-chart in Figure 6(a) presents the results for w2 and that in Figure 6(b) for w6. Our first observation

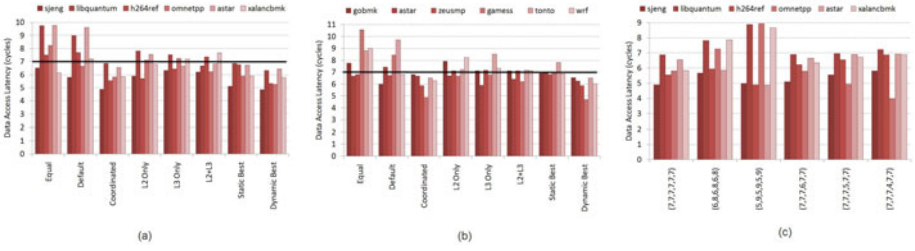


Fig. 6. Data access latency values for (a) workload w2 under default QoS specification and (b) workload w6 under default QoS specification. (c) workload w2 under different QoS specifications.

from these plots is that our scheme satisfies the specified QoS for all applications. In comparison, in w2, *L2 Only*, *L3 Only* and *L2+L3* could not satisfy QoS for 3, 3, and 2 applications, respectively. *Equal* and *Default* perform even worse, each not being able to satisfy QoS for 4 out of 6 applications in w2.

Sensitivity Study. The results presented above are for a fixed QoS value for all applications. To study sensitivity of our approach to QoS values, we now present results with workload w2 when executed with different QoS values. Each point on the x-axis in Figure 6(c) corresponds to a set of QoS values for the applications in this workload (i.e., ith entry in the vector corresponds to the QoS (in cycles) for the ith application in the workload), and the y-axis represents data access latencies achieved by our approach (*Coordinated*), when targeting the specified QoS values. One can observe that, in the first five scenarios, our scheme is able to satisfy all QoS values. In the last scenario, on the other hand, our scheme could not satisfy all QoS values. Clearly, by requiring very low access latencies (as QoS specifications), one can surely fail any partitioning scheme. However, what is important here is that our scheme successfully adapts to the specified QoS values, which is observed from the first five scenarios.

6 Concluding Remarks

We demonstrate that coordinated multilayer cache partitioning is critical for extracting maximum performance from on-chip caches of emerging multicore architectures. We also present a fully-automated, model based, multilayer cache partitioning scheme. This scheme tries to achieve two objectives. First, it tries to satisfy specified QoS values for all applications, and second, it distributes the remaining excess cache capacity (if any) across applications such that a global performance metric (weighted speedup) is maximized. In our experimental evaluation, we compare our approach against several alternate schemes as well as an optimal scheme which is not implementable in practice but guarantees the best workload performance. Our experience with this scheme shows that it is very successful in practice and the results obtained using it cannot be achieved by applying cache partitioning to each layer independently.

References

1. Alfs, G., Knupferr, N.: Intel's Multicore Architecture Briefing (2008), http://www.intel.com/pressroom/archive/releases/20080317fact.htm
2. Chang, J., Sohi, G.: Cooperative Cache Partitioning for Chip Multiprocessors. In: ICS (2007)
3. Duchesne, P., Remillard, B.: Statistical Modeling and Analysis for Complex Data Problems. Springer, Heidelberg (2005)
4. Herdrich, A., et al.: Rate-based QoS Techniques for Cache/Memory in CMP Platforms. In: ICS (2009)
5. Ko, B., et al.: Scalable Service Differentiation in a Shared Storage Cache. In: ICDCS (2003)
6. Guo, F., et al.: A Framework for Providing Quality of Service in Chip Multi-Processors. In: MICRO (2007)
7. Nesbit, K., et al.: Fair Queuing Memory Systems. In: MICRO (2006)
8. Martin, M., et al.: Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. In: SIGARCH Comput. Archit. News (2005)
9. Magnusson, P.S., et al.: Simics: A Full System Simulation Platform. IEEE Transactions on Computer (2002)
10. Iyer, R., et al.: CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In: ICS (2004)
11. Iyer, R., et al.: QoS Policies and Architecture for Cache/Memory in CMP Platforms. In: SIGMETRICS (2007)
12. Borkar, S.Y., et al.: Intel Processor and Platform Evolution for the Next Decade. Technical Report, Intel (2005)
13. Henning, J.L., et al.: SPEC CPU2006 Benchmark Descriptions. SIGARCH Comput. Archit. News (2006)
14. Qureshi, M.K., et al.: Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In: MICRO (2006)
15. Rafique, N., et al.: Architectural Support for Operating System-Driven CMP Cache Management. In: PACT (2006)
16. Smith, J.E.: Characterizing Computer Performance with a Single Number. ACM Communications Journal (1988)
17. Srikantaiah, S., et al.: SHARP Control: Controlled Shared Cache Management in Chip Multiprocessors. In: MICRO (2009)

Backfilling with Guarantees Granted upon Job Submission

Alexander M. Lindsay[1,⋆], Maxwell Galloway-Carson[2], Christopher R. Johnson[2], David P. Bunde[2], and Vitus J. Leung[3]

[1] iBASEt
aml.lindsay@gmail.com
[2] Knox College
{mgallowa,crjohnso,dbunde}@knox.edu
[3] Sandia National Laboratories
vjleung@sandia.gov

Abstract. In this paper, we present scheduling algorithms that simultaneously support guaranteed starting times and favor jobs with system-desired traits. To achieve the first of these goals, our algorithms keep a profile with potential starting times for every unfinished job and never move these starting times later, just as in Conservative Backfilling. To achieve the second, they exploit previously unrecognized flexibility in the handling of holes opened in this profile when jobs finish early. We find that, with one choice of job selection function, our algorithms can consistently yield a lower average waiting time than Conservative Backfilling while still providing a guaranteed start time to each job as it arrives. In fact, in most cases, the algorithms give a lower average waiting time than the more aggressive EASY backfilling algorithm, which does not provide guaranteed start times. Alternately, with a different choice of job selection function, our algorithms can focus the benefit on the widest submitted jobs, the reason for the existence of parallel systems. In this case, these jobs experience significantly lower waiting time than Conservative Backfilling with minimal impact on other jobs.

1 Introduction

Backfilling has been a standard feature of multiprocessor scheduling algorithms since it was introduced by Lifka [7] in the Extensible Argonne Scheduling sYstem (EASY). In a survey of parallel job scheduling, Feitelson et al [4] characterize backfilling with three parameters, the number of reservations or jobs with guaranteed start times, the order of queue jobs, and the amount of lookahead into the queue. In this paper, we describe variations of backfilling where all jobs are given a guarantee upon their arrival, Conservative Backfilling [8]. However, unlike Conservative Backfilling, we are interested in supporting job priorities other than First-Come-First-Serve (FCFS) [10]. Also, while we do not use any lookahead into the queue, one of our algorithms does delay making decisions

⋆ Work done while Alex was a student at Knox College.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 142–153, 2011.

until more data is available. Thus, our algorithms add a fourth parameter, when decisions are made, to the three parameters mentioned above.

A key benefit of Conservative Backfilling is that each job is granted a guaranteed starting time when it is submitted. (It may start earlier, but will not be delayed later than this time.) These guarantees lead Conservative Backfilling to benefit wide jobs, jobs requiring many processors, relative to other backfilling strategies (e.g. [13]). From a fairness standpoint, this guarantee ensures that wide or long jobs, which are less likely to benefit from backfilling, are not harmed by jobs that backfill more easily. These guarantees also make the scheduler more predictable since each user has a bound on when their jobs will run.

Conservative backfilling maintains a profile containing a tentative schedule for all jobs. When a job arrives, it is placed in the earliest possible spot within the profile, i.e. it is scheduled to start at the earliest time that does not disturb any previously-placed job. The only other profile changes occur when a job finishes early, creating a "hole" that potentially allows other jobs to move earlier. In this case, Conservative initiates *compression*, the reexamination of each job in the order of its current starting time in the profile. Each job is removed from the schedule and then reinserted at the earliest possible time. Compression never delays a job since the job can always fit back into the profile at the same spot, but some jobs move earlier, into a hole or spaces vacated by jobs that have themselves moved. Since no job's planned start time is ever delayed, each job's initial reservation is an upper bound on its actual starting time.

Because Conservative compression reschedules jobs based on the profile's order, intuition suggests that it tends to preserve job order, closing holes by sliding the end of the profile earlier. (Of course, job order does change when a job fits into a hole that earlier jobs could not use.) Since the profile is built as jobs arrive, this gives Conservative a FCFS tendency. This is desirable from a fairness perspective, but may not support a specific system's goals. For example, some systems may wish to favor short jobs to improve average response time and systems oriented toward capability computing may wish to favor wide jobs.

Backfilling algorithms have been designed to support these goals (e.g. [14,5,1,11]), but they do so by reordering the profile, which sacrifices the key benefit of Conservative scheduling: its ability to give jobs guaranteed starting times when they are submitted. In this paper, we present scheduling algorithms that simultaneously support guaranteed starting times and favor jobs with system-desired traits. To achieve the first of these goals, our algorithms keep a profile with potential starting times for every unfinished job and never delay these starting times, just as in Conservative. To achieve the second, they exploit previously unrecognized flexibility in the handling of holes that appear in the profile. Specifically, we present two algorithms using the following kinds of flexibility:

– *job selection*: Although Conservative always tries to move the next job in the profile into a hole, any job that fits can be moved into a hole. (This idea is also used in [9].)

– *timing*: Although Conservative closes holes as soon as they form, the scheduler is only required to identify jobs that it wants to start immediately. Thus, some decisions can be deferred until more information (e.g. more job arrivals or early completions) is available.

We analyze our algorithms using an event-based simulator run with traces from the Parallel Workloads Archive [3]. From the traces, our simulator takes an arrival time, a required number of processors, a running time, and an estimated running time for each job. The estimated running time gives the scheduler an upper bound on the job's running time, but most jobs "end early", with actual running time less than their estimate. Throughout, we assume that jobs need exactly the requested number of processors (rigid jobs), that each processor can run at most one job at a time (pure space-sharing) and that each job finishes in exactly its given running time (no interference between jobs).

We find that, with one choice of job selection function, our algorithms consistently yield a lower average waiting time than Conservative while still providing each job a guaranteed start time when it arrives. In fact, in most cases, our algorithms give better waiting times than the more aggressive EASY algorithm [7], which does not provide guaranteed start times. Alternately, with another job selection function, our algorithms significantly lower waiting times for the widest jobs with minimal impact on other jobs.

The rest of the paper is organized as follows. We describe our algorithms in Section 2 and relevant related work in Section 3. Then we give our experimental results in Section 4 and conclude in Section 5.

2 Algorithms

Now we present our algorithms to exploit the flexibility discussed above.

2.1 Prioritized Compression

Our first algorithm is *conservative with Prioritized Compression* (PC). This algorithm maintains two data structures, a profile with the tentative schedule and a *compression queue* of jobs ordered by a system-specific priority function.

When a job arrives into the system, it is placed into the profile exactly as in Conservative and also added to the compression queue. When a job finishes early and creates a hole, PC compresses the schedule by trying to reschedule each job in the order given by the compression queue; it tries to reschedule the first job in the compression queue, then the second, and so on. This differs from Conservative, which considers jobs in the order they occur in the profile, but PC preserves the key feature that no job moves later in the profile; a job accepts rescheduling only when it benefits and a job is only permitted to make moves that do not interfere with any other job.

By using a customized order for compression, PC allows high-priority jobs to benefit from the hole even if they begin much later in the profile. Doing so adds another wrinkle to the compression operation, however. Consider the profile

shown in Figure 1(a); time is on the x-axis, with the current time at the far left. Suppose job A finishes early and is removed. If the resulting profile is compressed with the order E, C, D (Longest Job First), only jobs C and D are rescheduled. This yields the profile shown in Figure 1(b), with job E delayed even though it could also be started. To avoid unnecessary idle time like this, the compression algorithm for PC returns to the front of the compression queue each time a job is rescheduled. (Conservative does not need to do so since rescheduling one job cannot benefit a previously-considered job when the profile order is used.)

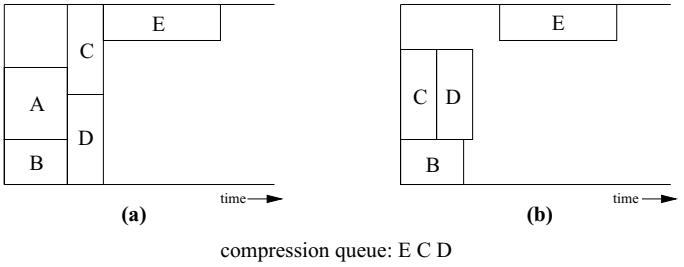


Fig. 1. Profile showing need to return to beginning of the compression queue after each successful rescheduling. (a) Initial profile before job A terminates early. (b) Profile after rescheduling jobs E, C, and D once each in that order.

The downside of returning to the beginning of the compression queue after each successful rescheduling operation is that jobs can be moved more than once. For example, consider the profile depicted in Figure 2(a) and suppose again that job A finishes early. If the profile is compressed with the order D, C (Widest Job First), the first rescheduling operation improves the planned start time of job D, producing the profile shown in Figure 2(b). Once job C is rescheduled, however, job D can be moved again, resulting in the profile shown in Figure 2(c).

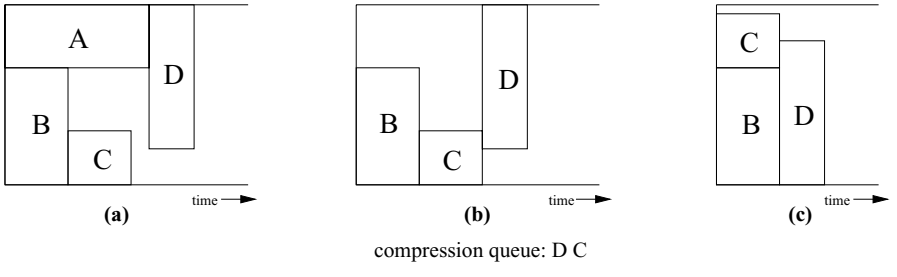


Fig. 2. Example where PC compression moves the same job twice. (a) Initial profile before job A terminates early. (b) Profile after first compression of job D. (c) Profile after compressing job C and then job D again.

Since jobs can move more than once, a natural question is how long compression will take. We return to this question in Section 4.3.

2.2 Delayed Compression

Our second algorithm is *conservative with Delayed prioritized Compression* (DC). It keeps a prioritized compression queue just like PC, but also exploits flexibility in the timing of compression by deferring some rescheduling operations. Specifically, when a job finishes early, DC's compression operation only reschedules jobs that can begin immediately, deliberately leaving holes in the profile. For example, consider the profile depicted in Figure 3(a) and suppose job A finishes early. If DC compresses with order D, E, F (Longest Job First), it would leave the profile as depicted in Figure 3(b), with a hole after job C even though the planned starting time of job F could be improved. By deferring this improvement, algorithm DC leaves itself flexibility in case a high-priority job arrives or another job finishes early. Note that once the running system reaches the hole, the scheduler must fill the hole; this requires an additional check when a job finishes and the profile indicates idle time for some of its processors.

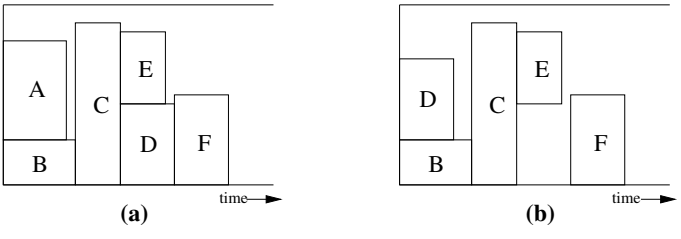


Fig. 3. Example where the DC algorithm deliberately leaves a hole in the profile. (a) Initial profile before job A terminates early. (b) Profile after compression.

One issue with deliberately leaving holes in the profile is that newly-arrived jobs can backfill into them. For example, suppose a short job arrived after the compression operation shown in Figure 3. If this job fits into the hole left when job D was moved, it can backfill there and bypass job F as well as any later jobs. While this backfill operation may be fine if the scheduler wishes to favor short jobs, it can completely undermine the scheduler's priority mechanism if a different priority function is being used. To avoid this, DC also handles job arrivals differently than Conservative. Rather than immediately adding a new job to the profile, DC instead adds it to the compression queue. The algorithm then reschedules any job before the new job in the compression queue whose new start time would be before the estimated completion of the new job, i.e. those higher-priority jobs that could be delayed by the new job. Once the new job is reached in the compression queue, it is scheduled and compression ends. This modified treatment of job arrivals closes holes when necessary to protect rescheduling opportunities for high-priority jobs. In the example shown in Figure 3, DC would reschedule F if a lower-priority job arrives and could be scheduled to finish after the end of C (the earliest possible start time of F). Alternately, the hole could be occupied by a new job with higher priority.

3 Related Work

Backfilling was introduced by Lifka [7] in the Extensible Argonne Scheduling sYstem (EASY). In a survey of parallel job scheduling, Feitelson et al [4] characterize variations in backfilling with three parameters, the number of reservations, the order of queue jobs, and the amount of lookahead into the queue. We add a fourth parameter, when the profile can be reordered.

Reservations have been used since the early days of parallel batch schedulers [2]. EASY [7] uses one reservation. At the other extreme, Conservative Backfilling [8] gives all jobs a reservation. Talby and Feitelson [14] and Srinivasan et al [13] suggest an adaptive number of reservations. The Maui Scheduler [5] has a parameterized number of reservations. Chiang et al [1] suggest that four is a good number of reservations.

EASY and Conservative Backfilling use First-Come-First-Serve (FCFS) order. The FCFS Scheduling Algorithm has been analyzed by Schwiegelshohn and Yahyapur [10]. Perkovic and Keleher [9] study Conservative Backfilling with random queue ordering both with and without sorting by length and random reordering as well. Reordering the backfill queue for EASY is proposed by Tsafrir et al [15].

Talby and Feitelson [14] combine three types of priorities in the order of queue jobs. The Maui Scheduler has even more components in its order of queue jobs. Chiang et al [1] propose generalizations of the Shortest Job First (SJF) scheduling algorithm to order queue jobs. They also use *fixed* and *dynamic* reservations. With dynamic reservations, job reservations and the ordering of job reservations can change with each new job arrival or if the priorities of waiting jobs change. With fixed reservations, job reservations can only move earlier in order, even if a job has no reservation or a job that has a later reservation attains a higher priority. Leung et al [6] study fixed and dynamic variations of Conservative Backfilling in the context of fairness.

All the above algorithms use no lookahead. Shmueli and Feitelson [11] use one reservation, various queue orderings, and lookahead into the queue. All of these algorithms reorder the profile when a job arrives or terminates early. All of our algorithms give every job a reservation, use various queue orderings based on the length or width of the jobs, and use no lookahead into the queue, a combination that is not used by any of the algorithms above. Additionally, some of our algorithms delay to varying degrees when the profile is reordered. Our PC algorithm reorders the profile when a job arrives or terminates early like all of the algorithms above. Our DC algorithm reorders the profile only when a job arrives or can run immediately.

4 Experimental Results

As described in the introduction, we evaluate our algorithms with an event-based simulator running traces from the Parallel Workloads Archive [3]. Figure 4 lists the traces used. These are all traces with estimated running times except

for LLNL-uBGL, which is omitted because its waiting time shows almost no variation for any of the algorithms we examined. Jobs in these traces without user estimates are given accurate estimates. (Simulations by Smith et al. [12] suggest that better estimates reduce average waiting time for Conservative scheduling. The effect of inaccurate estimates on EASY is the subject of many papers; Tsafrir and Feitelson [16] summarize and attempt to settle the issue.)

Name	Full file name	# jobs	% w/ estimates
CTC-SP2	CTC-SP2-1996-2.1-cln.swf	77,222	99.99
DAS2-fs0	DAS2-fs0-2003-1.swf	219,571	100
DAS2-fs1	DAS2-fs1-2003-1.swf	39,348	100
DAS2-fs2	DAS2-fs2-2003-1.swf	65,380	100
DAS2-fs3	DAS2-fs3-2003-1.swf	66,099	100
DAS2-fs4	DAS2-fs4-2003-1.swf	32,952	100
HPC2N	HPC2N-2002-1.1-cln.swf	202,876	100
KTH-SP2	KTH-SP2-1996-2.swf	28,489	100
LANL-CM5	LANL-CM5-1994-3.1-cln.swf	122,057	90.75
LLNL-Atlas	LLNL-Atlas-2006-1.1-cln.swf	38,143	84.85
LLNL-Thunder	LLNL-Thunder-2007-1.1-cln.swf	118,754	32.47
LPC-EGEE	LPC-EGEE-2004-1.2-cln.swf	220,679	100
SDSC-BLUE	SDSC-BLUE-2000-3.1-cln.swf	223,669	100
SDSC-DS	SDSC-DS-2004-1.swf	85,006	100
SDSC-SP2	SDSC-SP2-1998-3.1-cln.swf	54,041	99.94

Fig. 4. Traces used in simulations

The trace job counts given in Figure 4 differ from the values given in the Parallel Workloads Archive [3] because we ignored jobs that were partial executions (they were checkpointed and swapped out; status 2, 3, or 4) and jobs that were cancelled before starting (status 5 and running time ≤ 0). We also ignored 8 jobs in the SDSC-DS trace with running time -1 (unknown).

4.1 Increasing Responsiveness

Since user-perceived performance is the typical goal of scheduling, we first consider how our algorithms can improve average waiting time. For this metric, it is beneficial to run short jobs before long ones so we use Shortest Job First as our priority function. Figure 5 presents the results as a percent improvement over the average waiting time achieved by Conservative. We also include EASY for comparison since it backfills aggressively, benefiting short jobs since they backfill more easily. The exact results vary by traces, but our algorithms outperform Conservative on all traces except DAS2-fs3. In fact, they outperform EASY in the majority of cases. The most notable exception is the LLNL-Thunder trace, which has the lowest percent of jobs with user estimates (only 32%; see Figure 4). This may explain the relatively poor performance of our algorithms on that trace since jobs without estimates do not finish early, reducing the number of holes our algorithms can exploit. Of our algorithms, DC generally beats PC.

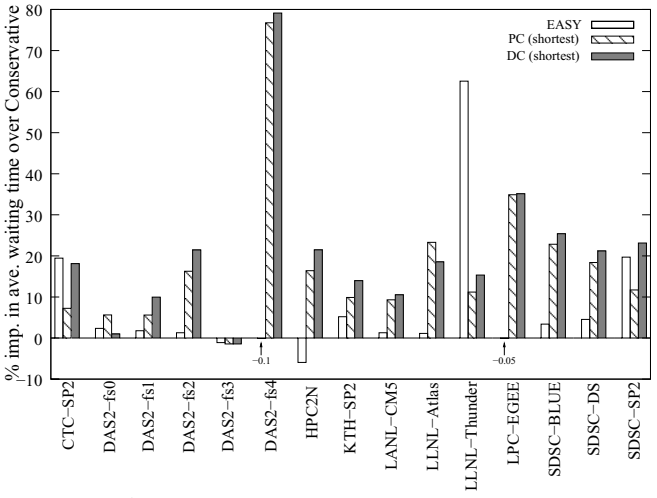


Fig. 5. Average waiting time relative to Conservative

Furthermore, our algorithms achieve these benefits without greatly delaying other jobs. To see this we looked at the average waiting time for the 5% of jobs with the greatest waiting time. See Figure 6 for the results, again presented as a percent improvement over Conservative's performance on the same measure. As in the overall average waiting time, our algorithms generally outperform Conservative, though there are more exceptions (DAS2-fs3, LANL-CM5, and SDSC-SP2). Comparing to EASY yields a similar picture as well, again with LLNL-Thunder as the outlier. The pattern remains when looking at the 1% of jobs with the greatest waiting time (see Figure 7); our algorithms give significantly better performance for the DAS2-fs2, DAS2-fs4, LLNL-Atlas, and LPC-EGEE traces, significantly worse performance for the LANL-CM5 and SDSC-SP2 traces, and comparable (within 10%) or mixed performance for the others.

We have shown that our algorithms significantly improve the average waiting time when using the shortest job first priority function. It is worth noting that they mostly outperform Conservative under this measure with other natural priority functions as well. Specifically, we considered the priority functions FIFO, Widest (most requested processors) Job First, Longest (in estimated time) Job First, Shortest Job First, and Narrowest (fewest requested processors) Job First for both of our algorithms. Out of 150 combinations of trace, algorithm, and priority function, only 49 (33%) of them were worse than Conservative. Most of the differences were small (generally $< 10\%$, many $< 2\%$), with a majority of the big improvements appearing in Figure 5 and the significantly negative values generally associated with the Longest Job First or Widest Job First priority functions. (The worst single value is -38% for DC with Longest Job First.)

Overall, DC with Shortest Job First seems to be a very good choice for increasing responsiveness. It gave better average waiting time than Conservative and EASY in eleven out of fifteen traces. It only had worst average waiting time than both Conservative and EASY in one trace and just EASY in three others.

4.2 Favoring Wide Jobs

To demonstrate the flexibility of our algorithms, we also look at a different scheduling goal: improving the performance of wide jobs. These are jobs that, because of a large computational or memory requirement, must run on many processors. From a capability perspective, wide jobs are the reason to build large systems since they cannot run otherwise.

To benefit these jobs, we run our algorithms with the Widest Job First priority function. We measure schedule quality with the average waiting time of the

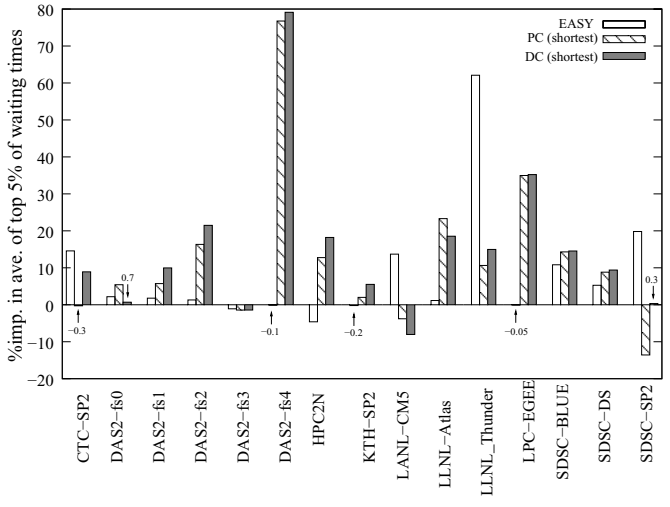


Fig. 6. Average of top 5% of waiting times relative to Conservative

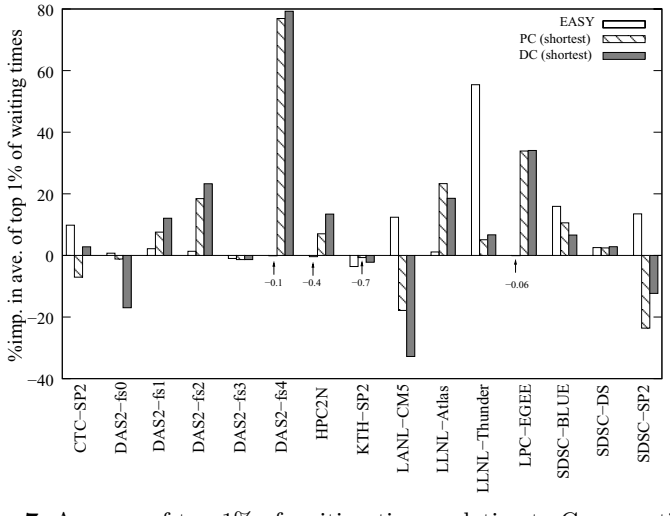


Fig. 7. Average of top 1% of waiting times relative to Conservative

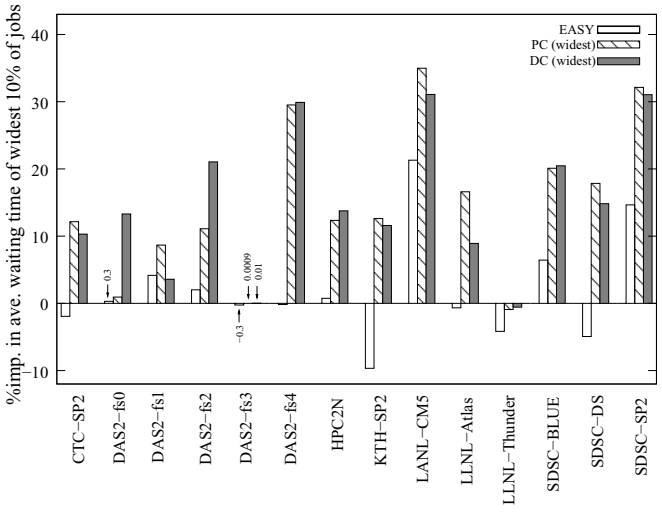


Fig. 8. Average waiting time of widest 10% of the jobs relative to Conservative

widest 10% of the jobs in each trace. Figure 8 shows the results as a percent improvement over Conservative. The LGC-EGEE trace is not included since each of its jobs requests a single processor. On the other traces, our algorithms outperform Conservative on all traces except LLNL-Thunder, the trace with relatively few user estimates. (The improvement on the DAS2-fs3 trace is admittedly negligible.) It is unclear which of them is preferable. Our algorithms also outperform EASY, which is not surprising since wide jobs have difficulty backfilling and thus benefit from the guaranteed start times given by our algorithms.

As when we tried to improve overall system responsiveness, we investigate the performance of non-favored jobs. Figure 9 plots average waiting time of all jobs, again relative to Conservative. The results are mixed, but not consistently bad and the negative values are of fairly small magnitude. Thus, it seems that our algorithms benefit wide jobs without greatly impairing overall performance.

4.3 Scheduler Running Time

As mentioned in Section 2.1, there is a question as to how long compression will take with our algorithms, particularly PC. We instrumented our simulations of Conservative and PC to measure the work required for compression. Specifically, we counted how many times the algorithms looked at an event (a job's planned start or end time) in the profile. In the worst case (Longest Job First priority on DAS2-fs0), PC examined nearly 580 times as many events as Conservative. This case is an extreme outlier; in only two other traces (HPC2N and DAS2-fs3) did PC examine more than 43 times as many events as Conservative (127 and 72 times, respectively). Even in the outlier case, however, the scheduler running

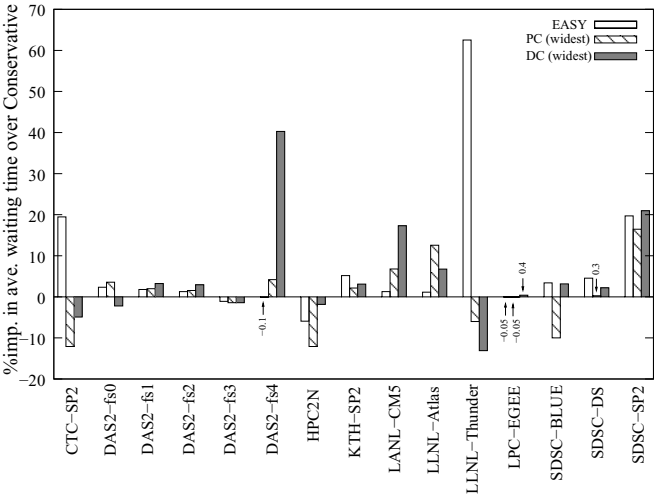


Fig. 9. Average waiting time relative to Conservative

time was not excessive; the total simulation time for that trace was less than 24 hours on a laptop, meaning the scheduler spent less than 0.4 seconds scheduling and rescheduling each job on average.

5 Discussion

We have presented a couple of algorithms that exploit flexibility in Conservative backfilling to improve various measures of performance while still retaining its ability to give jobs a guaranteed starting time as they arrive. We are impressed by the potential of these algorithms, but there is ample room for future research. More work is needed to understand why the algorithms perform better on some traces than others and to distinguish between the algorithms. It would also be interesting to consider other priority functions, including user-assigned job priorities, to further explore the flexibility in job selection. For the flexibility in timing, one of our algorithms closes holes as soon as possible and the other closes holes only when more jobs arrive or a job can run. We can further explore the flexibility in timing by closing holes only when a job can run.

Acknowledgments. A.M. Lindsay, M. Galloway-Carson, C.R. Johnson, and D.P. Bunde were partially supported by contracts 763836 and 899808 from Sandia National Laboratories. Sandia is a multipurpose laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under Contract No. DE-AC04-94AL85000. We also thank all those who contributed traces to the Parallel Workloads Archive.

References

1. Chiang, S.-H., Arpaci-Dusseau, A., Vernon, M.K.: The impact of more accurate requested runtimes on production job scheduling performance. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 103–127. Springer, Heidelberg (2002)
2. Das Sharma, D., Pradhan, D.K.: Job scheduing in mesh multicomputers. In: Proc. Intern. Conf. on Parallel Processing Workshops, pp. 251–258 (1994)
3. Feitelson, D.: The parallel workloads archive, http://www.cs.huji.ac.il/labs/parallel/workload/index.html
4. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U.: Parallel job scheduling — A status report. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2004. LNCS, vol. 3277, pp. 1–16. Springer, Heidelberg (2005)
5. Jackson, D., Snell, Q., Clement, M.: Core algorithms of the maui scheduler. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 2001. LNCS, vol. 2221, pp. 87–102. Springer, Heidelberg (2001)
6. Leung, V., Sabin, G., Sadayappan, P.: Parallel job scheduling policies to improve fairness - a case study. In: Proc. 6th Intern. Workshop on Scheduling and Resource Management for Parallel and Distributed Syst. (2010)
7. Lifka, D.: The ANL/IBM SP scheduling system. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1995 and JSSPP 1995. LNCS, vol. 949, pp. 295–303. Springer, Heidelberg (1995)
8. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. IEEE Trans. on Parallel and Distributed Syst. 12(6), 529–543 (2001)
9. Perković, D., Keleher, P.J.: Randomization, speculation, and adaptation in batch schedulers. In: Proc. 2000 ACM/IEEE Conf. on Supercomputing (2000)
10. Schwiegelshohn, U., Yahyapour, R.: Analysis of first-come-first-serve parallel job scheduling. In: Proc. 9th ACM/SIAM Symp. on Discrete Algorithms, pp. 629–638 (1998)
11. Shmueli, E., Feitelson, D.G.: Backfilling with lookahead to optimize the performance of parallel job scheduling. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 228–251. Springer, Heidelberg (2003)
12. Smith, W., Taylor, V., Foster, I.: Using run-time predictions to estimate queue wait times and improve scheduler performance. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 1999, IPPS-WS 1999, and SPDP-WS 1999. LNCS, vol. 1659, pp. 202–219. Springer, Heidelberg (1999)
13. Srinivasan, S., Kettimuthu, R., Subramani, V., Sadayappan, P.: Selective reservation strategies for backfill job scheduling. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 55–71. Springer, Heidelberg (2002)
14. Talby, D., Feitelson, D.G.: Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling. In: Proc. 13th Intern. Parallel Processing Symp., pp. 513–517 (1999)
15. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Backfilling using system-generated predictions rather than user runtime estimates. IEEE Trans. on Parallel and Distributed Systems 18(6), 789–803 (2007)
16. Tsafrir, D., Feitelson, D.G.: The dynamics of backfilling: Solving the mystery of why increased inaccuracy may help. In: Proc. IEEE Intern. Symp. on Workload Characterization, pp. 131–141 (2006)

Introduction

Leonel Sousa, Frédéric Suter, Alfredo Goldman,
Rizos Sakellariou, and Oliver Sinnen

Topic chairs

Scheduling and load balancing are fundamental issues for deploying applications on parallel and distributed systems. Static and dynamic techniques, deterministic and stochastic methods have been researched to tackle the hard problem of achieving the minimum span and the optimal load balancing, making the best use of parallel and distributed systems by maintaining the resources busy and minimizing energy consumption. Although research has been done for years and years, namely for static scheduling and dynamic load balancing, these are old but very timely topics of research in the era of multicore computers and cloud computing. New challenges arise with the increased interest in applications with real-time constraints, the continuous growth of algorithms complexity and sophistication of applications, and the heterogeneity of systems and the diversity of their conditions of operation. This year, the contributions in the scheduling and load-balancing topic of Euro-Par provide a very good coverage of different perspectives and aspects, with a focus on both theoretical aspects and practical questions. Some of these papers are focused on heterogeneous systems, in particular on more hierarchical systems, some also considering failures, there are a few that address theoretical aspects and one mainly presents experimental work. The papers continue to cover the two ends of the hardware spectrum, tightly-coupled multicore systems and clusters of workstations. Energy awareness has become important for all types of computing and it is addressed in the accepted papers for the small scale, in embedded systems, as well as for larger computing facilities, such as clusters. Modern scheduling and load balancing is dominated by the inclusion of more aspects into the scheduling decisions, be it communication and memory location aspects or even the social influence. The task of selecting the papers to be presented at the conference was hard, because the number of submissions was high and the quality excellent. Only 9 papers were accepted for publication, which led to quite a low acceptance rate in this topic. All papers were reviewed by at least four independent reviewers. We would like to thank all the reviewers for their time and effort. The quality of the reviews simplified the selection process. At the same time, we would like to thank all authors, in particular the ones that did not have their manuscripts accepted. Their contributions allow Euro-Par to maintain its position as one of the premier scientific conferences where innovative scheduling research for parallel and distributed systems is presented year after year.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, p. 154, 2011.

Greedy "Exploitation" Is Close to Optimal on Node-Heterogeneous Clusters*

Arnold L. Rosenberg

Colorado State University, Fort Collins, CO 80523, USA
Northeastern University, Boston, MA 02115, USA
rsnbrg@cs.umass.edu

Abstract. The Cluster-Exploitation Problem (CEP) challenges a *master* computer to schedule a "borrowed" node-heterogeneous cluster $\mathcal{C}$ of *worker* computers in a way that maximizes the amount of work that $\mathcal{C}$'s computers complete within a fixed time period. This challenge is heightened by the fact that "completing" work requires $\mathcal{C}$'s computers to return results from their work to the master. It has been known for some time that the greedy *LIFO* protocol, which orchestrates $\mathcal{C}$'s computers to finish working in the *opposite* of their starting order, *does not* solve the CEP optimally; in fact, the *FIFO* protocol, which has $\mathcal{C}$'s computers finish working in the *same* order as they start, *does* solve the CEP optimally (over sufficiently long time periods). That said, the LIFO protocol has features (aside from its intuitive appeal) that would make it attractive to implement when solving the CEP—as long as its solution to the problem was not too far from optimal. This paper shows this to be the case. Specifically:

1. The LIFO protocol provides approximately optimal solutions to the CEP, in the following sense. For every cluster $\mathcal{C}$, there is a fixed fraction $\varphi_{\mathcal{C}} > 0$ *that does not depend on how heterogeneous cluster* $\mathcal{C}$ *is* (as measured by the relative speeds of its fastest and slowest computers) such that $\mathcal{C}$ completes at least the fraction $\varphi_{\mathcal{C}}$ as much work under the LIFO protocol as under the optimal FIFO protocol.

Our analysis of the CEP uncovers an unexpected property of the LIFO protocol:

2. In common with the FIFO protocol, the LIFO protocol's work production is independent of the order in which the master supplies work to the workers—no matter what the relative speeds of the workers are.

Within the literature of *divisible load scheduling*, the CEP follows the master-worker paradigm under the "single-port with no overlap" model.

Keywords: Scheduling divisible workload; Worksharing; Heterogeneous cluster.

1 Introduction

A *master* computer C_0 has a large uniform computational workload of independent tasks. It has temporary access to a cluster[1] $\mathcal{C}$ comprising n *worker* computers,

* Research supported in part by US NSF Grant CNS-0905399.

[1] We call $\mathcal{C}$ a "cluster" for convenience: the C_i may be geographically dispersed and more diverse in power than that term usually connotes.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 155–166, 2011.

$C_1, \ldots, C_n$, that may differ in computing power: each C_i can execute a unit of work in ρ_i time units—and these n *computing rates* can be very different. All $n+1$ computers intercommunicate across a single network that they access with uniform cost. (Our model is, thus, *node-heterogeneous* and *link-homogeneous*.) The elements of C_0's workload are *divisible,* in the sense that each can be subdivided at will to accommodate the differing computing rates of $\mathcal{C}$'s computers. The Cluster-Exploitation Problem (CEP, for short) is a simple scheduling problem under which C_0 has access to $\mathcal{C}$'s computers for some predetermined "lifespan" of L time units, during which:

1. For each $i \in \{1, \ldots, n\}$, in some order, C_0 sends some "personalized" number, w_i, units of work to each worker computer C_i, in a single message;
2. Each worker computer executes the work it receives and returns its results to C_0, in a single message.

The challenge is to orchestrate the preceding process so that $\mathcal{C}$'s computers collectively complete as much work as possible during the L time units—while ensuring that *at most one intercomputer message is in transit in the network at any step*. A unit of work is *completed* once C_0 has sent it to some C_i, and C_i has executed the unit and returned results to C_0. Within the literature of *divisible load scheduling*, the CEP follows the master-worker paradigm under the "single-port with no overlap" model. We call a schedule for solving the CEP a *worksharing protocol*. The significance of the CEP stems from the demonstration in [2] that the optimal work-production of a cluster $\mathcal{C}$ depends *only* on $\mathcal{C}$'s vector of *computing rates*, $\langle \rho_1, \ldots, \rho_n \rangle$, which we call $\mathcal{C}$'s *heterogeneity profile*.

What is the optimal schedule for solving the CEP on an n-computer cluster $\mathcal{C}$? We cite a hyperbolic instance of the CEP to garner some intuition. For convenience, let us index $\mathcal{C}$'s computers in nonincreasing order of speed, so that $\rho_1 \leq \cdots \leq \rho_n$. (Recall that each ρ_i is the time to complete one unit of work, so a smaller ρ-value means a faster computer.) Now (here's the hyperbole) say that $\mathcal{C}$'s computers are *very* different in speed: each C_i is 10^{10} times faster than C_{i-1}: formally, $\rho_{i+1} = 10^{10}\rho_i$. It is "intuitively obvious" that the optimal solution to this instance of the CEP is for the master C_0 to proceed as follows:

1. Saturate C_1 ($\mathcal{C}$'s fastest computer) with work that takes it L time units to complete.
2. Recursively solve the CEP for $C_2, \ldots, C_n$, for the lifespan determined by the portion of the L time units when neither C_n's work nor its results are in transit.

This "obviously optimal" *greedy* solution embodies the *LIFO* protocol (with workers served in order of speed): $\mathcal{C}$'s computers are orchestrated to finish working (and return results to C_0) in the *opposite* of the order in which they are served. The first surprise concerning the CEP appeared in [10], where it was shown that the LIFO protocol *does not* solve the CEP optimally. The second surprise was the demonstration in [2] that, over sufficiently long lifespans, the *FIFO* protocol, which has $\mathcal{C}$'s computers finish working (and return results to C_0) in the *same* order as they start, *does* solve the CEP optimally.[2] (The non-idle intervals in Fig. 1 suggest the origin of the names "LIFO," "FIFO.")

The results in [2,10] apparently lessen the importance of the LIFO protocol—but this view may be shortsighted. This paper revisits the LIFO protocol and shows it to have

[2] Simulations in [1] suggest that "sufficiently long" lifespans have quite modest lengths.

advantages that may make it an attractive protocol for the CEP—even though it is not optimal. We show that, in addition to having a simple recursive structure, which makes the protocol easy to specify, implement, and analyze:

> *The LIFO protocol is approximately as powerful as the FIFO protocol in solving the* CEP (Theorem 4).
> Specifically, for every cluster $\mathcal{C}$, there is a fixed fraction $\varphi_{\mathcal{C}} > 0$ *that does not depend on how heterogeneous cluster* $\mathcal{C}$ *is* (as measured by the relative speeds of its fastest and slowest computers) such that $\mathcal{C}$ completes at least the fraction $\varphi_{\mathcal{C}}$ as much work under the LIFO protocol as under the optimal FIFO protocol.

On the road to this result, we uncover a rather surprising property of the LIFO protocol.

> *The LIFO protocol's work production is independent of the order in which the master* C_0 *supplies work to the workers in cluster* $\mathcal{C}$ (Theorem 3).
> Even in our extreme example, $\mathcal{C}$'s computers complete the same amount of work when the slowest worker (C_n) is allocated the longest time slot as when the fastest one (C_1) is. (This independence can be derived from results in [4], but the proof we present is quite elegant and may have further application.)

Related work. Employing a model that is very similar to ours, [3] derives efficient optimal or near-optimal schedules for the four variants of the CEP for clusters $\mathcal{C}$ that correspond to the four paired answers to the questions: "Do tasks produce nontrivial-size results?" "Is $\mathcal{C}$'s network pipelined?" For those variants that are **NP**-Hard, *near*-optimality is the most that one can expect to achieve efficiently—and this is what [3] achieves. More details on this and related work are available in the survey [11]. One finds in [12] a study of heterogeneity in computing that is based on the fact (from [2]) that optimal solutions to the CEP for a cluster $\mathcal{C}$ depend only on $\mathcal{C}$'s heterogeneity profile; this study explores features of $\mathcal{C}$'s profile that determine its work-completion rate and that give one cluster a higher rate than another. A variant of the CEP in which clusters are *node-homogeneous* but *link-heterogeneous* is studied in [5]; in that setting, the FIFO protocol loses its advantage over the LIFO protocol: neither protocol dominates the other. Less directly related to our study is the large body of work that studies the scheduling of "divisible workloads." While parts of sources such as [6,7,9] and their kin share our interest in the CEP, their focus on tasks that do not produce measurable output that must be returned to the master allows much simpler algorithmics; e.g., the FIFO and LIFO protocols coincide in their model.

2 Formal Details

We adapt the model of [8], which is the basis of [2,5,10,12].

The computing environment. The *master* computer C_0's workload is composed of work units that are identical in size and complexity.[3] *The tasks' (common) complexity can be an arbitrary function of their (common) size.* Our model posits that the cost of transmitting work grows *linearly* with the total amount of work performed. This allows us to *measure both time and message-length in the same units as work.*

[3] "Size" refers to specification length, "complexity" to computation time.

This linear communication model ignores *fixed* transmission costs—the end-to-end latency of a message's first packet and the per-message set-up overhead—because their impacts fade over long lifespans. Thus, we replace the *affine* communication model of [2,8] with a *linear* model. We justify this simplification via two facts that hold asymptotically, i.e., over "sufficiently long life-spans." (a) For the CEP, the linear and affine models coincide asymptotically. (b) The optimality result from [2] that motivates the current study (cited as our Theorem 1) holds only asymptotically.

For $i \in \{1, \ldots, n\}$, a *worker* computer C_i that belongs to the cluster $\mathcal{C}$ of interest can execute one unit of work in ρ_i time units; this ρ-value is C_i's *computing rate*. For convenience, we normalize the computing rates of $\mathcal{C}$'s computers, so that if $\mathcal{C}$'s *(heterogeneity) profile* is $\langle \rho_1, \ldots, \rho_n \rangle$, then for each $i \in \{1, \ldots, n\}$, $0 < \rho_i \leq 1$. (Recall: *A smaller rate means a faster computer.*) We posit a uniform communication fabric for all computers: The time to send a single packet either from the master C_0 to some worker C_i or from C_i to C_0 is τ time units.[4] Within the context of the CEP, every intercomputer message is either a work-allocation that C_0 sends to some $C_i \in \mathcal{C}$ or the results of executed work that C_i sends to C_0. We posit that *each unit of work produces* $0 < \delta < 1$ *units of results*. The entire L-time-unit "exploitation" episode must be orchestrated so that *at most one intercomputer message is in transit in the network at a time*. Before any C_i sends a message of length ℓ to another C_j, C_i *packages* the message, at a cost of $\pi_i \ell$ time-units; symmetrically, when C_j receives the message, it *unpackages* it, at a cost of $\overline{\pi}_i \ell$ time-units. (Packaging a message could be as computationally "lightweight" as packetizing and compressing it or as "heavyweight" as encoding it.)

Worksharing protocols. When there is only *one* $C_i \in \mathcal{C}$, C_0 shares w units of work with C_i via the process summarized in the following schematic time-line of worksharing with one worker computer (not to scale).

C_0 packages work for C_i	work is in transit	C_i unpackages the work	C_i computes the work	C_i packages its results	results are in transit	C_0 unpackages the results
$\pi_0 w$	τw	$\overline{\pi}_i w$	$\rho_i w$	$\pi_i \delta w$	$\tau \delta w$	$\overline{\pi}_0 \delta w$

When there are *many* (i.e., more than one) $C_i \in \mathcal{C}$, we use two ordinal-indexing schemes for $\mathcal{C}$'s computers to help orchestrate communications while solving the CEP. The *startup order* specifies the order in which C_0 transmits work within $\mathcal{C}$; it labels the workers $C_{s_1}, \ldots, C_{s_n}$, to indicate that C_{s_i} receives work—hence, begins working—before $C_{s_{i+1}}$. Dually, the *finishing order* labels the workers $C_{f_1}, \ldots, C_{f_n}$, to specify the order in which they return their results to C_0. Protocols proceed as follows.

1. *Transmit work.* C_0 prepares and transmits w_{s_1} units of work for C_{s_1}. It immediately prepares and sends w_{s_2} units of work to C_{s_2} via the same process. Continuing thus, C_0 supplies the C_{s_i} with w_{s_i} units of work seriatim—with no intervening gaps.
2. *Compute.* Upon receiving work from C_0, C_i unpackages and performs the work.
3. *Transmit results.* As soon as C_i completes its work, it packages its results and transmits them to C_0.

[4] We find the transit rate τ a more convenient cost measure than its reciprocal, bandwidth.

We choose the work-allocations $\{w_i\}_{i=1}^{n}$ so that, with no gaps, $\mathcal{C}$'s computers:
- receive work and compute in the startup order $\Sigma = \langle s_1, \ldots, s_n \rangle$;
- complete work and transmit results in the finishing order $\Phi = \langle f_1, \ldots, f_n \rangle$;
- complete all work and communications by time L.

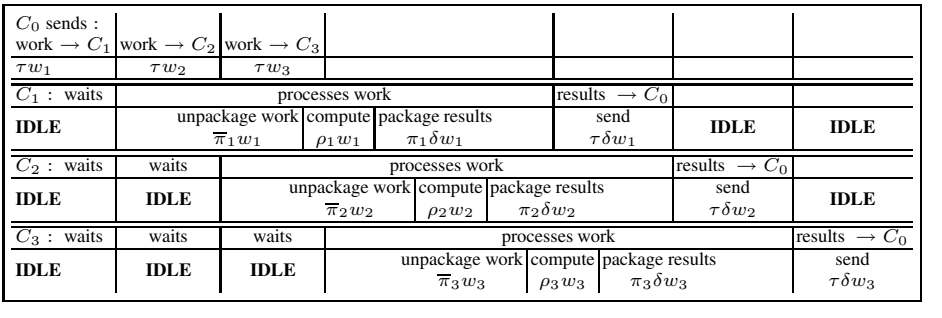
Our goal is to maximize $\mathcal{C}$'s *aggregate completed work*, $\mathcal{W}(\mathcal{C}; L) \stackrel{\text{def}}{=} w_1 + \cdots + w_n$.

In depicting and analyzing multiworker protocols, we have all computing by the master C_0—i.e., its packaging work-allocations for $\mathcal{C}$'s workers and unpackaging their results—take place *offline*, so that we focus solely on a worksharing episode as it appears to the workers. Although this choice differs from that in [2], one verifies easily that *all qualitative conclusions in [2]*—notably the optimality of FIFO protocols (cited as our Theorem 1)—*are independent of this choice*.

The timelines for two instantiations of the generic multiworker worksharing protocol appear in Fig. 1. (*To save space while preserving legibility we have each* $s_i \equiv i$.) In the top protocol, Σ and Φ *coincide:* $(\forall i)[f_i = s_i]$, which specifies the (optimal) *FIFO* protocol. In the bottom one, Σ and Φ *are reversed:* $(\forall i)[f_i = s_{n-i+1}]$, which specifies the *LIFO* protocol. Neither relationship is true of general protocols; cf. [2].

The following abbreviations enhance the legibility of complicated expressions.
- For $1 \le i \le n$, $\mathsf{R}_i \stackrel{\text{def}}{=} \overline{\pi}_i + \rho_i + \delta\pi_i$
  is the *effective computing rate* of computer C_i per work unit, i.e., the "round-trip" time-cost for [work-unpackaging + work-performing + result-packaging]
- $\widetilde{\tau} \stackrel{\text{def}}{=} (1+\delta)\tau$
  is the (common) per-unit "round-trip" communication rate for each computer.

Henceforth, we focus on a cluster $\mathcal{C}$ *with* effective *(heterogeneity) profile* $\langle \mathsf{R}_1, \ldots, \mathsf{R}_n \rangle$.



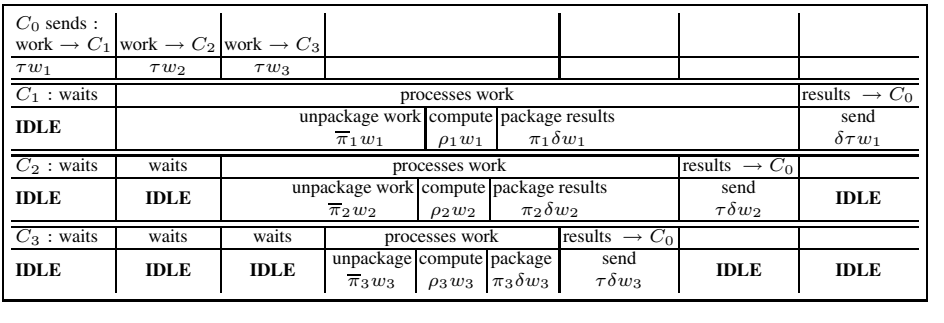


Fig. 1. Time-lines of 3-worker FIFO (top) and LIFO (bottom) protocols (not to scale)

3 Work Production under the LIFO and FIFO Protocols

We invoke the observation from [2] that the work production of any n-computer cluster $\mathcal{C}$ under any worksharing protocol can be calculated by solving a system of n linear equations in n unknowns (the w_i). Assuming that $\mathcal{C}$'s computers are served in the startup order $s_1, \ldots, s_n$, the asymptotic[5] work-allocations to its computers under a given worksharing protocol P, denoted $w_{s_1}^{(\mathsf{P})}, \ldots, w_{s_n}^{(\mathsf{P})}$, are specified by the following system.

$$\mathsf{C}^{(\mathsf{P})} \cdot \begin{pmatrix} w_{s_1}^{(\mathsf{P})} \\ w_{s_2}^{(\mathsf{P})} \\ \vdots \\ w_{s_n}^{(\mathsf{P})} \end{pmatrix} = \begin{pmatrix} L \\ L \\ \vdots \\ L \end{pmatrix}; \tag{1}$$

$\mathsf{C}^{(\mathsf{P})}$ is the *coefficient matrix* that specifies the details of protocol P. One can "read off" the coefficient matrices for the LIFO protocol L and the FIFO protocol F from Fig. 1:

$$\mathsf{C}^{(\mathsf{L})} = \begin{pmatrix} \mathsf{R}_{s_1} + \widetilde{\tau} & 0 & \cdots & 0 \\ \widetilde{\tau} & \mathsf{R}_{s_2} + \widetilde{\tau} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \widetilde{\tau} & \widetilde{\tau} & \cdots & \mathsf{R}_{s_n} + \widetilde{\tau} \end{pmatrix}; \quad \mathsf{C}^{(\mathsf{F})} = \begin{pmatrix} \mathsf{R}_{s_1} + \widetilde{\tau} & \delta\tau & \cdots & \delta\tau \\ \tau & \mathsf{R}_{s_2} + \widetilde{\tau} & \cdots & \delta\tau \\ \vdots & \vdots & \ddots & \vdots \\ \tau & \tau & \cdots & \mathsf{R}_{s_n} + \widetilde{\tau} \end{pmatrix} \tag{2}$$

One can solve system (1) instantiated with the coefficients from (2) to determine the work-allocations $\{w_{s_i}^{(\mathsf{L})}\}_{i=1}^n$ and $\{w_{s_i}^{(\mathsf{F})}\}_{i=1}^n$, hence also the resulting amounts of aggregate completed work, $\mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L) = \sum_i w_{s_i}^{(\mathsf{L})}$ and $\mathcal{W}^{(\mathsf{F})}(\mathcal{C}; L) = \sum_i w_{s_i}^{(\mathsf{F})}$.

Theorem 1 ([2]). (a) *For any cluster $\mathcal{C}$ and lifespan L,*

$$\mathcal{W}^{(\mathsf{F})}(\mathcal{C}; L) \;=\; \frac{X}{1 + \delta\tau X} \cdot L \quad \textit{where} \quad X \;=\; \sum_{k=1}^{n} \frac{1}{\mathsf{R}_{s_k} + \tau} \cdot \prod_{i=1}^{k-1} \frac{\mathsf{R}_{s_i} + \delta\tau}{\mathsf{R}_{s_i} + \tau}. \tag{3}$$

(b) *No worksharing protocol has greater work production than the FIFO protocol.*

Theorem 2. *For any cluster $\mathcal{C}$ and lifespan L,*

$$\mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L) \;=\; L \cdot \sum_{k=1}^{n} \frac{1}{\mathsf{R}_{s_k} + \widetilde{\tau}} \cdot \prod_{i=1}^{k-1} \frac{\mathsf{R}_{s_i}}{\mathsf{R}_{s_i} + \widetilde{\tau}}. \tag{4}$$

Proof (Sketch). By considering the equation for $w_{s_1}^{(\mathsf{L})}$, plus all pairs of adjacent equations (i.e., equations whose indices have the forms s_i and s_{i+1}), we find that

$$\left[w_{s_1}^{(\mathsf{L})} \;=\; \frac{1}{\mathsf{R}_{s_1} + \widetilde{\tau}} \cdot L \right] \quad \text{and} \quad \left[w_{s_k}^{(\mathsf{L})} \;=\; \frac{\mathsf{R}_{s_{k-1}}}{\mathsf{R}_{s_k} + \widetilde{\tau}} \cdot w_{s_{k-1}}^{(\mathsf{L})} \;\; \text{for } k \in \{2, \ldots, n\} \right] \tag{5}$$

[5] Throughout, *asymptotic* means "as L grows without bound."

By unfolding the recurrent portion of (5), we find explicit expressions for each $w_{s_k}^{(\mathsf{L})}$:

$$w_{s_k}^{(\mathsf{L})} \;=\; \frac{1}{\mathsf{R}_{s_k}+\widetilde{\tau}} \cdot \prod_{i=1}^{k-1} \frac{\mathsf{R}_{s_i}}{\mathsf{R}_{s_i}+\widetilde{\tau}} \cdot L. \qquad \square$$

Theorems 1(a) and 2 specify the work productions of (respectively) the FIFO and LIFO protocols for a given, but unspecified, startup order $s_1, \ldots, s_n$. The fact that the notations $\mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L)$ and $\mathcal{W}^{(\mathsf{F})}(\mathcal{C}; L)$ do not specify this order presages the fact that these quantities are, in fact *independent of startup order*.

Theorem 3 ([2,4]). *When cluster $\mathcal{C}$ is scheduled according to either the LIFO protocol or the FIFO protocol, its aggregate completed work is independent of the order in which $\mathcal{C}$'s computers are served. That is, for all startup orders Σ_1 and Σ_2:*

$$\big[\mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L; \Sigma_1) = \mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L; \Sigma_2)\big] \quad \textit{and} \quad \big[\mathcal{W}^{(\mathsf{F})}(\mathcal{C}; L; \Sigma_1) = \mathcal{W}^{(\mathsf{F})}(\mathcal{C}; L; \Sigma_2)\big].$$

This result appears in [2] for the FIFO protocol and can be derived from results in [4] for the LIFO protocol. We present an alternative proof for the LIFO protocol by emulating the elegant proof strategy used in [12] for the FIFO protocol.

A function $F(x_1, \ldots, x_n)$ is symmetric if its value is unchanged by every reordering of values for its variables. When $n = 3$, for instance, we must have

$$F(a,b,c) = F(a,c,b) = F(b,a,c) = F(b,c,a) = F(c,a,b) = F(c,b,a)$$

for all values a, b, c for the variables x_1, x_2, x_3. For integers $n > 1$ and $k \in \{1, \ldots n\}$, $F_k^{(n)}$ denotes the *multilinear* symmetric function[6] that has n variables grouped as products of k variables. The four functions $F_k^{(4)}$ of R-values appear in the following table.

$$\begin{array}{l}
F_1^{(4)}(\mathsf{R}_1, \mathsf{R}_2, \mathsf{R}_3, \mathsf{R}_4) = \mathsf{R}_1 + \mathsf{R}_2 + \mathsf{R}_3 + \mathsf{R}_4 \\
F_2^{(4)}(\mathsf{R}_1, \mathsf{R}_2, \mathsf{R}_3, \mathsf{R}_4) = \mathsf{R}_1\mathsf{R}_2 + \mathsf{R}_1\mathsf{R}_3 + \mathsf{R}_1\mathsf{R}_4 + \mathsf{R}_2\mathsf{R}_3 + \mathsf{R}_2\mathsf{R}_4 + \mathsf{R}_3\mathsf{R}_4 \\
F_3^{(4)}(\mathsf{R}_1, \mathsf{R}_2, \mathsf{R}_3, \mathsf{R}_4) = \mathsf{R}_1\mathsf{R}_2\mathsf{R}_3 + \mathsf{R}_1\mathsf{R}_2\mathsf{R}_4 + \mathsf{R}_1\mathsf{R}_3\mathsf{R}_4 + \mathsf{R}_2\mathsf{R}_3\mathsf{R}_4 \\
F_4^{(4)}(\mathsf{R}_1, \mathsf{R}_2, \mathsf{R}_3, \mathsf{R}_4) = \mathsf{R}_1\mathsf{R}_2\mathsf{R}_3\mathsf{R}_4
\end{array}$$

Two notational simplifications will enhance legibility. (1) We allow k to assume the value 0 and set $F_0^{(n)} \equiv 1$. (2) Because our arguments to the functions $F_i^{(n)}$ are always $\mathsf{R}_1, \ldots, \mathsf{R}_n$, we abbreviate "$F_i^{(n)}(\mathsf{R}_1, \ldots, \mathsf{R}_n)$" by "$F_i^{(n)}$."

Theorem 3 is immediate from the following lemma.

Lemma 1. *For all lifespans L:*

$$\mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L) \;=\; L \cdot \frac{F_{n-1}^{(n)} + F_{n-2}^{(n)}\widetilde{\tau} + \cdots + F_1^{(n)}\widetilde{\tau}^{n-2} + F_0^{(n)}\widetilde{\tau}^{n-1}}{(\mathsf{R}_1+\widetilde{\tau}) \times (\mathsf{R}_2+\widetilde{\tau}) \times \cdots \times (\mathsf{R}_{n-1}+\widetilde{\tau}) \times (\mathsf{R}_n+\widetilde{\tau})}. \tag{6}$$

Thus, $\mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L)$ is symmetric *in the effective computing rates, $\mathsf{R}_1, \ldots, \mathsf{R}_n$.*

Proof. We proceed by induction on the sizes (i.e., numbers of computers) of clusters. To aid legibility, we embellish the "name" of each cluster $\mathcal{C}$ with a subscript that specifies

[6] The qualifier "multilinear" tells us that no variable occurs to a power > 1; this excludes symmetric functions such as $F(a,b) = a^2b + ab^2$.

its size. The notational convention is that the n-computer cluster $\mathcal{C}_n$ has effective profile $\langle \mathsf{R}_1, \ldots, \mathsf{R}_n \rangle$. As the base of our induction, we note from equation (4) that

$$\mathcal{W}^{(\mathsf{L})}(\mathcal{C}_1; L) = \frac{1}{(\mathsf{R}_1 + \widetilde{\tau})} = \frac{F_0^{(1)}}{(\mathsf{R}_1 + \widetilde{\tau})};$$

$$\mathcal{W}^{(\mathsf{L})}(\mathcal{C}_2; L) = \frac{1}{(\mathsf{R}_1 + \widetilde{\tau})} + \frac{1}{(\mathsf{R}_2 + \widetilde{\tau})} \cdot \frac{\mathsf{R}_1}{(\mathsf{R}_1 + \widetilde{\tau})} = \frac{F_1^{(2)} + F_0^{(2)}\widetilde{\tau}}{(\mathsf{R}_1 + \widetilde{\tau}) \times (\mathsf{R}_2 + \widetilde{\tau})}.$$

Now assume, for induction, that (6) holds for all cluster sizes up through n. Combining our two specifications of $\mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L)$, viz., equations (6) and (4), we then have

$$\begin{aligned}\mathcal{W}^{(\mathsf{L})}(\mathcal{C}_{n+1}; L) &= \mathcal{W}^{(\mathsf{L})}(\mathcal{C}_n; L) + \frac{1}{\mathsf{R}_{n+1} + \widetilde{\tau}} \cdot \prod_{i=1}^{n} \frac{\mathsf{R}_i}{\mathsf{R}_i + \widetilde{\tau}} \\ &= \frac{\left(F_{n-1}^{(n)} + F_{n-2}^{(n)}\widetilde{\tau} + \cdots + F_0^{(n)}\widetilde{\tau}^{n-1}\right)(\mathsf{R}_{n+1} + \widetilde{\tau}) + \prod_{i=1}^{n} \mathsf{R}_i}{(\mathsf{R}_1 + \widetilde{\tau}) \times (\mathsf{R}_2 + \widetilde{\tau}) \times \cdots \times (\mathsf{R}_n + \widetilde{\tau}) \times (\mathsf{R}_{n+1} + \widetilde{\tau})}.\end{aligned}$$

The proof is now completed by invoking the following easily verified identities:

$$\text{For all } i \in \{1, \ldots, n\}, \quad \mathsf{R}_{n+1}F_{n-i}^{(n)} + F_{n-i+1}^{(n)} = F_{n-i+1}^{(n+1)}.$$

In verifying these identities, recall that $\prod_{i=1}^{n} \mathsf{R}_i = F_n^{(n)}$. In applying these identities, note the role of the $\widetilde{\tau}$ in the induction-extending factor $(\mathsf{R}_{n+1} + \widetilde{\tau})$. □

A conjecture. Olivier Beaumont has reported, in personal communication, that informal simulations have failed to turn up any other worksharing protocol that shares the LIFO and FIFO protocols' independence from the order in which cluster $\mathcal{C}$'s computers are supplied with work. It is intriguing to conjecture that these two protocols are, in fact, unique in this independence. A plausible place to start trying to prove this is to exploit the fact that the LIFO and FIFO protocols are the only ones whose coefficient matrices—cf. (2)—have upper triangles and lower triangles that are all constant. (This fact about the matrices can be verified from the development in [2].)

Theorem 3 combines with equations (3) and (4) to reveal the following simple but consequential facts. The first fact is that *LIFO and FIFO protocols complete more work on faster clusters.*

Proposition 1. *Let clusters $\mathcal{C}$ and $\mathcal{C}'$ have respective profiles $\langle \mathsf{R}_1, \ldots, \mathsf{R}_i, \ldots, \mathsf{R}_n \rangle$ and $\langle \mathsf{R}_1, \ldots, \mathsf{R}'_i, \ldots, \mathsf{R}_n \rangle$. If $\mathsf{R}'_i < \mathsf{R}_i$,[7] then for all L:*

$$\left[\mathcal{W}^{(\mathsf{L})}(\mathcal{C}'; L) > \mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L)\right] \text{ and } \left[\mathcal{W}^{(\mathsf{F})}(\mathcal{C}'; L) > \mathcal{W}^{(\mathsf{F})}(\mathcal{C}; L)\right].$$

Proof. The proof for the FIFO protocol appears in [12], so we focus on the LIFO protocol. We refine equation (4) to specifies the startup order Σ. (We actually already do this in the righthand expression in (4).) We choose any startup order Σ for $\mathcal{C}$, for which $s_n = i$; i.e., Σ has the form $\Sigma = \langle s_1, \ldots, s_{n-1}, i \rangle$. We then form the versions of

[7] Recall: the indicated inequality means that $\mathcal{C}'$'s ith computer is *faster* than cluster $\mathcal{C}$'s.

equation (4) that use startup order Σ with both of the indicated profiles. To enhance perspicuity, we write these versions in a way that emphasizes the fact that $\mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L; \Sigma)$ and $\mathcal{W}^{(\mathsf{L})}(\mathcal{C}'; L; \Sigma)$ differ only in their first terms.

$$\mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L; \Sigma) = \left(\frac{1}{\mathsf{R}_{s_n} + \widetilde{\tau}} \cdot \prod_{i=1}^{n-1} \frac{\mathsf{R}_{s_i}}{\mathsf{R}_{s_i} + \widetilde{\tau}} + \sum_{k=1}^{n-1} \frac{1}{\mathsf{R}_{s_k} + \widetilde{\tau}} \cdot \prod_{i=1}^{k-1} \frac{\mathsf{R}_{s_i}}{\mathsf{R}_{s_i} + \widetilde{\tau}} \right) \cdot L$$

$$\mathcal{W}^{(\mathsf{L})}(\mathcal{C}'; L; \Sigma) = \left(\frac{1}{\mathsf{R}'_{s_n} + \widetilde{\tau}} \cdot \prod_{i=1}^{n-1} \frac{\mathsf{R}_{s_i}}{\mathsf{R}_{s_i} + \widetilde{\tau}} + \sum_{k=1}^{n-1} \frac{1}{\mathsf{R}_{s_k} + \widetilde{\tau}} \cdot \prod_{i=1}^{k-1} \frac{\mathsf{R}_{s_i}}{\mathsf{R}_{s_i} + \widetilde{\tau}} \right) \cdot L$$

Because $\mathsf{R}'_{s_n} < \mathsf{R}_{s_n}$, we thereby find that

$$\mathcal{W}^{(\mathsf{L})}(\mathcal{C}'; L; \Sigma) - \mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L; \Sigma) = L \cdot \left(\frac{1}{\mathsf{R}'_{s_n} + \widetilde{\tau}} - \frac{1}{\mathsf{R}_{s_n} + \widetilde{\tau}} \right) \cdot \prod_{i=1}^{n-1} \frac{\mathsf{R}_{s_i}}{\mathsf{R}_{s_i} + \widetilde{\tau}} > 0,$$

so that $\mathcal{W}^{(\mathsf{L})}(\mathcal{C}'; L; \Sigma) > \mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L; \Sigma)$. □

The second fact is that *adding an additional computer to cluster $\mathcal{C}$ increases $\mathcal{C}$'s aggregate completed work under the LIFO protocol*. (It is shown in [4], by example, that this need not be the case with the FIFO protocol.)

Proposition 2. *Let cluster $\mathcal{C}'$ be obtained by adding a $(n+1)$th computer to cluster $\mathcal{C}$. Then $\mathcal{W}^{(\mathsf{L})}(\mathcal{C}'; L) > \mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L)$.*

Proof. Let the new computer, C, have effective computing rate R. Let each of $\mathcal{C}$'s computers retain its starting index within $\mathcal{C}'$, and let us assign C startup index s_{n+1}, so that $\mathsf{R} = \mathsf{R}_{s_{n+1}}$. (This is just for convenience: Theorem 3 tells us that we could assign C any starting order without changing the result.) Invoking equation (4), we find that

$$\mathcal{W}^{(\mathsf{L})}(\mathcal{C}'; L) - \mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L) = \frac{1}{\mathsf{R}_{s_{n+1}} + \widetilde{\tau}} \cdot \prod_{i=1}^{n} \frac{\mathsf{R}_{s_i}}{\mathsf{R}_{s_i} + \widetilde{\tau}}.$$

This difference is positive because each factor in the product is. □

4 The LIFO Protocol Is Approximately Optimal

This section is devoted to proving that every cluster completes a fixed fraction as much work under the LIFO protocol as under the optimal FIFO protocol.

Theorem 4. *The LIFO protocol is "approximately" optimal, in the following sense. For every cluster $\mathcal{C}$, there exists a fixed constant $\varphi_{\mathcal{C}} > 0$ that does not depend on how heterogeneous cluster $\mathcal{C}$ is (as measured by the relative speeds of its fastest and slowest computers) such that $\mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L) > \varphi_{\mathcal{C}} \cdot \mathcal{W}^{(\mathsf{F})}(\mathcal{C}; L)$.*[8]

We prove Theorem 4 by computing a lower bound on $\mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L)$ and an upper bound on $\mathcal{W}^{(\mathsf{F})}(\mathcal{C}; L)$ and comparing the results. We focus on an n-computer cluster $\mathcal{C}$ whose effective heterogeneity profile is $\langle \mathsf{R}_1, \ldots, \mathsf{R}_n \rangle$, where

- $\mathsf{R}^{(\text{slow})} \stackrel{\text{def}}{=} \max\{\mathsf{R}_1, \ldots, \mathsf{R}_n\}$ is the effective rate of $\mathcal{C}$'s *slowest* computer;
- $\mathsf{R}^{(\text{fast})} \stackrel{\text{def}}{=} \min\{\mathsf{R}_1, \ldots, \mathsf{R}_n\}$ is the effective rate of $\mathcal{C}$'s *fastest* computer.

[8] Recall that $\mathcal{W}^{(\mathsf{F})}(\mathcal{C}; L)$ is the most work that cluster $\mathcal{C}$ can complete in time L under *any* worksharing protocol (Theorem 1).

4.1 A Lower Bound on the LIFO Work Production $\mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L)$

Lemma 2. *For every cluster $\mathcal{C}$ and lifespan L, $\mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L) > \dfrac{1}{\mathsf{R}^{(\text{slow})} + \widetilde{\tau}} \cdot L$.*

Proof. By combining equation (4) for $\mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L)$ with the "order-independent" Theorem 3 and the "faster-clusters-are-better" Proposition 1, we find that

$$\mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L) \geq \frac{1}{\mathsf{R}^{(\text{slow})} + \widetilde{\tau}} \sum_{k=0}^{n-1} \left(\frac{\mathsf{R}^{(\text{slow})}}{\mathsf{R}^{(\text{slow})} + \widetilde{\tau}} \right)^k \cdot L = \frac{1}{\widetilde{\tau}} \cdot \left(1 - \left(\frac{\mathsf{R}^{(\text{slow})}}{\mathsf{R}^{(\text{slow})} + \widetilde{\tau}} \right)^n \right) \cdot L \tag{7}$$

We next invoke the "bigger-clusters-are-more-powerful" Proposition 2 to remark that

$$\mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L) \geq \mathcal{W}^{(\mathsf{L})}(\mathcal{C}'; L), \tag{8}$$

where $\mathcal{C}'$ is the two-computer subcluster of $\mathcal{C}$ whose profile is $\langle \mathsf{R}_1, \mathsf{R}_2 \rangle$, and $\mathsf{R}^{(\text{slow})} \in \{\mathsf{R}_1, \mathsf{R}_2\}$. We then combine inequalities (7) and (8) to see that

$$\begin{aligned}
\mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L) \geq \mathcal{W}^{(\mathsf{L})}(\mathcal{C}'; L) &\geq \frac{1}{\widetilde{\tau}} \cdot \left(1 - \left(\frac{\mathsf{R}^{(\text{slow})}}{\mathsf{R}^{(\text{slow})} + \widetilde{\tau}} \right)^2 \right) \cdot L \\
&= \frac{1}{\widetilde{\tau}} \cdot \left(1 - \frac{\mathsf{R}^{(\text{slow})}}{\mathsf{R}^{(\text{slow})} + \widetilde{\tau}} \right) \cdot \left(1 + \frac{\mathsf{R}^{(\text{slow})}}{\mathsf{R}^{(\text{slow})} + \widetilde{\tau}} \right) \cdot L \\
&> \frac{1}{\widetilde{\tau}} \cdot \left(1 - \frac{\mathsf{R}^{(\text{slow})}}{\mathsf{R}^{(\text{slow})} + \widetilde{\tau}} \right) \cdot L \\
&= \frac{1}{\mathsf{R}^{(\text{slow})} + \widetilde{\tau}} \cdot L.
\end{aligned}$$

The lemma follows. □

4.2 An Upper Bound on the FIFO Work Production $\mathcal{W}^{(\mathsf{F})}(\mathcal{C}; L)$

Lemma 3. *For every cluster $\mathcal{C}$ and lifespan L, $\mathcal{W}^{(\mathsf{F})}(\mathcal{C}; L) < \dfrac{1}{\tau} \cdot L$.*

Proof. We simplify the development by replacing cluster $\mathcal{C}$'s characterizing parameters by a composite parameter that bounds its *computation-to-communication complexity:* the ratio $\kappa_{\mathcal{C}} \stackrel{\text{def}}{=} \mathsf{R}^{(\text{fast})}/\tau$. By combining equation (3) for $\mathcal{W}^{(\mathsf{F})}(\mathcal{C}; L)$ with the "faster-clusters-are-better" Proposition 1, we then find that

$$\begin{aligned}
\mathcal{W}^{(\mathsf{F})}(\mathcal{C}; L) &\leq \frac{1}{\tau} \cdot \left(1 - \frac{(1-\delta)(\mathsf{R}^{(\text{fast})} + \delta\tau)^n}{(\mathsf{R}^{(\text{fast})} + \tau)^n - \delta \cdot (\mathsf{R}^{(\text{fast})} + \delta\tau)^n} \right) \cdot L \\
&= \frac{1}{\tau} \cdot \left(1 - \frac{(1-\delta)(\kappa_{\mathcal{C}} + \delta)^n}{(\kappa_{\mathcal{C}} + 1)^n - \delta \cdot (\kappa_{\mathcal{C}} + \delta)^n} \right) \cdot L \\
&= \frac{1}{\tau} \cdot \left(1 - \frac{1-\delta}{\Big((\kappa_{\mathcal{C}} + 1)/(\kappa_{\mathcal{C}} + \delta)\Big)^{\kappa_{\mathcal{C}} + (n - \kappa_{\mathcal{C}})} - \delta} \right) \cdot L.
\end{aligned}$$

We now invoke the classical inequality

$$\left(1 + \frac{x}{m}\right)^m \leq e^x,$$

which holds for all real positive x and m, to observe that

$$\left(\frac{\kappa_{\mathcal{C}}+1}{\kappa_{\mathcal{C}}+\delta}\right)^{\kappa_{\mathcal{C}}} = \left(1+\frac{1-\delta}{\kappa_{\mathcal{C}}+\delta}\right)^{\kappa_{\mathcal{C}}} \leq \left(1+\frac{1-\delta}{\kappa_{\mathcal{C}}+\delta}\right)^{\kappa_{\mathcal{C}}+\delta} \leq e^{1-\delta}$$

This inequality combines with our assumption that $\delta < 1$ to allow us to extend the preceding chain of inequalities on $\mathcal{W}^{(\mathsf{F})}(\mathcal{C}; L)$. We find that

$$\mathcal{W}^{(\mathsf{F})}(\mathcal{C}; L) \leq \frac{1}{\tau} \cdot \left(1 - \frac{1-\delta}{e^{(1-\delta)(n-\kappa_{\mathcal{C}})} - \delta}\right) \cdot L < \frac{1}{\tau} \cdot L.$$

The lemma follows. □

4.3 The LIFO-FIFO Bounding Ratio

Finally, we combine the bounds of Lemmas 2 and 3 to conclude that for all clusters $\mathcal{C}$ and lifespans L,

$$\mathcal{W}^{(\mathsf{L})}(\mathcal{C}; L) > \frac{\tau}{\mathsf{R}^{(\text{slow})} + \widetilde{\tau}} \cdot \mathcal{W}^{(\mathsf{F})}(\mathcal{C}; L).$$

The fraction $\varphi_{\mathcal{C}} = \dfrac{\tau}{\mathsf{R}^{(\text{slow})} + \widetilde{\tau}}$ thus satisfies Theorem 4. □

Clearly, the fraction $\varphi_{\mathcal{C}}$ does not depend on how heterogeneous cluster $\mathcal{C}$ is as measured by the relative speeds of its fastest and slowest computers—as exposed, say, by the size of the ratio $\mathsf{R}^{(\text{slow})}/\mathsf{R}^{(\text{fast})}$.

5 Conclusions

We have exposed unexpected properties of the LIFO worksharing protocol, the structurally attractive and intuitively compelling *greedy* solution to the Cluster Exploitation Problem (CEP). These properties involve both the structure of the LIFO protocol and its behavior, measured via its work-production in the CEP.

In terms of the LIFO protocol's behavior, our main result shows that the protocol's work-production when solving the CEP is at least a fixed constant fraction of optimal (Theorem 4). In view of the ease of specifying and analyzing the LIFO protocol, this result may promote interest in the protocol—and in pursuing analogous performance bounds for other as-yet unanalyzed scheduling heuristics.

In terms of the LIFO protocol's structure, we have shown that a cluster's work-production under the LIFO protocol is independent of the order in which the cluster's computers are supplied with work (Theorem 3). This independence is shared by the optimal FIFO worksharing protocol; we conjecture that it is shared by no protocols other than FIFO and LIFO. This unexpected result joins companions in [2, 10, 12] in reminding us of the subtlety of the phenomenon of heterogeneity in computing—even with respect to as simple a scheduling problem as the CEP.

Future work will attempt to settle the order-independence conjecture and will explore the CEP when work complexity is not linear in work size.

Acknowledgments. It is a pleasure to thank the anonymous referees for their careful reviews, and Olivier Beaumont and Sanjay Rajopadhye for stimulating conversations.

References

1. Adler, M., Gong, Y., Rosenberg, A.L.: Asymptotically Optimal Worksharing in HNOWs: How Long Is 'Sufficiently Long'? In: 36th Annual Simulation Symposium, pp. 39–46 (2003)
2. Adler, M., Gong, Y., Rosenberg, A.L.: On "Exploiting" Node-Heterogeneous Clusters Optimally. Theory of Computing Systems 42, 465–487 (2008)
3. Beaumont, O., Legrand, A., Robert, Y.: The Master-Slave Paradigm with Heterogeneous Computers. IEEE Transactions on Parallel and Distributed Systems 14, 897–908 (2003)
4. Beaumont, O., Marchal, L., Robert, Y.: Scheduling Divisible Loads with Return Messages on Heterogeneous Master-Worker Platforms. In: Bader, D.A., Parashar, M., Sridhar, V., Prasanna, V.K. (eds.) HiPC 2005. LNCS, vol. 3769, pp. 498–507. Springer, Heidelberg (2005)
5. Beaumont, O., Rosenberg, A.L.: Link-Heterogeneity vs. Node-Heterogeneity in Clusters. In: 17th International High-Performance Computing Conference (2010)
6. Bharadwaj, V., Ghose, D., Mani, V.: Optimal Sequencing and Arrangement in Distributed Single-Level Tree Networks. IEEE Transactions on Parallel and Distributed Systems 5, 968–976 (1994)
7. Bharadwaj, V., Ghose, D., Mani, V., Robertazzi, T.G.: Scheduling Divisible Loads in Parallel and Distributed Systems. J. Wiley & Sons, New York (1996)
8. Cappello, F., Fraigniaud, P., Mans, B., Rosenberg, A.L.: An Algorithmic Model for Heterogeneous Clusters: Rationale and Experience. International Journal of Foundations of Computer Science 16, 195–216 (2005)
9. Dutot, P.-F.: Complexity of Master-Slave Tasking on Heterogeneous Trees. European Journal of Operational Research 164, 690–695 (2005)
10. Rosenberg, A.L.: On Sharing Bags of Tasks in Heterogeneous Networks of Workstations: Greedier Is Not Better. In: 3rd IEEE International Conference on Cluster Computing, pp. 124–131 (2001)
11. Rosenberg, A.L.: Changing Challenges for Collaborative Algorithmics. In: Zomaya, A. (ed.) Handbook of Nature-Inspired and Innovative Computing: Integrating Classical Models with Emerging Technologies, pp. 1–44. Springer, New York (2006)
12. Rosenberg, A.L., Chiang, R.C.: Toward Understanding Heterogeneity in Computing. In: 24th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2010 (2010)

Scheduling JavaSymphony Applications on Many-Core Parallel Computers*

Muhammad Aleem, Radu Prodan, and Thomas Fahringer

Institute of Computer Science, University of Innsbruck,
Technikerstraße 21a, A-6020 Innsbruck, Austria
{aleem,radu,tf}@dps.uibk.ac.at

Abstract. JavaSymphony is a Java-based programming and execution environment for programming and scheduling the performance oriented applications on multi-core parallel computers. In this paper, we present a multi-core aware scheduling extension to JavaSymphony capable of mapping parallel applications on large multi-core machines and heterogeneous clusters. JavaSymphony scheduler considers several multi-core specific performance parameters and application types, and uses these parameters to optimise the mapping of JavaSymphony objects and tasks. We evaluate the performance of JavaSymphony scheduler using several real scientific applications and benchmarks on a multi-core shared memory machine and a heterogeneous cluster.

1 Introduction

Multi-core processors [2] add an additional level of parallelism to the existing parallel computers and scheduling parallel applications becomes even more challenging. To speedup applications, a scheduler is required to consider the application properties (e.g., communication and computation needs) and architectural characteristics (e.g., network and memory latencies, heterogeneity of machines, memory hierarchies, machine load). Today, there are many research efforts [3,4,6,8,9], which target application scheduling on multi-core parallel computers. Some of them [3,6,8], however, either consider at most one architectural characteristic or they [9] are limited to a specific parallel computing architecture (e.g., shared or distributed memory computers). We extend our Java-based parallel programming paradigm with a scheduler capable of scheduling jobs on multi-core parallel computers. To the best of our knowledge, we are the one who provides a scheduler for Java applications which considers the low level multi-core specific characteristics (e.g., network and memory latencies, bandwidth, processor speed, shared cache, machine load).

In previous work [1], we developed JavaSymphony (JS) as a Java-based programming paradigm for parallel and distributed infrastructures such as shared

* This research is partially funded by the "Tiroler Zukunftsstiftung", Project name: "Parallel Computing with Java for Manycore Computers".

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 167–179, 2011.

memory multi-cores and heterogeneous clusters. JS's design is based on the concept of dynamic virtual architecture, which allows the programmer to fully define a hierarchical structure of the underlying computing resources (e.g., cores, processors, machines, and clusters) and to control load balancing and locality.

A main drawback of the JS's design is the fact that the mapping of objects and tasks to computing cores has to be performed manually by the programmer.

To fill this gap, we extend JS with a scheduler based on a non-preemptive static scheduling algorithm capable of mapping the JS parallel applications (e.g., shared, distributed, and hybrid memory applications with high degree of regularity). Many architecture specific factors (e.g., processor speed, memory and network latencies, resource sharing, and machine load) influence the performance of a parallel application. Considering alone the target architecture is not sufficient to determine the sensitivity of a performance factor. The application class (e.g., communication and computation needs) and the architectural features collectively determine the performance sensitivity of a factor. Therefore, we propose a method based on training experiments that determines the sensitivities of performance factors with respect to the application classes (e.g., communication and computation-intensive) and multi-core architectures (e.g., shared memory machines, heterogeneous clusters). The training data consists of sorted lists of Performance Factor (PF), which is used by the JS scheduler as guidelines.

The paper is organised as follows. Next section discusses the related work. Section 3 presents the JS overview. Section 4 presents the JS scheduler, including its architecture, methodlogy, and algorithm. Section 5 presents experimental results and section 6 concludes the paper.

2 Related Work

Jcluster [8] is a Java-based message passing parallel environment. It provides a load balancing task scheduler based on transitive random stealing algorithm. The scheduler allows the idle nodes to steal tasks from the busy nodes. In contrast to the JS, they consider only the load balancing factor on clusters.

Proactive [3] is a Java-based parallel environment providing high-level programming abstractions based on the concept of active objects. Alongside programming, Proactive provides deployment-level abstractions for applications on multi-core machines, clusters, and Grids. In contrast to the JS, Proactive does not provide functionality to map an active object to a specific core or processor.

Parallel Java [6] is a Java-based programming environment. It provides programming constructs similar to the OpenMP and MPI. Parallel Java's scheduler keeps track of the busy and idle nodes in a cluster, and schedules the jobs by selecting one of the idle nodes. In contrast to the JS, they only consider the availability of nodes (free machines) as the main scheduling criteria.

In [9] the authors studied the impact of the shared resource contention on the application performance. To avoid the shared resource contention, they proposed a scheduling algorithm which allocates jobs in order to balance the cores' cache miss rates. In contrast to our approach, their scheduler only considers shared memory multi-cores and does not schedule applications on clusters.

In [4], the authors presented an energy-aware scheduling algorithm for heterogeneous multi-core machines. They use profiling to collect the different characteristics of a parallel program and then fuzzy logic is applied to estimate the suitability among program characteristics and cores. In contrast to our approach, they do not consider several important performance-sensitive factors such as processor computing power, co-scheduling of threads, and latencies.

Most of the related work, either considers few multi-core characteristics or they are limited to the specific target architectures. To the best of our knowledge, no Java-based scheduler considers the low-level multi-core and application related characteristics.

3 JavaSymphony

JavaSymphony is a Java-based programming paradigm for developing parallel and distributed applications. JS's high-level constructs abstract low-level infrastructure details and simplify the tasks of controlling parallelism and locality. It offers a unified solution for user-controlled locality-aware mapping of objects and tasks on shared and distributed memory architectures. Here, we provide an overview of some of the JS features, while complete details can be found in [1].

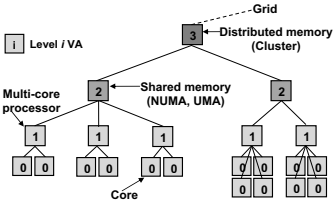


Fig. 1. Four-level VA

JS's design is based on the concept of the dynamic Virtual Architecture (VA) [1]. A VA allows the programmer to define the structure of heterogeneous computing resources and to control mapping, load balancing and migration of objects. Most existing work [3,6] assumes a flat hierarchy of computing resources. In contrast, JS allows the programmer to fully specify the multi-core architectures [1] by defining as a tree structure, where each VA element has a certain level representing a specific resource granularity. Figure 1 depicts a four-level VA representing a heterogeneous cluster.

Writing a parallel JS application requires encapsulating Java objects into so called *JS objects*, which are distributed and mapped onto the hierarchical VA nodes (levels 0 to n). The object agent system [1], a part of JS runtime environment, processes remote as well as local shared memory jobs. It is responsible for creating jobs, mapping objects to VAs, migrating, and releasing objects.

Previously, the JS programmer was responsible to create the required VAs and to manually map the objects and tasks onto the VA nodes which we plan to automatise by developing a scheduler that automatically creates the required VAs and manages the mappings of the JS objects and tasks.

4 JavaSymphony Scheduler

JavaSymphony scheduler is a multi-core aware scheduler that operates in a two stages. In stage-1, training experiments are conducted offline to study the performance impacts of the different factors (e.g., processor speed, memory/network

latencies, bandwidth, co-scheduling) with respect to the two architectures and application classes (for simplicity). In stage-2, the JS scheduler uses the collected training data as guidelines to optimise the selection of the target computing resources (e.g., machines, processors, and cores). For the training and the validation experiments, we use several applications which we classify in two classes (e.g., compute-intensive and the communication-intensive). The classification of the applications is performed by measuring the computational needs of the applications, Section 5.1 describes in detail the classification experiment and the related results.

4.1 System Architecture

Figure 2 shows the JS system architecture. The JS runtime [1] is an agent-based system and has two main components: object and network agent system. The object agent system has two components: the Public Object Agent (PubOA, one for each machine), and the Application Object Agent (AppOA, one for each JS application). The network agent system monitors and interacts with the corresponding multi-core machine.

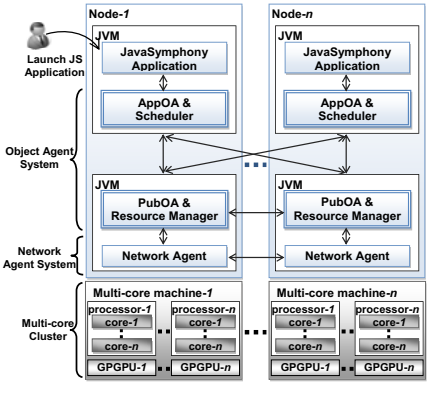


Fig. 2. The JS System Architecture

The JS scheduler has two modules: resource manager and scheduler. The resource manager is part of the PubOA and interacts with the multi-core machine with the help of the network agent. The resource manager acquires and keeps track of the physical computing resources (e.g., cores, processors, and multi-core machine) and collects the machine related information: network and memory latencies, memory hierarchies and bandwidth and processor details (e.g., topology, speed). The scheduler is part of the AppOA and runs along with the executing JS application. The scheduler uses the information provided by the resource manager and the programmer (in form of PF lists) to sub-optimally schedule the JS objects and tasks on the multi-core resources.

4.2 Scheduling Methodology

The JS scheduler considers following performance factors together with two application classes (e.g., compute and communication intensive) to determine the performance sensitivities for the factors.

1. *`Network latency`*: The amount of time required by a message to travel from one machine to another in a cluster.
2. *`Memory latency`*: The time delay which occurs for a message to travel from a main memory module to a processor (e.g., NUMA latencies).

3. *Bandwidth*: The amount of data transferred (in a second), from one machine to another in a cluster (network bandwidth) or from a memory module to a processor (memory bandwidth, such as in NUMA-based machines).
4. *Co-scheduling*: Co-scheduling of the parallel threads is achieved by mapping n threads on a multi-core processor (having n cores and shared last level cache). The not-co-scheduled execution is obtained by mapping threads on different processors (avoiding the sharing of the last level cache). The co-scheduling ratio (cos) shows the performance improvements or degradations and is calculated as follows: $cos = \frac{T(p)_{not-co-scheduled}}{T(p)_{co-scheduled}}$, where $T(p)$ is the parallel execution time of the application. The *cos* value greater than 1 shows improved, less than 1 shows degraded, and the *cos* value 1 shows no change in the performance of the application.
5. *Machine load*: The number of parallel tasks mapped on a multi-core machine. In this work, we only consider the load during the computational stage (excluding the external load). It plays significant role in the performance of the hybrid memory applications. If a machine's load is not balanced, then the over-loaded machines will face more contention on the shared resources.
6. *Processor speed*: The processor speed represents the computing power (clock frequency) of a processor.

In stage-1, the training experiments are conducted to determine the significance of the above mentioned factors for the available multi-core architectures and application classes (e.g., communication and computation intensive). We manually search a sub-optimal VA node (e.g., core, processor, or machine) by considering a performance factor and map the JS tasks onto the selected VA node. For example, to search a sub-optimal VA node with respect to the *processor speed* factor, we select the fastest available resources (e.g., cores, processors, and machines) and manually map the JS tasks onto the selected VA nodes. Using this methodology, we collect the performance impact data for all the factors with respect to the two parallel architectures and the application classes.

In stage-2, the JS scheduler utilises the collected training data (in the form of PF lists) and makes the scheduling decisions using that. To schedule a JS application on a parallel architecture, the scheduler requires the training data for the similar application class on the target parallel architecture. The JS scheduler uses a repetitive optimisation method (to find a sub-optimal VA node) by considering all the performance factors in the PF list. After optimising with respect to all the factors, the scheduler selects one of the VA node (e.g., core, processor, or machine) and maps the task onto the selected resource.

4.3 Algorithm

A JS application consists of two schedule entities: the coarse-grained JS objects and the fine-grained JS tasks. Therefore, we designed the scheduling algorithm to operate in two phases: in phase-1 the JS objects are scheduled and in phase-2 the JS tasks.

Algorithm 1. JavaSymphony Scheduler (main part)

```
   Input: AppPrgMdl,AppClass,Arch,JSObjectQ,JSTaskQ,Rlist
   Output: JS scheduled application
 1 begin
 2   while true do
 3     phase ⟵ 1;                                   /* phase-1 (object) scheduling */
 4     while JSObjectQ ≠ ∅ do
 5       Object obj ⟵ pop(JSObjectQ);                        /* get next JS object */
 6       VA v ⟵ GetOptNode(phase, AppClass, Arch, AppPrgMdl);
 7       Schedule(obj, v);                        /* map JS object obj to VA node v */
 8       UpdateResources(Rlist, obj, v);
 9     end
10     phase ⟵ 2;                                    /* phase-2 (task) scheduling */
11     while JSTaskQ ≠ ∅ do
12       Task tsk ⟵ pop(JSTaskQ);                          /* get next JS task */
13       VA v ⟵ GetOptNode(phase, AppClass, Arch, AppPrgMdl);
14       Schedule(tsk, v);                        /* map JS task tsk to VA node v */
15       UpdateResources(Rlist, tsk, v);
16     end
17   end
18 end
```

Algorithm 1 shows the main scheduling algorithm. First, the input data items are declared: *AppPrgMdl* (JS application's programming model e.g., shared, distributed, or hybrid), *AppClass* (compute-/communication-intensive), *Arch* (target architecture e.g., shared or distributed memory computer), *JSObjectQ* (JS object queue), *JSTaskQ* (JS task queue), and *Rlist* (resource status). In line 2, main scheduling loop starts. First, the scheduling phase is updated (line 3) and phase-1 scheduling starts (line 4). In line 5, the object *obj* is extracted from the object queue. Then, *GetOptNode* method (Algorithm 2) is invoked (line 6) which returns a sub-optimal VA node v, by considering the scheduling phase (*phase*), the application class (*AppClass*), the architecture (*Arch*), and the programming model (*AppPrgMdl*). Then, the *obj* object is mapped to the VA node v (line 7). Afterwards, the mapping details are passed to the resource manager (line 8). In the phase-2, the JS tasks are scheduled (lines $10-16$). First, the scheduling phase is updated (line 10) and the phase-2 scheduling starts at line 11. A task *tsk* is extracted from the task queue (line 12). Then, the *GetOptNode* method is invoked (line 13) that returns a sub-optimal VA node v, by considering the scheduling phase, application class, architecture, and programming model. Then, the task *tsk* is mapped to the VA node v (line 14) and the mapping details are passed to the resource manager (line 15).

Algorithm 2 shows the *GetOptNode* method. First, a VA node v is created (line 2). The existing VA nodes (e.g., cores, processors, machines, and cluster) are acquired in *vaNodes* (line 3). The performance factors list (*PFlist*) is read (line 4). For the object scheduling, the *getAppVaNode* method is invoked with parameters: the programming model, the application class, the architecture, all VA nodes, and the PFlist (lines $5-6$). The *getAppVaNode* method returns a sub-optimal VA node by considering the application programming model (e.g., shared, distributed, or hybrid memory). For example, if a hybrid memory JS application is scheduled, then it returns a VA node representing a multi-core

Algorithm 2. JavaSymphony Scheduler - *GetOptNode* method

```
   Input: AppPrgMdl,AppClass,Arch,phase
   Output: VA v
 1 begin
 2     VA v ⟵ ∅;
 3     VA[ ] vaNodes ⟵ ReadSystemVaNodes();                    /* read all VA nodes */
 4     Vector PFlist ⟵ ReadSystemPFList(Arch, AppClass);
 5     if phase=1 then                                        /* phase-1, object scheduling */
 6         v ⟵ getAppVaNode(AppPrgMdl, AppClass, Arch, vaNodes, PFlist)
 7     else if phase=2 then                                   /* phase-2, task scheduling */
 8         VA[ ] optNodes ⟵ vaNodes;
 9         while PFlist.hasNext() do                          /* find a sub-optimal VA node */
10             Object pfct ⟵ PFlist.getNext();
11             optNodes ⟵ getBestFitNodes(pfct, optNodes);
12         end
13         v ⟵ optNodes[0];
14     end
15     return v;
16 end
```

machine in a cluster (optimising only the network performance factors). In phase-2 (line 8), first all VA nodes are assigned to *optNodes*. After the start of the loop in line 9, a performance factor *pfct* is obtained (line 10). In line 11, *getBestFitNodes* method is invoked, this method returns a subset of VA nodes by considering optimisation with respect to a factor (*pfct*). For example, when the method is called using the performance factor *processor speed*, then it returns a subset of the sub-optimal VA nodes which represents the fastest available machines, processors, and cores. After optimising all the factors in *PFlist*, a sub-optimal VA node is assigned to v (line 13). In line 15, the VA node v is returned.

5 Experiments

We developed several JS-based real applications and benchmarks and experimented using two types multi-core parallel computers: shared memory machines ($\mathtt{m01-02}$) and a heterogeneous cluster ($\mathtt{HC}$, an aggregation of $\mathtt{m01-02}$ and $\mathtt{k01-03}$ machines). Table 1 outlines the details of the experimental setup.

5.1 Experimental Methodology

We perform two types of experiments for training and validation of the JS scheduler. Before the experiments, we classify all the applications in two classes: the *communication-intensive* and the *computation-intensive*.

Table 1. The Experimental Setup

Nodes	*Node architecture*	*Shared caches*	*Processor*	*Processor cores per node*	*Network*
m01 – 02	NUMA	L3	Quad-core Opteron 8356	32 (8 × 4)	Gigabit Ethernet
k01 – 03	UMA	-	Dual-core Opteron 885	8 (4 × 2)	Gigabit Ethernet

To classify an application, we measured the application execution time performing pure computational tasks (e.g., add, multiply, divide). For that, we measure the performance counters `RETIRED_X87_FLOATING_POINT_OPERATIONS` and `CPU_CLOCK_UNHALTED` and calculate the time (in seconds) consumed by each of the compute operations using the formula: $Time(op) = \frac{Count(op) \times CyclesPer(op)}{CpuFrequency}$, where *op* denotes addition, multiplication or division. Then, we sum the time consumed by the compute-operations and calculate the percentage of computation time from the overall execution. The results (shown in Figure 3) are then used to classify all the applications in compute-intensive (more than 50% time in computations) and the communication-intensive (less than 50% time in computations) classes. Within each class, we use some of the applications for the training phase and the rest for the validation experiments.

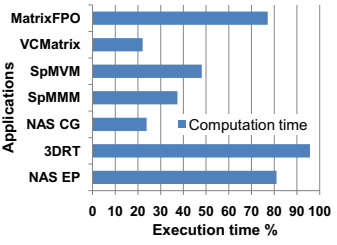


Fig. 3. Application classification

In the training experiments, we manually map JS tasks to VA nodes by searching for a sub-optimal node (considering a performance factor e.g., processor speed, latency, co-scheduling). For example, to search a sub-optimal node with respect to the *latency* factor results in the subset of the VA nodes which minimize the latency (both network and memory) of the mapped JS tasks. Similarly, we collect the performance gains achieved by each factor (whichever applicable, e.g., bandwidth on `m01-02` machines is modeled by the latency, therefore it is not listed) with respect to the two parallel architectures and the application classes. Table 2 shows the results (along the performance gains achieved as compared to the default executions) as a sorted list of the performance factors (PF). The PF lists highlight the significance of the different performance factors with respect to the target architectures and the application classes.

In the validation experiments, the JS applications are scheduled by the scheduler based on the application programming model (shared, distributed, or hybrid), architecture type (shared or distributed memory computer), and application class (compute or communication intensive). The PF lists are also supplied to the scheduler (Algorithm 1), which uses them as guidelines for optimising the search and the selection of the target VA nodes. The JS scheduler creates the required VAs and sub-optimally maps the JS entities (objects and tasks) onto the VA nodes. The experiments conducted on the `m01/02` machines contain up to three executions: an optimised (using the PF lists) execution by the scheduler, a

Table 2. Performance factors lists - average speedup gains

	Compute-intensive application class	*Communication-intensive application class*
$m01 - 02$	latency (13.28%), co-scheduling (5.19%)	latency (16.69%), co-scheduling (9.05%)
HC cluster	processor speed (30.15%) machine load (24.7%) co-scheduling (15.16%) latency (10.99%)	processor speed (25.5%) machine load (24.35%) latency (6.35%) co-scheduling (2.68%)

default Linux-scheduled execution (denoted as LSO), and a scheduler-based execution with optimisations applied using the shuffled factors (the *co-scheduling* factor is used before the *latency*) of the PF lists (PRM). The experiments conducted on the HC cluster contain up to four executions: a JS scheduler-based optimised execution (using PF lists), a Linux scheduled un-optimised execution (LSO) with the machine access order: m01–02 and k01–k03, the LSOR (Linux scheduled) with reverse machine access order: k01 − 03 and m01 − 02, and the JS scheduled execution with shuffled factors (for compute-intensive applications the *latency* factor is swapped with the *co-scheduling* and for communication-intensive applications the *machine load* factor is swapped with the *processor speed*) in the PF list (PRM).

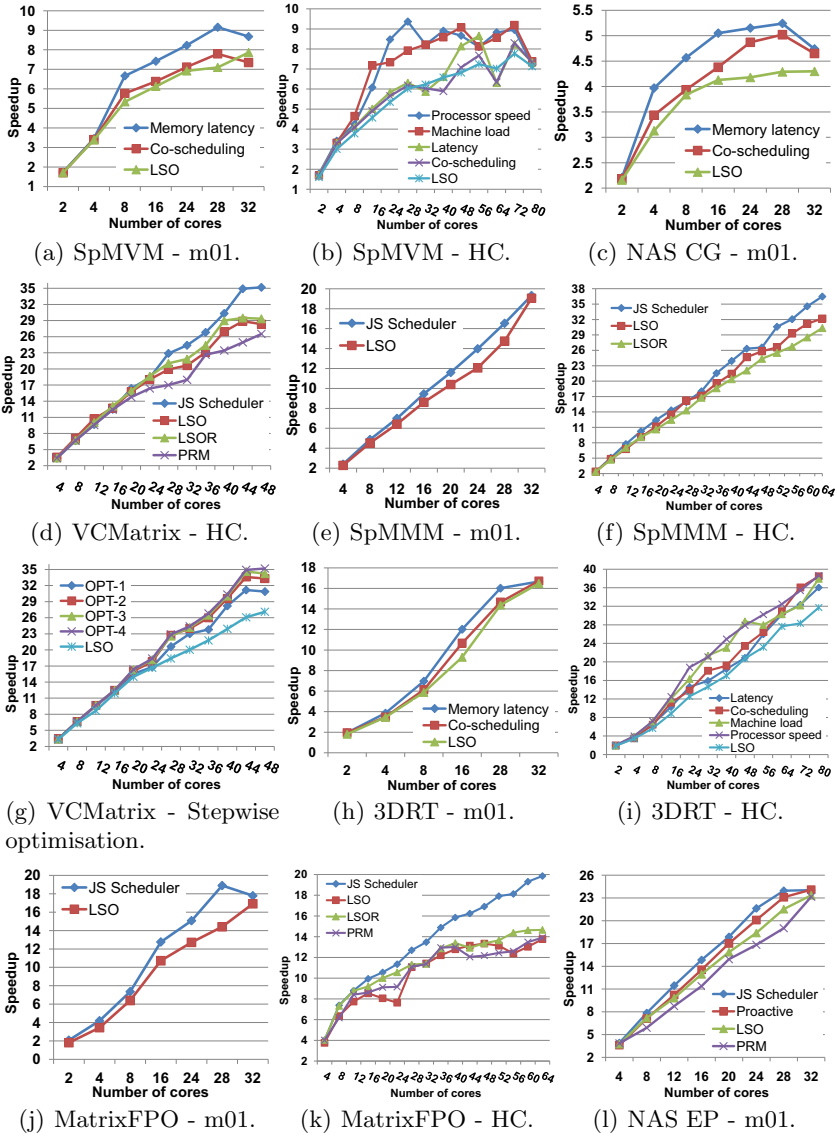
5.2 Communication-Intensive Applications

We used four communication-intensive applications for the training and the validation experiments: the Sparse Matrix-Vector Multiplication (SpMVM), the NAS parallel benchmarks CG kernel, the Variance Co-Variance Matrix computation (VCMatrix), and the Sparse Matrix-Matrix Multiplication (SpMMM).

Training Experiments. The SpMVM kernel computes $y = A \cdot x$ where A is a sparse matrix, and x, y are the dense vectors. The y is computed as follows: $y_i = \sum_{j=1}^{n} a_{ij} \cdot x_j$. We use 15000×15000 matrix, with 4000 non-zeros/row. Figure 4(a) shows the experiment results on the m01. The results show that, the *memory latency* based mappings of the JS tasks result in 15.41% improved speedup (on average) as compared to the LSO. The mappings of the parallel tasks based on *co-scheduling* factor achieves 2.69% better speedups as compared to the LSO. Figure 4(b) shows the SpMVM experimental results on the HC cluster. The results show that, the mapping of the tasks based on the *processor speed* factor achieves 25.5% speedup (on average) as compared to the LSO (using the HC machines in a random order). The other mappings (considering other factors) achieve: *machine load* (24.35%), *latency* (memory and network, 6.35%), and *co-scheduling* (2.68%), better speedups as compared to the LSO.

The CG kernel uses the power and conjugate gradient method to compute an approximation to the smallest eigenvalues. We used the Java-based implemention [5] of the kernel to develop the JS CG version. Figure 4(c) shows the experiment (size: C) results of the JS CG on m01. The *memory latency* factor based mapping of the application achieves on average 17.97% and the *co-scheduling* based mappings achieves 8.72% better speedups as compared to the LSO.

Validation Experiments. The VCMatrix computes the co-variances and variances of a matrix. The diagonal values of the resultant matrix represent variances and the off-diagonal represent co-variances. Figure 4(d) shows the experiment (matrix size: 2200×2200) results on the HC cluster. The JS scheduled execution outperformed the other three executions (PRM, LSO, and LSOR) for most of the machine sizes and achieves better speedup up-to 24.18% as compared to the LSO and LSOR, and up-to 39.96% as compared to the PRM. The JS scheduler optimises the performance of the application on the HC cluster by utilising



(a) SpMVM - m01. (b) SpMVM - HC. (c) NAS CG - m01.

(d) VCMatrix - HC. (e) SpMMM - m01. (f) SpMMM - HC.

(g) VCMatrix - Stepwise optimisation. (h) 3DRT - m01. (i) 3DRT - HC.

(j) MatrixFPO - m01. (k) MatrixFPO - HC. (l) NAS EP - m01.

Fig. 4. JS scheduler experimental results - training and validation

the fastest available resources (e.g., cores, processors, and machines), reducing the data contention on $\mathtt{m01-02}$ (by balancing the machine load), and reducing memory latencies (in the $\mathtt{m01-02}$ NUMA machines).

The SpMMM multiplies two sparse matrices to computes the resultant sparse matrix. Figure 4(e) shows the experiment (matrix size: 10000×10000, 1000 non-zeros/row) results on the $\mathtt{m01}$ machine. The JS scheduled execution achieves better performances, and gains up-to 15.92% more speedups as compared to

the LSO. Figure 4(f) shows the experiment results on the HC cluster. The JS scheduler-based execution achieves better results and achieves up-to 15.13% (as compared to LSO) and 21.17% (as compared to LSOR) more speedups.

To investigate the optimisation effects for the different factors, we experimented with VCMatrix application on the HC cluster and step-wise optimised the LSO execution. First, we use the PF list with only one factor (most significant) and scheduled the application. We then add other factors stepwise and schedule the application. The results (shown in Figure 4(g)) show that, the OPT-1 (using the top most factor) achieves up-to 19.47% speedup as compared to the LSO. The OPT-2 (using the top two factors) achieves up-to 10.18% more speedups as compared to the OPT-1. The OPT-3 and the OPT-4 (using the top three and four factors) achieves further speedups up-to 3.04% and 2.75%. The results show that, the most of the performance (here, up-to 83.66%) can be achieved by optimising the two most significant factors.

5.3 Computation-Intensive Applications

We use three compute-intensive applications for the training and validation experiments. These are: the 3D Ray Tracing (3DRT), the NAS benchmarks EP kernel, and the Matrix Transposition with Floating Point Operations (MatrixFPO).

Training Experiments. We developed the JS-based versions of the 3DRT application using the Java-based version from Java Grande benchmarks [7]. It is a large-scale application that creates several ray tracers, initialises them with scene data (64 spheres), and renders at N×N resolution. Figure 4(h) shows the experiment (image size: 4000 × 4000) results on the m01. The results show that, the *memory latency* factor based mapping of the 3DRT achieves on average 13.28% improved speedup as compared to the LSO. The mappings of the application based on the *co-scheduling* factor achieves on average 5.19% better speedups as compared to the LSO. Figure 4(i) shows the experiment results on the HC cluster. The *processor speed* based mapped application achieves best speedup (on average 30.15% more) as compared to the LSO. The speedups achieved by mappings (considering the other factors) are: *machine load* (24.7%), *co-scheduling* (15.16%), and *latency* (memory and network, 10.99%), better speedups as compared to the LSO.

Validation Experiments. The MatrixFPO transposes a matrix, and performs several floating point operations (e.g., addition, multiplication, and division) at each transpose step. Figure 4(j) shows the experiment (matrix size: 10000 × 10000) results on the m01. The JS scheduled execution of the application achieves up-to 31.09% better speedups as compared to the LSO. Figure 4(k) shows the experiment results on the HC cluster. The JS scheduler based execution achieves better speedups for most of the machine sizes and achieves up-to 48.75% (as compared to the LSO), 35.47% (as compared to the LSOR), and 44.05% more speedups (as compared to the PRM execution). The JS scheduled execution

achieves better performance results as compared to other executions because of the optimisations (using the PF list for the target application class and the architecture) applied by the JS scheduler.

The NAS EP kernel is used to measure the computational performance of parallel computers. Figure 4(1) shows the experiment (data size: 16777216×100) results on the `m01`. We experimented and compared the JS scheduled, LSO, PRM, and the Proactive [3] based executions. The results show that, the JS scheduled execution achieves better speedups as compared to all the other executions and achieves up-to 16.82% (as compared to the LSO), up-to 12.13% (as compared to the Proactive), and up-to 30.32% (as compared to the PRM) more speedups. Proactive exhibits the low performance since it has no capability to map the *active objects* on specific cores. Although the EP kernel has small memory footprints still the remotely scheduled Proactive objects cause some performance degradations as compared to the JS scheduled execution.

6 Conclusions

In this paper, we presented a multi-core aware scheduling extension to JavaSymphony. JS scheduler provides the locality controlled mapping of JS tasks and objects. JS scheduler makes scheduling decisions by considering both the application class and the multi-core specific performace factors. We presented JS scheduler's architecture, methodlogy, and algorithm. We developed several real applications and benchmarks, and experimented using two real multi-core parallel computers. The experimental results show that, the JS scheduled parallel applications outperform other scheduling heuristics and technologies such as Proactive and operating system relying application scheduling.

References

1. Aleem, M., Prodan, R., Fahringer, T.: JavaSymphony: A programming and execution environment for parallel and distributed many-core architectures. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6272, pp. 139–150. Springer, Heidelberg (2010)
2. Barroso, L.A., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R., Verghese, B.: Piranha: a scalable architecture based on single-chip multiprocessing. In: In the 27th ISCA 2000, p. 282. ACM, New York (2000)
3. Caromel, D., Leyton, M.: Proactive parallel suite: From active objects-skeletons-components to environment and deployment. In: Euro-Par Workshops, pp. 423–437. Springer, Heidelberg (2008)
4. Chen, J., John, L.K.: Energy aware program scheduling in a heterogeneous multi-core system. In: Proceedings of the IEEE International Symposium on Workload Characterization, 2008, pp. 1–9. IEEE Computer Society, Los Alamitos (2008)
5. Frumkin, M.A., Schultz, M., Jin, H., Yan, J.: Performance and scalability of the NAS parallel benchmarks in Java. In: IPDPS, p. 139a. IEEE Computer Society, Los Alamitos (2003)

6. Kaminsky, A.: Parallel Java: A unified API for shared memory and cluster parallel programming in 100% Java. In: 21st IPDPS. IEEE Computer Society, Los Alamitos (2007)
7. Smith, L.A., Bull, J.M.: A multithreaded Java grande benchmark suite. In: Third Workshop on Java for High Performance Computing (2001)
8. Zhang, B.-Y., Yang, G.-W., Zheng, W.-M.: Jcluster: an efficient Java parallel environment on a large-scale heterogeneous cluster: Research articles. Concurr. Comput.: Pract. Exper. 18(12), 1541–1557 (2006)
9. Zhuravlev, S., Blagodurov, S., Fedorova, A.: Addressing shared resource contention in multicore processors via scheduling. In: ASPLOS 2010, pp. 129–142. ACM, New York (2010)

Assessing the Computational Benefits of AREA-Oriented DAG-Scheduling

Gennaro Cordasco[1], Rosario De Chiara[2], and Arnold L. Rosenberg[3]

[1] Seconda Università di Napoli, Italy
gennaro.cordasco@unina2.it
[2] Università di Salerno, Italy
dechiara@dia.unisa.it
[3] Colorado State Univ. and Northeastern Univ., USA
rsnbrg@cs.umass.edu

Abstract. Many modern computing platforms, including "aggressive" multicore architectures, proposed exascale architectures, and many modalities of Internet-based computing are "task hungry"—their performance is enhanced by always having as many tasks eligible for allocation to processors as possible. The *AREA-Oriented scheduling* (*AO-scheduling*) paradigm for computations with intertask dependencies—modeled as DAGs—was developed to address the "hunger" of such platforms, by executing an input DAG so as to render tasks eligible for execution quickly. AO-scheduling is a weaker, but more robust, successor to *IC-scheduling*. The latter renders tasks eligible for execution maximally fast—a goal that is not achievable for many DAGs. AO-scheduling coincides with IC-scheduling on DAGs that admit optimal IC-schedules—and optimal AO-scheduling is possible *for all* DAG*s*. The computational complexity of optimal AO-scheduling is not yet known; therefore, this goal is replaced here by a multi-phase heuristic that produces optimal AO-schedules for *series-parallel* DAGs but possibly suboptimal schedules for general DAGs. This paper employs simulation experiments to assess the computational benefits of AO-scheduling in a variety of scenarios and on a range of DAGs whose structure is reminiscent of ones encountered in scientific computing. The experiments pit AO-scheduling against a range of heuristics, from lightweight ones such as FIFO scheduling to computationally more intensive ones that mimic IC-scheduling's *local* decisions. The observed results indicate that AO-scheduling does enhance the efficiency of task-hungry platforms, by amounts that vary according to the availability patterns of processors and the structure of the DAG being executed.

Keywords: Area-oriented DAG-scheduling; Scheduling for task-hungry platforms.

1 Introduction

Many modern computing platforms, including "aggressive" multicore architectures [24], proposed exascale architectures [9], and many modalities of Internet-based computing [11,14,15,22], are "task hungry"—their performance is enhanced by always having as many tasks eligible for allocation to processors as possible. In earlier work, we developed the master-worker *IC-scheduling* paradigm for computations with intertask dependencies—modeled as DAGs—to address the "hunger" of such platforms

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 180–192, 2011.

[3,5,18,19,22,23]. IC-schedules attempt to execute an input DAG so as to render tasks eligible for execution as fast as possible, with a dual goal: (1) Prevent a computation's stalling pending the return of already allocated tasks. (2) Increase "parallelism" by enhancing the effective utilization of workers. Because many DAGs do not admit optimal IC-schedules [19], we have developed a new paradigm—*AREA-Oriented scheduling* (*AO-scheduling*)—to address this deficiency. Optimal AO-schedules—or, *AREA-max schedules*—coincide with optimal IC-schedules on DAGs that admit such schedules; and, AREA-max schedules exist *for every* DAG. AO-scheduling achieves its universal optimizability by weakening IC-scheduling's often-unachievable demand of maximizing the number of eligible tasks at *every* step of a DAG-execution to the always-achievable demand that this number be maximized *on average*. The foundations of AO-scheduling are developed for general DAGs in [6] and for series-parallel DAGs (*SP-DAGs*, see [10,13,20]) in [7]. Because optimal AO-scheduling may be computationally intractable, we develop in Sec. 3 a multiphase heuristic, AO, that produces AREA-max schedules for SP-DAGs but possibly suboptimal AO-schedules for general DAGs.

As with IC-scheduling, it is not clear *a priori* that AO-scheduling enhances the efficiency of executing a DAG. The enhancement of efficiency via *IC-scheduling* is verified experimentally in [4,12,17] for many families of DAGs that admit optimal IC-schedules. But, as noted earlier, many DAGs do not admit such schedules—which fact motivates AO-scheduling and the current study. The current paper adapts the methodology of [12] to *assess the potential computational benefits of AO-scheduling*. We model a "task-hungry" computing platform as a stream of task-seeking workers that arrive according to a random process. We focus on two populations of DAGs:

1. We study AREA-max schedules for randomly constructed *SP-DAGs*.
2. We study the AO-schedules produced by our multiphase heuristic AO for DAGs that are random compositions of small "building-block" DAGs [19]. The DAGs we schedule model computations each of whose subcomputations has the structure of *an expansion* (as in a search tree), *a reduction* (as in an accumulation), *a parallel-prefix* (a/k/a *scan*), *an all-to-all communication* (as in a "gossip"). Such compositions have structures reminiscent of ones that arise in scientific computing.

Thus, all of the AO-schedules that we study can be constructed *efficiently* from the DAGs being scheduled. We simulate executing each generated DAG on our platform model: (a) using the schedule produced by AO and (b) using schedules produced by several popular heuristics, ranging from lightweight ones such as a version of CONDOR's FIFO scheduling [1] to computationally intensive ones that mimic IC-scheduling's *local* decisions. The results we observe indicate that, statistically, *AO-scheduling does significantly enhance the efficiency of task-hungry platforms*, by amounts that vary according to the availability patterns of processors and the structure of the DAG being executed.

2 Background

A. Basics. We study computations that are described by DAGs. Each DAG $\mathcal{G}$ is specified by two sets: its *nodes* $V_{\mathcal{G}}$, each denoting a *task*,[1] and its (directed) *arcs* $A_{\mathcal{G}}$. Each arc

[1] We henceforth refer to DAG *tasks*, rather than *nodes*, to emphasize our computational focus.

$(u \rightarrow v)$ denotes a *dependency* between parent-task u and child-task v. When one executes $\mathcal{G}$, task $v \in V_{\mathcal{G}}$ becomes *eligible* (for execution) only after all of its parents have been executed; hence, all sources (= parentless tasks) are eligible at the beginning of an execution. The goal is to render all sinks (= childless tasks) eligible. Informally, a *schedule* Σ for $\mathcal{G}$ is a rule for selecting which eligible task to execute at each step of an execution; formally, Σ is a *topological sort* of $\mathcal{G}$ (see [8]).

B. Quality metrics. We measure the quality of a schedule Σ for an n-task DAG $\mathcal{G}$ via the rate at which Σ renders tasks of $\mathcal{G}$ eligible: the faster, the better. To this end, we define $E_{\Sigma}(t)$, *the quality of* Σ *at step* t, as the number of tasks that are eligible after Σ has executed t tasks[2] ($t \in [1, n]$). *IC-scheduling strives* to execute $\mathcal{G}$'s tasks in an order that maximizes $E_{\Sigma}(t)$ *at every step* $t \in [1, n]$ *of the execution*; formally: $(\forall t \in [1, n])\ E_{\Sigma^\star}(t) = \max_{\Sigma \text{ a schedule for } \mathcal{G}}\{E_{\Sigma}(t)\}$. *AO-scheduling strives* to find a schedule Σ for $\mathcal{G}$ of maximum *AREA*, where $AREA(\Sigma) \stackrel{\text{def}}{=} E_{\Sigma}(0) + E_{\Sigma}(1) + \cdots + E_{\Sigma}(n)$. (Note the analogy with Riemann sums.) For such an *AREA-max schedule* $\Sigma^\star$,

$$AREA(\Sigma^\star) = \max_{\Sigma \text{ a schedule for } \mathcal{G}} AREA(\Sigma).$$

Many simple DAGs, even tree-DAGs[3] and SP-DAGs, do not admit optimal IC-schedules [19]. Thus, even well-structured DAGs benefit from the universality of optimal AO-scheduling.

3 Finding Good AO-Schedules Efficiently

A. The complexity of AREA-maximization. Every DAG admits an AREA-max schedule. If a DAG admits an optimal IC-schedule, then every such schedule is AREA-max, and vice-versa. This good news from [6] is tempered by the fact that we do not yet know how to find AREA-max schedules for arbitrary DAGs *efficiently*. Indeed, a result in [6] makes it plausible that one *cannot* always produce AREA-max schedules efficiently. Fortunately, efficient AO-scheduling algorithms exist for several significant families of DAGs [6, 7]. Most significantly (for our study): *One can find an AREA-max schedule for any* n*-task SP-*DAG $\mathcal{G}$ *in time* $O(n^2)$. The validating algorithm in [7] exploits $\mathcal{G}$'s structure by (a) decomposing $\mathcal{G}$ to produce a tree $\mathcal{T}_{\mathcal{G}}$ that exposes $\mathcal{G}$'s series-parallel structure, (b) recursively unrolling $\mathcal{T}_{\mathcal{G}}$ to craft an AREA-max schedule for $\mathcal{G}$.

B. Toward efficient AO-scheduling. We develop a four-phase heuristic AOH that AO-schedules any n-task DAG efficiently—specifically, in time $O(n^2)$. Given a DAG $\mathcal{G}$:

Phase 1: *Find* $\mathcal{G}$*'s transitive skeleton* $\mathcal{G}'$. Removing all shortcut arcs from $\mathcal{G}$ reduces the overall complexity of finding an AO-schedule. Formally, $\mathcal{G}'$ is a smallest sub-DAG of $\mathcal{G}$ that shares $\mathcal{G}$'s task-set and transitive closure. Easily, $\mathcal{G}$ and $\mathcal{G}'$ share all of their AREA-max schedules, because removing shortcuts does not impact tasks' dependencies.

Phase 2: *Convert* $\mathcal{G}'$ *to an SP-*DAG $\sigma(\mathcal{G}')$. Invoke an *SP-ization algorithm* to convert $\mathcal{G}'$ to $\sigma(\mathcal{G}')$. Choose an algorithm that: (a) maintains in $\sigma(\mathcal{G}')$ all of the intertask dependencies from $\mathcal{G}'$; (b) (approximately) retains the degree of parallelism inherent in $\mathcal{G}'$ (which precludes, e.g., having $\sigma(\mathcal{G}')$ simply linearize $\mathcal{G}'$); (c) operates within time $O(n^2)$. $\sigma(\mathcal{G}')$ will generally contain *extra* tasks that are not tasks of $\mathcal{G}'$; see Fig. 1. SP-ization algorithms that fit our requirements appear in, e.g., [10, 13, 20].

[2] $[a, b]$ denotes the set of integers $\{a, a + 1, \ldots, b\}$.

[3] A *tree-*DAG is a DAG that remains acyclic even ignoring arc orientations.

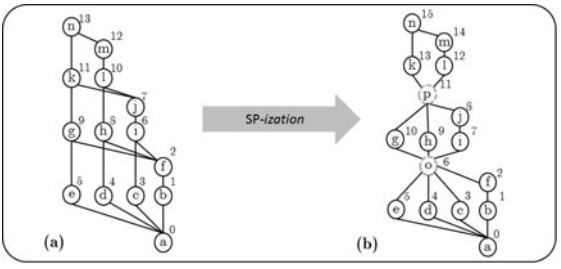


Fig. 1. A sample SP-ization of the LU-decomposition DAG. The task-numbering describes an AREA-max schedule

Phase 3: *Find an AREA-max schedule Σ' for $\sigma(\mathcal{G}')$*, using, e.g., the algorithm of [7].
Phase 4: *"Filter" the AREA-max schedule Σ' for $\sigma(\mathcal{G}')$ to obtain the AO-schedule Σ for $\mathcal{G}$*. "Filtering" Σ' removes the extra tasks added by the SP-ization algorithm. For each extra task u, we assign a *priority* to u's parents in $\mathcal{G}$, that equals the priority of u in Σ'. Σ then schedules equal-priority tasks of $\mathcal{G}$ greedily, by their *yield*—the number of eligible tasks their execution produces.

We illustrate this heuristic on the LU-decomposition DAG $\mathcal{G}$ of Fig. 1(a). $\mathcal{G}$ contains no shortcut arcs, so $\mathcal{G}' = \mathcal{G}$. One possible SP-ization $\sigma(\mathcal{G}')$ of $\mathcal{G}'$ appears in Fig. 1(b); note the two extra tasks o and p. The SP-DAG scheduling algorithm of [7] produces the Area-max schedule $\Sigma' = (a, b, f, c, d, e, o, i, j, g, h, p, l, k, m, n)$ for $\sigma(\mathcal{G}')$; note the task-numbering in Fig. 1(b). Finally, we obtain an AO-schedule Σ for $\mathcal{G}$ by simply removing tasks o and p from Σ'.

4 Experiments to Assess the Quality of AOH

4.1 Experimental Design

A. Overview. We randomly generate DAGs that share structural characteristics with a variety of "real" computation-DAGs, especially ones encountered in scientific computing. We craft five schedules for each generated DAG, one using an AO-scheduling heuristic based on AOH and four using heuristics that represent a range of sophistication and computational intensiveness. We compare the five schedules using two metrics:

1. *Batched makespan*. We overlay our DAG-scheduling with a probabilistic model that specifies the arrival patterns of "hungry" workers and the execution time of each allocated task;
2. *AREA*. We seek to verify or refute the positive correlation between larger AREA and smaller makespan observed on small examples.

B. The DAGs that we execute. We generate DAGs randomly from two populations:

1. *Random n-task SP-DAGs*. We generate a random binary tree $\mathcal{T}$ and randomly (50% uniform choice) designate each internal node of $\mathcal{T}$ either a series- or a parallel-composition node. We then view $\mathcal{T}$ as the composition tree $\mathcal{T}_{\mathcal{G}}$ of a SP-DAG $\mathcal{G}$.

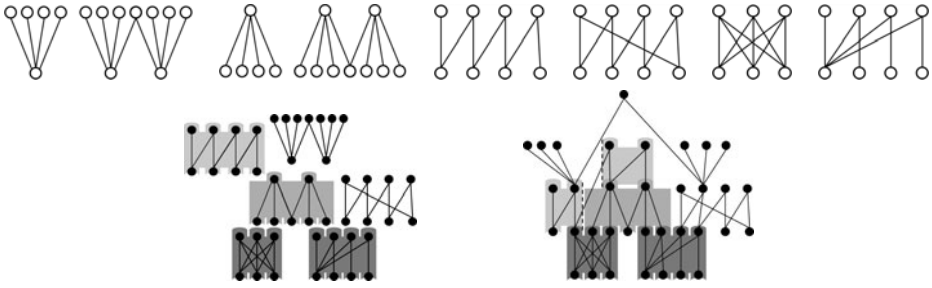


Fig. 2. (Top) A sequence of eight BBBs (All arcs point upward). (Bottom) Composing six BBBs into a LEGO®-DAG.

2. *Random n-task LEGO®-DAGs* (named for the toy). We select a sequence of *Bipartite Building Blocks* (*BBB*s) (see Fig. 2 (top)), randomized according to both size and structure. We *compose* the BBBs in the manner described in [19] and depicted schematically in Fig. 2 (bottom).

C. The AO heuristic. We compare all schedulers to the *AO-scheduler* AO, which produce an AO-schedule for an n-task DAG $\mathcal{G}$ in time $O(n^2)$, in one of two ways:

1. If $\mathcal{G}$ is an SP-DAG, then AO uses the algorithm of [7] to craft an AREA-max schedule for $\mathcal{G}$.
2. If $\mathcal{G}$ is *not* an SP-DAG, then AO uses the multi-phase heuristic AOH of Sec. 3 to craft an AO-schedule for $\mathcal{G}$.

D. The competing schedulers. The heuristics that compete against AO differ in the data structure used to store $\mathcal{G}$'s currently eligible tasks. (See [8] for specifications of the data structures.) All load newly eligible tasks in random order.

1. The FIFO (*first-in, first-out*) scheduler organizes $\mathcal{G}$'s eligible tasks in a FIFO queue. FIFO is, essentially, the scheduler used by systems such as Condor [1].
2. The LIFO (*last-in, first-out*) scheduler organizes $\mathcal{G}$'s current eligible tasks in a stack.
3. The STATIC-GREEDY scheduler organizes tasks that are newly rendered eligible in a MAX-priority queue whose entries are (partially) ordered by *outdegree*.
4. The DYNAMIC-GREEDY scheduler organizes tasks that are newly rendered eligible in a structure that is (partially) ordered by tasks' *yields*. The *yield* of an eligible task v at time t is the number of non-eligible tasks that would be rendered eligible if v were executed at this step. DYNAMIC-GREEDY thus makes the same *local* decisions as does an optimal IC-scheduler (when one exists)—but it lacks the latter's tie-breaking foresight. *Complexity:* The following all take time $O(n)$: (a) initializing the list, (b) serving a "hungry" worker (using EXTRACT-MAX), (c) adding the new eligible tasks after an outdegree-d task v has completed. Thus, DYNAMIC-GREEDY and AO have proportional worst-case computational complexities.

E. The computing platform. Our batched-makespan experiment demands a model of the computing platform in which DAGs are executed. We employ a master-centric model similar to that in [12], the IC-scheduling precursor to this paper. We model the simulated execution of a DAG $\mathcal{G}$ by scheduling heuristic[4] HEUR via a discrete time-ordered queue of "events." Each "event" is represented by the not-yet-executed *residue* of $\mathcal{G}$, together with the current set of eligible tasks, organized as mandated by HEUR. The initial residue of $\mathcal{G}$ is $\mathcal{G}$ itself; the initial set of eligible tasks comprises $\mathcal{G}$'s sources. The transition from one "event" to its successor proceeds as follows:

1. The master polls the available "hungry" workers and allocates (using HEUR's priority measure) one eligible task of $\mathcal{G}$ each to some of these workers. (Only some "hungry" workers get served when there are not enough eligible tasks to serve them all.) Once allocated, a task is no longer eligible.
2. Independently, and asynchronously, served workers execute their tasks.
3. When a worker completes its allocated task, call it v, the master removes v from the current residue of $\mathcal{G}$ and adds the tasks that v's completion renders eligible to the set of current eligible tasks, in the manner mandated by HEUR.

Our model for the computing platform is completed by specifying two probability distributions, one specifying the arrival pattern of "hungry" workers and one specifying tasks' completion times.

• *Worker arrivals*. At each step t of a simulated DAG-execution, we generate a number c_t of "hungry" workers that are seeking tasks, from an exponential distribution with rate parameters $\lambda \in \{1,\ 1/2,\ 1/4,\ 1/8,\ 1/16,\ 1/32\}$, so there are $\mu = 1/\lambda$ workers per step on average.
• *Task execution times*. The master does not know which workers are more powerful than others, so it treats all workers equally. Possible differences in worker power are modeled via the distribution of task execution times. The execution time, t, of an allocated task v is chosen randomly from the "positive half" of a normal distribution with mean 1. We have studied two distributions, one with standard deviation 0.1 and one with standard deviation 0.5. The latter parameter, in particular, allows us to observe the performance of our heuristics on platforms having a rather high level of *heterogeneity*.

4.2 Experimental Methodology

A. DAG sizes. Our experiments simulate the execution of DAGs that range in size from 200 tasks to 4000 tasks. We thereby observe the performance of our heuristics on DAGs that model subcomputations to those that model full computations.

B. BBB structures and sizes. We generate three families of LEGO®-DAGs by composing BBBs whose structures are chosen uniformly among the six structures depicted in Fig. 2 (top) and whose sizes are selected randomly from three distributions: (a) a *uniform* distribution from the set $[2, 20]$; (b) an *exponential* distribution and (c) a *harmonic* distribution; both of the latter generate BBBs having 10 tasks on average.

[4] HEUR $\in$ {AO, FIFO, LIFO, STATIC-GREEDY, DYNAMIC-GREEDY}.

C. Experimental procedures. Both our makespan-comparison and AREA-comparison experiments involved executing four sets of 45 DAGs each: 5 DAGs of each size $n \approx 200, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000$. Each trial executed each DAG 100 times. The results on like-sized DAGs were averaged; we used the means and variances of the schedulers' makespans or AREAs for our comparisons and analyses.

4.3 Experimental Results and Discussion

A. Makespan-comparison. This experiment evaluates AO-scheduling in a simulated "real" computational setting. We considered 120 different test settings, each identified by a triple (D, H, μ). D specifies the class of DAGs:

$D \in \{$SP-DAGs, (Uniform or Exponential or Harmonic) LEGO®-DAGs$\}$;

H specifies the scheduling heuristic:

$H \in \{$AO, FIFO, LIFO, STATIC-GREEDY, DYNAMIC-GREEDY$\}$;

μ specifies the mean number of "hungry" workers per step: $\mu \in \{1, 2, 4, 8, 16, 32\}$. For this experiment, the standard deviation of task-execution time is fixed at 0.1.

We compared the performance of heuristic AO against its competitors via the *timing-ratios* $T(H) \div T(\text{AO})$, where $T(H)$ denotes the simulated makespan observed using heuristic $H \in \{$FIFO, LIFO, STATIC-GREEDY, DYNAMIC-GREEDY$\}$. Note that larger values of the ratio favor heuristic AO. We present both means and 95% confidence intervals of the results in Fig. 3. To enhance legibility, we present a separate plot for each value of D and μ; to conserve space, we present results for random SP-DAGs and only *uniformly* distributed LEGO®-DAGs. (The three families of LEGO®-DAGs exhibit very similar behaviors; see [2].) In each plot, the X-axis indicate the size of DAG instances, while the Y-axis indicates the timing-ratios for AO's four competitors.

Our first observation is that AO-scheduling decreases makespans only for "intermediate" arrival rates μ of "hungry" workers. This is not surprising. When workers arrive very infrequently, i.e., $\mu \approx 1$, *any* heuristic will require roughly n steps to execute an n-task DAG; one observes this in the top plots of Fig. 3. At the other extreme, when workers "flood" the system, there is so much "parallelism" that the only hard limitation for *any* heuristic is the length of a DAG's inherently sequential "critical path." In neither extreme does makespan depend on the scheduling heuristic. Between these extremes, though, there is a range of values of μ where the scheduling heuristic strongly influences makespan: In our trials, when $1 < \mu \leq 32$, AO always completed executing the DAG in less (simulated) time than its competitors. Importantly, we observed that:

> *Within a broad range of worker arrivals, the makespan of a heuristic, as exposed in Fig. 3, has a strong positive correlation with the AREAs of heuristics' schedules, as exposed in Fig. 5. In other words, schedules with higher AREAs executed* DAG*s with smaller makespans.*

The *amount* of observed advantage in makespan depended on three factors: the value of μ, the size of the DAG being executed, and the family of DAGs. Several cases (e.g., $\mu = 8, 16$) show an improvement in the range of 7–12% for LEGO®-DAGs and 10–14% for SP-DAGs. Recall that AO always provides an *AREA-max* schedule for each SP-DAG but not necessarily for each LEGO®-DAG.

Comparing the performance of AO's competitors, we observe first that DYNAMIC-GREEDY always outperforms the other competitors by a considerable margin. This is

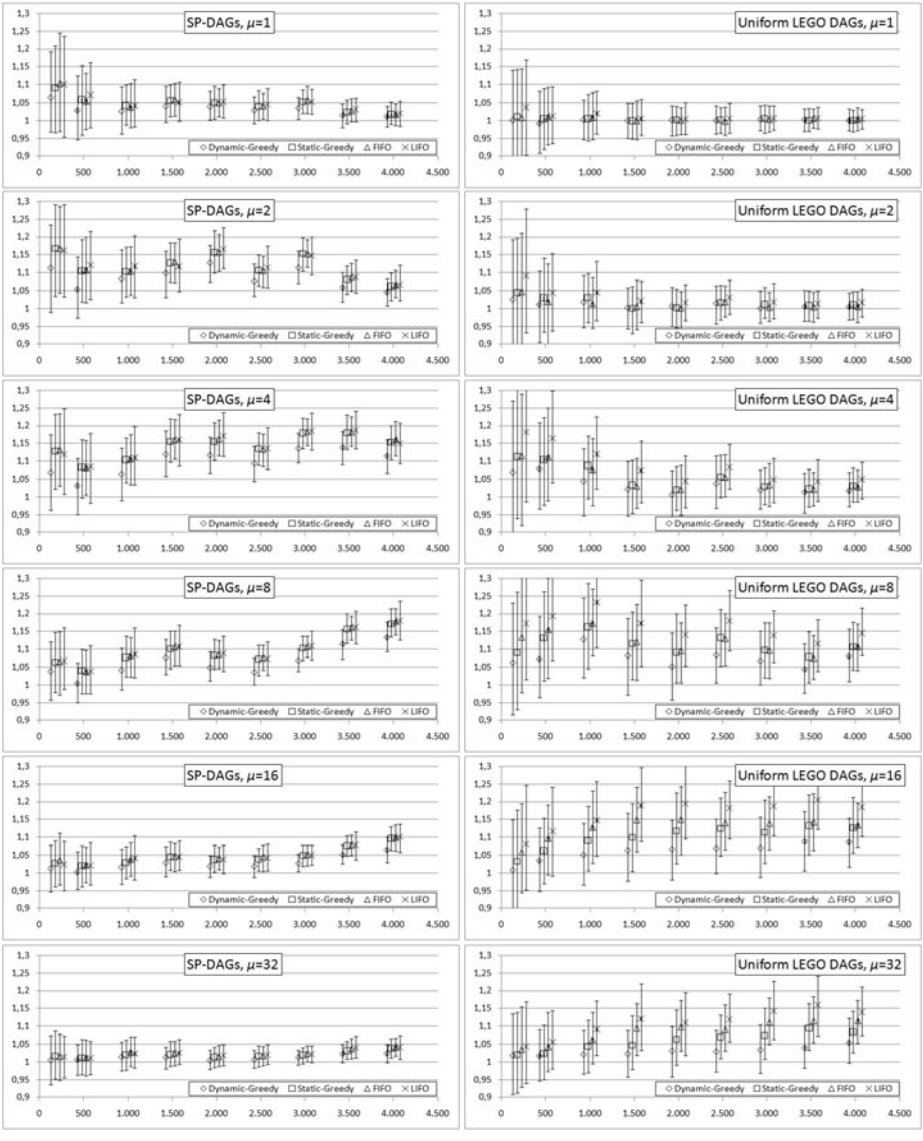


Fig. 3. Timing-ratios for (left) *random SP*-DAG*s* and (right) *Uniform LEGO*®-DAG*s* when the average number of "hungry" workers is $\mu = 1, 2, 4, 8, 16, 32$ (top to bottom).

not surprising because DYNAMIC-GREEDY makes the same *local* decision as an optimal IC-schedule. DYNAMIC-GREEDY "pays for" its superiority among the competitors by its much greater computational expense. Among the other three competitors: LIFO is always the worst heuristic; STATIC-GREEDY and FIFO perform roughly equivalently much of the time, but STATIC-GREEDY sometimes significantly outperforms FIFO; cf., (LEGO®-DAGs, STATIC-GREEDY, 16) vs. (LEGO®-DAGs, FIFO, 16). For SP-DAGs, the three static heuristics: STATIC-GREEDY, FIFO, and LIFO, do not differ substantially.

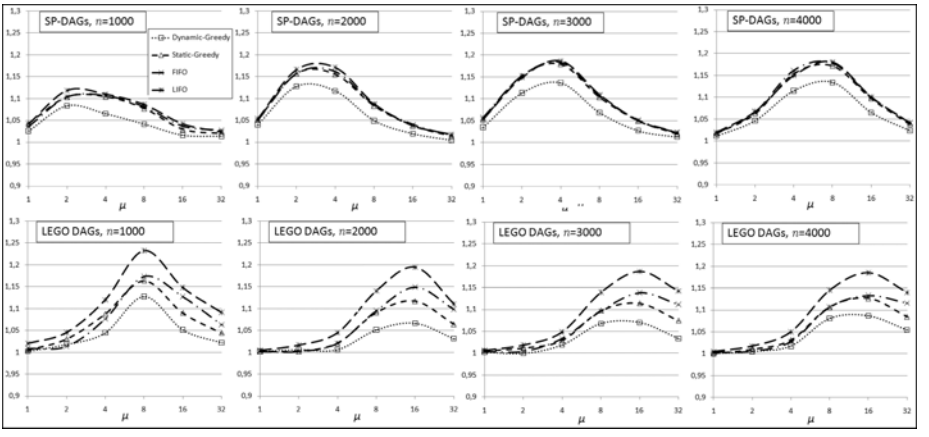


Fig. 4. Timing-ratios for random (top) SP-DAGs, (bottom) Uniform LEGO®-DAGs of different sizes. Left-to-right: 1000 tasks, 2000 tasks, 3000 tasks, 4000 tasks. The X-axes indicate the average number of "hungry" workers at each poll.

The impact of worker-arrival rates. We have just noted that average worker-arrival rate μ influences the performance of AO relative to its competitors. In order to refine this observation, with an eye toward better understanding how μ influences the relative qualities of schedules, we provide, in Fig. 4, plots that show the performance advantage of AO (in terms of timing-ratios) as a function of μ; the values of μ appear logarithmically along the X-axes of the plots. The most notable similarity in the plots is that all are unimodal: as μ increases, AO's relative performance improves up to a unique peak and thereafter degrades. Moreover, AO's peak advantage is comparable for all DAGs of similar sizes, both LEGO®-DAGs and SP-DAGs. However, there are also notable differences in the plots, particularly between LEGO®-DAGs as a class and SP-DAGs as a class. Specifically, we observe the advantage of AO peaking at a higher value of μ for LEGO®-DAGs than for SP-DAGs. Moreover, while the value of μ that maximizes AO's advantage for SP-DAGs grows roughly linearly with DAG-size (the maximizing values range from 2 for 1000-task DAGs to 8 for 4000-task DAGs), this does not appear to happen with LEGO®-DAGs (for which the maximizing values start at 8, for 1000-task DAGs, then jump to 16 for the other three DAG-sizes).

In an attempt to understand why our two DAG families' makespans react differently to the average worker-arrival rate, we analyzed certain characteristics of DAGs from these families. Based on the data in the following table, we conjecture that *the maximizing value of μ depends on the inherent degree of parallelism in the* DAG *being executed.*

	SP-DAGS			LEGO-DAGS		
DAG-size (nodes)	DAG-size (arcs)	Normalized AREA	Critical path lenght	DAG-size (arcs)	Normalized AREA	Critical path lenght
1000	1219	70	150	2885	76	58
2000	2429	75	328	5644	132	74
3000	3666	106	411	8332	189	92
4000	4920	181	445	11114	255	119

We observe that LEGO®-DAGs have smaller critical-path lengths and higher normalized AREAs than do SP-DAGs. (The observed difference would be even larger if we used AREA-max schedules for LEGO®-DAGs rather than the often-suboptimal schedules provided by heuristic AO.) Thus, the values of normalized AREA and critical-path length suggest that LEGO®-DAGs are more "parallelizable" than SP-DAGs.

Modeling heterogeneity via large variance in task execution-times. A major motivation for the development of IC-scheduling (see [22])—hence also of AO-scheduling—was the observed *temporal unpredictability* of many modern computing platforms, which precludes the reliable use of classical, critical-path based, DAG-scheduling strategies as in, e.g., [16]. As noted in sources such as [14,22], we seldom know literally *nothing* quantitative about the computing platform; it is more that our knowledge is very indefinite. A basic tenet of both IC-scheduling and AO-scheduling is that one does not have to deal explicitly with the temporal unpredictability of task execution-times when scheduling a DAG—as long as one enhances the rate of rendering tasks eligible for execution. We test this tenet experimentally by allowing greater variability in task execution-times, expressed via a larger standard deviation in these times. Our primary model allows 10% standard deviation in task execution-times: a mean time of 1 time-unit/task and a standard deviation of 0.1 time-units. How would our results change if we allowed 50% deviation, by raising the allowed standard deviation to 0.5 time-units? We repeated the experiments presented in earlier sections with this larger standard deviation—with rather surprising results. Increasing the allowable standard deviation from 0.1 to 0.5—a truly significant change!—produces a negligible change in the observed advantage of heuristic AO! The observed differences in the average timing-ratios obtained with the two standard deviations in task execution-times, 0.1 and 0.5, do not exceed 0.05%. This suggests that the quality of AO-schedules—as generated by heuristic AO—relative to the four competing heuristics is virtually unaffected by both heterogeneity and temporal unpredictability in "task-hungry" platforms.

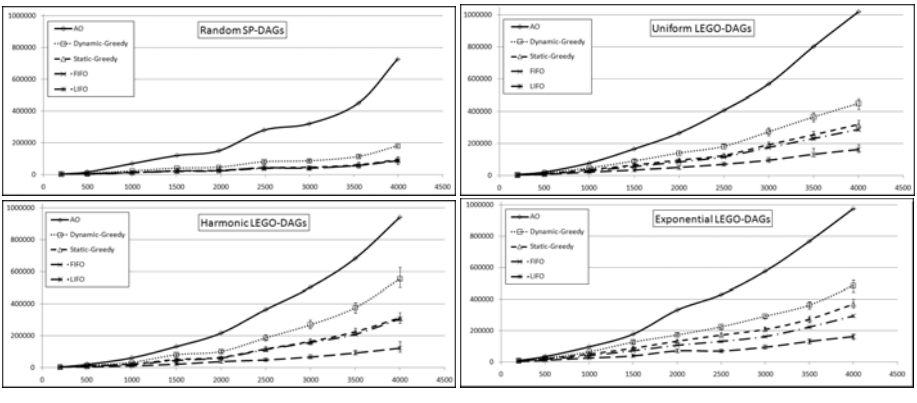


Fig. 5. AREA comparison. Clockwise from the top-left: Random SP-DAGs, Uniform LEGO®-DAGs, Exponential LEGO®-DAGs, Harmonic LEGO®-DAGs.

B. AREA-comparison. Our makespan-oriented experiment suggest that AO-scheduling, as implemented by heuristic AO, has a benign impact on computational performance. This inference has led us to wonder: (a) How do the AREAs of the schedules produced by heuristic AO compare to those of the schedules produced by its four heuristic competitors? (b) How well do the observed differences in makespan of the four competitors track the differences in the AREAs of schedules produced by these heuristics?

We have studied these questions via an experiment that compared the AREAs of AO's schedules to those of schedules for the same DAGs that are produced by the four competing heuristics. We considered 20 test settings, each characterized by the class of DAGs considered and the scheduling heuristic analyzed. For each setting, we executed each DAG 100 times. Fig. 5 presents the mean observed AREA values, as well as their ranges $[\min, \max]$. There is one plot for different-size DAGs from each family indicated in the caption; the sizes of DAG-instances appear along the X-axes. Notable among the observed results: As expected, the schedules provided by heuristic AO for SP-DAGs, being AREA-max, always have the largest AREAs. But, not obviously, the AREA-superiority of AO's schedules persist for general DAGs—which suggests that AO produces high-AREA schedules. This suggestion is reinforced by the fact that difference between the AREAs of AO's schedules and those produced by the competitor heuristics grows more than linearly with the size of the DAG being scheduled.

C. Summing up the experiments. When we consider the results of both the makespan-comparison experiment and the AREA-comparson experiments, as exposed in Figs. 3, 4, and 5, we observe three factors that support our hypothesis that *there is a strong positive correlation between the AREA of a schedule and its makespan.*

- *The schedules provided by all five heuristics*—AO and its four competitors—*have the same relative ordering in the makespan-comparison experiment as in the AREA-comparison experiment.*
- *When the three lightweight competitor heuristics,* FIFO, LIFO, *and* STATIC-GREEDY, *produce schedules for SP-*DAG*s, these schedules have roughly the same AREAs and roughly the same makespans.*
- *The ratio of the AREAs of* AO*'s schedules to those produced by the four competitor heuristics is roughly* 4 *for SP-*DAG*s and only roughly* 2 *for LEGO®-*DAG*s. This correlates positively with the relative improvements in makespan for the same families of* DAG*s.*

In the interest of full disclosure, we do not yet know if the observed differences between results for SP-DAGs and for LEGO®-DAGs are *inherent*, due to the different characteristics of such DAGs (such as degree of inherent parallelism), or algorithmic, due to a possible loss of quality introduced by the heuristics of Sec. 3.

5 Conclusion

Our contributions. Building on the novel *AREA-oriented* (*AO*) scheduling paradigm of [6], we have assessed the quality of AO-schedules for a variety of artificially generated DAGs whose structures are reminiscent of those encountered in real scientific computations. AO-schedules strive to maximize the rate at which DAG-tasks are rendered eligible for allocation to workers with the hope that this will make such schedules

computationally advantageous for modern "task-hungry" computing platforms, such as Internet-based, aggressively multi-core, and exascale platforms. Our assessment pitted our new efficient AO-scheduling heuristic AO against four common scheduling heuristics that represent different points in the sophistication-complexity space of schedulers. We have shown via simulation experiments that:

- The schedules produced by AO have *AREAs that are closer to optimality* than are the schedules produced by the four competing heuristics.
- The schedules produced by AO have *lower makespans* than do the four competing heuristics, based on a probabilistic model of the computing platform and the DAG-executing process.

Importantly, our experiments suggest a strong positive relationship between the AREA of a DAG-schedule and the schedule's performance, as measured by its makespan.

We view the new scheduling heuristic, AO, which operates within time quadratic in the size of the DAG being scheduled, as an important advance because AO *represents the first efficient scheduling mechanism that provably enhances the rate of producing allocation-eligible tasks for* every *computation*-DAG [6].

Finally our experiments have a high degree of *robustness*. The demonstrated computational benefits of AO-scheduling persist even when the "task-hungry" platforms have a high degree of heterogeneity and/or a high degree of temporal unpredictability.

Where we are going. Our demonstration of the computational benefits of AO-scheduling reinforces the importance of two algorithmic questions.

- Does there exist an algorithm for crafting AREA-max schedules for SP-DAGs that is more efficient than the quadratic-time algorithm of [7]?
- Does there exist an algorithm for SP-izing arbitrary DAGs whose use would improve the AREAs and makespans of schedules provided by heuristic AO?

Additionally, the "success" of our experiments suggests the desirability of assessing the value of AO-scheduling via experiments with *real* computations rather than simulated artificial ones. We hope to follow this path in the not-distant future, beginning with experiments using actual traces.

Acknowledgement. The research of A. Rosenberg was supported in part by NSF Grant CNS-0905399. The authors thank A. Gonzáles-Escribano and his team for providing access to their DAG-SP-ization code.

References

1. The Condor Project, Univ. of Wisconsin condor, http://www.cs.wisc.edu/
2. Cordasco, G., De Chiara, R., Rosenberg, A.L.: Assessing the computational benefits of AREA-oriented DAG-scheduling. Tech. Rpt., U. Salerno (2011), http://www.isislab.it/papers/TR0711.pdf
3. Cordasco, G., Malewicz, G., Rosenberg, A.L.: Advances in IC-scheduling theory: scheduling expansive and reductive dags and scheduling dags via duality. IEEE Trans. Parallel and Distributed Systems 18, 1607–1617 (2007)
4. Cordasco, G., Malewicz, G., Rosenberg, A.L.: Applying IC-scheduling theory to some familiar computations. Wkshp. on Large-Scale, Volatile Desktop Grids (PCGrid 2007) (2007)

5. Cordasco, G., Malewicz, G., Rosenberg, A.L.: Extending IC-scheduling via the Sweep algorithm. J. Parallel and Distributed Computing 70, 201–211 (2010)
6. Cordasco, G., Rosenberg, A.L.:: On scheduling dags to maximize area. In: 23rd IEEE Int'l Par. and Distr. Processing Symp. IPDPS 2009 (2009)
7. Cordasco, G., Rosenberg, A.L.: Area-maximizing schedules for series-parallel dAGs. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6272, pp. 380–392. Springer, Heidelberg (2010)
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press, Cambridge (1999)
9. Dongarra, J., et al.: International Exascale Software Project Roadmap. Tech. Rpt. UT-CS-10-652, Univ. Tennessee (2010)
10. González-Escribano, A., van Gemund, A., Cardeñoso-Payo, V.: Mapping unstructured applications into nested parallelism. In: Palma, J.M.L.M., Sousa, A.A., Dongarra, J., Hernández, V. (eds.) VECPAR 2002. LNCS, vol. 2565. Springer, Heidelberg (2003)
11. Foster, I., Kesselman, C. (eds.): The Grid: Blueprint for a New Computing Infrastructure, 2nd edn. Morgan Kaufmann, San Francisco (2004)
12. Hall, R., Rosenberg, A.L., Venkataramani, A.: A comparison of dag-scheduling strategies for Internet-based computing. In: 21st IEEE Int'l Par. and Distr. Proc. Symp. (IPDPS 2007) (2007)
13. Jayasena, S., Ganesh, S.: Conversion of NSP DAGs to SP DAGs. MIT Course Notes 6.895 (2003)
14. Kondo, D., Casanova, H., Wing, E., Berman, F.: Models and scheduling mechanisms for global computing applications. Int'l Par. and Distr. Processing Symp., IPDPS 2002 (2002)
15. Korpela, E., Werthimer, D., Anderson, D., Cobb, J., Lebofsky, M.: SETI@home: massively distributed computing for SETI. In: Dubois, P.F. (ed.) Computing in Science and Engineering, IEEE Computer Soc. Press, Los Alamitos (2000)
16. Kwok, Y.-K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Computing Surveys 31, 406–471 (1999)
17. Malewicz, G., Foster, I., Rosenberg, A.L., Wilde, M.: A tool for prioritizing DAGMan jobs and its evaluation. J. Grid Computing 5, 197–212 (2007)
18. Malewicz, G., Rosenberg, A.L.: On batch-scheduling dags for Internet-based computing. In: 11th Int'l Conf. on Parallel Computing, Euro-Par 2005 (2005)
19. Malewicz, G., Rosenberg, A.L., Yurkewych, M.: Toward a theory for scheduling dags in Internet-based computing. IEEE Trans. Comput. 55, 757–768 (2006)
20. Mitchell, M.: Creating minimal vertex series parallel graphs from directed acyclic graphs. In: 2004 Australasian Symp. on Information Visualisation, vol. 35, pp. 133–139 (2004)
21. Papadimitriou, C.H., Yannakakis, M.: Optimization, approximation, and complexity classes. J. Computer and System Scis. 43, 425–440 (1991)
22. Rosenberg, A.L.: On scheduling mesh-structured computations for Internet-based computing. IEEE Trans. Comput. 53, 1176–1186 (2004)
23. Rosenberg, A.L., Yurkewych, M.: Guidelines for scheduling some common computation-dags for Internet-based computing. IEEE Trans. Comput. 54, 428–438 (2005)
24. Tomov, S., Dongarra, J., Baboulin, M.: Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Computing 36(5-6), 232–240 (2010)

Analysis and Modeling of Social Influence in High Performance Computing Workloads

Shuai Zheng[2], Zon-Yin Shae[1], Xiangliang Zhang[2],
Hani Jamjoom[1], and Liana Fong[1]

[1] IBM T. J. Watson Research Center, Hawthorne, NY
[2] King Abdullah University of Science and Technology, Thuwal, Saudi Arabia

Abstract. Social influence among users (e.g., collaboration on a project) creates bursty behavior in the underlying high performance computing (HPC) workloads. Using representative HPC and cluster workload logs, this paper identifies, analyzes, and quantifies the level of social influence across HPC users. We show the existence of a social graph that is characterized by a pattern of dominant users and followers. This pattern also follows a power-law distribution, which is consistent with those observed in mainstream social networks. Given its potential impact on HPC workloads prediction and scheduling, we propose a fast-converging, computationally-efficient online learning algorithm for identifying social groups. Extensive evaluation shows that our online algorithm can (1) quickly identify the social relationships by using a small portion of incoming jobs and (2) can efficiently track group evolution over time.

1 Introduction

Wide-use and expansion of collaboration technologies (e.g., social networking) are influencing user behavior across all daily activities. Almost completely overlooked, this paper analyzes the effects of *social influence* on high performance computing (HPC) workloads. The intuition is that user collaboration affects the underlying job submission characteristics. For example, students in a class will likely exhibit correlated workload characteristics, especially considering project deadlines, homework, etc.

Discovering the underlying social patterns and dependencies within groups of correlated users—or *communities*, for short—will help improve workload prediction and job scheduling. Our work is akin to those in *community centric web search*, and more recently to Lin *et al.* [7], which discovers the communities based on mutual awareness from observable blogger actions. Unlike existing studies, this paper—to the best of our knowledge—is the first attempt to propose a social-influence-aware method for discovering correlated users and modeling their corresponding workloads in HPC environments.

In an HPC environment, community discovery has several challenges. First, not all the users are regular users of HPC/clusters. Ephemeral users need to be identified and discarded. Second, computing similarities between users is difficult. Since each user submits a different number of jobs to HPC/clusters, measuring

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 193–204, 2011.

the pair wise similarity of users based on their submitted jobs is both complex and unreliable, especially when jobs are described by a complex structured language, e.g., Job Description Language (JDL) [10]. Finally, the community discovery process must be computationally efficient—especially for large-scale workloads—so that it can be used to improve the underlying job scheduling. The challenges outlined above limit the applicability of standard clustering techniques (e.g., double-clustering approach in [12]). In this paper, an efficient method is proposed to identify the *correlated users* in HPC/Cluster workloads.

Following similar analysis of social networks [4], we show that our discovered communities exhibit power-law characteristics. Also, depending on a user's activity, we show that he/she can be categorized as either a *dominant user* or a *follower* to a dominant user. This has profound implications on the importance of dominant users in job scheduling. Basically, identifying dominant users and their followers allows job schedulers to better predict change in future resource demands. To enable effective usage of this new insight, we propose an online learning algorithm that dynamically learns the social characteristics of a given workload. Experimental results show that our online algorithm can efficiently identify stable social groups by observing only a small portion of workload arrivals and can track the group evolution over time.

The remainder of this paper is organized as follows. Section 2 introduces the data sources that we used. We describe our proposed method in Section 3. We then characterize various aspects of the discovered communities in Section 4. Section 5 describes the online learning mechanism. We discuss the related work in Section 6 and conclude the paper in Section 7.

2 Data Sources

The first dataset we used is the Grid 5000 traces [2]—a popular HPC workload testbed. Grid 5000 is an experimental grid platform consisting of nine geographically distributed sites across France. Each site comprises of one or more clusters, for a total of 15 clusters. We use the traces recorded by the individual Grid 5000 clusters from the beginning of the Grid 5000 project (during the first half of 2005) to November 10^{th}, 2006. While there are many useful parameters for each job record in the trace, we extract only `UserID`, `GroupID`, and `SubmittedTime`. In the Grid 5000 trace, there are a total of 10 groups, more than 600 users, and more than 100,000 jobs.

The second dataset that we used is the job trace (i.e., the Logging and Bookkeeping (L&B) files) from the Enabling Grid for EScienc[1] (EGEE) grid. EGEE currently supports up to 300,000 jobs per day on a 24×7 basis. Similar to Grid 5000, we extract the submission `timestamp` and `userID` as job parameters. We use two sets of EGEE L&B files: one contains 229,340 jobs submitted by 53 users in 2005; the other contains 347,775 jobs submitted by 74 users in 2007.

[1] http:// www.eu-egee.org/

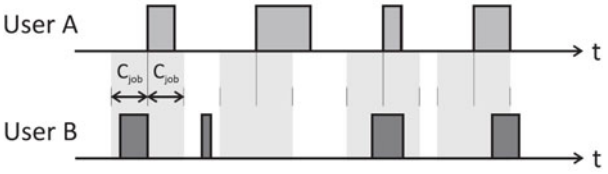


Fig. 1. An example of socially-connected jobs. User B has 3/4 (75%) of jobs within a C_{job} before or after a job by User A.

3 Social Influence Model

In this section, we define *social influence matrix* (SIM) between users. We focus on studying social relationships based on the submission time of jobs. As an example, consider two users working on a project, which consists of many of jobs. If the two users are working closely on the project (e.g., paper deadline), their job submission times will be close. We refer to the two users and their jobs as being *socially influenced*. In this paper, we use two key features, `UserID` and `SubmittedTime`, to analyze the social influence between users. We do not consider the duration time of jobs.

Take Group 1 in the Grid 5000 dataset as an example. In total, we have 38 different users, each submitting hundreds of jobs. Furthermore, consider two users: User A (submitting 1000 jobs) and User B (submitting 800 jobs). Social influence between these two users is captured by two factors:

- **Socially-connected jobs**: for a job of User B, if the minimum time between the submission time of this job and at least one of the 1000 jobs of User A is small enough (e.g., less than half hour before/after submission time), then we say this job is socially connected to User A. We refer to this minimum time threshold between jobs as C_{job}.
- **Socially-connected users**: for the entire 800 jobs of User B, if more than $x\%$ (e.g., $x\% = 50\%$ or 80%) of the jobs are socially-connected jobs to User A, then we say User B is socially connected to User A. We refer to the $x\%$ threshold as C_{user}. Furthermore, we define User A as a *dominant user*, and User B as a *follower* of User A.

It should be noted that the social connections are directed relationships. The fact that User A is socially connected to User B does not mean that User B is also socially connected to User A.

Social influence between User A and User B is depicted in Figure 1. In this example, three out of User B's four jobs have at least one submission from User A within the time interval C_{job}. We can say that User B has 75% of his/her jobs socially connected with User A, and that he/she is a follower to User A if $C_{user} \leq 75\%$.

Next, we turn our attention to building the SIM. For ease of processing, we sort the 38 users in Group 1 of the Grid 5000 according to their `UserID`. The SIM is a 38 by 38 matrix, noted as M. The element $M(i, j)$ of i-th row and j-th column denotes the corresponding percentage of the jobs of user i that are socially connected to user j. We propose Algorithm 1 to calculate the SIM[2].

Algorithm 1. Algorithm of Calculating the Social Influence Matrix (SIM)

Input: Data set of jobs $D = \{UserID_i,\ JobTime_i\}$,
the $UserID_i$ submitted a job at time $JobTime_i$
$\mathcal{U}$=$\{U_j\}, j = 1...|\mathcal{U}|$, Distinct UserIDs
Criterion $C_{job} = 0.5\ hour, 1\ hour, 6\ hours$
Criterion $C_{user} = 50\%, 80\%$
Result: Social Influence Matrix (SIM) M
for $U_j \in \mathcal{U}$ **do**
  $\mathcal{Y} = \{JobTime_q | \forall q, UserID_q = U_j\}$
  ($JobTime$ of all jobs submitted by U_j)
  for $U_i \in \mathcal{U}$ **do**
    $\mathcal{X} = \{JobTime_q | \forall q, UserID_q = U_i\}$
    ($JobTime$ of all jobs submitted by U_i)
    for $k = 1$ *to* $|\mathcal{X}|$ **do**
      $d(k) = min(|X(k) - Y(q)|),\ q = 1, ..., |\mathcal{Y}|$
    $M(i, j) = \sum_{k=1}^{|\mathcal{X}|} (d(k) < C_{job}) / |\mathcal{X}|$
    if $M(i, j) > C_{user}$ **then**
      user U_i is socially connected to user U_j, and
      user U_i is a follower to the dominant user U_j

Table 1 shows the SIM of Group 1 when C_{job} is one hour. We give SIM of the first five users. In the first column, the first element is 1, which means that 100% of User 1's jobs are socially connected to himself/herself. Obviously, all diagonal values of the matrix are 1. The second element in the first column is 0, which means that none of the jobs of second user are socially connected to the first user.

As described earlier, social connections between users are influenced by the threshold value C_{user}—the minimum percentage of socially-connected jobs. If we set the criterion C_{user} to 80%, the *followers* to User i are the ones who have values larger than 0.8 in i-th column. We show the number of *followers* in Table 2 for C_{user}=80% and C_{user}=50%. As expected, when C_{user} is decreased, we have an increase in the number of followers. For example, User 2 has 5 followers when C_{user} is reduced to 50% from 80%. In the following section, we will plot the

[2] For a large set of users, we have two solutions to handle them in practice. First, we can parallelize Algorithm 1 to efficiently calculate SIM M, because the calculations of elements M_{ij} are independent. Second, we can use the later proposed Algorithm 2 to incrementally update M online.

Table 1. One-Hour Social Influence Matrix (SIM)

	User 1	User 2	User 3	User 4	User 5	...
User 1	1	0	0	0	0	...
User 2	0	1	0	0.029	0.084	...
User 3	0	0	1	0	0	...
User 4	0	0.131	0	1	0.239	...
User 5	0	0.048	0	0.056	1	...
...	...	...	...	...	...	...

Table 2. Number of Followers. A value of 1 means that there is only one follower to the corresponding user, or this user is only socially connected to himself/herself.

C_{job} C_{user}	User 1	User 2	User 3	User 4	User 5	...
1 hr 80%	1	3	1	2	1	...
1 hr 50%	1	5	1	2	2	...

results like Table 2 for the two Grid 5000 datasets and the two EGEE datasets to explore their number of followers distribution.

4 Analysis of Social Influence

In this section, we analyze Grid 5000 and EGEE datasets to discover socially-connected users. As expected, the values for C_{job} and C_{user} will impact the resulting analysis. We present our results for a number of value combinations for both parameters. One challenge was choosing a reasonable parameter range for C_{job}. In particular, C_{job} should be set as small as possible to capture true dependencies. To reason about value for C_{job}, we plot the Cumulative Distribution Function (CDF) of the jobs' interarrival time for the four datasets in Figure 2. As the figure shows, the interarrival times vary significantly over multiple orders of magnitude. Picking a small value of C_{job} will unnecessarily filter out longer-range (i.e., minutes rather than seconds) dependencies, which are common in human interactions. Figure 2 indicates that—on average—the probability that at least one job will be submitted within 30 minutes for all the groups under study is larger than 95%. Thus, we decide to use C_{job} values in the range from 30 minutes to 6 hours in our investigation.

4.1 Community Extraction from HPC Workloads

Figure 3 (a) shows the number of followers for each user identified from Group 1 of Grid 5000 trace with criterion C_{user}=50%. When C_{user}=50% and C_{job}=6 hours, User 1, 2, and 3 have 13, 11, and 10 followers, respectively. When C_{user} =50% and C_{job}=1 or 0.5 hours, the number of followers for every user decreases as expected. Figure 3 (b) shows the number of followers for each user in Group 1 with criterion C_{user}=80%. From Figure 3 (a) and (b), we can see that all users

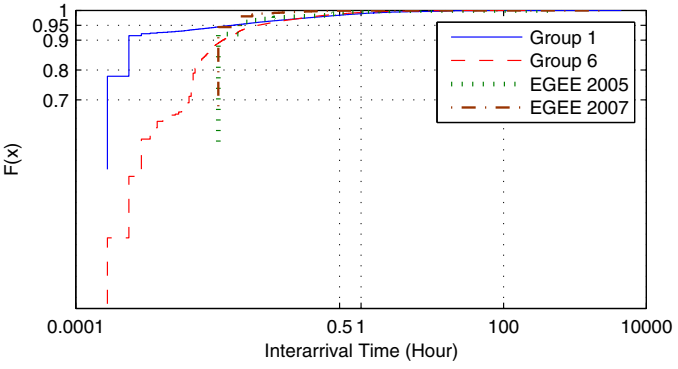


Fig. 2. CDF of jobs' interarrival time for Group 1, Group 6, EGEE 2005 and EGEE 2007

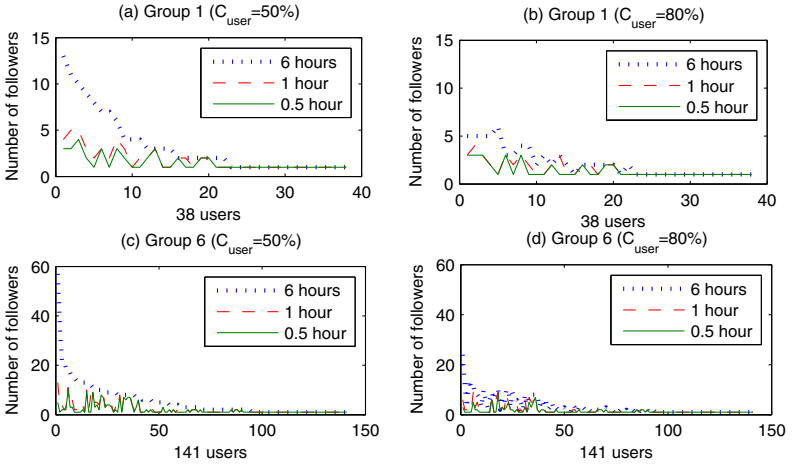


Fig. 3. Social groups discovered in Group 1 and Group 6 of Grid 5000 traces showing the number of followers to each dominant user with different criteria of $C_{user} = 50\%$, 80% and $C_{job} = 6\ hours$, 1 *hour*, 0.5 *hour*

in Group 1 consistently have followers. Similarly, Figure 3 (c) and (d) show that nearly 60% of all users in Group 6 consistently have followers. Figure 4 shows that around 45% of all users in EGEE 2005 consistently have followers, and around 55% of all users in EGEE 2007 consistently have followers.

4.2 Power-Law Distribution of Discovered Communities

A common property of many large networks is that the vertex connectivities follow a scale-free power-law distribution [4]. We would like to investigate if

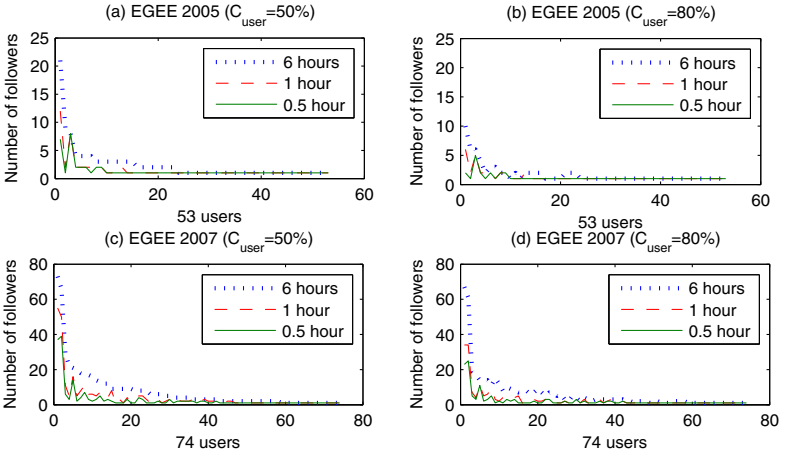


Fig. 4. Social groups discovered in EGEE 2005 and EGEE 2007 traces showing the number of followers to each dominant user with different criteria of $C_{user} = 50\%$, 80% and $C_{job} = 6\ hours$, 1 *hour* and 0.5 *hour*

such a power-law property exists among HPC users. In our study, each vertex is a person. The connectivity among vertices captures the interactions among users.

Let $P(k)$ denote the probability that a user has a number of followers k, where k is a positive integer number. To study if our discovered social groups have the same property as the common networks, we investigate whether the number of followers k of each dominant user has the power-law distribution: $P(k)=a \times k^b$. Figure 5 shows the power-law distribution of the number of followers identified from Group 1 and 6 of Grid 5000, as well as from EGEE 2005 and EGEE 2007. The power-law distributions of other groups exhibit similar characteristics (unless the group size is very small); they are not shown for space consideration. All social followers are discovered with socially-connected criterion $C_{job} = 0.5\ hour$ and $C_{user} = 50\%$. From Figure 5, we can see that the number of followers fits very well the power-law distribution with different parameters a and b.

5 Design of Online Learning Mechanism

Our earlier analysis used an offline mechanism to identify social influence in HPC workloads. For our analysis to be consumable by HPC job schedulers and resource managers, a real-time (online) mechanism is needed. Algorithm 2 shows the proposed mechanism for computing the social influence matrix (SIM) on the fly while jobs are arriving. Suppose that at time step, t, a user submits a job. We have $(UserID^t, JobTime^t)$ as a sample in our streaming workload, where

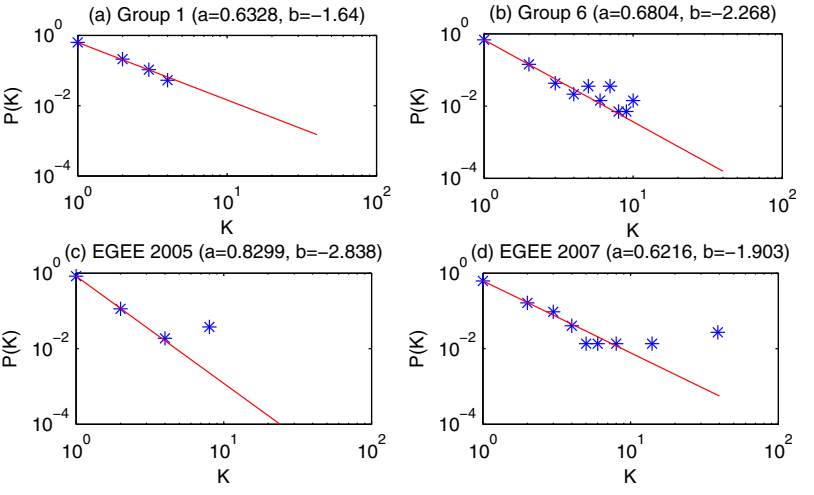


Fig. 5. The power-law distribution of the number of users k following each dominant user identified in Group 1 and Group 6 of Grid 5000, EGEE 2005 and EGEE 2007

time step $t \geq 1$ is an integer[3]. Given the threshold C_{job} of socially-connected jobs, we maintain a $Un^t \times Un^t$ matrix M^t, where Un^t is the number of unique users until t. Each element of M^t is a 2-tuple object $M^t_{ij} = \{R^t_{ij}, C^t_i\}$, where C^t_i is the number of jobs submitted by user $UserID = U_i$ until time t, and R^t_{ij} is the number of jobs of $UserID = U_i$ that are socially connected to $UserID = U_j$ until t. The threshold C_{job} and C_{user} are set to half hour and 50% respectively by taking experience from the offline mechanism for enabling further comparison.

We use MATLAB to implement and simulate Algorithm 2. Unlike Algorithm 1, we use a sliding time window along the workload flow. This window only includes past job submissions within the interval of C_{job}. Note, this is different to the method described in Algorithm 1 for checking the socially-connected jobs. In Algorithm 1, we consider the absolute value of the time difference, which includes both sides of the current time point on the time axis (as shown in Figure 1).

We use *cosine similarity* [13] to measure the difference between the distribution of followers as obtained by Algorithms 1 and 2. We set C_{job}=0.5 hour and C_{user}=50%. Figure 6 shows the cosine similarity between online results as compared to its offline counterpart as a function of the percentage of observed jobs (in chronological order). The cosine similarity increases towards the value 1 as additional jobs are observed. When all jobs have been observed (at the 100% value on the x-axis), the online and offline algorithm produce the same results (cosine similarity = 1).

A very promising characteristic of the online algorithm is its ability to track group evolution over time. For example, Figure 6 shows how Group 6 and EGEE

[3] Time step t is used to order the streaming jobs by arrival time. If $t1 < t2$, $UserID^{t1}$ submitted a job earlier than $UserID^{t2}$, i.e., $JobTime^{t1} < JobTime^{t2}$.

Algorithm 2. Online Algorithm of Calculating Socially Influence Matrix (SIM)

Input: Streaming jobs $S = \{UserID^t,\ JobTime^t\}$,
the $UserID^t$ submitted a job at time $JobTime^t$
Criterion $C_{job} = 0.5\ hour$
Criterion $C_{user} = 50\%$
Result: Socially Influence Matrix M^t, $M^t_{ij} = \{R^t_{ij}, C^t_i\}$
where R^t_{ij} is the number of U_i's jobs socially connected to U_j until t, and C^t_i is the number of jobs submitted by user U_i until t
Initialization:
Distinct users set $\mathcal{U} = \{\}$, and the number of distinct users $Un^t = 0$,
$M^t_{ij} = \{R^t_{ij} = 0, C^t_i = 0\}$
Maintain M^t
for $t = 1$ *to* ... **do**
  if $UserID^t \notin \mathcal{U}$ **then**
    $\mathcal{U} = \mathcal{U} \cup UserID^t$
    $Un^t = Un^{t-1} + 1$
  $i \leftarrow UserID^t = U_i$
  $C^t_i = C^{t-1}_i + 1$
  $\mathcal{X} = \{UserID^q\ |\forall q, JobTime^q >= JobTime^t - C_{job}\}$
  a set of $UserID$ whose jobs arrived within a time-window C_{job}
  for $U_j \in \mathcal{X}$ **do**
    $R^t_{ij} = R^{t-1}_{ij} + 1$
    $R^t_{ji} = R^{t-1}_{ji} + |X^{j,t'}|$
    where $|X^{j,t'}|$ is the number of U_j in time window of $[max\{t - C_{job}, t' + C_{job}\}, t]$ and t' is the last time when U_i appeared
  Socially-connected users
  for $j = 1$ *to* Un^t **do**
    if $R^t_{ij}/C^t_i > C_{user}$ *in* M^t_{ij} **then**
      user U_i is socially connected to user U_j, and
      user U_i is a follower to the dominant user U_j

2005 have sudden increase in the number of discovered social groups after processing 50% and 80% of all job flows, respectively. In contrast, Group 1 and EGEE 2007 have stable social groups after processing 5% of all jobs, with little change beyond that point.

The discovered changes in social groups reflect variations in HPC system usage. In order to verify the changes in social relationships, we investigate the number of distinct users as a function of job arrivals (Figure 7). Comparing Figures 7 and 6, we find that the curves move in tandem (as a function of job arrival). For example, Group 6 and EGEE 2005 have more new users after observing 50% and 80% of all job flows. The number of users in Group 1 and EGEE 2007 grow slowly after 5% of all jobs.

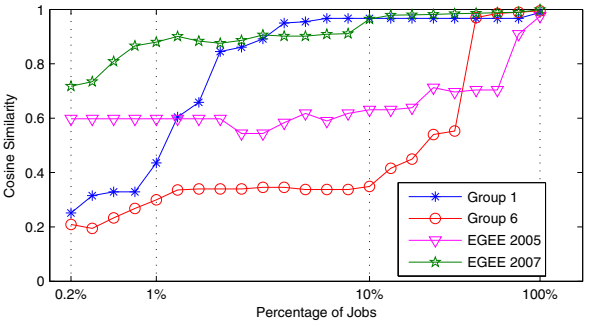


Fig. 6. Online Learning Convergence for Group 1, Group 6, EGEE 2005 and 2007

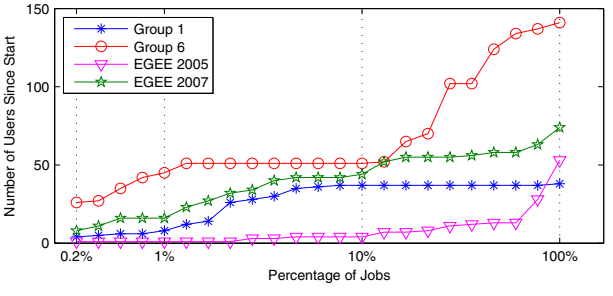


Fig. 7. Online Users Count for Group 1, Group 6, EGEE 2005 and EGEE 2007

We are still interested in how fast the similarity can reach a stable state. Figure 6 shows that our online algorithm can converge and reach a stable state quickly in real time. For example in Group 1, we see that the user population (number of distinct users as shown in Figure 7) is stable within 1% of job flows, while the cosine similarity (in Figure 6) reaches the stable state within about 0.4% of all jobs. When the user population of Group 1 changes dramatically from 1% to 7%, Figure 6 shows that the cosine similarity follows the changes quickly and reaches a stable state closing to 1.

6 Related Work

Many generally available workload traces [3,2,1] have been used to study design alternatives for resource scheduling algorithms, resource capacity planning in both grid cluster and cloud environments, building performance modeling, etc. Most of these studies use—as input—the arrival pattern of individual jobs and their resource requirements [8]. Only recently, Iosup *et al.* [5] presented their first investigation of the grouping of jobs by looking at their submission patterns and their impact on computing resource consumption. More recently, Ostermann *et al.* [9] studied job flows focusing on how sub-jobs are submitted in parallel or

in sequence [9]. Unlike existing work, we investigate job submission patterns from the perspective of social connections (e.g., group association, virtual organization, etc.). Especially in cloud environments, the ability to predict future demands is critical to managing the underlying computing resources.

Community extraction has been commonly studied as a graph problem. A graph is constructed by taking users/persons as nodes and connecting two nodes if they are correlated in some activities. A community discovered in a graph is a subgraph including a group of nodes and their edges, where the nodes have high similarity with each other. Two approaches have been used to identify the subgraphs. One is based on a clustering method, weighted kernel k-means, which groups together the nodes that are similar to each other by measuring a type of random walk distance [14]. The other approach uses graph partitioning algorithm (or called spectral clustering) to cut the graph into a set of subgraphs by optimizing the cost of cutting edges under the normalized cut criterion [11].

The most relevant work to our paper is the Mixed User Group Model (MUGM) proposed by Song *et al.* [12]. MUGM forms the groups of users through characterizing each user by job clusters, which were obtained by using CLARA (also known as k-medoids) clustering method [6] on the whole jobs. There are two difficulties in applying MUGM on analyzing HPC workloads. First, using clustering method to group jobs requires the computation of job similarities, which is difficult for jobs described by mixture of features. Second, representing users based on job clusters cannot be easily adapted as an online technique, because job clusters need to be computed on the whole data before analyzing the user groups. Our approach identifies social groups by measuring their tasks' submission behavior. Furthermore, our approach does not need the computation of job similarities and can be efficiently used in an online fashion.

7 Conclusions and Future Work

This paper identified and validated the existence of social influence on HPC workloads. We suspect that this influence stems from how HPC applications are developed and run. Given its potential importance on job scheduling and resource management, we proposed a method to discover *socially-connected users* based on measuring the proportion of *socially-connected jobs* they have. We showed the existence of a social graph characterized by a pattern of dominant users and followers. We applied this proposed method to traces of Grid 5000 and EGEE. Both in the Grid 5000 and the EGEE workloads, we consistently found that around half of the users had *followers* irrespective of how the thresholds C_{job} and C_{user} were set. Additionally, the corresponding social graph followed a power-law distribution, which is consistent with mainstream social networks. We developed both an offline and a fast-converging online algorithm to implement our proposed method. The online version was shown to require a small number of arrived jobs to discover the social groups and be able to track the group evolution over time.

Identifying dominant users and their followers may have profound implication on the prediction of workload and, consequently, on resource demand patterns. Thus, our future work will focus on quantifying social characteristics of the discovered user connections and using this quantitative information to improve the prediction of workload arrival patterns and resource demands. The prediction will enable the development of new algorithms in job scheduling, resource utilization, and resource capacity planning.

References

1. Google Cluster Data, http://code.google.com/p/googleclusterdata/
2. Grid Workloads Archive, http://gwa.ewi.tudelft.nl/pmwiki/
3. Parallel Workloads Archive, http://www.cs.huji.ac.il/labs/parallel/workload/
4. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science 286(5439), 509 (1999)
5. Iosup, A., Jan, M., Sonmez, O.O., Epema, D.H.J.: The characteristics and performance of groups of jobs in grids. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 382–393. Springer, Heidelberg (2007)
6. Kaufman, L., Rousseeuw, P., Corporation, E.: Finding groups in data: an introduction to cluster analysis, vol. 39. Wiley Online Library, Chichester (1990)
7. Lin, Y., Sundaram, H., Chi, Y., Tatemura, J., Tseng, B.: Blog community discovery and evolution based on mutual awareness expansion. In: Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence, pp. 48–56. IEEE Computer Society, Los Alamitos (2007)
8. Mishra, A.K., Hellerstein, J.L., Cirne, W., Das, C.R.: Towards characterizing cloud backend workloads: insights from google compute clusters. ACM SIGMETRICS Performance Evaluation Review 37(4), 34–41 (2010)
9. Ostermann, S., Prodan, R., Fahringer, T., Iosup, R., Epema, D.: On the characteristics of grid workflows. In: CoreGrid Technical Report TR-0132 (2008)
10. Pacini, F.: Job description language howto (2003)
11. Shi, J., Malik, J.: Normalized cuts and image segmentation. IEEE Transactions on Pattern Analysis and Machine Intelligence 22(8), 888–905 (2000)
12. Song, B., Ernemann, C., Yahyapour, R.: User group-based workload analysis and modelling. In: IEEE International Symposium on Cluster Computing and the Grid, CCGrid 2005, vol. 2, pp. 953–961. IEEE, Los Alamitos (2005)
13. Tan, P., Steinbach, M., Kumar, V.: Introduction to data mining. Pearson Addison Wesley, Boston (2006)
14. Yen, L., Vanvyve, D., Wouters, F., Fouss, F., Verleysen, M., Saerens, M.: Clustering using a random walk based distance measure. In: Proceedings of the 13th Symposium on Artificial Neural Networks (ESANN 2005), pp. 317–324. Citeseer (2005)

Work Stealing for Multi-core HPC Clusters

Kaushik Ravichandran, Sangho Lee, and Santosh Pande

College of Computing, Georgia Institute of Technology, USA
kaushikr@gatech.edu, slee431@gatech.edu, santosh@cc.gatech.edu

Abstract. Today a significant fraction of HPC clusters are built from multi-core machines connected via a high speed interconnect, hence, they have a mix of shared memory and distributed memory. Work stealing algorithms are currently designed for either a shared memory architecture or for a distributed memory architecture and are extended to work on these multi-core clusters by assuming a single underlying architecture. However, as the number of cores in each node increase, the differences between a shared memory architecture and a distributed memory architecture become more acute. Current work stealing approaches are not suitable for multi-core clusters due to the dichotomy of the underlying architecture. We combine the best aspects of both the current approaches in to a new algorithm. Our algorithm allows for more efficient execution of large-scale HPC applications, such as UTS, on clusters which have large multi-cores. As the number of cores per node increase, which is inevitable given today's processor trends, such an approach is crucial.

Keywords: dynamic load balancing, unbalanced tree search, multi-core.

1 Introduction

Today, a large portion of HPC systems are built from multi-core machines connected through high speed interconnects such as InfiniBand. This kind of architecture is seen in systems in the Top 500 list such as Jaguar (Oak Ridge National Laboratory), Hopper (NERSC) and Kraken (National Institute of Computational Sciences). These systems have two distinct kinds of parallelism: intra-node and inter-node. Intra-node parallelism (through shared memory) is due to the existence of multiple cores in a single node, while inter-node parallelism (through distributed memory) is due to the presence of a large number of such nodes. The number of cores per node, and hence intra-node parallelism, is increasing as larger and larger multi-cores become commonplace.

Work stealing algorithms are currently designed for either a shared memory architecture or for a distributed memory architecture and are extended to work on these multi-core clusters in a straight forward manner. The distributed memory implementations extend naturally into a multi-core cluster environment by running separate tasks on different cores. Shared memory implementations are typically extended by using PGAS (Partitioned Global Address Space) languages such as UPC (Berkley Unified Parallel C). PGAS languages present a

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 205–217, 2011.

unified address space over the underlying distributed memory allowing the algorithm to scale across the cluster. Termination detection algorithms are similarly extended.

However, as the number of cores in each node increase (due to larger multi-cores), the differences between a shared memory architecture and a distributed memory architecture become more acute. Current approaches which extend a completely shared memory paradigm or a distributed memory paradigm to a cluster can be improved. Combining the best aspects of both approaches results in a new algorithm which allows for a more efficient execution of large-scale HPC applications, like UTS, which need efficient dynamic load balancing. As the number of cores per node increase, which is inevitable given today's processor trends, such an approach is crucial.

The UTS (Unbalanced Tree Benchmark) [14] is representative of the class of applications which process highly unbalanced workloads. We demonstrate the effectiveness of our approach on the UTS benchmark and report significant speedups on large multi-core clusters over current shared memory implementations and distributed memory implementations.

2 Work Stealing

A fundamental problem in achieving maximum performance from HPC applications is that of dynamic load balancing. Many applications exhibit a large variability in the amount of work they dynamically generate. This variability gives rise to an imbalance in the parallel execution of these applications. Variability could be caused by many reasons. For example, the random nature of input data could cause imbalance between the workloads of different parallel processing elements. The UTS (Unbalanced Tree Search) benchmark [14], is representative of the class of parallel applications which require dynamic load balancing. The benchmark has been carefully designed to be *the* optimal adversary to load balancing strategies. In this paper we shall focus on the UTS benchmark to test our algorithm, both because of its popularity and because of its ability to stress dynamic load balancing algorithms effectively.

Load balancing can be broadly divided into two categories: static and dynamic. Static approaches have been well studied [16,12]. These approaches, however, are not suitable for the kind of applications we are concerned about due to the unpredictability of the problem space and dependence on the input parameters and dataset. Dynamic load balancing algorithms have been proposed to address the types of applications we are looking at.

Many dynamic load balancing algorithms have been proposed. Two popular algorithms used on both architectures (shared as well as distributed memory) are work sharing and work stealing. Work sharing involves balancing of the workload using a globally shared task queue. Work stealing on the other hand follows a completely distributed approach where idle processing elements take on the onus of finding work by looking around the system and has been found to be more efficient [6]. Its effectiveness lies in the fact that it puts a majority of the overhead on idle processors, minimizing the load on busy processors and

minimizing the need for global information. Work stealing has been proven to be optimal for a large class of problems and has tight space bounds [2], thus, making it the method of choice for large scale distributed clusters.

Termination detection is a critical postlude to work stealing algorithms. This step allows programs to recognize when there is no more work in the system. As we have described before, in the work stealing method, once a processing element becomes idle, it looks around for work that it can steal from other processing elements. Indeed, it is possible that all processing elements have completed their work and are simply looking around for more work endlessly. The process of detecting such a system state and ending the execution is termination detection. Termination detection algorithms, akin to work stealing algorithms have different implementations on shared memory architectures and on distributed memory architectures.

Shared memory architecture. On shared memory architectures, work stealing has been used to effectively parallelize unbalanced workloads. Implementations such as Cilk [9], have popularized work stealing by implementing it in the runtime system. Typically, each processing element maintains a double ended queue in shared memory. When it needs to process a task, it takes a task off the front of its queue and processes it. Any new tasks are added to the front of the queue. When a processing element completes all the tasks in its queue, it becomes idle and begins work stealing. It looks for other processing elements which have excess tasks in their task queues. Once it locates such a queue, it takes tasks from the back of that queue and pushes it onto its own. Of course, the implementation needs to be highly tuned to reduce excess locking overhead. Methods like "THE" [9] eliminate a majority of the situations in which locks are needed. Such methods split the queues into private and public sections, eliminating the need for locking in the private sections while still requiring locking in the public section.

Termination detection in shared memory architectures can be achieved using special kinds of barriers, called cancellable barriers which allow threads to "check-in" and "check out". These barriers are especially suited for shared memory work stealing algorithms.

Attempts have been made to extend such work stealing algorithms to HPC clusters which contain both shared memory and distributed memory [6]. To extend these algorithms to a cluster, an abstraction is needed to apply a shared memory paradigm over the entire cluster. Partitioned Global Address Space (PGAS) languages allow precisely this. These languages allow programmers to write code, assuming a shared memory architecture and the PGAS programming model takes care of the rest. It implicitly converts any cross machine memory accesses into messages using interfaces such as MPI.

While this approach leads to a correct implementation, it is not necessarily efficient, for several reasons. The cross machine memory accesses that are converted into messages cause increased contention, runtime overhead and latency. These overheads can be minimized to some extent by careful tuning of the messages that are sent by using one sided reads and writes and RDMA (Remote

Direct Memory Access). However, using one sided reads and writes is a much more complicated affair and involves a lot more effort. A more serious disadvantage of using RDMA is that often, the underlying system needs to dedicate one core from each SMP to address these accesses transparently and efficiently [6].

Distributed memory architecture. Work stealing algorithms for distributed memory architectures differ from shared memory architectures for several reasons. For one, on shared memory architectures, synchronization primitives, caching and coherence protocols are often taken for granted as the underlying hardware takes care of these issues. Implementing these global operations in a distributed memory architecture often results in high runtime overheads and latencies. Another reason is that some algorithms simply do not scale. For example, it is no longer feasible to allow different tasks to spin on a common memory location, due to the absence of shared memory. Attempting to do so, would introduce a terrible amount of contention at certain nodes due to messaging. Clearly, different algorithms are needed which are suited for distributed memory architectures.

Solutions which use direct management of communication operations using explicit message passing have been shown to be viable [1]. Different algorithms such as Dijikstra's Termination Detection algorithm [4] are more suitable than cancellable barriers for termination detection in a distributed setting.

Practical solutions are typically designed assuming a completely distributed memory, using MPI or a similar message passing interface. A typical multi-core HPC cluster, however, has both shared and distributed memory. It is extremely straight forward to extend the implementation to an entire multi-core cluster by simply running different tasks on different cores, irrespective of whether or not they share any common memory. This is not an optimal solution, since communicating via MPI is certainly slower than through shared memory when it exists. However, this solution is still correct and allows for the execution of these work stealing algorithms across an entire cluster.

Our approach. Though extending the shared memory paradigm or the distributed memory paradigm to an entire multi-core cluster maybe correct, it is not necessarily efficient. As multi-core machines become more and more prevalent it is crucial to recognize the differences between shared memory architectures and distributed memory architectures. Our approach uses aspects from both methodologies to come up with a new more efficient algorithm.

Our approach uses two different load balancing strategies. One inside a multi-core node and one across multi-core nodes. For intra-node load balancing we use a popular algorithm which uses lockless task queues in shared memory and cancellable barriers for termination detection, while for inter-node load balancing we switch over to a pure MPI implementation and a termination detection algorithm similar to Dijikstra's. We show that such an approach is more efficient than previous approaches when there are a larger number of cores per node.

The ideas behind our approach can also be used to improve previous implementations. We would like to stress, that our ideas are orthogonal to previous approaches and they too can benefit from our techniques.

3 Design for Our Approach

3.1 Shared Memory Design

Our approach uses a popular algorithm for work stealing in shared memory multi-cores [6]. Split queues are used to alleviate locking overhead and a cancellable barrier is used to detect termination. Each core runs a single thread which is responsible for executing tasks.

The task queues are accessed very frequently and hence operations on them must provide efficient access. Each thread has one local task queue that it uses to maintain its list of tasks. When a thread generates more tasks and needs to add it to its local task queue, it enqueues the tasks at the front of the queue. When a thread needs to remove tasks from the queue, it dequeues them, again, from the front of the queue. When threads become idle, they search other task queues for work. If they find work in some other task queue, they will steal it by dequeue-ing it from the back of the queue.

First and foremost, this task queue should provide efficient access to the local thread, since it is on the critical path of execution. Any delays on task queue operations will directly be reflected in the execution time of the application. Other threads also need access to the task queue to enable work stealing. Concurrent access can be achieved by using a simple locking mechanism on the task queue. This would however add locking overhead for the local thread as well with every enqueue and dequeue operation. To alleviate the locking overhead we can use a single queue, but divide it into two distinct regions: a local region and a global region. The local region would remain lock free for access by the local thread and the global region would be synchronized through a lock. This is called a split queue (Figure 1).

Split queues need additional operations on top of the regular enqueue and dequeue operations to function properly. A thread continuously inserts tasks into the local portion of the task queue. If there is a sufficient amount of work in the local region it can choose to expose the excess work into the global region of the task queue. This operation involves invoking the lock of the global region of the queue. This operation of moving work into the global region of the queue is called the *release* operation. The *release* operations must be performed periodically ensuring enough work for other threads to steal. Correspondingly, when work in the local region of the split queue has been completed, it is then necessary to get some of the work from the global region of the queue (if it exists) back into the local part of the queue. This can be accomplished by simply moving the boundary between the local and global regions of the queue, further towards the global region. This operation must also be locked and is known as the *reacquire* operation. The *reacquire* operation is only performed when the work in the local portion is exhausted. Using the *release* and *reacquire* operations, locking is minimized to the global portions of the queue and the

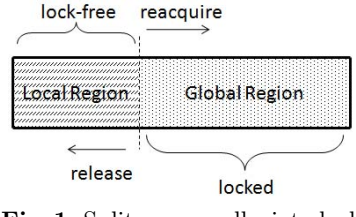


Fig. 1. Split queues alleviate locking overhead

accesses to the local region are lock free, except for the *release* and *reacquire* operations. This contributes minimally to the critical path overhead.

This kind of work stealing follows the depth-first work and breadth-first steal technique [2] that is used by many implementations including Cilk [9]. Our shared memory design is implemented in OpenMP.

Termination detection. Termination detection is achieved using a cancellable barrier. Cancellable barriers allow threads to check-in and also check-out when more work has been released into the system. Once a thread finds that the global regions are empty, it checks into the cancellable barrier. Indeed, this does not mean that there is no work in the system, it simply means that there is no work in the global region of any of the task queues. Other threads could be processing tasks and they could have tasks in their local regions which have not yet been released into the global region. Barriers can be canceled by another thread when it releases work into the system by performing a *release* operation. This kind of cancellable barrier does not scale across nodes in a cluster because of its centralized nature.

3.2 Distributed Memory Design

So far we have described the design of a shared memory implementation of work stealing. To expand this across the cluster we need a way to steal across nodes. We could have used a programming model like PGAS to convert any shared memory steals across nodes into messages sent through the underlying interconnect. However, due to remote locking latencies and overheads of the PGAS language and runtime, we found that co-ordinating cross node steals through MPI was much more efficient.

Hence, MPI is used to achieve load balancing across nodes. MPI provides for efficient message passing which is under our control rather than a PGAS language [14,7]. One important consideration which guided our design was that stealing from a thread in the same node is many times faster than stealing from another node in the cluster (in our experiments, about 50 times faster). Hence, we always perform work stealing inside a node before we cross the node boundary. This important guiding principle can be applied to any work stealing algorithm for improved performance.

One approach to enable work stealing across nodes would be to allow individual threads to send MPI steal requests to different nodes when they detect that there is no more work in the local node. This design requires a multi-threaded implementation of MPI and introduces locking overheads in the MPI runtime. To avoid any locking overhead and to maximize portability of our implementation we needed to use a single threaded implementation. This led us to choose a design in which we designate one thread per node to be in charge of cross node MPI steal requests. This designated thread will send cross node steal requests only when all the work on the local node has completed since stealing intra-node is much more quick than stealing across nodes. The designated thread (hereafter referred to as *thread0*) is also in charge of responding to steal requests from other nodes as well as termination detection.

thread0 will begin sending out cross node steals when it has exhausted all the tasks on its task queue and when all the other threads have checked into the cancellable barrier. We provide a special check-in mechanism for *thread0* so that it can "peek" into the current state of the barrier and wait till all the other threads have arrived at the barrier. By using this "peek" check-in,*thread0*, can determine when all the other threads have finished. If some thread generates work or if *thread0* finds work, the barrier gets canceled and all the other threads resume work stealing.

thread0 of a given node will choose a victim and send out a steal request to that victim. The *thread0* of every node processes tasks from its task queue just like every other thread, but in addition, it also periodically checks for any new steal messages it might need to service. If it has work in its local task queue, it will dequeue several nodes (controlled by parameter *chunksize*) it and send it out the thief. If it has no work, it will respond with the fact that it has no work. If *thread0* of a node, sends out a steal request and it gets back work, it will enqueue it onto its local task queue and cancel the barrier to wake all other thread up. If on the other hand, it gets a message saying that there is no work at the victim node, it will choose another victim and proceed. Using this approach, the maximum number of outstanding messages in the system will be bounded by the number of nodes since each node sends out steal requests one at a time. Different methodologies can be adopted for choosing the order in which victims are selected, however, we employ random work stealing which has been proven to be optimal [2].

If we had used a shared memory paradigm (a PGAS language like UPC) the steal operations will disturb the working threads in other nodes because these threads will be forced to wait for the global regions of these task queues to be unlocked to perform any *release* or *reacquire* operations. The cost of these interfering remote locking operations is typically an order of magnitude greater than the cost of a shared variable reference [15]. However, our implementation using MPI similar to [7], enables the thread to perform operations on their local task queues without waiting for locks from threads external to the node (which have the highest latency). They simply service steal requests at regular intervals removing the necessity of locks from external threads. Our distributed memory design is implemented in MPI.

Termination detection. The described techniques enable efficient work stealing across nodes in a HPC cluster, but this is not enough. We also need to detect termination across all the nodes in the cluster. If after attempting to steal from other nodes, *thread0* does not find any work in other nodes it will begin the global (across cluster nodes) termination detection process.

For termination detection in a single node we employed a cancellable barrier. This approach is simply not scalable to an entire cluster for reasons explained previously. We need a different algorithm to detect termination across the cluster. Many algorithms have been proposed in literature. We chose to use a modified version of the well known Dijikstra's termination detection algorithm [5] similar to [7] which is a token based termination detection algorithm. We refrain from describing the algorithm here due to a lack of space. Details can be found in [5].

Recall, that for *thread0* to have participated in the global termination detection process, it must have finished sending out all its cross node steal requests. And to have sent out cross node steal requests, it must have been the case that the other threads in the node are still waiting on the cancellable barrier. Once, *thread0* determines that global termination has been reached, it performs a complete check-in to the cancellable barrier on the node (as against a "peek" check-in that it normally performs). Once, *thread0* checks in, threads waiting at the cancellable barrier are notified that global termination has been reached and all threads terminate execution.

3.3 Combined Approach

The combination of the above schemes provides for a very efficient work stealing approach for HPC clusters. To summarize, we prioritize intra-node steals over inter-node steals and use a different algorithm for intra-node steals (asynchronous steals and cancellable barriers) and for inter-node steals (polling for steal requests and Dijikstra's termination detection). Note that while better victim selection in the previous approaches would improve their performance it is insufficient and the use of two different algorithms is vital. Figure 2 depicts all the interactions in the form of a state diagram.

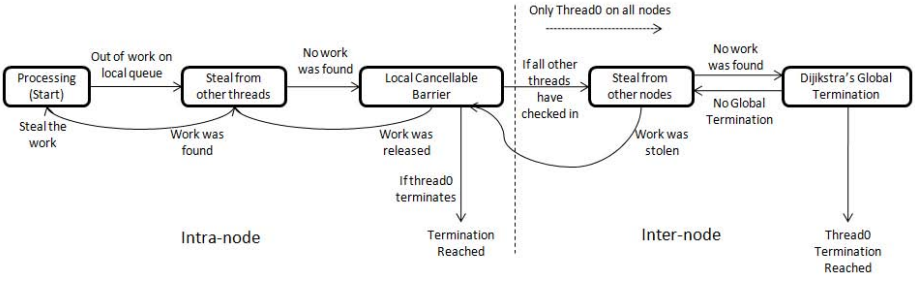


Fig. 2. Interactions of the algorithm

4 Evaluation

4.1 UTS

The Unbalanced Tree Search benchmark has been designed to be representative of a class of HPC applications which require substantial dynamic load balance [14]. Applications that fit this category include many search and optimization problems that must enumerate a large state space of unknown or unpredictable structure. UTS has become *the* benchmark with which load balancing algorithms are benchmarked. The benchmark is itself based on a simple problem - the parallel exploration of an unbalanced tree. The problem is to count the total number of nodes that this tree generates. The tree is generated through an implicit construction that is parametrized in shape, depth, size and imbalance. The tree is generated by traversing down the tree from the root node. When we process the root, its children (nodes) are generated. Now, each of these nodes

are recursively processed to generate the entire tree. While the processing time of each node is fairly constant, there is a high amount of variance in the sizes of each of the sub-trees leading to imbalance in workloads. [14] can be referred for a more detailed explanation. Figure 3 depicts an unbalanced tree showing a weblog file [10]. There are two types of trees that are used as part of the benchmark: Geometric Trees and Binomial Trees. Details can be found in [14].

4.2 Results

In this section we compare our implementation with two highly tuned state of the art implementations: one which extends a shared memory paradigm to a multi-core cluster using UPC [15] and another which extends a distributed memory paradigm to a multi-core cluster using purely MPI [7].

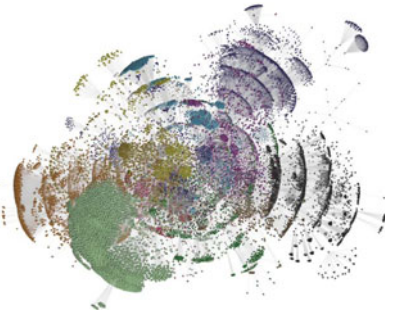


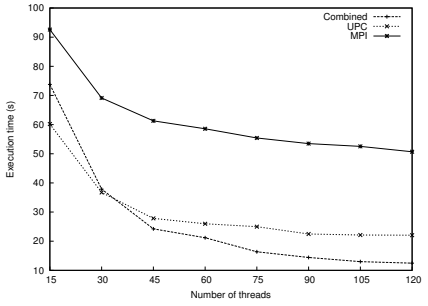
Fig. 3. Unbalanced tree showing a weblog

The UPC implementation extends a shared memory paradigm to the entire multi-core cluster. Accesses to remote memory are converted under the hood into cross node MPI messages. It has been found that simply using UPC and extended a shared memory algorithm is simply not scalable [15,14,7] and the authors of this implementation had to use several techniques such as polling instead of asynchronous stealing to improve performance to an acceptable level. In our experiments we have found this tuned UPC implementation to be faster than the MPI version, consistent with previous findings [15]. Our approach is significantly faster than the current implementations at higher thread/node counts. In this section we will refer to the three implementations as: the Combined approach (our method), the UPC approach and the MPI approach.

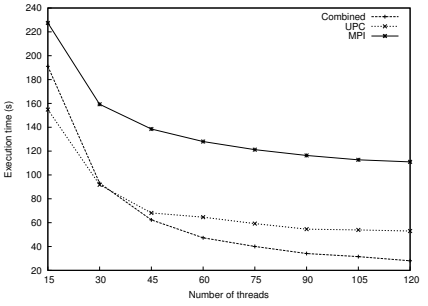
For our experiments we used a 15-node, 120-core IBM BladeCenter H Linux cluster with 2 socket x Core2 quad processors. Each node supported the parallel execution of up to 8 threads. This allowed us to observe the behavior of the three implementations as we increased the number of threads on each core from 1 to 8. The nodes in the cluster were connected using Ethernet and we used MPI for message passing across nodes. The UPC implementation was compiled with Pthreads support which enabled intra-node communication to happen through shared memory.

Tests were performed using 3 trees generated by the UTS benchmark. We used two geometric trees (GEO1 and GEO2) and one binomial tree (BIN1)[1]. We increased the number of threads running per core from 1 to 8 hence scaling

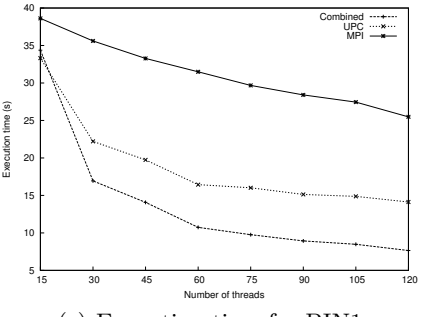
[1] For reproducibility we provide the exact parameters used to generate the tree.
GEO1 (1635119272 nodes): Geometric (Fixed), $d = 15$, $b_0 = 4$, $rootseed = 29$.
GEO2 (4230646601 nodes): Geometric (Fixed), $d = 15$, $b_0 = 4$, $rootseed = 19$.
BIN1 (1060980001 nodes): Binomial, $b_0 = 2000$, $q = 0.024999999975$, $m = 40$, $rootseed = 316$.



(a) Execution time for GEO1



(b) Execution time for GEO2



(c) Execution time for BIN1

Fig. 4. Execution Time

Table 1	Inter-node	Intra-node
Combined	5233	132919
UPC	13128	1501
MPI	800501	N/A

Table 2	Inter-node	Intra-node
No. steals	5233	132919
Total time	0.25923	0.13852
Time/steal (s)	0.000049539	0.000001042

(a) Table 1: Number of steals for different approaches. Table 2: Time taken for each steal in the Combined Approach in sec.

Fig. 5. Steals Breakdown

the implementation from 15 threads (one on each node) to 120 threads (8 on each node). Figure 4 shows the results we obtained.

Discussion. Our implementation which combined the use of two different algorithms consistently performed better than the two other implementations at higher core counts on GEO1, GEO2 and BIN1. The MPI implementation is slower than both the combined approach and the UPC approach in all the tests. Let us consider the case when only 1 thread runs on each node of the cluster. We call this the *baseline case*. Each additional test increases the number of threads on the node by 1. In the *baseline case*, the Combined approach is slower than

the UPC approach by as much as up to 23%. With 2 threads running per node, the Combined approach is slower than the UPC approach by approximately 2%. However, as we increase the number of threads per node, the Combined approach consistently performs better, with almost a 20% improvement when using 4 threads/node which only increases as we increase the number of threads/node.

The fact that the Combined approach is slower than the UPC implementation when using a smaller number of threads points to two facts. That at lower thread counts, the overheads of using two separate termination detection and stealing algorithms slows down the overall execution and also that the base UPC implementation is highly tuned and performs very well on machines with low threads/node. However, we observe that at higher threads/node counts the Combined approach provides significant speedups over the UPC approach and the MPI approach.

Table 1 in 5(a) shows the breakdown of the number of inter-node and intra-node steals performed by the various approaches. These numbers were obtained by running GEO2 with a total of 90 threads on 15 nodes. The Combined approach prioritizes intra-node steals over inter-node steals. The UPC implementation however, performs a much higher number of inter-node steals while compared to intra-node steals (however, roughly the same proportion considering that there are 15 nodes). The MPI implementation performs only inter-node steals since we assume a completely distributed memory. Table 2 in 5(a) points to the fact that inter-node steals are almost 50 times slower than intra-node steals!

The underlying reason behind the speedups is the fact that we use two different algorithms for inter-node steals and intra-node steals. The speedup cannot be attributed to the use of shared memory alone, since the UPC implementation was compiled with Pthreads support which enables use of shared memory for threads which are collocated on the same node. Similarly the speedup cannot be attributed to improved victim selection (based on locality). In our experiments we found that tuning victim selection in the UPC and MPI approach improved performance by only a modest 2% (at 8 threads/node). We conclude that the most important factor in the speedup is the use of a different algorithm for inter-node steals and intra-node steals. Retrofitting either a shared memory paradigm or a distributed memory paradigm on top of a multi-core cluster does not perform as well at higher thread/node count.

5 Related Work

A large amount of effort has been invested in studying different kinds of load balancing algorithms. Load balancing methods have been broadly classified into static methods and dynamic methods. Static methods such as [16,12] are suitable in situations where work can be divided fairly before execution. Task graph scheduling [13] finds a schedule given a set of tasks which are organized as a graph. Dynamic approaches so far have focused on using a distributed memory approach using MPI [7] and techniques like RDMA [17] or using a shared memory approach [14] using OpenMP. PGAS languages extend the shared memory

paradigm across an entire cluster (containing nodes with distributed memory) hence allowing load balancing algorithms to work across an entire cluster. Certain languages like X10 [3], Cilk [9] employ dynamic load balancing techniques. Hierarchical techniques have been proposed in ATLAS [11]. Work stealing techniques, in general, have been well studied and have been shown to be applicable to various applications on distributed memory machines such as [17,8].

6 Conclusion and Future Work

In this paper we propose an algorithm which uses different algorithms for inter-node and intra-node steals and demonstrate marked improvements on the UTS benchmark over current implementations. Future work involves optimizing our implementation using various techniques such as work pushing and using smarter victim identification schemes. We strongly believe that the concepts that are presented in this paper can be applied to current implementations and can be used in designing work stealing algorithms in the future. We would like to thank the anonymous reviewers for their comments. We also gratefully acknowledge the support of NSF grants CCF-1018544 and CCF-0916962.

References

1. Berlin, K., Huan, J.: Evaluating the impact of programming language features on the performance of parallel applications on cluster architectures. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 194–208. Springer, Heidelberg (2004)
2. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM 46, 720–748 (1999)
3. Charles, P., Grothoff, C., Saraswat, V.: X10: an object-oriented approach to non-uniform cluster computing. SIGPLAN Not. 40, 519–538 (2005)
4. Scholten, C.S., Dijikstra, E.W.: Termination detection for diffusing computations (1980)
5. Dijkstra, E.W., Scholten, C.S.: Termination detection for diffusing computations. Information Processing Letters 11(1), 1–4 (1980)
6. Dinan, J., Larkins, D.B., Sadayappan, P., Krishnamoorthy, S., Nieplocha, J.: Scalable work stealing. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009, pp. 53:1–53:11. ACM, New York (2009)
7. Dinan, J., Olivier, S., Sabin, G., Prins, J., Sadayappan, P., Tseng, C.-W.: Dynamic load balancing of unbalanced computations using message passing. In: IPDPS 2007, IEEE International, pp. 1–8 (2007)
8. Dowaji, S., Roucairol, C.: Load balancing strategy and priority of tasks in distributed environments (1994)
9. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the cilk-5 multithreaded language. SIGPLAN Not. 33, 212–223 (1998)
10. Isenberg, P.: Phyllotactic patterns for tree layout, `http://pages.cpsc.ucalgary.ca/~pneumann/wiki/pmwiki.php?n=MyUniversity.PhylloTrees`
11. Eric Baldeschwieler, J., Blumofe, R.D., Brewer, E.A.: Atlas: An infrastructure for global computing (1996)

12. Kim, C., Kameda, H.: An algorithm for optimal static load balancing in distributed computer systems. IEEE Trans. Comput. 41, 381–384 (1992)
13. Kwok, Y.-K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput. Surv. 31, 406–471 (1999)
14. Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P., Tseng, C.-W.: Uts: an unbalanced tree search benchmark. In: Almási, G.S., Caşcaval, C., Wu, P. (eds.) KSEM 2006. LNCS, vol. 4382, pp. 235–250. Springer, Heidelberg (2007)
15. Olivier, S., Prins, J.: Scalable dynamic load balancing using upc. In: ICPP 2008, pp. 123–131. IEEE Computer Society, Washington, DC, USA (2008)
16. Tantawi, A.N., Towsley, D.: Optimal static load balancing in distributed computer systems. J. ACM 32, 445–465 (1985)
17. van Nieuwpoort, R.V., Kielmann, T., Bal, H.E.: Efficient load balancing for wide-area divide-and-conquer applications. SIGPLAN Not. 36, 34–43 (2001)

A Dynamic Power-Aware Partitioner with Task Migration for Multicore Embedded Systems

José Luis March, Julio Sahuquillo, Salvador Petit, Houcine Hassan, and José Duato

Department of Computer Engineering (DISCA)
Universitat Politècnica de València, Spain
jomarcab@gap.upv.es,
{jsahuqui,spetit,husein,jduato}@disca.upv.es

Abstract. Nowadays, a key design issue in embedded systems is how to reduce the power consumption, since batteries have a limited energy budget. For this purpose, several techniques such as Dynamic Voltage Scaling (DVS) or task migration can be used. DVS allows reducing power by selecting the optimal voltage supply, while task migration achieves this effect by balancing the workload among cores.

This paper first analyzes the impact on energy due to task migration in multicore embedded systems with DVS capability and using the well-known Worst Fit (WF) partitioning heuristic. To reduce overhead, migrations are only performed at the time that a task arrives to and/or leaves the system and, in such a case, only one migration is allowed.

The huge potential on energy saving due to task migration, leads us to propose a new dynamic partitioner, namely DP, that migrates tasks in a more efficient way than typical partitioners. Unlike WF, the proposed algorithm examines which is the optimal target core before allowing a migration. Experimental results show that DP can improve energy consumption in a factor up to 2.74 over the typical WF algorithm.

1 Introduction

Embedded systems is an important segment of the microprocessor market since they are becoming ubiquitous in our life. Systems like PDAs, smart phones, or automotive, provide an increasing number of functionalities such as voice communication, navigation, or gaming, so that computational power is becoming more important every day. However, increasing computational power impacts on battery lifetime, so how to improve power management is a major design concern.

To deal with both computational and power management requirements, many systems use multicore processors. These processors allow a better power management than complex monolithic processors for the same level of performance. Moreover, many manufacturers (Intel, IBM, Sun, etc.) deliver processors providing multithreading capabilities, that is, they provide support to run several threads simultaneously. Some examples of current multithreaded processors are Intel's Montecito [14] and IBM Power 5 [10]. Also, leading manufacturers of the embedded sector, like ARM, plan to include multithreading technology in next-generation processors [16].

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 218–229, 2011.

A power management technique that is being implemented in most current microprocessors is Dynamic Voltage Scaling (DVS) [9]. This technique allows the system to improve its energy consumption by reducing the frequency when the processor has a low level of activity (e.g., a mobile phone that is not actively used). In a multicore system, the DVS regulator can be shared among several cores also referred to as *global* or private to each core. In the former case, all cores are forced to work at the same speed but less regulators are required so it is a cheaper solution. The latter case, enables more energy savings since each core frequency can be properly tuned to its applications requirements but it is more expensive [15].

Energy consumption in systems with a global DVS regulator can be further improved by properly balancing the workload [7,13]. To this end, a partitioner module is in charge of distributing tasks according to a given algorithm (e.g., Worst Fit [1] or First Fit) that selects the target core to run the task. Unfortunately, the nature of some workload mixes prevents the partitioner from achieving a good balancing. To deal with this drawback some systems allow tasks to migrate (move their execution) from one core to another, which results in energy saving improvements.

This work presents a dynamic power-aware partitioner, namely DP, for a multicore multithreaded system that dynamically (at run-time) assigns tasks to cores and allows task migration to improve energy consumption. Our focus is on tasks presenting real-time constraints, that is, tasks must end their execution before a given deadline or run during several periods before leaving the system. The proposed partitioner readjusts possible dynamic imbalances (due to new arrivals or exits of tasks) by reallocating tasks among cores. In this way, the workload can be more fairly balanced, so system frequency -in many cases- can be reduced, thus enabling further energy consumption improvements. In addition, the number of migrations has been limited in order to reduce overhead.

Finally, as the aim of migration is to reduce imbalance, it makes sense to analyze the benefits of applying migration when the workload changes. Three cases have been analyzed: when a task arrives to the system, when a task leaves the system, and both cases together. Experimental results show that enabling migration *only on arrival* in the classical WF algorithm allows achieving energy improvements in a factor up to 2.18 with respect to the case where no migration is allowed, while in the proposed DP algorithm these improvements can be up to 2.74.

The remaining of this paper is structured as follows. Section 2 discusses the related research on energy management and task migration. Section 3 describes the modeled system, including the partitioner and the power-aware scheduler. Section 4 presents the proposed workload partitioning algorithms. Section 5 analyzes experimental results of performance and energy. Finally, Section 6 presents some concluding remarks.

2 Related Work

Scheduling in multiprocessor systems can be performed in two main ways depending on the task queue management: *global scheduling*, where a single task queue is shared by all the processors, or *partitioned scheduling*, that uses a private task queue for each processor. The former allows task migrations since all the processors share the same

task queue. In the latter case, the scheduling in each processor can be performed by applying well-established uniprocessor theory algorithms such as EDF (Earliest Deadline First) or RMS (Rate Monotonic Scheduling). An example of global scheduling for sporadic tasks can be found in [11].

In the partitioned scheduling case, research can focus either on the partitioner or the scheduler. Acting in the partitioner, recent works have addressed the energy-aware task allocation problem [19,2,1]. For instance, Wei et al. [19] reduce energy consumption by exploiting parallelism of multimedia tasks on a multicore platform combining DVS with switching-off cores. Aydin et al. [2] present a new algorithm that reserves a subset of processors for the execution of tasks with utilization not exceeding a threshold. Unlike our work, none of these techniques use task migration among cores.

Some proposals have been dealing with task migration. Brandenburg et al. [4] evaluate some scheduling algorithms (both global and partitioned) in terms of scalability, although no power consumption were investigated. In [21] Zheng divides tasks into fixed and migration tasks, allocating each of the latter to two cores, so they can migrate from one to another. Unlike our work, in this paper there is no consideration about dynamic workload changes (tasks arriving to and leaving the system), instead, all tasks are assumed to arrive at the same instant, so migrations can be scheduled off-line. Seo et al. [15] present a dynamic repartitioning algorithm with migrations to balance the workload and reduce consumption. In [5] Brião et al. analyze how soft tasks migration affects NoC-based MPSoCs in terms of deadline misses and energy consumption. These two latter works focus on non-threaded architectures.

Regarding the scheduler, in [8] El-Haj-Mahmoud et al. virtualize a simultaneous multithreaded (SMT) processor into multiple single-threaded superscalar processors with the aim of combining high performance with real-time formalism. In order to improve real-time tasks predictability, Cazorla et al. [6] devise an interaction technique between the Operating System (OP) and an SMT processor. Notice that these works do not tackle energy consumption.

3 System Model

Figure 1 shows a block diagram of the modeled system. When a task reaches the system, a partitioner module allocates it into a task queue associated to a core, which contains the tasks that are ready for execution in that core. These task queues are components of the power-aware scheduler that communicates with a DVS regulator, in charge of adjusting the working frequency of the cores in order to satisfy the workload requirements. To focus our research, experiments considered a two-core processor implementing three hardware threads each.

Processor cores implement the coarse-grain multithreading paradigm that switches the running thread when a long latency event occurs (i.e., a main memory access). Thus, the running thread issues instructions to execute while the other threads access memory, so overlapping their execution. In the modeled system, the issue slots are always assigned to the thread executing the task with the highest real-time priority. If this thread stalls due to a long latency memory event, then the issue slots are temporarily reassigned until the event is resolved.

3.1 Task Real-Time Behavior

The system workload executes periodic hard real-time tasks. There is no task dependency and each task has its own period of computation. A task can be launched to execute at the beginning of each active period, and it must end its execution before reaching its deadline (hard real-time). The end of the period and the deadline of a task are considered to be the same for a more tractable scheduling process. There are also some periods where tasks do not execute since they are not active (i.e., inactive periods). In short, a task arrives to the system, executes several times repeatedly, leaves the system, remains out of the system for some periods, and then it enters the system again. This sequence of consecutive active and inactive periods allows to model real systems mode changes.

Besides its period and deadline, a task is also characterized by its Worst Case Execution Time (WCET). This parameter is used to obtain the task utilization: $U = \frac{WCET}{Period}$. Different partitioning algorithms may use this value in the process of allocating incoming tasks to a core, guaranteeing schedulability.

3.2 Power-Aware Scheduler

Once a task is allocated to a core, it is inserted into the task queue of that core, where incoming tasks are ordered according to the EDF policy [3], which priorizes the tasks with the closest deadlines. Thus, the three tasks with the closest deadlines will be always mapped into the three hardware threads implemented in each core.

The scheduler is also in charge of calculating the target speed of each core according to the tasks's requirements. In this sense, in order to minimize power consumption, each core will choose the minimum frequency that fulfills the temporal contraints of its task set. This information is sent to the DVS regulator that selects the maximum frequency/voltage level among the requested by the cores.

The target frequency is recalculated to check if it has to be updated, but only when the workload changes, that is, when a task arrives to or/and leaves the system. In the former case, a higher speed can be required because the workload increases. In the latter

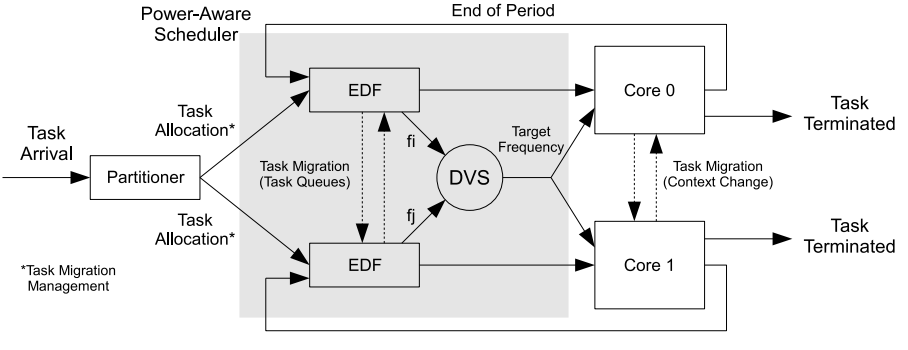


Fig. 1. Modeled system

Table 1. Energy (E) used per frequency (F)

F[MHz]	500	400	300	200	100
E[pJ/cycle]	450	349.2	261.5	186.3	123.8

case, it could happen that a lower frequency could satisfy the deadline requirements of the remaining tasks.

Different speed values are considered for the power-aware scheduler, based on the frequency levels of a Pentium M [18] that are shown in Table 1. The 5L configuration allows the system to work at any of these five levels, whereas the 3L mode permits running tasks at the highest, the lowest and the intermediate (300 Mhz) frequency. Futhermore, the overhead of changing the frequency/voltage level has been modeled according to the voltage transition rate in the Pentium M processor, that is approximately $1mv/1\mu s$ [20].

4 Partitioning Heuristics with Task Migration

There are several partitioning heuristics that can be used to distribute tasks among cores as they arrive to the system. The Worst Fit (WF) partitioning heuristic is considered one of the best choices in order to balance the workload, thus improving energy savings [1]. WF balances workload by assigning the incoming task to the least loaded core. If more than one task arrives to the system at the same time, it arranges the incoming tasks by increasing utilization order and assigns them to the cores beginning with the task with highest utilization. This algorithm was initially used in partitioned scheduling, thus, it does not support task migration among cores by design. Therefore, once WF has assigned an incoming task to a given core, the task remains in that core until it leaves the system (i.e., it has executed all its active periods).

4.1 Extending Worst Fit to Support Task Migration

Figure 2 shows an example of how task migration could improve workload balancing. At the beginning of the execution (time t0), task 0 and task 1 are the only tasks assigned to core 0 and core 1, respectively. Task 0 presents a utilization around 25% (i.e., its WCET occupies a quarter of its period), while the utilization of task 1 is around 33%. At point t2, task 2, whose utilization is around 66%, arrives to the system, and the WF algorithm assign it to core 0 (since it is the least loaded core). Consequently, the system would exhibit a high workload imbalance since the global utilization of core 0 and core 1 would be 91% and 33%, respectively. To solve this imbalance, task 0 can bee migrated to core 1, so providing a better balance (66% in core 0 versus 58% in core 1).

The system can become unbalanced when the workload changes, that is, when a task arrives to or leaves the system. Thus, migration policies should apply in these points in order to be effective. This leads to three variants of the WF policy: WF_{in}, WF_{out}, and WF_{in-out}. WF_{in} allows migration only when a new task arrives to the system, WF_{out} when a task leaves the system, and WF_{in-out} allows migration in both previous cases. To avoid performing too much migrations, which could lead to excessive overhead, we

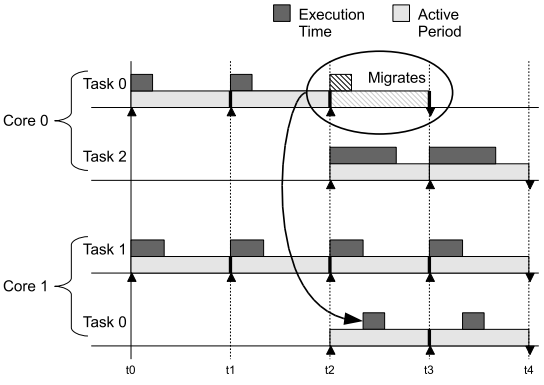


Fig. 2. Task periods and migrations

limit the number of migrations performed when a task arrives to or leaves the system to only one.

Figure 3 shows the Migration Attempt (MA) algorithm. This routine calculates the imbalance by subtracting the utilization of the least loaded core from the utilization of the most loaded one. This result is divided by two (since there are two cores) to obtain a theoretical utilization value that represents the amount of work that should migrate to achieve a perfect balancing. Then, it searches the task in the most loaded core whose utilization is the closest to this one. Notice that it could happen that by migrating that task the workload balancing would not improve (e.g., consider a situation where only one task is assigned to the most loaded core). Therefore, the algorithm performs the migration only if it improves the workload balancing.

1: $imbalance \leftarrow max\_core\_utilization - min\_core\_utilization$
2: $target\_utilization \leftarrow imbalance/2$
3: $minimum\_difference \leftarrow MAX\_VALUE$
4: **for all** task in most_loaded_core **do**
5: **if** $|U_{task} - target\_utilization| < minimum\_difference$ **then**
6: $minimum\_difference \leftarrow |U_{task} - target\_utilization|$
7: $candidate \leftarrow task$
8: **end if**
9: **end for**
10: $new\_max\_core\_utilization \leftarrow max\_core\_utilization - U_{candidate}$
11: $new\_min\_core\_utilization \leftarrow min\_core\_utilization + U_{candidate}$
12: $new\_imbalance \leftarrow |new\_max\_core\_utilization - new\_min\_core\_utilization|$
13: **if** $new\_imbalance < imbalance$ **then**
14: $migrate(candidate)$
15: **end if**

Fig. 3. Migration Attempt algorithm

4.2 Dynamic Partitioner

This subsection presents the proposed Dynamic Partitioner (DP). As done by the WF algorithm, DP also arranges the tasks arriving to the system by increasing utilization order. However, before assigning any incoming task to a given core, DP checks how the workload balancing would become if the incoming task were assigned to the first core. Then, it also calculates the effect of performing a migration attempt (as shown in Figure 3). These testings are performed for each core in the system. Finally, the core assignment that provides the best overall balance is applied. Two versions of DP are considered: DP_{in} and DP_{in-out}. DP_{in} refers to the described DP algorithm, where a migration can be performed only when a task arrives to the system, while DP_{in-out} also performs a migration attempt when a task leaves the system.

Figure 4 depicts an example where the DP_{in} heuristic improves the behavior of WF_{in}. The latter allocates the incoming task to core 0, and then performs a migration attempt, but in this case, there is not any possible migration enabling a better workload balancing. Thus, the final imbalance becomes 20% (i.e., $90\% - 70\%$). In contrast, when DP_{in} is applied, it also checks the result of allocating the new task to core 1 (DP_{in} B arrow) and then considering one migration. In this case, the migration enables a better balance since both cores remain equally loaded with 80% of utilization, which will be the distribution selected by DP_{in}.

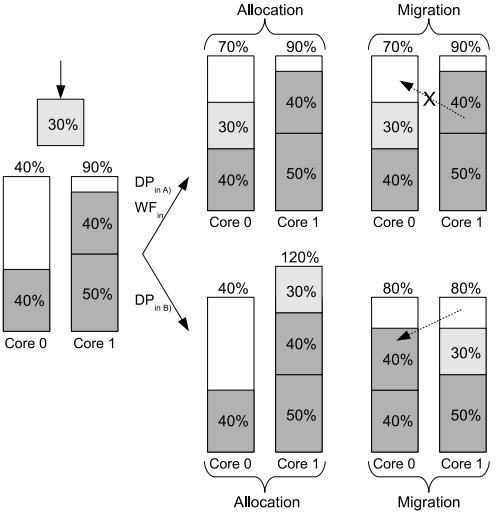


Fig. 4. WF_{in} vs DP_{in}

To sum up, the main difference between WF_{in} and DP_{in} is that the former selects only one core and performs a migration attempt, whereas the proposed heuristic checks different cores, and choses the best option in terms of workload balance.

5 Experimental Results

Experimental evaluation has been conducted by extending the Multi2Sim simulation framework [17], to model the system described in Section 3. As stated before, experiments considered a two-core processor implementing three hardware threads each. Internal core features have been modeled like an ARM11 MPCore based processor, but modified to work as a coarse-grain multithreaded processor with in-order execution, two-instruction issue width, and a 100-cycle memory latency.

Table 2 shows the benchmarks from the WCET analysis project [12] that were used to prepare real-time workload mixes. These mixes have been designed taking into account aspects such as task utilization, number of repetitions (task periodicity), and the sequence of active and inactive periods. The global system utilization varies in a single execution from 35% to 95%, in order to test the algorithms behavior across a wide range of situations. In addition, all results are presented and analyzed for a system implementing three and five voltage levels.

5.1 Impact of Applying Migrations at Different Points of Time

This section analyzes the best points of time to carry out migrations focusing on the standard WF algorithm (no migration is supported) and its variants supporting migration (WF_{out}, WF_{in}, WF_{in-out}). Figure 5 shows the relative energy consumption compared to the energy consumed by the system working always at the maximum speed for diverse benchmark mixes and DVS configurations.

As observed, migration can provide huge energy savings with respect to no migration (WF) regardless when migration is applied. For instance, in the 5-level system with task migration mixes 3 and 4 improve their energy consumption in a factor up to 1.33 and 2.18, respectively, when compared with their execution in the same system without migrations. This trend is also followed, although to a lesser extent, in the 3-level system.

Comparing the three WF versions with task migration, it can be observed that if migration can apply only each time a new task arrives instead of when a task terminates,

Table 2. Benchmark description

Name	Function Description	Name	Function Description
adpcm	Adaptive pulse code modulation algorithm	insertsort	Insertion sort on a reversed array of size 10
bs	Binary search for a 15-element array	janne_complex	Nested loop program
bsort100	Bubblesort program	jfdctint	Discrete-cosine transformation
cnt	Counts non-negative numbers in a matrix	lcdnum	Read ten values, output half to LCD
compress	Data compression program	lms	LMS adaptive signal enhancement
cover	Program for testing many paths	ludcmp	LU decomposition algorithm
crc	Cyclic redundancy check on 40-byte data	matmult	Matrix multiplication of two 20x20 matrices
duff	Copy 43-byte array	minver	Inversion of floating point matrix
edn	FIR filter calculations	ns	Search in a multi-dimensional array
expint	Series expansion for integral function	nsichneu	Simulate an extended Petri Net
fac	Factorial of a number	qsort-exam	Non-recursive version of quick sort algorithm
fdct	Fast Discrete Cosine Transform	qurt	Root computation of quadratic equations
fft1	1024-point Fast Fourier Transform	select	Nth largest number in a floating point array
fibcall	Simple iterative Fibonacci calculation	sqrt	Square root function
fir	Finite impulse response filter	statemate	Automatically generated code

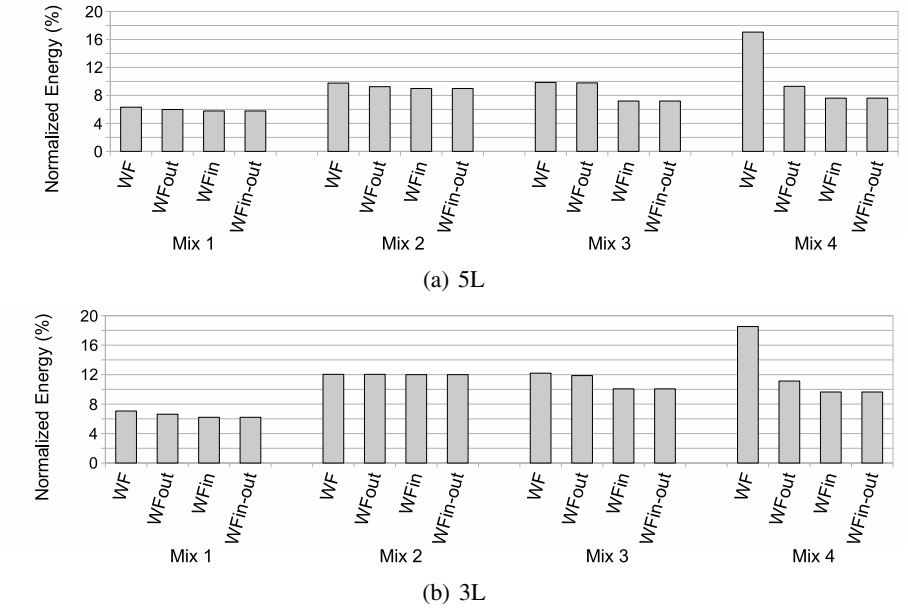


Fig. 5. Worst Fit variants comparison for different DVS levels

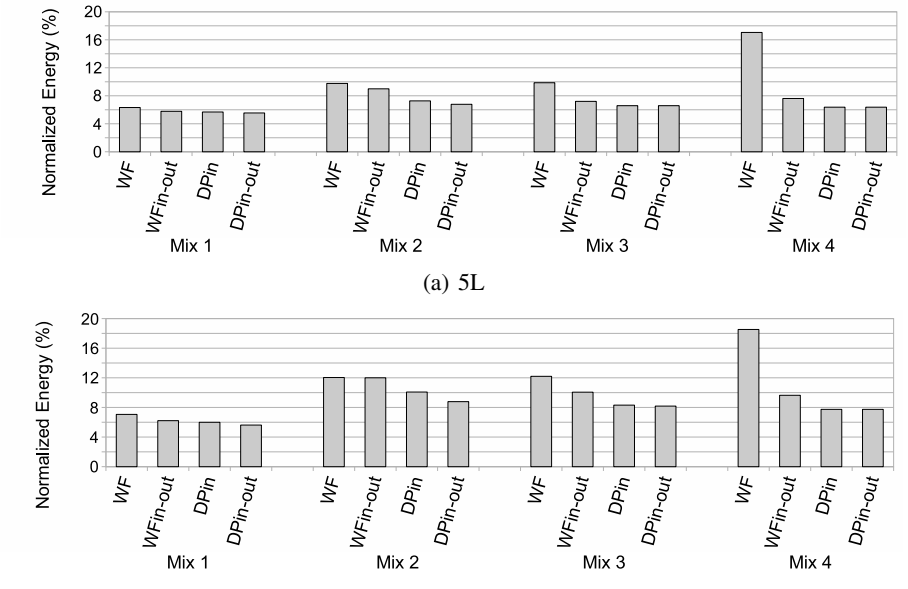
then much higher energy savings can be achieved. The main reason is that the inter-arrival time standard deviation is higher than that of the inter-leaving time, since several tasks reach the system at the same time. Inter-arrival standard deviation values of the mixes are 3.87, 24.48, 43.98, and 14.65 Mcycles for mix 1, mix 2, mix 3, and mix 4, respectively. On the other hand, the inter-leaving time is, on average, 3.65, 22.50, 36.40, and 12.32 Mcycles. Finally, WF_{in-out} offers scarce benefits over WF_{in} since it only adds a low number of extra migrations.

Notice that if the system implements more DVS frequency levels (5 levels in the figure), then more energy savings can be obtained since the system can select a frequency closer to the optimal estimated by the scheduler. However, despite this fact, an interesting observation is that energy benefits due to migration in the 3-level system can reach or even surpass the benefits of having the 5-level system without migrations. For example, the energy consumption of WF_{out} for mix 4 in the 3-level system is around 11% of the consumption of the baseline, whereas the same value of WF in the 5-level system is 17%.

5.2 Comparing DP versus WF Variants

This section analyzes the energy improvements of two variants of the proposed DP algorithm (DP_{in} and DP_{in-out}) over the WF algorithm. For comparison purposes the best variant of the WF (WF_{in-out}) with migration has been also included in the plots. Figure 6 shows the results.

Results show that, regardless the mix and system-level, both variants of DP always consume less power than WF_{in-out}. DP_{in-out} achieves, for mixes 3 and 4, energy



(a) 5L

(b) 3L

Fig. 6. WF versus DP for different DVS levels

improvements over WF in a factor up to 2.74 and 1.56, respectively. Moreover, for mix 2, where WF_{in-out} is only able to find scarce benefits over WF, the proposed DP improves the energy consumption of WF around 1.51.

For a better understanding of the algorithms behavior, we define the *migration rate* metric as the number of migrations performed by the algorithm divided by the number of times that the migration algorithm is executed. For instance, regarding the *in* variant of the WF and DP algorithms, the migration rates of WF_{in} are 64%, 62%, 54%, 45% for mix 1, mix 2, mix 3, and mix 4, respectively; while for DP_{in} the corresponding values are 64%, 76%, 68%, and 73%. This means that the proposal performs migrations in some cases where the WF is not able to find any candidate to migrate at all.

6 Conclusions

Workload balancing has been already proved to be an efficient power technique in multicore systems. Unfortunately, unexpected workload imbalances can rise at run-time provided that the workload is dynamically changing since new tasks arrive to or leave the system. To palliate this situation this paper has analyzed the impact on energy consumption of task migration combined with workload balancing.

To prevent excessive overhead, task migration has been strategically applied at three different execution times where the workload changes (at task arrival, at task termination, and in both cases). Results with respect to the WF algorithm showed that applying migration at arrival time can save results in a factor up to around 2.18. This results can be slightly improved if migration is also applied when tasks terminate.

Due to the potential of migration, this paper has proposed the DP algorithm, which achieves much better energy improvements than classical partitioning algorithms like WF. The proposal improves energy consumption in a factor of 1.51 in some workloads where WF with migrations provides scarce benefits, and energy can be improved in a factor up to 2.73 in the analyzed workloads.

Experimental results also showed that migration can provide energy consumption improvements with respect to a more complex system with a higher number of frequency/voltage levels. A final remark is that achieving a better workload balancing by allowing task migrations not only results in energy savings, but also allows a wider set of tasks to be scheduled.

Acknowledgments. This work was supported by Spanish CICYT under Grant TIN2009-14475-C04-01, and by Consolider-Ingenio under Grant CSD2006-00046.

References

1. AlEnawy, T.A., Aydin, H.: Energy-Aware Task Allocation for Rate Monotonic Scheduling. In: Proceedings of the 11th Real Time on Embedded Technology and Applications Symposium, March 7-10, pp. 213–223. IEEE Computer Society, San Francisco (2005)
2. Aydin, H., Yang, Q.: Energy-Aware Partitioning for Multiprocessor Real-Time Systems. In: Proceedings of the 17th International Parallel and Distributed Processing Symposium, Workshop on Parallel and Distributed Real-Time Systems, April 22-26, p. 113. IEEE Computer Society, Nice (2003)
3. Baker, T.P.: An Analysis of EDF schedulability on a multiprocessor. IEEE Transactions on Parallel and Distributed Systems 16(8), 760–768 (2005)
4. Brandenburg, B.B., Calandrino, J.M., Anderson, J.H.: On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study. In: Proceedings of the 29th Real-Time Systems Symposium, November 30-December 3, pp. 157–169. IEEE Computer Society, Barcelona (2008)
5. Brião, E., Barcelos, D., Wronski, F., Wagner, F.R.: Impact of Task Migration in NoC-based MPSoCs for Soft Real-time Applications. In: Proceedings of the International Conference on VLSI, October 15-17, pp. 296–299. IEEE Computer Society, Atlanta (2007)
6. Cazorla, F., Knijnenburg, P., Sakellariou, R., Fernández, E., Ramirez, A., Valero, M.: Predictable Performance in SMT Processors: Synergy between the OS and SMTs. IEEE Transactions on Computers 55(7), 785–799 (2006)
7. Donald, J., Martonosi, M.: Techniques for Multicore Thermal Management: Classification and New Exploration. In: Proceedings of the 33rd Annual International Symposium on Computer Architecture, June 17-21, pp. 78–88. IEEE Computer Society, Boston (2006)
8. El-Haj-Mahmoud, A., AL-Zawawi, A., Anantaraman, A., Rotenberg, E.: Virtual Multiprocessor: An Analyzable, High-Performance Architecture for Real-Time Computing. In: Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, September 24-27, pp. 213–224. ACM Press, San Francisco (2005)
9. Hung, C., Chen, J., Kuo, T.: Energy-Efficient Real-Time Task Scheduling for a DVS System with a Non-DVS Processing Element. In: Proceedings of the 27th Real-Time Systems Symposium, December 5-8, pp. 303–312. IEEE Computer Society, Rio de Janeiro (2006)
10. Kalla, R., Sinharoy, B., Tendler, J.M.: IBM Power5 Chip: A Dual-Core Multithreaded Processor. IEEE Micro 24(2), 40–47 (2004)

11. Kato, S., Yamasaki, N.: Global EDF-based Scheduling with Efficient Priority Promotion. In: Proceedings of the 14th International Conference on Embedded and Real-Time Computing Systems and Applications, August 25-27, pp. 197–206. IEEE Computer Society, Kaohisung (2008)
12. Malardalen Real-Time Research Center, Vasteras, Sweden: WCET Analysis Project. WCET Benchmark Programs (2006), [Online], http://www.mrtc.mdh.se/projects/wcet/
13. March, J., Sahuquillo, J., Hassan, H., Petit, S., Duato, J.: A New Energy-Aware Dynamic Task Set Partitioning Algorithm for Soft and Hard Embedded Real-Time Systems. To be published on The Computer Journal (2011)
14. McNairy, C., Bhatia, R.: Montecito: A Dual-Core, Dual-Thread Itanium Processor. IEEE Micro 25(2), 10–20 (2005)
15. Seo, E., Jeong, J., Park, S., Lee, J.: Energy Efficient Scheduling of Real-Time Tasks on Multicore Processors. IEEE Transactions on Parallel and Distributed Systems 19(11), 1540–1552 (2008)
16. Shah, A.: Arm plans to add multithreading to chip design. ITworld (2010), [Online], http://www.itworld.com/hardware/122383/arm-plans-add-multithreading-chip-design
17. Ubal, R., Sahuquillo, J., Petit, S., López, P.: Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors. In: Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing, October 24-27, pp. 62–68. IEEE Computer Society, Gramado (2007)
18. Watanabe, R., Kondo, M., Imai, M., Nakamura, H., Nanya, T.: Task Scheduling under Performance Constraints for Reducing the Energy Consumption of the GALS Multi-Processor SoC. In: Proceedings of the Design Automation and Test in Europe, April 16-20, pp. 797–802. ACM, Nice (2007)
19. Wei, Y., Yang, C., Kuo, T., Hung, S.: Energy-Efficient Real-Time Scheduling of Multimedia Tasks on Multi-Core Processors. In: Proceedings of the 25th Symposium on Applied Computing, March 22-26, pp. 258–262. ACM, Sierre (2010)
20. Wu, Q., Martonosi, M., Clark, D.W., Reddi, V.J., Connors, D., Wu, Y., Lee, J., Brooks, D.: A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. In: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture, November 12-16, pp. 271–282. IEEE Computer Society, Barcelona (2005)
21. Zheng, L.: A Task Migration Constrained Energy-Efficient Scheduling Algorithm for Multiprocessor Real-time Systems. In: Proceedings of the International Conference on Wireless Communications, Networking and Mobile Computing, September 21-25, pp. 3055–3058. IEEE Computer Society, Shanghai (2007)

Exploiting Thread-Data Affinity in OpenMP with Data Access Patterns*

Andrea Di Biagio, Ettore Speziale**, and Giovanni Agosta

Dipartimento di Elettronica ed Informazione, Politecnico di Milano
andrea.dibiagio@gmail.com, {speziale,agosta}@elet.polimi.it

Abstract. In modern NUMA architectures, preserving data access locality is a key issue to guarantee performance. We define, for the OpenMP programming model, a type of architecture-agnostic programmer hint to describe the behaviour of parallel loops. These hints are only related to features of the program, in particular to the data accessed by each loop iteration. The runtime will then combine this information with architectural information gathered during its initialization, to guide task scheduling, in case of dynamic loop iteration scheduling. We prove the effectiveness of the proposed technique on the NAS parallel benchmark suite, achieving an average speedup of 1.21x.

Current trends in computer architectures tend to increase the number of cores per chip to cope with the power and frequency walls while exploiting the transistor density increase. This is driving designers towards multi- and many-core architectures, where *Non-Uniform Memory Access* (NUMA) designs are needed [9,2]. In NUMA architectures, cores incur in greater delays when accessing non-local memories. Since NUMA machines preserve the shared memory abstraction, it is possible to program them using programming models such as OpenMP [3], which hide the complexity of the underlying memory hierarchy.

To achieve performance in NUMA architectures, it is essential to provide *data access locality*, that is, data located in a given node are accessed as much as possible from the cores of the same node, and as little as possible from the other ones [14,21]. Recent works targeting OpenMP on Linux focus on exploiting specialized page allocation policies [4] such as explicit data distribution, which allows the programmer to select a precise distribution to be implemented at initialization time. The *next-touch* policy, introduced in [8,6], allows dynamic data relocation by exploiting memory protection mechanisms.

However, such works incur in one or more of the following drawbacks: they rely on programmer knowledge of the underlying architecture, thus negating a major benefit of OpenMP, architecture independence [15]; they lack dynamism, since they provide only a single data distribution strategy which might not cover all the access patterns the program employs during different phases of its execution;

* This work was supported in part by the European Commission under Grant 2PARMA FP7-248716 and ARTEMIS-SMECY.
** This author was supported in part by a grant from ST Microelectronics.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 230–241, 2011.

or, they do not deal with workload balancing, which in turn adversely affects irregular parallel applications.

In this work, we take into account these issues, providing a solution to maintain thread-data affinity across the lifetime of the application, which relies on programmer hints describing only the application behavior, and exploiting them through a specialized runtime, balancing the workload by means of work-stealing.

The rest of this paper is organized as follows. Section 1 introduces the syntax and semantics of the proposed hints, while Section 2 provides details on our runtime design and implementation, and Section 3 provides an experimental evaluation. Finally, Section 4 provides a brief survey of related works, and Section 5 draws some conclusions and highlights future research directions.

1 The Data Access Pattern Approach

The current OpenMP standard provides support for parallel loops through the `omp for` and `omp do` directives[1]. The parallel loop syntax is restricted to force the loop bounds to be loop invariants, since the runtime must always be able to evaluate the iteration space. Once the iteration space has been computed, iterations are first grouped into *chunks*[2] and then mapped to the active threads of the parallel team according to the scheduling policy implemented by the runtime. Programmers can influence the behaviour of the runtime system only by forcing a iteration scheduling policy and specifying a minimum chunk size.

Even though OpenMP allows the programmer to choose among different scheduling strategies, to address the problem of mapping iterations over the threads in a team, there is no support for expressing thread-data affinity [5,19].

The key idea of our approach is to allow the runtime to identify the portion of data which will be accessed by the iterations of a parallel loop. These iterations will then be scheduled to threads according to a novel dynamic scheduling policy, which will try to preserve locality as much as possible.

To this end, we extend the existing OpenMP parallel loop directive through a new clause representing the *data access pattern*, that is the way loop iterations access the data. The runtime will then use the thread-data affinity information derived from the data access pattern to improve the existing dynamic iteration scheduling policy, by scheduling threads on the cores nearest to the memory where the related data are stored. While automated approaches to page placement do not require changes to the API, identifying and exploiting thread-data affinity at compile time might not be feasible, and is in general a very complex task [5]. By contrast, a skilled programmer is able to identify more effectively the patterns used by threads when accessing data, and thus provide precise hints to the runtime. This is, anyway, mandatory if a fine-tuning of the application performances is desired [4,20,16].

[1] `omp for` and `omp do` model the same type of parallel loop, in C and Fortran respectively. For brevity in the rest of the paper we will refer to `omp for` but the same considerations apply to `omp do` as well.

[2] We use the term *chunk* to refer to a set of iterations as specified in OpenMP [3].

A key difference with respect to previous works [4], including *PGAS* languages [18,1], is that to minimize the programming efforts when writing parallel programs, our approach does not rely on explicit data distribution and exploitation of the processor space.

1.1 Data Access Pattern Definition

A data access pattern binds iterations in a parallel loop with the portion of memory accessed at runtime. We formally define the data access pattern and the OpenMP syntactic extension needed to support it as follows.

Definition 1. *A* data access pattern *is an equivalence relation over the elements of a k-dimensional array data structure. An equivalence class under the data access pattern relation is called* tile*. Data access pattern relations are described by means of* pattern clauses*, defined by the grammar in Figure 1 and its associated semantics.*

$$
\begin{array}{lll}
\mathit{Axiom} \rightarrow \texttt{pattern(}\ \mathit{Clause}\ \texttt{)} & & \mathit{PatternExpr} \rightarrow \mathit{RangeExpr} \mid \mathit{SliceExpr} \\
\mathit{Clause} \rightarrow \mathit{DataStructure}\ \texttt{[}\ \mathit{PESeq}\ \texttt{]} & & \mathit{RangeExpr} \rightarrow \mathit{Expr}\ \texttt{:}\ \mathit{Expr} \\
\mathit{PESeq} \rightarrow \mathit{PESeq}\ \texttt{,}\ \mathit{PatternExpr} & & \quad \mid \mathit{Expr} \mid \texttt{*} \\
\quad \mid \mathit{PatternExpr} & & \mathit{SliceExpr} \rightarrow \texttt{\^{}}\ \mathit{Expr}
\end{array}
$$

Fig. 1. Pattern Clause Syntax. *Expr* is any expression of runtime constants, while *DataStructure* can be any array or pointer variable name.

In our OpenMP extension, a *pattern clause* (or, for brevity, a pattern) is associated to a loop directive. The first argument of a pattern clause is a reference to the shared data structure that is concurrently accessed by iterations in the loop. The rest of the pattern clause consists of a sequence of *pattern expressions*, one for each dimension. A pattern expression can be either a *range expression* or a *slice expression*. A range expression is used to identify a range of indices in a given dimension of the data structure, that are associated to all tiles. A slice expression identifies the size of each tile in a given dimension.

A range expression has the form `[n:m]`. Both `n` and `m` must be loop invariant. Their value is thus known at runtime before the loop execution starts. The lower bound of a range expression may be omitted when it matches exactly the lower bound of the associated dimension. Hence, a pattern expression `m` is an alias for `[lb:m]`, where `lb` is the lower bound of the index for the dimension considered. The `*` operator is also a shorthand for `[lb:ub]`, where `lb` and `ub` are the lower bound and the upper bound values of the index for a given dimension. The latter range expression variants allow a more compact definition of the pattern clause in many practical cases, but do not add any expressive power.

A slice expression takes the form `^n`, where `n` is a runtime constant.

Figure 2 demonstrates the data access pattern semantics. The two slice expressions define bi-dimensional tiles of size `RSLICE` × `CSLICE` on matrix `A`, thus representing the block-wise accesses performed by the loop nest.

```
#pragma omp for collapse(2) \
 pattern(A[^RSLICE,^CSLICE])
for(i = 0; i < ROWS; i += RSLICE)
  for(j = 0; j < COLS; j += CSLICE)
    for(k = 0; k < RSLICE; ++k)
      for(h = 0; k < CSLICE; ++h)
        A[i+k][j+h] = ...;
```

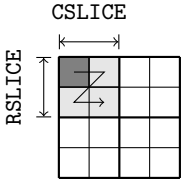


Fig. 2. Pattern example. Matrix `A` is accessed in a block-wise fashion by the collapsed parallel loop.

The mapping between tiles and iterations is defined as follows: if there is no slice expression in the pattern there is a single tile which is accessed by all iterations; otherwise, tiles and iterations are associated by a bijective relation, that depends on both the iteration indices and the sign of the loop increment expressions. In a normalized loop nest, each slice expression is associated to one loop index i_l and the tiles can be ordered with respect to the indices i_d of the dimension d associated to the slice expression divided by the tile size n. Iterations of loop index i_l are mapped to tiles with $i_d/n = i_l$.

Back to the example in Figure 2, assuming `RSLICE` = `CSLICE` = 2, and `A` a 4×4 square matrix, the pattern identifies four tiles. The iterations with index `i` = 0 are associated to data items of indices 0 and 1 on the first dimension. The same holds for loop index `j`, which is associated to the slice expression corresponding to the second dimension of `A`. Thus, iteration `i`, `j` = $\langle 0, 0 \rangle$ is mapped to the data in `A[0][0]`, `A[0][1]`, `A[1][0]`, and `A[1][1]`.

2 Runtime Extensions to Exploit Patterns

To employ the information encoded in the pattern clauses, we propose an extension of the OpenMP runtime. The runtime analyzes each pattern expressions to identify the size of the memory tiles accessed by iterations. The tile information can then be exploited at runtime to group together iterations that will probably touch the same set of virtual memory pages. Since at runtime the base address of the patterned data structure is known, it is always possible to identify the set of memory pages that are expected to be touched by the iterations of the loop. This is true also for dynamically allocated data-structures for which the size can be assumed equal to the tile size times the size of the iteration space.

Since the runtime aims at maximizing the number of local accesses, while avoiding, if possible, to incur in the penalty of long latency due to remote memory accesses, the information obtained analyzing pattern clauses is used to identify groups of iterations (*blocks*) that need to be scheduled together on the same node. Iterations that access the same memory pages (or different pages physically mapped to the same node) are grouped within the same block. The dynamic scheduling policy is thus driven by the collected pattern information.

The implementation used in this work is based on the *libgomp* [7] OpenMP runtime and uses the Linux NUMA API [12] to detect virtual page mappings.

2.1 Iteration Space Partitioning

To exploit the hints provided by the pattern information, the runtime has to partition the iteration space so to minimize the number of remote accesses.

Finding an optimal partition is known to be NP-complete. Obviously, such complexity cannot be handled at runtime even with moderate numbers of iterations. Therefore, we propose a straightforward heuristic approach to minimize the time spent by the runtime in analyzing pattern information while still providing a good, even if potentially sub-optimal, partitioning. To further reduce the overhead, we base the partitioning of the iterations of each loop on the information obtained from a single pattern.

The algorithm implemented in the proposed heuristic approach performs a linear scan of the iteration space in search of opportunities for grouping adjacent iterations. Let a and b be two adjacent iterations of the analyzed parallel loop. Both a and b will be mapped to the same block if at least one of the following conditions is satisfied: iteration a accesses to the same set of memory pages touched by iteration b; the set of pages touched by a are physically mapped to a node that is the same for the pages touched by b; pages touched by both a and b are not physically mapped to any node in the system.

Let us now formally introduce the concept of iteration block.

Definition 2. *Let lb and ub be respectively the lower and upper bound of the iteration space I of the analyzed loop. A block of iterations is defined as a range of indices of the form* $[base, last]$*, where* $base \geq lb$ *and* $last \leq ub$*.*

Let B be the set of blocks obtained from the partitioning phase, and let $b \in B$ be a block of iterations. We call $r(b)$ the range of indices described by b.

The runtime limits the maximum number of blocks to reduce the algorithm complexity while maintaining the required flexibility to cope with irregular workloads. The limit has been set, considering the outcome of an experimental campaign, to twice the number of available nodes in the system. To cope with the imposed constraints, different blocks of iterations may be merged.

When no pattern clause is specified for a given parallel loop, the iteration space is evenly partitioned into a number of blocks equal to the number of available nodes. Since the output of the partitioning algorithm is not necessarily the optimal partition, we later introduce a runtime work stealing mechanism to reduce the effects of an unbalanced distribution of the workload.

2.2 A Dynamic Scheduling Policy for Pattern Enabled OpenMP Runtimes

At the end of the partitioning stage, the iteration space of the parallel loop is divided into blocks of iterations. When a loop has associated pattern information, the runtime knows exactly which pages are touched by each iteration block. The

runtime assigns a *work queue* to each NUMA node. The work queue is used to store information about iteration blocks. A global work queue is reserved for those blocks that are not related to any of the active NUMA nodes.

The algorithm that maps blocks to work queues uses the iteration-data affinity information coming from the analysis of the pattern. Each thread of the parallel team analyzes the set of blocks in parallel. Let b be a block and let P_b be the set of pages touched by iterations of b. The algorithm counts how many pages in P_b are mapped to each node. The node with the highest number of mapped pages is finally selected as the target node for the block b. If none of the nodes is related to any of the pages in P_b, b is assigned to the global queue.

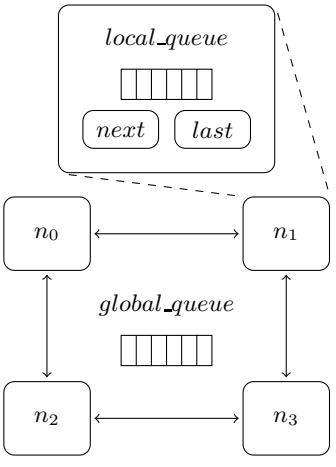


Fig. 3. Runtime system with four distributed work queues and a global queue

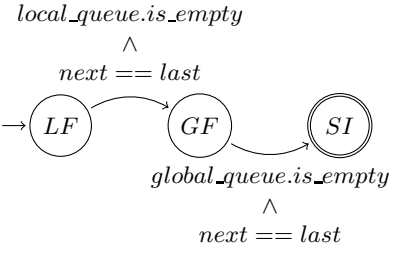


Fig. 4. Runtime behaviour of a sub-team. *Local Fetch* (*LF*): fetch blocks from the *local_queue* of the local node; *Global Fetch* (*GF*): fetch blocks from the *global_queue*; *Steal Iterations* (*SI*): steal blocks from the *local_queue* of neighbour nodes

Figure 3 shows the internal state of the runtime system in the case of a cc-NUMA architecture with four nodes. The internal state of each node is composed of a working queue called *local_queue* and two integer fields *next* and *last*, used respectively to store the lower bound and the upper bound index of the range of iteration indices associated to the *current block*.

At runtime, parallel teams are split into sub-teams, each associated to a distinct NUMA node. A sub-team associated to a node n is composed only by threads of the team that are running on node n. Threads are mapped to sub-teams at runtime when a new team starts. The runtime behaviour of a sub-team can be formally described by a finite state automaton as shown in Figure 4.

Each sub-team starts executing in the initial state LF. Threads of a sub-team whose working state is LF, are only allowed to fetch blocks from the local work queue of the node in which they are running. When the local block queue is empty and no iterations are available in the current block, the sub-team moves

from *LF* to *GF*, where the sub-team fetches blocks from the global queue. In both states, iterations are selected using the Guided Self Scheduling algorithm [17].

The idea is to exploit the locality of accesses preventing when possible threads from accessing remote pages. To this end, at first threads are forced to execute iterations from the local queue to maximize the probability of local accesses. Only when there are no more iterations associated to the local node, threads start fetching iterations from the global queue. Since global queue only stores blocks related to virtual pages that are still not mapped, there is an high probability that threads accessing the global queue will own those pages because of the *first-touch* policy implemented by the OS. The first-touch policy is the default policy for NUMA-aware Linux systems. It consists of placing memory pages on those nodes that first access the data during the program execution.

2.3 Work Stealing Strategy

When the global queue is empty and there are no iterations available in the local queue, the sub-team transitions from *GF* to *SI*. While in *SI*, threads start stealing blocks of iterations from the queues associated to other nodes. According to the implemented work stealing policy, threads in *SI* start stealing from the work queues of the nearest neighbour nodes. Since the runtime is aware of the distance between nodes (identified by means of calls to the Linux NUMA API), each sub-team knows which nodes are the best candidates for stealing.

The work stealing procedure iterates over the neighbours set of a node n in search of available blocks of iterations. By default the current neighbour node (n_{neigh}) is initially set equal to the node that hosts the current sub-team (n).

As long as there are iterations to fetch from the work queue of n_{neigh}, threads fetch new iterations from their work queues. Eventually, when the work queue of the current neighbour becomes empty, a new neighbour is selected.

The selection strategy is based on the *NUMA distance* between nodes of the underlying architecture. In the case shown in Figure 3, $\langle dist(n_0, n_i) | i \in [0 : 3] \rangle = \langle 0, 1, 1, 2 \rangle$. The distance relation $dist(n, n_i)$ imposes a partial ordering of the nodes $n_i \in K$. We need, for each node, a sequence of nodes to poll for the next neighbour, called a *neighbours vector*. To obtain the vectors, we make this a total ordering by imposing that, when $dist(n, n_i) = dist(n, n_j)$, $n_i \prec n_j$ if $i < j$.

3 Experimental Results

In this Section, we provide an experimental validation of our approach. The main findings are that the proposed approach based on pattern clauses is able to consistently reduce the number of remote memory accesses, and that the reduction directly translates into a significant performance improvement.

The experimental campaign has been conducted on a AMD ccNUMA machine with four nodes, each a quad core Opteron 8378 processor. Each core has a two-level private cache hierarchy. L1 cache is composed by a 64KBytes data cache and by a 64KBytes instruction cache. L2 cache is an unified 512KBytes

cache. All cores within a node share an unified 6144KBytes L3 cache. Inter-node communication is supported by a ring network topology.

AMD event based counters have been used to measure memory accesses. Separate runs have been used for performance and memory access profiling, to avoid memory access counter sampling overhead in timing measurements.

3.1 Benchmark Suite

We employ the NAS Parallel Benchmark suite, OpenMP version 3.3 [11]. We do not report on *DC* and *EP*, since these benchmarks do not have any OpenMP loop constructs (`omp for` and `omp do`). The benchmarks have been modified in order to make use of dynamic scheduling. Table 1 shows the number of total loops, dynamically scheduled loops, and loops tagged with the `pattern` clause.

Table 1. Benchmark characterization

Bench	Parallel loops	Dynamic loops	Patterns	Bench	Parallel loops	Dynamic loops	Patterns
bt.c	28	14	9	lu.c	26	10	9
cg.c	18	16	16	mg.b	14	11	11
ft.b	8	6	6	sp.c	33	20	20
is.c	9	2	2	ua.c	68	56	56

We compare the baseline *libgomp* runtime implementation opportunely extended to support a Guided Self Scheduling strategy for dynamically scheduled loop iterations with our optimized runtime. This choice is dictated by the fact that the *libgomp* dynamic scheduler provides only poor performance, thus comparing with it would result in a significant bias due to Guided Self Scheduling.

For all experiments we use 16 threads, each pinned on a different core.

3.2 Performance Analysis

Table 2 describes the runtime behaviour of the benchmarks, showing the percentage of blocks fetched in each of the states of the automaton in Figure 4 along with the percentage of the execution time spent in loops tagged with `pattern` clauses. A high percentage of fetches from local queues denotes a good distribution of the data structures, which is effectively exploited by the iteration scheduling thanks to correct pattern information. On the other hand, blocks fetched through work stealing have higher probability of resulting in remote accesses since they were originally intended to be executed on a different node.

Table 3 shows the speedups obtained by our optimized runtime with respect to the baseline. Two scenarios are provided: *Best*, where the proposed work stealing policy based on NUMA distances is used; and *Worst*, where neighbour vectors are reversed. This shows that the order of the neighbours counts: the last column (Δ) shows the maximum performance loss in case of random neighbours

Table 2. Runtime behaviour

Bench	Blocks fetched from Local [%]	Global [%]	Steal [%]	Time in opt. loops [%]
bt.c	65.72	0.01	34.27	90.55
cg.c	99.61	0.03	0.36	87.26
ft.b	76.40	0.00	23.60	66.69
is.c	66.67	0.00	33.33	51.30
lu.c	80.21	0.21	19.58	26.49
mg.b	35.16	22.26	42.58	66.82
sp.c	70.03	0.00	29.97	91.92
ua.c	88.36	0.13	11.51	78.28

Table 3. Speedups

Bench	Speedup Worst	Best	Δ
bt.c	1.14	1.27	0.13
cg.c	1.81	1.82	0.01
ft.b	1.12	1.19	0.07
is.c	1.00	1.00	0.00
lu.c	1.02	1.05	0.03
mg.b	1.00	1.00	0.00
sp.c	1.18	1.23	0.05
ua.c	1.07	1.08	0.01

selection. However, the results also show that the impact of this policy is not so large as to make the runtime less effective than the baseline. Thus, the *Worst* scenario shows the impact of the iteration scheduling optimization, while the *Best* scenario adds the impact of an effective work-stealing policy.

We can see that, for most benchmarks, we obtain a speedup between 1.05x and 1.27x for the *Best* scenario. There are three exceptions: *MG*, *IS* and *CG*.

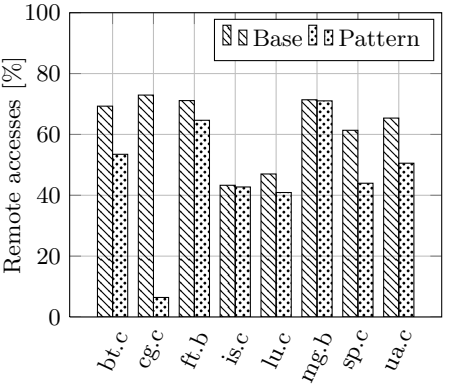
MG is the only benchmark where the initial distribution of frequently accessed data structures is performed by the master thread alone. Since we rely on the first-touch policy to provide the initial distribution, a large number of remote accesses is generated regardless of the iteration scheduling policy. Note that the pattern definition leads the runtime to place most of the iterations on the node where the master thread resides, thus leading to a reduced amount of blocks fetched from local queues.

IS benchmark implements a bucket sort algorithm. Excluding the time spent in initializing data structures, most of the time is spent on a fast data parallel loop used to sort keys of each bucket. There are several instances of non-linear accesses where array indices are obtained from table lookups. This type of access cannot be optimized, since it is by design hard to predict, to provide the required randomness. While the proposed technique cannot obtain a speedup, it still does not impose an overhead with respect to the baseline.

CG obtains the highest speedup, a remarkable 1.82x. It performs sparse matrix multiplication, which can easily lead to irregular accesses. However, the benchmark provides an initial data distribution that combined with the data access pattern information allows a massive improvement in data access regularity, which immediately translates into a performance improvement.

3.3 Remote Memory Access Analysis

Figure 5 shows the reduction in remote memory accesses obtained by our runtime with respect to the baseline. Memory access reduction is at the base of performance improvement, so these results mirror the performance speedups.



(a) Remote accesses percentage

Bench	Base	Pattern	Savings [%]
bt.c	70,777.56	54,787.53	22.59
cg.c	40,969.86	3,592.73	91.23
ft.b	4,824.42	4,494.31	0.90
is.c	851.71	844.01	6.84
lu.c	37,674.95	32,731.91	13.12
mg.b	2,504.46	2,486.34	0.72
sp.c	269,485.03	192,971.48	28.39
ua.c	115,912.76	85,196.04	26.50

(b) Raw results (millions of accesses)

Fig. 5. Memory accesses performed by benchmarks

It is especially interesting to consider the reduction in *CG*, where remote memory accesses are strongly minimized thanks to the pattern information.

In *IS*, the data access patterns are mostly unpredictable, as memory accesses are defined through non-affine array functions. This makes it hard to find good pattern information for most of the parallel loops in the code. While the savings in terms of remote accesses are small, they are sufficient to offset the overhead imposed by the pattern evaluation and iteration space partitioning phases.

In *MG*, most frequently accessed data structures are allocated on a single node, which forces all threads on other nodes to perform remote accesses. Thus, no significant reduction is obtained. Moreover, the high amount of global fetches shows that part of data structures were not preallocated at all.

4 Related Work

Several different approaches are proposed in literature to mitigate the memory latency penalty due to remote accesses. Some of these approaches rely on the ability of the runtime system [6] or the OS itself [10] to implicitly trigger the migration of worker threads to avoid the cost of remote accesses.

Other approaches, such as PGAS languages [18,1] rely on the ability of the programmer to manually distribute data structures concurrently accessed by threads at runtime. These languages provide the programmer a mean to force a specific dynamic page placement policy for those shared data structures that will be heavily accessed by loops. On the other hand, our solution does not rely on explicit distribution hints, though it can take advantage of an initial data distribution provided by means of the *first-touch* policy.

Dynamic data distribution based on memory protection mechanisms has been introduced in [8,6,13]. Memory pages forming shared data structures can be dynamically tagged, to trigger a page migration to the next node touching them

(*next-touch* strategy). Our approach is orthogonal with respect to this strategy, since we reduce the number of remote accesses without triggering redistributions.

In [15] the authors propose a dynamic data redistribution solution similar to [8,6] but based on information akin to our proposed data access pattern, which is, contrary to our solution, computed at runtime by means of profiling.

5 Conclusions

We propose an optimized OpenMP runtime design for NUMA machines to exploit thread-data affinity in parallel programs by means of programmer hints that take into account only the application behavior. Our experimental campaign shows a reduction in the number of remote accesses for most NAS benchmarks.

The approach could be further improved by removing unnecessary pattern evaluations when multiple subsequent loops share the same pattern. Moreover, opportunities for data redistribution could be automatically detected at compile-time by analysing pattern variations between subsequent loops.

Future extensions could include adding thread migration to handle the cases of multiple concurrent applications as well as the case of applications with multiple phases, alternating I/O bound phases with CPU bound ones. We also expect that combining our technique with a next-touch strategy would further reduce the remote accesses, while limiting the number of pages moved.

Furthermore, identifying patterns requires skill and time. It would be worth exploring both static analysis and profiling based techniques to provide recommended patterns to the programmer.

References

1. Allen, E., Chase, D., Hallet, J., Luchangco, V., Maessen, J., Ryu, S., Steele Jr., G.L., Tobin-Hochstadt, S.: The Fortress Language Specification. In: Sun Microsystems (2008)
2. AMD: AMD Direct Connect Architecture (2010), `http://www.amd.com/us/products/technologies/direct-connect-architecture`
3. ARB: OpenMP Application Program Interface, version 3.0 (2008), `http://www.openmp.org`
4. Bircsak, J., Craig, P., Crowell, R., Cvetanovic, Z., Harris, J., Nelson, C.A., Offner, C.D.: Extending OpenMP for NUMA Machines. In: SC (2000)
5. Broquedis, F., Diakhaté, F., Thibault, S., Aumage, O., Namyst, R., Wacrenier, P.-A.: Scheduling Dynamic OpenMP Applications over Multicore Architectures. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 170–180. Springer, Heidelberg (2008)
6. Broquedis, F., Furmento, N., Goglin, B., Namyst, R., Wacrenier, P.-A.: Dynamic Task and Data Placement over NUMA Architectures: An OpenMP Runtime Perspective. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 79–92. Springer, Heidelberg (2009)
7. GNU: GNU libgomp (2010), `http://gcc.gnu.org/onlinedocs/libgomp/`
8. Goglin, B., Furmento, N.: Enabling High-performance Memory Migration for Multithreaded Applications on LINUX. In: IPDPS, pp. 1–9. IEEE, Los Alamitos (2009)

9. Intel: Intel QuickPath Architecture (2010), www.intel.com/technology/quickpath/whitepaper.pdf
10. Jenks, S., Gaudiot, J.-L.: Exploiting Locality and Tolerating Remote Memory Access Latency Using Thread Migration. Int. J. Parallel Program. 25(4), 281–304 (1997)
11. Jin, H., Frumkin, M.: The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Tech. rep., NASA (1999)
12. Kleen, A.: An NUMA API for Linux (2004), http://www.halobates.de/numaapi3.pdf
13. Lankes, S., Bierbaum, B., Bemmerl, T.: Affinity-On-Next-Touch: An Extension to the Linux Kernel for NUMA Architectures. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009. LNCS, vol. 6067, pp. 576–585. Springer, Heidelberg (2010)
14. Marathe, J., Mueller, F.: Hardware Profile-guided Automatic Page Placement for ccNUMA Systems. In: PPOPP, pp. 90–99. ACM, New York (2006)
15. Nikolopoulos, D.S., Artiaga, E., Ayguadé, E., Labarta, J.: Scaling Non-regular Shared-memory Codes by Reusing Custom Loop Schedules. Scientific Programming 11(2), 143–158 (2003)
16. Nikolopoulos, D.S., Papatheodorou, T.S., Polychronopoulos, C.D., Labarta, J., Ayguadé, E.: A Transparent Runtime Data Distribution Engine for OpenMP. Scientific Programming 8(3), 143–162 (2000)
17. Polychronopoulos, C.D., Kuck, D.J.: Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. IEEE Trans. Computers 36(12), 1425–1439 (1987)
18. University, R.: High Performance Fortran Language Specification. SIGPLAN Fortran Forum 12(4), 1–86 (1993)
19. Robertson, N., Rendell, A.P.: OpenMP and NUMA Architectures I: Investigating Memory Placement on the SGI Origin 3000. In: Sloot, P.M.A., Abramson, D., Bogdanov, A.V., Gorbachev, Y.E., Dongarra, J., Zomaya, A.Y. (eds.) ICCS 2003. LNCS, vol. 2660, pp. 648–656. Springer, Heidelberg (2003)
20. Terboven, C., Mey, D.a., Schmidl, D., Jin, H., Reichstein, T.: Data and Thread Affinity in OpenMP Programs. In: MAW 2008: Proceedings of the 2008 workshop on Memory access on future processors, pp. 377–384. ACM, New York (2008)
21. Tikir, M.M., Hollingsworth, J.K.: Using Hardware Counters to Automatically Improve Memory Performance. In: SC, p. 46. IEEE Computer Society, Los Alamitos (2004)

Workload Balancing and Throughput Optimization for Heterogeneous Systems Subject to Failures

Anne Benoit[1], Alexandru Dobrila[2,⋆], Jean-Marc Nicod[2], and Laurent Philippe[2]

[1] ENS Lyon, Université de Lyon, LIP laboratory (ENS, CNRS, INRIA, UCBL), France
[2] Université de Franche-Comté, LIFC laboratory, (UFC), France
adobrila@lifc.univ-fcomte.fr

Abstract. In this paper, we study the problem of optimizing the throughput of streaming applications for heterogeneous platforms subject to failures. The applications are linear graphs of tasks (pipelines), and a type is associated to each task. The challenge is to map tasks onto the machines of a target platform, but machines must be specialized to process only one task type, in order to avoid costly context or setup changes. The objective is to maximize the throughput, i.e., the rate at which jobs can be processed when accounting for failures. For identical machines, we prove that an optimal solution can be computed in polynomial time. However, the problem becomes NP-hard when two machines can compute the same task type at different speeds. Several polynomial time heuristics are designed, and simulation results demonstrate their efficiency.

1 Introduction

Most of the distributed environments are subject to failures, and each component of the environment has its own failure rate. Assuming that a failure may be tolerated, as for instance in asynchronous systems [1] or production systems, the failures have an impact on the system performance. When scheduling an application onto such a system, either we can account for failures to help improve the performance in case of failures, or ignore them. In some environments, such as computing grids, this failure rate is so high that we cannot ignore failures when scheduling applications that last for a long time as a batch of input data processed by pipelined tasks for instance. This is also the case for micro-factories where a production is composed of several instances of the same micro-component that must be processed by cells.

In this paper, we deal with scheduling and mapping strategies for *coarse-grain* workflow applications [18,19]. The applications are linear graphs of tasks (pipelines), and a type is associated to each task. The target platform is a set

⋆ Corresponding author.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 242–254, 2011.

of execution resources (generically called *machines*), such as a grid or a micro-factory, on which the tasks must be mapped. A series of jobs enters the workflow and progresses from task to task until the final result is computed. Once a task is mapped onto a set of dedicated resources (known in the literature as *multi-processor tasks* [3,7]), the computation requirements and the failure rates for each machine when processing one job of the workflow are known. After an initialization delay, a new job is completed every period, where the period is the inverse of the throughput. It is defined as the longest cycle-time of a machine. Note that we target coarse-grain applications and platforms on which the cost of communications is negligible in comparison to the cost of computations.

In the distributed computing system context, a use case of a streaming application is for instance an image processing application where images are processed in batches, on a SaaS (Software as a service) platform. In this context, failures may occur because of the nodes, but they also may be impacted by the complexity of the service [9]. On the production side, a use case is a micro-factory [13,5,12] composed of several cells that provides functions as assembly or machining. But, at this scale, the physical constraints are not totally controlled and it is mandatory to take failures into account in the automated command. A common property of these systems is that we cannot use replication, as for instance in [4,14,10], to overcome the failures. For streaming applications, it may impact the throughput to replicate each task. For a production which deals with physical objects, replication is not possible. Fortunately, losing a few jobs may not be a big deal; for instance, the loss of some images in a stream will not alter the result, as far as the throughput is maintained, and losing some micro-products is barely more costly than the occupation of the processing resources that have been dedicated to it. The failure model is based on the Window-Constrained [16] model, often used in real-time environment. In this model, only a fraction of the messages will reach their destination. The losses are not considered as a failure but as a guarantee: for a given network, a Window-Constrained scheduling [15,17] can guarantee that no more than x messages will be lost for every y sent messages.

In this paper, we therefore solely concentrate on the problem of period minimization (i.e., throughput maximization), where extra jobs are processed to account for failures. For instance, if there is a single task, mapped on a single machine, with a failure rate of $1/2$, a throughput of x jobs per unit time will be achieved if the task processes $2 \times x$ jobs per time unit.

The paper is organized as follows. Section 2 presents the framework and formalizes the optimization problems tackled in the paper. An exhaustive complexity study is provided in Section 3: we exhibit some particular polynomial problem instances, and prove that the remaining problem instances are NP-hard problems. In Section 4, we design a set of polynomial-time heuristics to solve the most general problem instance, building upon complexity results, and in particular linear program formulations to solve sub-problems. Moreover, we conduct extensive simulations to assess the relative and absolute performance of the heuristics. Finally, we conclude in Section 5.

2 Framework and Optimization Problems

Applicative framework. The application consists of a linear chain of n tasks, $T_1, T_2, \ldots, T_n$. A type is associated to each task: we have a set of p task types with $n \geq p$, and a function t which returns the type of a task. Hence, $t(i)$ is the type of task T_i. A series of jobs enters the workflow and progresses from task to task until the final result is computed, and x_i is the average number of jobs processed by task T_i to output one job out of the system. Note that x_{i+1} depends on x_i and on the failure rate of the machine processing T_i (see below).

Target platform. The target platform is distributed and heterogeneous. It consists of a set of m machines (a cell in the micro-factory or a host in a grid platform), $M_1, M_2, \ldots, M_m$. The task processing time depends on the machine that performs it: it takes $w_{i,u}$ units of time to machine M_u to execute task T_i on one job. Each machine is able to process all the task types. However, to avoid costly context or setup changes during execution, the machines may be specialized to process only one task type. Note that we do not take communication times into account as we consider that the processing time is much greater than the communication time (coarse-grain applications).

Failure model. It may happen that a job (or product) is lost (or damaged) while a task is being executed on this job. For instance, an electrostatic charge may be accumulated on an actuator and a piece will be pushed away rather than caught, or a message will be lost due to network contention. Note that we deal only with transient failures, as defined in [8]: the tasks are failing for some jobs, but we do not consider a permanent failure of the machine responsible of the task, as this would lead to a failure for all the remaining jobs to be processed and the inability to finish them. In order to deal with failures, we process more jobs than needed, so that at the end, the required throughput is reached. The failure rate of task T_i performed onto machine M_u is the percentage of failure for this task and it is denoted $f_{i,u}$.

Objective function. Our goal is to assign tasks to machines so as to optimize some key performance criteria. A task can be allocated to several machines, and $q(i,u)$ is the quantity of task T_i executed by machine M_u; if $q(i,u) = 0$, T_i is not assigned to M_u. Recall that x_i is the average number of jobs processed by task T_i to output one job out of the system. We must have, for each task T_i, $\sum_{u=1}^{m} q(i,u) = x_i$, i.e., enough jobs are processed for task T_i in the system.

The objective function is to maximize the number of jobs that exit the system per time unit, making abstraction of the initialization and clean-up phases. This objective is important when a large number of jobs must be processed. Actually, we deal with the equivalent optimization problem that minimize the *period*, the inverse of the throughput. One challenge is that we cannot compute the number x_i of jobs that must be processed by task T_i before allocating tasks to machines, since x_i depends on the failure rates incurred by the allocation. However, each task T_i has a unique successor task T_{i+1}, and x_{i+1} is the amount of jobs needed by T_{i+1} as input. Since T_i is distributed on several machines

with different failure rates, we have $\sum_{u=1}^{m}(q(i,u)\times(1-f_{i,u}))=x_{i+1}$, where $q(i,u)\times(1-f_{i,u})$ represents the amount of jobs output by the machine M_u if $q(i,u)$ jobs are treated by that machine. For each task, we sum all the instances treated by all the machines. We are now ready to define the cycle-time ct_u of machine M_u: it is the time needed by M_u to execute all tasks T_i with $q(i,u)>0$: $ct_u=\sum_{i=1}^{n}q(i,u)\times w_{i,u}$. The objective function is to minimize the maximum cycle-time, which corresponds to the period of the system: $\min\max_{1\leq u\leq m}ct_u$.

Rules of the game. Different rules of the game may be enforced to define the allocation, i.e., the $q(i,u)$ values. For *one-to-many* mappings, we enforce that a single task must be mapped onto each machine: $\forall i,i':1\leq i,i'\leq n\ s.t.\ i\neq i'$, $q(i,u)>0\Rightarrow q(i',u)=0$. This kind of mapping is quite restrictive because we must have at least as many machines as tasks. Note that a task can be allocated to several machines. We relax this rule to allow for *specialized* mappings, in which several tasks of the same type can be mapped onto the same machine: $\forall i,i':1\leq i,i'\leq n\ s.t.\ t(i)\neq t(i'),q(i,u)>0\Rightarrow q(i',u)=0$. Note that if each task has a different type, the specialized mapping and the one-to-many mapping are equivalent. Finally, *general* mappings have no constraints: any task (no matter the type) can be mapped on any machine.

Problem definition. For the optimization problem that we consider, the three important parameters are: (i) the rules of the game (*one-to-many (o2m)* or *specialized (spe)* or *general (gen)* mapping); (ii) the failure model (f if failures are all identical, f_i if the failure for a same task is identical on two different machines, f_u if the failure rate depends only on the machine, and the general case $f_{i,u}$); and (iii) the computing time (w if the processing times are all identical, w_i if it differs only from one task to another, w_u if it depends only on the machine, and $w_{i,u}$ in the general case). We are now ready to define the optimization problem:

Definition 1. MinPer(R,F,W)*: Given an application and a target platform, with a failure model* $F=\{f|f_i|f_u|f_{i,u}|*\}$ *and computation times* $W=\{w|w_i|w_u|w_{i,u}|*\}$*, find a mapping (i.e., values of* $q(i,u)$ *such that for each task* T_i *with* $1\leq i\leq n$, $\sum_{u=1}^{m}q(i,u)=x_i$*) following rule* $R=\{o2m|spe|gen|*\}$*, which minimizes the period of the application,* $\max_{1\leq u\leq m}\sum_{i=1}^{n}q(i,u)\times w_{i,u}$.

Note that $*$ is used to express the problem with any variant of the corresponding parameter; for instance, MinPer$(*,f_{i,u},w)$ is the problem of minimizing the period with any mapping rule, where failure rates are general, while execution times are all identical.

3 Complexity Results

We assess the complexity of the different instances of the MinPer(R,F,W) problem. First we provide the complexity of the problems with $F=f_i$, and then we discuss the most general problems with $F=f_{i,u}$. Even though the general problem is NP-hard, we show that once the allocation of tasks to machines is known, we can optimally decide how to share tasks between machines, in polynomial time. Also, we give an integer linear program to solve the problem.

3.1 Complexity of the MinPer$(*, f_i, *)$ Problems

We first show how the MinPer$(*, f_i, *)$ problems can be simplified. Indeed, in this case, the number of products that should be computed for task T_i at each period, x_i, is independent of the allocation of tasks to machines. We can therefore ignore the failure probabilities, and focus on the computation of the period of the application. The following Lemma 1 allows us to further simplify the problem: tasks of similar type can be grouped and processed as a single *equivalent* task.

Lemma 1. *For* MinPer$(*, f_i, w_i)$ *or* MinPer$(*, f_i, w_u)$*, there exists an optimal solution in which all tasks of the same type are executed onto the same set of machines, in equal proportions:* $\forall i, j : 1 \leq i, j \leq n$ *with* $t(i) = t(j)$*,*

$$\exists \alpha_{i,j} \in \mathbb{Q} \text{ s.t. } \forall u : 1 \leq u \leq m,\ q(i,u) = \alpha_{i,j} \times q(j,u)\ . \tag{1}$$

The proof consists in building an optimal solution which follows Equation (1), from an existing one. We redistribute the work and define the $\alpha_{i,j}$ values for each problem instance. The detailed proof is available in the companion research report [2].

Corollary 1. *For* MinPer$(*, f_i, w_i)$ *or* MinPer$(*, f_i, w_u)$*, we can group all tasks of same type* t *as a single* equivalent *task* $T_t^{(eq)}$*, s.t.* $x_t^{(eq)} = \sum_{1 \leq i \leq n | t(i)=t} x_i$*. Then, we can solve this problem with the one-to-many rule, and deduce the solution of the initial problem.*

Proof. Following Lemma 1, we search for the optimal solution which follows Equation (1). Since all tasks of the same type are executed onto the same set of machines in equal proportions, we can group them as a single equivalent task. The amount of work to be done by the set of machines corresponds to the total amount of work of the initial tasks, i.e., for a type t, $\sum_{1 \leq i \leq n | t(i)=t} x_i$.

The one-to-many rule decides on which set of machines each equivalent task is mapped, and then we share the initial tasks in equal proportions to obtain the solution to the initial problem: if task T_i is not mapped on machine M_u, then $q(i,u) = 0$, otherwise $q(i,u) = \frac{x_i}{x_{t(i)}^{(eq)}} \times \frac{P}{w_{i|u}}$, where $w_{i|u} = \{w_i \mid w_u\}$.

We are now ready to establish the complexity of the MinPer$(*, f_i, *)$ problems. Recall that n is the number of tasks, m is the number of machines, and p is the number of types. We start by providing polynomial algorithms for one-to-many and specialized mappings with w_i (Theorem 1 and Corollary 2). Then, we discuss the case of general mappings, which can also be solved in polynomial time (Theorem 2). Finally, we tackle the instances which are NP-hard (Theorem 3).

Theorem 1. MinPer$(o2m, f_i, w_i)$ *can be solved in time* $O(m \times \log n)$*.*

Proof. First, note that solving this one-to-many problem amounts to decide on how many machines each task is executed (since machines are identical), and then split the work evenly between these machines to minimize the period. Hence, if T_i is executed on k machines, $q(i,u) = \frac{x_i}{k}$, where M_u is one of these k machines, and the corresponding period is $\frac{x_i}{k} \times w_i$.

We provide a greedy algorithm to solve the problem. The idea is to assign initially one machine per task (note that there is a solution only if $m \geq n$), sort the tasks by non-increasing period, and then iteratively add a machine to the task whose machine(s) have the greater period, while there are some machines available. Let g_i be the current number of machines assigned to task T_i: the corresponding period is $\frac{x_i}{g_i} \times w_i$. At each step, we insert the task whose period has been modified in the ordered list of tasks, which can be done in $O(\log n)$ (binary search). The initialization takes a time $O(n \log n)$ (sorting the tasks), and then there are $m - n$ steps of time $O(\log n)$. Since we assume $m \geq n$, the complexity of this algorithm is in $O(m \times \log n)$. To prove that this algorithm returns the optimal solution, let us assume that there is an optimal solution of period P_{opt} that has assigned o_i machines to task T_i, while the greedy algorithm has assigned g_i machines to this same task, and its period is $P_{greedy} > P_{opt}$. Let T_i be the task which enforces the period in the greedy solution (i.e., $P_{greedy} = x_i w_i / g_i$). The optimal solution must have given at least one more machine to this task, i.e., $o_i > g_i$, since its period is lower. This means that there is a task T_j such that $o_j < g_j$, since $\sum_{1 \leq i \leq n} o_i \leq \sum_{1 \leq i \leq n} g_i = m$ (all machines are assigned with the greedy algorithm). Then, note that since $o_j < g_j$, because of the greedy choice, $x_j w_j / o_j \geq x_i w_i / g_i$ (otherwise, the greedy algorithm would have given one more machine to task T_i). Finally, $P_{opt} \geq x_j w_j / o_j \geq x_i w_i / g_i = P_{greedy}$, which leads to a contradiction, and concludes the proof.

Corollary 2. MinPer(spe, f_i, w_i) *can be solved in time* $O(n + m \times \log p)$.

Proof. For the specialized mapping rule, we use Corollary 1 to solve the problem: first we group the n tasks by types, therefore obtaining p equivalent tasks, in time $O(n)$. Then, we use Theorem 1 to solve the problem with p tasks, in time $O(m \times \log p)$. Finally, the computation of the mapping with equal proportions is done in $O(n)$, which concludes the proof.

Theorem 2. MinPer$(gen, f_i, *)$ *can be solved in polynomial time.*

Proof. We exhibit a linear program to solve the problem for the general case with $w_{i,u}$. Note however that the problem is trivial for w_i or w_u: we can use Corollary 1 to group all tasks as a single equivalent task, and then share the work between machines as explained in the corollary.

In the general case, we solve the following (rational) linear program, where the variables are P (the period), and $q(i,u)$, for $1 \leq i \leq n$ and $1 \leq u \leq m$.

$$\begin{array}{l} \text{Minimize } P, \text{ subject to} \\ \text{(i) } q(i,u) \geq 0 \text{ for } 1 \leq i \leq n, 1 \leq u \leq m \\ \text{(ii) } \sum_{1 \leq u \leq m} q(i,u) = x_i \text{ for each task } T_i \text{ with } 1 \leq i \leq n \\ \text{(iii) } \sum_{1 \leq i \leq n} q(i,u) \times w_{i,u} \leq P \text{ for each machine } M_u \text{ with } 1 \leq u \leq m \end{array} \tag{2}$$

The size of this linear program is polynomial in the size of the instance, all $n \times m + 1$ variables are rational. Therefore, it can be solved in polynomial time [11].

Finally, we prove that the remaining problem instances are NP-hard (one-to-many or specialized mappings, with w_u or $w_{i,u}$). Since MinPer$(o2m, f_i, w_u)$ is

a special case of all other instances, it is sufficient to prove the NP-completeness of the latter problem.

Theorem 3. *The* MinPer$(o2m, f_i, w_u)$ *problem is NP-hard in the strong sense.*

Proof. We consider the following decision problem: given a period P, is there a one-to-many mapping whose period does not exceed P? The problem is obviously in NP: given a period and a mapping, it is easy to check in polynomial time whether it is valid or not. The NP-completeness is obtained by reduction from 3-PARTITION [6], which is NP-complete in the strong sense.

We consider an instance $\mathcal{I}_1$ of 3-PARTITION: given an integer B and $3n$ positive integers $a_1, a_2, \ldots, a_{3n}$ such that for all $i \in \{1, \ldots, 3n\}$, $B/4 < a_i < B/2$ and with $\sum_{i=1}^{n} a_i = nB$, does there exist a partition $I_1, \ldots, I_n$ of $\{1, \ldots, 3n\}$ such that for all $j \in \{1, \ldots, n\}$, $|I_j| = 3$ and $\sum_{i \in I_j} a_i = B$? We build the following instance $\mathcal{I}_2$ of our problem with n tasks, such that $x_i = B$, and $m = 3n$ machines with $w_u = 1/a_u$. The period is fixed to $P = 1$. Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. We now show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does.

Suppose first that $\mathcal{I}_1$ has a solution. For $1 \leq i \leq n$, we assign task T_i to the machines of I_i: $q(i, u) = a_u$ for $u \in I_i$, and $q(i, u) = 0$ otherwise. Then, we have $\sum_{1 \leq u \leq m} q(i, u) = \sum_{u \in I_i} a_u = B$, and therefore all the work for task T_i is done. The period of machine M_u is $\sum_{1 \leq i \leq n} q(i, u) \times w_u = a_u / a_u = 1$, and therefore the period of 1 is respected. We have a solution to $\mathcal{I}_2$.

Suppose now that $\mathcal{I}_2$ has a solution. Task T_i is assigned to a set of machines, say I_i, such that $\sum_{u \in I_i} q(i, u) = B$, and $q(i, u) \leq a_u$ for all $u \in I_i$. Since all the work must be done, by summing over all tasks, we obtain $q(i, u) = a_u$, and the solution is a 3-partition, which concludes the proof.

3.2 Complexity of the MinPer$(*, f_{i,u}, *)$ Problems

When we consider problems with $f_{i,u}$ instead of f_i, we do not know in advance the number of jobs to be computed by each task in order to have one job exiting the system, since it depends upon the machine on which the task is processed. However, we are still able to solve the problem with general mappings, as explained in Theorem 4. For one-to-many and specialized mappings, the problem is NP-hard with w_u, since it was already NP-hard with f_i in this case (see Theorem 3). We prove that the problem becomes NP-hard with w_i in Theorem 5, which illustrates the additional complexity of dealing with $f_{i,u}$ rather than f_i.

Theorem 4. MinPer$(gen, f_{i,u}, *)$ *can be solved in polynomial time.*

Proof. We modify the linear program (2) of Theorem 2 to solve the case with general failure rates $f_{i,u}$. Indeed, constraint (ii) is no longer valid, since the x_i are not defined before the mapping has been decided. It is rather replaced by constraints (iia) and (iib):

(iia) $\sum_{1 \leq u \leq m} q(n, u) \times (1 - f_{n,u}) = 1$;
(iib) $\sum_{1 \leq u \leq m} q(i, u) \times (1 - f_{i,u}) = \sum_{1 \leq u \leq m} q(i + 1, u)$ for each $T_i (1 \leq i < n)$.

Constraint (iia) states that the final task must output one job, while constraint (iib) expresses the number of jobs that should be processed for task T_i, as a function of the number for task T_{i+1}. There are still $n \times m + 1$ variables which are rational, and the number of constraints remains polynomial, therefore this linear program can be solved in polynomial time [11].

Theorem 5. *The* MinPer$(o2m, f_{i,u}, w_i)$ *problem is NP-hard.*

The proof of this theorem is quite involved, and we refer to the companion research report [2] for the details.

However, if the allocation of tasks to machines is known, then we can optimally decide how to share tasks between machines, in polynomial time. We build upon the linear program of Theorem 4, and we add a set of parameters: $a_{i,u} = 1$ if T_i is allocated to M_u, and $a_{i,u} = 0$ otherwise (for $1 \leq i \leq n$ and $1 \leq u \leq m$). The variables are still the period P, and the amount of task per machine $q(i, u)$. The linear program writes:

$$\begin{array}{l} \text{Minimize } P, \text{ subject to} \\ \text{(i) } q(i,u) \geq 0 \text{ for } 1 \leq i \leq n, 1 \leq u \leq m \\ \text{(iia) } \sum_{1 \leq u \leq m} q(n,u) \times (1 - f_{n,u}) = 1 \\ \text{(iib) } \sum_{1 \leq u \leq m} q(i,u) \times (1 - f_{i,u}) = \sum_{1 \leq u \leq m} q(i+1,u) \text{ for } 1 \leq i < n \\ \text{(iii) } \sum_{1 \leq i \leq n} q(i,u) \times w_{i,u} \leq P \text{ for } 1 \leq u \leq m \\ \text{(iv) } q(i,u) \leq a_{i,u} \times F_{\max} \text{ for } 1 \leq i \leq n \text{ and } 1 \leq u \leq m \end{array} \tag{3}$$

We have added constraint (iv), which states that $q(i, u) = 0$ if $a_{i,u} = 0$, i.e., it enforces that the fixed allocation is respected. $F_{\max} = \prod_{1 \leq i \leq n} \max_{1 \leq u \leq m} f_{i,u}$ is an upper bound on the $q(i, u)$ values, it can be pre-computed before running the linear program. The size of this linear program is clearly polynomial in the size of the instance, all $n \times m + 1$ variables are rational, and therefore it can be solved in polynomial time [11].

The linear program of Equation (3) allows us to find the solution in polynomial time, once the allocation is fixed. We also propose an integer linear program (ILP), which computes the solution to the MinPer$(spe, f_{i,u}, w_{i,u})$ problem, even if the allocation is not known. However, because of the integer variables, the resolution of this program takes an exponential time. Note that this ILP can also solve the MinPer$(o2m, f_{i,u}, w_{i,u})$: one just needs to assign a different type to each task. We no longer have the $a_{i,u}$ parameters, and therefore we suppress constraint (iv). Rather, we introduce a set of Boolean variables, $x(u, t)$, for $1 \leq u \leq m$ and $1 \leq t \leq p$, which is set to 1 if machine M_u is specialized in type t, and 0 otherwise. We then add the following constraints:

$$\begin{array}{ll} \text{(iva)} & \sum_{1 \leq t \leq p} x(u,t) \leq 1 \text{ for each machine } M_u \text{ with } 1 \leq u < m \text{ ;} \\ \text{(ivb)} & q(i,u) \leq x(u,t_i) \times F_{\max} \text{ for } 1 \leq i \leq n \text{ and } 1 \leq u \leq m \text{ .} \end{array}$$

Constraint (iva) states that each machine is specialized into at most one type, while constraint (ivb) enforces that $q(i, u) = 0$ when machine M_u is not specialized in the type t_i of task T_i. This ILP has $n \times m + 1$ rational variables, and $m \times p$ integer variables. The number of constraints is polynomial in the size of the instance. Note that this ILP can be solved for small problem instances with ILOG CPLEX (`www.ilog.com/products/cplex/`).

4 Heuristics and Simulations

From the complexity study, we are able to find an optimal general mapping. In this section, we provide practical solutions to solve MinPer($spe, f_{i,u}, w_{i,u}$), which is NP-hard. Indeed, general mappings are not feasible in some cases, since it involves reconfiguring the machines between the execution of two tasks whose type is different. This additional setup time may be unaffordable. We design in Section 4.1 a set of polynomial time heuristics which return a specialized mapping, building upon the complexity results of Section 3. Finally, we present exhaustive simulation results in Section 4.2.

4.1 Polynomial Time Heuristics

Since we are able to find the optimal solution once the tasks are mapped onto machines, the heuristics are building such an assignment, and then we run the linear program of Equation (3) to obtain the optimal solution in terms of $q(i, u)$. The first heuristic is random, and serves as a basis for comparison. Then, the next three heuristics (H2, H3 and H4) are based on an iterative allocation process in two stages. In the first *top-down* stage, the machines are assigned from task T_1 to task T_n depending on their speed $w_{i,u}$: the machine with the best $w_{1,u}$ is assigned to T_1 and so on. The motivation is that the workload of the first task is larger than the last task because of the job failures that arise along the pipeline. In the second *bottom-up* stage, the remaining machines are assigned from task T_n to task T_1 depending on their reliability $f_{i,u}$: the machine with the best $f_{n,u}$ is assigned to T_n and so on. The motivation is that it is more costly to lose a job at the end of the pipeline than at the beginning, since more execution time has been devoted to it. We iterate until all the machines have at least one task to perform. Finally, H5 performs only a *top-down* stage, repetitively. The heuristics are described below.

H1: Random heuristic. The first heuristic randomly assigns each task to a machine when the allocation respects the task type of the chosen machine.

H2: Without any penalization. The *top-down* stage assigns each task to the fastest possible machine. At the end of this stage, each task of the same type is assigned onto the same machine, the fastest. Then, the already assigned machines are discarded from the list. In the same way, the *bottom-up* stage assigns each task of the same type to the same machine starting from the more reliable one. We iterate on these two steps until all machines are specialized.

H3: Workload penalization. The difference with H2 is in the execution of the *top-down* stage. Each time a machine is assigned to a task, this machine is penalized to take the execution of this task into account and its $w_{i,u}$ is changed to $w_{i,u} \times (k+1)$ where k is the number of tasks already mapped on the machine M_u. This implies that several machines can be assigned to the same task type in this phase of the algorithm: if a machine is already loaded by several tasks then we

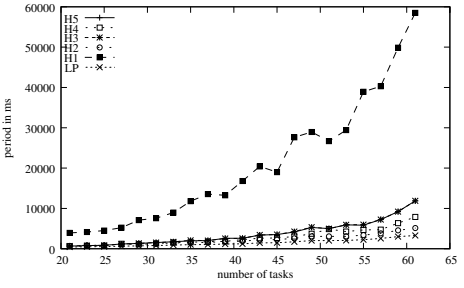


Fig. 1. $m = 20$, $p = 5$.
Heuristics against the linear program.

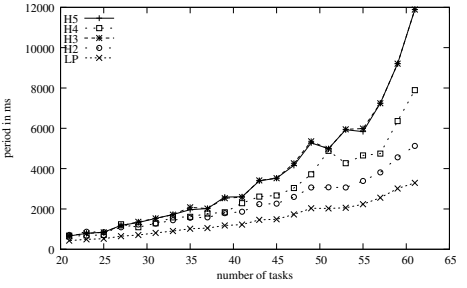


Fig. 2. $m = 20$, $p = 5$.
Without H1.

may find a faster machine and assign it to this task type. The *bottom-up* stage has the same behavior as for H2.

H4: Cooperation work. In this heuristic, a new machine is assigned to each task, depending on its speed, during the *top-down* stage; then the *bottom-up* stage has the same behavior as the heuristic H2.

H5: Focus on speed. The heuristic H5 focuses only on the speed by repeating the *top-down* stage of heuristic H3, until all the machines are allocated to at least one task.

4.2 Simulations

In this section, we evaluate the performance of the five heuristics. The period returned by each heuristic is measured in *ms*. Recall that m is the number of machines, p the number of types, and n the number of tasks. Each point in a figure is an average value of 30 simulations where the $w_{i,u}$ are randomly chosen between 100 and 1000 *ms* (these values are chosen to show the high level of heterogeneity of the machines, and they are randomly chosen since machines and tasks are unrelated), for $1 \leq i \leq n$ and $1 \leq u \leq m$, unless stated otherwise. Similarly, failure rates $f_{i,u}$ are randomly chosen between 0.2 and 10 % unless stated otherwise. Indeed, we observed that failure rates over 10 % do not change the behavior of the heuristics.

Heuristics versus linear program. In this set of simulations, the heuristics are compared to the integer linear program which gives the optimal solution. The platform is such that $m = 20$, $p = 5$ and $21 \leq n \leq 61$. Figure 1 shows that the random heuristic H1 has poor performance. Therefore, for visibility reasons, H1 does not appear in the rest of the figures. Results in Figure 2 show that the heuristics are not far from the optimal. The best heuristics H2 and H4 have a ratio of 1.5 and 2 to the optimal solution. The platform used for these simulations is limited on cases where the integer linear program finds a result. With the same platform but $p = 10$, the percentage of success of the linear program is less than 50% with 61 tasks.

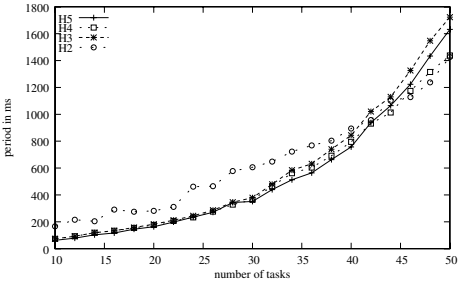


Fig. 3. $m = 50$, $p = 25$.
Heuristics with more machines than tasks.

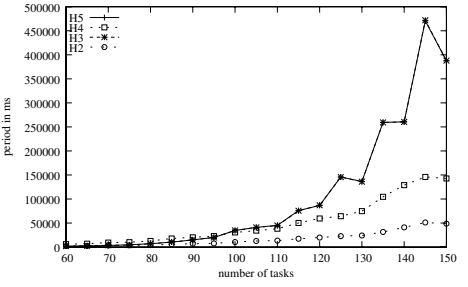


Fig. 4. $m = 50$, $p = 25$.
Heuristics with more tasks than machines.

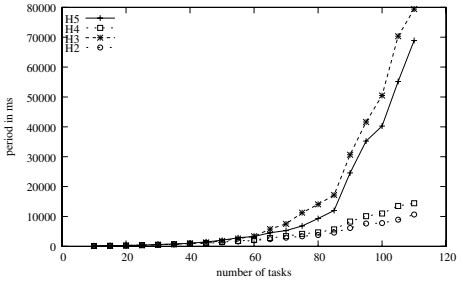


Fig. 5. $m = 40$, $p = 5$.
Small number of types.

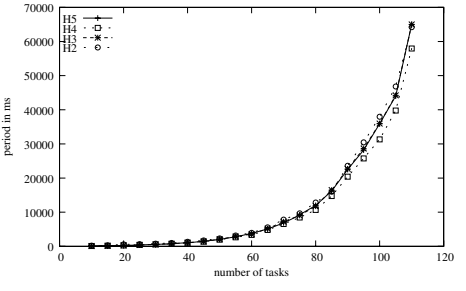


Fig. 6. $m = 40$, $p = 35$.
High number of types.

General behavior of the heuristics. In a second set of simulations, we focus on the behavior of the heuristics alone. First we compare settings with more tasks than machines, or the contrary. In Figure 3, we have $m = 50$, $p = 25$, and $10 \leq n \leq 50$. Results show that H2 is slightly worse than the other heuristics. This lack of performance of H2 becomes even clearer when p is closer to m (see [2] for further results). This is explained by the fact that H2 does not apply any penalization to the machines, thus using a good machine for many tasks of the same type. But when there is less tasks than machines, it is better to dedicate more machines to a given type. However, when the number of tasks is higher than the number of machines, H2 and H4 become clearly the best (see Figure 4). Indeed, at the end of the first stage of allocation, H3 and H5 will almost have used all the machines thus the second stage will not be decisive.

Also, we studied the impact of the number of types, for $m = 40$ and $10 \leq n \leq 110$. In Figure 5, we have $p = 5$, versus $p = 35$ in Figure 6. For $p = 5$, the possibilities to split groups are important. In this case, H2 and H4 are the best heuristics because the workload is shared on a higher number of machines and not only on those efficient for a given task. In the contrary, when the number of types is close to the number of machines ($p = 35$), the number of split tasks decreases. Indeed, each machine must be specialized to one type. In Figure 6, only 5 machines can be used to share the workload once each machine is dedicated to a type, and therefore the performance of the heuristics is pretty much alike.

Summary. Even though it is clear that H1 performs really poorly, the other heuristics can all be the most appropriate, depending upon the situation. If the number of tasks is greater than the number of machines, H2 is the best heuristic; otherwise, H4 becomes better than H2. Further simulations are done in [2], in particular to illustrate the impact of the failure rate on the results. Note that the comparison between the heuristics is made easier if the gap between the number of types and the number of machines is big. Indeed, with a small number of types, the tasks can be split many times because more machines are potentially dedicated to a same type. The choices made by a heuristic either to split a task or not have more impact on the result.

5 Conclusion

In this paper, we investigate the problem of maximizing the throughput of coarse-grain pipeline applications where tasks have a type and are subject to failures, with different mapping strategies (one-to-many, specialized or general). A task can be distributed on the platform so as to balance workload between the machines. From a theoretical point of view, an exhaustive complexity study is proposed. We prove that an optimal solution can be computed in polynomial time in the case of general mappings whatever the application/platform parameters, and in the case of one-to-many and specialized mappings when the failure rates only depend on the tasks, while the optimization problem becomes NP-hard in any other cases. Since general mappings do not provide a realistic solution because of unaffordable setup times when reconfiguration occurs, we propose to solve the specialized mapping problem by designing several polynomial heuristics. An exhaustive set of simulations demonstrate the efficiency of the heuristics: some of them return a throughput close to the optimal, while random mappings never give good solutions.

As future work, we plan to investigate other objective functions, such as the mean time to output one job out of the system, or other models: the failure rate associated to the task and/or the machine could be correlated with the time required to perform that task.

Acknowledgment. A. Benoit is with the Institut Universitaire de France. This work was supported in part by the ANR *StochaGrid* and *RESCUE* projects.

References

1. Bahi, J., Contassot-Vivier, S., Couturier, R.: Coupling dynamic load balancing with asynchronism in iterative algorithms on the computational grid. In: International Parallel and Distributed Processing Symposium, IPDPS 2003 (April 2003)
2. Benoit, A., Dobrila, A., Nicod, J.M., Philippe, L.: Workload balancing and throughput optimization for heterogeneous systems subject to failures. Research report, INRIA, France (February 2011), http://graal.ens-lyon.fr/~abenoit/
3. Blażewicz, J., Drabowski, M., Weglarz, J.: Scheduling multiprocessor tasks to minimize schedule length. IEEE Trans. Comput. 35, 389–393 (1986)

4. Cirne, W., Brasileiro, F., Paranhos, D., Góes, L.F.W., Voorsluys, W.: On the efficacy, efficiency and emergent behavior of task replication in large distributed systems. Parallel Computing 33(3), 213–234 (2007)
5. Descourvières, E., Debricon, S., Gendreau, D., Lutz, P., Philippe, L., Bouquet, F.: Towards automatic control for microfactories. In: IAIA 2007, 5th Int. Conf. on Industrial Automation (2007)
6. Garey, M.R., Johnson, D.S.: Computers and Intractability, a Guide to the Theory of NP-Completeness. W.H. Freeman and Company, New York (1979)
7. Gröflin, H., Klinkert, A., Dinh, N.P.: Feasible job insertions in the multi-processor-task job shop. European J. of Operational Research 185(3), 1308–1318 (2008)
8. Jalote, P.: Fault Tolerance in Distributed Systems. Prentice-Hall, Englewood Cliffs (1994)
9. Litke, A., Skoutas, D., Tserpes, K., Varvarigou, T.: Efficient task replication and management for adaptive fault tolerance in mobile grid environments. Future Generation Computer Systems 23(2), 163–178 (2007)
10. Parhami, B.: Voting algorithms. IEEE Trans. on Reliability 43(4), 617–629 (1994)
11. Schrijver, A.: Combinatorial Optimization: Polyhedra and Efficiency. Algorithms and Combinatorics, vol. 24. Springer, Heidelberg (2003)
12. Tanaka, M.: Development of desktop machining microfactory. Journal RIKEN Rev 34, 46–49 (2001) iSSN:0919-3405
13. Verettas, I., Clavel, R., Codourey, A.: Pocketfactory: a modular and miniature assembly chain including a clean environment. In: 5th Int. Workshop on Microfactories (2006)
14. Weissman, J.B., Womack, D.: Fault tolerant scheduling in distributed networks (1996)
15. West, R., Zhang, Y., Schwan, K., Poellabauer, C.: Dynamic window-constrained scheduling of real-time streams in media servers (2004)
16. West, R., Poellabauer, C.: Analysis of a window-constrained scheduler for real-time and best-effort packet streams. In: Proc. of the 21st IEEE Real-Time Systems Symp., pp. 239–248. IEEE, Los Alamitos (2000)
17. West, R., Schwan, K.: Dynamic Window-Constrained Scheduling for Multimedia Applications. In: ICMCS, vol. 2, pp. 87–91 (1999)
18. Wieczorek, M., Hoheisel, A., Prodan, R.: Towards a general model of the multi-criteria workflow scheduling on the grid. Future Gener. Comput. Syst. 25(3), 237–256 (2009)
19. Yu, J., Buyya, R.: A taxonomy of workflow management systems for grid computing. Research Report GRIDS-TR-2005-1, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia (April 2005)

On the Utility of DVFS for Power-Aware Job Placement in Clusters

Jean-Marc Pierson[1] and Henri Casanova[2]

[1] IRIT, University of Toulouse, Toulouse, France
pierson@irit.fr
[2] Dept. of Information and Computer Sciences, University of Hawai'i at Manoa, Honolulu, Hawai'i, U.S.A.
henric@hawaii.edu

Abstract. Placing compute jobs on clustered hosts in a way that optimizes both performance and power consumption has become a crucial issue. Most solutions to the *power-aware job placement problem* boil down to consolidating workload on a small number of hosts so as to reduce power consumption which achieving acceptable performance levels. The question we investigate in this paper is whether the capabilities provided by DVFS, i.e., the ability to configure a host in one of several power consumption modes, leads to improved solutions. We formalize the problem so that a bound on the optimal solution can be computed. We then study how the optimal, if it can be computed, and its bound vary across scenarios in which hosts provide various degrees of DVFS capabilities. We rely on a DVFS model that we instantiate based on real-world experiments. Our approach thus quantifies the potential improvements that hypothetical job placement algorithms can hope to achieve by exploiting DVFS capabilities.

1 Introduction

The problem of efficiently allocating resources among competing jobs and users on clusters has received considerable attention. Job scheduling algorithms have been developed that attempt to optimize job placement with respect to criteria related to job performance, system throughput, and/or fairness. Electrical power, although long been ignored in the job scheduling literature, has become a crucial issue for large-scale clusters. Consequently, in recent years many authors have studied the power-aware job placement problem, seeking theoretical as well as practical solutions [18,9,17,16,2]. A common approach for reducing the power consumption of a cluster is to place jobs so as to consolidate the workload on a small but sufficient number of cluster nodes, temporarily powering off unused nodes. Workload consolidation is enabled by virtual machine (VM) technology, which provides mechanisms to control and adapt resource shares allocated to VM instances, and to quickly migrate VM instances among cluster nodes. In addition to workload consolidation, a possibility is to configure the hardware via Dynamic Voltage Scaling (DVS), Dynamic Frequency Scaling (DFS), or a combination of

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 255–266, 2011.

the two (DVFS), which are now commonplace in modern processors. In this work, we seek to answer the following question: *What is the value added by DVFS when used in addition to workload consolidation when solving the power-aware job placement problem?* Some authors have proposed solutions that exploit DVFS capabilities [21,1,10,13]. However, obtained results are difficult to compare and do not provide a conclusive answer to the above question. Instead, we opt for an algorithm-agnostic approach and extend an existing formulation of the power-aware job placement problem proposed in the literature so that it accounts for cluster nodes that have DVFS capabilities. This problem can be formulated as an Mixed Integer Linear Program (MILP), which makes it possible to compute a bound on the optimal solution. Studying how the optimal and its bound vary as the DVFS capabilities of cluster nodes are enhanced provides a sound theoretical basis upon which to quantify the utility of DVFS when used in conjunction with workload consolidation. More specifically, our original contributions are:

- We combine and extend the power-job placement formulations in [4] and [23] and study two versions of the optimization problem: (i) optimize performance given a constraint on power consumption; and (ii) optimize power consumption given a constraint on job performance.
- We instantiate a model that captures the trade-off achieved between power consumption and performance using DVFS, based both on previous work and on experiments on a real-world platform.
- We compute the (bound on the) optimal for the optimization problem for several instantiations of the DVFS model, thereby quantifying the added benefit of increased DVFS capabilities.

Our main result is that using DVFS leads only to marginal improvements and that these improvements require only one additional intermediate power mode between the "powered off" and "powered on" modes.

2 Related Work

Since the power-aware job placement problem is NP-hard, most authors aim at designing heuristics. A prevalent technique consists in designing workload consolidation techniques so as to use a limited number of nodes, possibly relying on job migration. In [18,22], nodes are powered off when not used, and job placement decisions attempt to power a node back on only when absolutely necessary. Similarly, Hoyer et al. [16] propose statistical allocation planning for resource allocation while maintaining each job over a certain threshold on performance reduction. Others use various techniques to make online resource allocation decisions, leading to consolidation of jobs to a minimal set of nodes [17,20,15,2]. In addition to workload consolidation, it is possible to tune the hardware configuration of cluster nodes. For instance, users can specify that some hardware components can be slowed down or powered off for particular jobs [7]. More generally, the use of DVFS has been proposed in conjunction to workload consolidation techniques [9,21]. In the context of parallel applications, DVFS has

also been proposed as way to exploit and mitigate load imbalance and communication delays for the purpose of power consumption reduction [13,10]. Note that, like in this work, several authors have formulated various classes of power-aware job placement problems as linear programs [1,21,4]. But, to the best of our knowledge, no work to date has exploited such formulations to quantify the theoretical added benefit of using DVFS-enabled nodes.

3 Power-Aware Job Placement with DVFS

3.1 Problem Statement

We define the problem using [23] and [5] as foundations. We consider a cluster with H nodes, or *hosts*, and N jobs that must be placed and allocated resources on those hosts. Each job must be placed on exactly one host. We consider a static workload: no job enters or leaves the system and job resource needs are constant. Sound static job placement provides a good basis for job placement in the case of (more realistic) dynamic workloads. For instance, a static resource allocation can be recomputed periodically to account for changes in the workload. Alternately, a resource allocation can be recomputed for each job arrival/departure.

As in [23], which does not consider power consumption, we consider that hosts provide resources along an arbitrary number of resource dimensions (e.g., CPU time, RAM space, network bandwidth, disk space). Jobs have resource needs along the resource dimensions provided by hosts. We consider two kinds of resource needs: *rigid* and *fluid*. A rigid need denotes that a resource allocation is required. The job cannot benefit from a larger allocation and cannot operate with a smaller allocation. A fluid need specifies the maximum resource allocation that the job could use if alone on a reference host. The job cannot benefit from a larger allocation, but can operate with a smaller allocation at the cost of reduced performance. All that follows assume a single, reference host. A job could have two rigid needs: it could require 50% of the host's RAM and 20% of the host's disk space. The job could have two fluid needs: it could use up to 40% of the host's network bandwidth and up to 60% of the host's CPU time. In this example, the job cannot use both resources fully, for instance because of interdependence between I/O and computation. While not true in all cases [8], for simplicity we assume that rigid resource needs are completely independent from fluid resource needs (and from each other). Job resource needs can be discovered via benchmarking[24], analytical models [14], or runtime discovery [26,6].

Our metric for quantifying the performance of a particular job placement and resource allocation is the *scaled yield* [23]. For each fluid resource need the yield is defined as the ratio between the resource fraction allocated and the maximum resource fraction potentially used. For instance, if a job has a fluid CPU need of 60% but is allocated only 42% of the host's CPU, then the yield is $42/60 = 0.7$. Following the same rationale as in [23], we assume that the utilizations of all resources corresponding to the fluid needs of a job are linearly correlated. For the previous example, if the job were to be allocated only 20% of the host's I/O bandwidth (i.e., half of what it could potentially use), then it would use only

30% of the host's CPU (i.e., also half of what it could potentially use). The yield of a job is thus identical for all its fluid needs. We thus simply refer to the yield of a job, which takes values between 0 and 1. A job can, however, specify a QoS requirement as a minimum acceptable yield value. For the earlier example, the yield could be constrained to be higher than 0.4, which means that the CPU fraction allocated to the job would be at least $0.4 \times 60\% = 24\%$. The scaled yield of a job is then defined as:

$$scaled\ yield = \frac{yield - minimum\ yield}{1 - minimum\ yield}\ . \tag{1}$$

The scaled yield of a job thus takes values between 0 and 1. The objective is to maximize the minimum scaled yield over all jobs to optimize both aggregate performance and fairness among jobs [23]. Note that maximizing the average scaled yield is prone to starvation, as seen in [19] in the context of stretch optimization. However, in [23] a second optimization phase is used to maximize the average scaled yield while maintaining the previously maximized minimum scaled yield. The same approach could be used in this work as well.

As in [5], which studies power-aware job placement but considers only two resource dimensions, we assume that each host consumes power depending on utilization of its resources. Unlike [23,5], we consider that each host provides DVFS capabilities. More specifically, it is possible to reduce the power consumption of a host by reducing the capacity of one or more of its resources (e.g., lowering the number of CPU cycles per seconds by lowering the clock rate). Unlike [23,5], we allow for heterogeneous hosts. We quantify power consumption of the system as the sum of the power consumptions of all hosts, accounting for the load imposed on each host.

When faced with two distinct objectives (i.e., maximizing yield and minimizing power consumption) one possibility is to optimize a linear combination of them. This approach, albeit commonplace, is problematic because the coefficient of the linear combination must be chosen by the user, and also because the obtained solution is not guaranteed to be Pareto optimal. Instead, we consider two separate single-objective optimization problems:

1. BoundedPower: Maximize the minimum scaled yield given an upper bound on power consumption (i.e., a power budget).
2. BoundedYield: Minimize the power consumption given a lower bound on the minimum scaled yield (i.e., a performance budget).

3.2 Problem Formulation

In this section we formalize both optimization problems introduced in the previous section. We consider H hosts providing a d-dimensional resource and N jobs that must be placed on these hosts. Each host h can operate in $nv_h \geq 2$ distinct power modes. Each power mode corresponds to a different tradeoff between the resources available from the host and its power consumption. We ignore the overhead of modifying a host's power mode, including powering it on and off. In

$$\forall h \quad \sum_k p_{hk} = 1 \tag{2}$$

$$\forall i \quad \sum_{h,k} e_{ihk} = 1 \tag{3}$$

$$\forall i, h \quad e_{ih1} = 0 \tag{4}$$

$$\forall i, h, k \quad e_{ihk} \leq p_{hk} \tag{5}$$

$$\forall i, h, k \quad 0 \leq y_{ihk} \leq e_{ihk} \tag{6}$$

$$\forall i, h, k, j \quad w_{ihjk} = r_{ij}(y_{ihk}(1 - \delta_{ij}) + e_{ihk}\delta_{ij}) \tag{7}$$

$$\forall h, k, j \quad \sum_i w_{ihjk} \leq p_{hk} f_{hjk} \tag{8}$$

$$\forall i \quad \sum_{h,k} y_{ihk} \geq \hat{y}_i + Y(1 - \hat{y}_i) \tag{9}$$

$$E = \sum_{h,j,k} Power_{hjk}(\sum_i w_{ihjk}) \tag{10}$$

Fig. 1. Optimization constraints

power mode k, host h provides a capacity f_{hjk} for resource dimension j. Rather than quantifying f_{hjk} with absolute resource-specific units, we use a relative measure so as to easily account for heterogeneous hosts and/hosts in different power states. For resource dimension j, we identify the maximum resource fraction provided by any host in any power state in the platform. All f_{hjk} values are taken relative to this maximum and are thus between 0 and 1. For all h and j, f_{hj1} is zero, meaning that the first power mode for each host corresponds to being powered down. Job i's resource need along dimension j is denoted by r_{ij} and is relative to the aforementioned maximum. Furthermore, we define δ_{ij} to be 1 if job i's resource need in resource dimension j is rigid, and 0 if the need is fluid. The minimum yield required by job i, i.e., its Quality Of Service requirement, is denoted by $\hat{y}_i$. Finally, the power consumption of host h in power mode k due to resource dimension j is given by a function $Power_{hjk}(x)$, where x is the total resource usage in this resource dimension due to jobs placed on the host.

To formulate both our job placement problems as constrained optimization problems we define the following variables: (i) Y is a rational variable that quantifies the minimum scaled yield over all jobs; (ii) E is a rational variable that quantifies the power consumption of the platform; (iii) p_{hk} is a binary variable that is 1 if host h is in power mode k, and 0 otherwise; (iv) e_{ihk} is a binary variable that is 1 if job i is placed on host h in power mode k, and 0 otherwise; (v) y_{ihk} is a rational variable which quantifies the yield of job i on host h in power mode k; and (vi) w_{ihjk} is a rational variable which quantifies the resource usage due to job i on host h in power mode k for resource dimension j.

We can now write the set of constraints shown in Figure 1, where $i \in 1, \ldots, N$, $h \in 1, \ldots, H$, $j \in 1, \ldots, d$, and $k \in 1, \ldots, nv_h$. Constraint 2 states that a host operates in a single power mode. Constraint 3 states that a job can only be placed on a single host (which operates in a given power mode). Constraint 4 states that a job cannot be placed on a host that is powered down. Constraint 5 states that a job can only be placed on a host in a particular power mode if this host operates in this power mode. Constraint 6 states that the yield of a job can be non-zero only on the host (and its given power mode) on which the job

is placed. Constraint 7 defines the resource usage w_{ihjk}, which is non-zero only if job i is placed on host h that operates in power mode k. For a rigid resource need ($\delta_{ij} = 1$) the resource usage is equal to the job's resource need, while for a fluid need ($\delta_{ij} = 1$) the resource usage is scaled by the job's yield. Constraint 8 states that the resource usage on a host in a given power mode does not exceed the corresponding resource capacity. Constraint 9 simply states that, for each job, the minimum scaled yield, Y, is lower that the job's scaled yield. Finally, Constraint 10 states that the total power consumption of the platform is equal to the sum of the power consumption on all hosts (in their respective power states) along all resource dimensions. One can now formalize BoundedPower as maximizing Y subject to the constraints in Figure 1 with an additional $E \leq E_{max}$ constraint. Similarly, BoundedYield consists in minimizing E subject to the constraints in Figure 1 with an additional $Y \geq Y_{min}$ constraint.

In this work we make the simplifying assumption that the power consumption of a host is entirely driven by the power consumption of its CPU, i.e., along a single resource dimension. While it would be straightforward to incorporate other sources of power consumption in our problem formalization, models need to be developed for the corresponding $Power_{hjk}()$ functions. In fact, it could very well be that the power consumption of the different resources are not independent, in which case a single $Power_{hk}$ function that models power consumption given the d resource usages on host h in power mode k. For now, given that such models are not available and that the CPU does account for a large fraction of a host's dynamic power consumption, we assume the $Power_{hjk}()$ always returns 0 for $j \neq 1$. Arbitrarily choosing that the CPU resource dimension corresponds to $j = 1$, we define

$$Power_{h1k}(x) = C_{hk}^{min} + C_{hk}^{prop} \times x \ ,$$

where C_{hk}^{min} is the power consumption of host h in power mode k when idle, and C_{hk}^{prop} denotes the proportional increase in power consumption over C_{hk}^{min} due to a CPU load x (i.e the dynamic power consumption). This load is expressed as fraction of the maximum possible number of CPU cycles executed per time unit by a host, over all hosts in all their power states. This model and assumptions were experimentally shown close to real clusters power consumptions [11,25]. All constraints in Figure 1 are thus linear, and both optimization problems are thus MILPs. One can use a linear solver to compute exact solutions for small instances. For larger instances, one can relax the binary variables (e_{ihk} and p_{hk}) to take rational values between 0 and 1. The obtained solution is generally not feasible in practice, but provides an optimistic bound on the optimal. Note that for resources other than the CPU, the power consumption model may not be linear, in which case the theoretical approach used in this work would not apply.

4 DVFS/DFS Model

In our problem definition a host h is fully specified by C_{hk}^{min}, C_{hk}^{prop}, and f_{h1k}, for $k = 1, \ldots, nv_h$. In this section we explain how we instantiate these values so as to generate problem instances that are representative of real-world platforms.

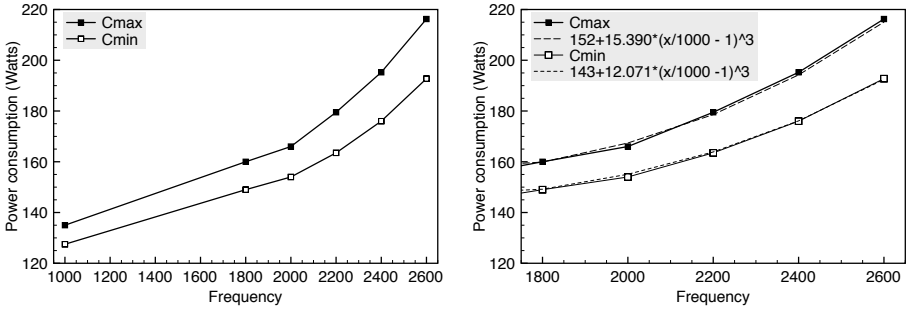


Fig. 2. Left: C^{min} and $C^{max} = C^{min} + C^{prop}$ (in Watts) vs. operating frequency (in MHz). Right: cubic fit on the 1800-2600 frequency range (in MHz).

Several authors have investigated the modeling of power consumption as a function of the frequency-voltage of the processor, which is correlated to compute speed. Following [12], Dynamic Frequency Scaling (DFS), or T-States, allows for running the processor at different frequencies. Dynamic Voltage and Frequency Scaling (DVFS), or P-States, allows for reducing both frequency and voltage, leading to better savings than DFS. Combined DVFS+DFS consists in applying DFS to the lowest power consumption mode available in DVFS.

The most comprehensive processor power consumption study to date is the one in [12]. The authors develop models that relate compute speed to power settings. The derived models are linear for DFS and DVFS, and cubic when both techniques are combined. While these models give power consumption as a function of the frequency, they say nothing regarding the evolution of the minimum and/or maximum power consumption at different frequencies. Consequently, we experimented on a real-world platform (AMD bi-processors dual-core in the Grid5000/Toulouse [3] platform) and we measured values for C^{min} (when hosts are idle) and $C^{max} = C^{min} + C^{prop}$ (when processors run cpuburn) at the 6 available operating frequencies on these hosts so as to generate representative problem instances. Results were consistent across hosts. Figure 2 shows result for one host corresponding to the case when DFS and DVFS are combined, which is what is assumed in all experiments hereafter. The frequency ranges from 1Ghz to 2.6Ghz, and C^{min} and C^{max} range from 127 Watts to 210 Watts, as seen in the graph on left-hand side. As in [12] we generate a cubic model for C^{min} and C^{max} from the experimental data, as seen in the graph on a right-hand side. The model is accurate on the 1.8GHz-2.6GHz range, which is the range used in all experiments hereafter. Using this model, and given a number of power modes and a frequency range, we can thus generate representative problem instances.

5 Numerical Results

5.1 Experimental Methodology

We generate sets of problem instances for BoundedPower and BoundedYield as follows. We generate instances with either $H = 4$ hosts ("small" instances for which we can compute the optimal solution) and $H = 32$ hosts ("large" instances). Given that our goal in this work is solely to investigate the effect of DVFS on job placement, we generate problem instances in which hosts provide only a CPU resource, for which jobs express fluid resource needs. Adding other (rigid) resources would complexify the job placement problem (i.e., in terms of bin packing) but have little impact on the bound on optimal we compute. Each host in the platform can be configured in the same number of power modes (i.e., $nv = nv_h$ does not depend on h). We use $nv \in \{2, 3, 4, 6, 8, 10, 15, 20\}$ for our instances, thus spanning the range from hosts that provide only two power modes (on and off) to hosts that allow for a fine-grain trade-off between compute speed and power consumptions. Although our problem formulation allows for heterogeneous hosts, we found that conclusions regarding the utility of DVFS are identical for homogeneous and heterogeneous platforms. Consequently, we only present results obtained for homogeneous platforms hereafter as the experimental scenarios are much simpler. Rather than picking a number of jobs N, we instead pick a load factor α and generate the smallest number of N random jobs such that the sum of their CPU resource needs is greater than or equal to $\alpha \times H$. For instance, $\alpha = 2$ corresponds to a scenario in which the platform is underprovisioned by (slightly more than) a factor two. For small instances ($H = 4$), we pick α between 1.0 and 1.5, while for large instances ($H = 32$), we pick α between 1.0 and 3.0. Job CPU needs, numbers between 0 and 1, are picked randomly using a truncated Gaussian distribution of mean 0.5. QoS requirements are set to 0 for all jobs.

For each instance specification, i.e., a (H, nv, α) triplet, we generate 100 random host and job specifications. We then generate a BoundedPower instance and a BoundedYield instance. For the BoundedPower instance, we constrain E to be below a fixed value. This fixed value is half of the power consumption obtained when placing jobs on hosts using a simple greedy algorithm (place the next job on the least loaded processor). For the BoundedYield instance, we arbitrarily constrain Y to be above 0.5. For each problem and for all instances we compute the solution of the relaxed MILP, which we denote by LPBound. For small instance, we compute the optimal solution, which we denote by Milp. We use the open-source GLPK linear solver to compute these solutions on an 3.2GHz Intel Xeon processor. We arbitrarily set a timeout value to 10 minutes and declare an instance not solvable if the timeout is exceeded.

5.2 Results for Small Instances

In this section we discuss results obtained on small instances. Our main objective here is to quantify the difference between Milp and LPBound. Figure 3 plots the average percentage difference between LPBound and Milp computed over

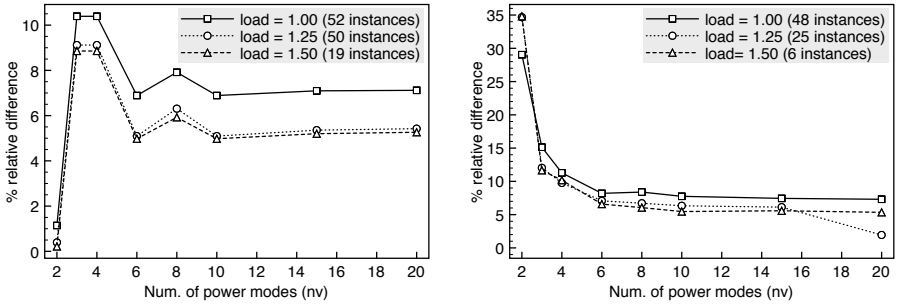


Fig. 3. Percent relative difference between LPBound and Milp vs. nv. Left-hand side: BoundedPower; Right-hand side: BoundedYield

successfully solved problem instances as the number of power modes increases, for small BoundedPower instances (left-hand side) and small BoundedYield instances (right-hand side). Each graph contains three curves, one for each value of $\alpha = 1.00, 1.25, 1.50$. The number of successfully solved instances out of the 100 generated instances is indicated in the legend. As expected, as α increases the number of solvable instances decreases since the number of optimization constraints increases. For BoundedPower, we see that LPBound is at most about 11% away from Milp. Interestingly the results for BoundedYield show a different pattern, with the relative different dropping sharply as nv increases. Regardless, in both cases, the relative difference is below 10% for large values of nv. It is these large values that are particularly relevant in this work since they correspond to scenarios in which hosts provide extensive DVFS capabilities. We conclude that LPBound tracks the Milp solution well for small instances. For large instances Milp cannot be computed, but we contend that a better LPBound value (i.e., higher for BoundedPower, lower for BoundedYield) is a strong indication of an opportunity for a better Milp value.

5.3 Results for Large Instances

Table 1 shows results for the averages and standard deviations of the percentage relative improvement of LPBound with $nv = 3$ over the case $nv = 2$ (i.e., no DVFS), both for BoundedPower and BoundedYield. Each row of the table corresponds to a particular α value, and percentage relative improvements are computed between two instances that are completely identical but for the nv values (i.e., they have the exact job sets). The table does not show any results for $nv > 3$ because results are identical to the $nv = 3$ case for all successfully solved instances. Our first important conclusion is thus that configuring hosts in more than 3 power modes does not lead to improved LPBound values.

For the BoundedPower problem, we see that α has no impact on the result. This is because LPBound is computed as the solution to a relaxed MILP, meaning that jobs are considered perfectly divisible. In this case, the platform can be conceptually considered as a single host whose compute capacity is simply bounded by the constraint on power consumption E. Changing the α value

Table 1. Average relative percentage LPBound improvement for $nv = 3$ instances over $nv = 2$ instances, for the BoundedPower and BoundedYield problems, and percentage of successfully solved instances. Standard deviations are in parentheses.

	BoundedPower		BoundedYield	
α	% imprvmnt	% success	% imprvmnt	% success
1.00	12.14 (8.43e-4)	100	10.83 (1.60e-7)	100
1.25	12.14 (1.05e-3)	100	10.83 (1.45e-7)	100
1.50	12.14 (1.25e-3)	100	9.59 (1.39)	100
1.75	12.14 (1.46e-3)	100	5.02 (1.40)	100
2.00	12.14 (1.53e-3)	100	1.85 (1.11)	88
2.25	12.14 (1.78e-3)	100	0.50 (0.36)	9
2.50	12.14 (2.07e-3)	100	n/a	0
2.75	12.14 (2.07e-3)	100	n/a	0
3.00	12.14 (2.56e-3)	100	n/a	0

simply amounts to scaling the yield of all jobs, meaning that the percentage difference between two solutions with different nv values does not change. The situation is different when solving BoundedYield. First, as α increases, GLPK fails to compute solutions. In some cases, these errors are due to 10-minute timeouts, likely due to the fact that the objective function is more complex than for BoundedPower. In other cases, the instances are not solvable, simply because job yields are bounded below by the constraint on the minimum scaled yield. Regardless, a more important observation is that the relative percentage improvement of using $nv = 3$ power modes over using $nv = 2$ power modes decreases as α increases. This is because a larger α implies a higher computational load on the system, which must be accommodated by configuring hosts in higher power modes (while staying within the power consumption budget if possible). Consequently, intermediate power modes are used increasingly less frequently in highly loaded scenarios and DVFS is increasingly less useful.

6 Conclusion

We have extended an existing formulation of the power-aware job placement problem proposed in the literature so that it accounts for cluster nodes that have DVFS capabilities. This formulation is a Mixed-Integer Linear Program, which makes it possible to compute a bound on the optimal solution, and relies on a DVFS model that we have instantiated based on a real-world platform. We have shown on small problem instances that the bound is reasonably close from the optimal solution, thereby indicating the the bound is likely a good indicator of the optimal. For large instances, in which case the optimal cannot be computed in a feasible amount of time, our main results can be summarized as: (i) using $nv > 3$ DVFS power modes never leads to improved LPBound values; (ii) using $nv > 2$ power modes leads to marginal improvements ($< 15\%$) when solving BoundedPower; (ii) using $nv > 2$ power modes leads to lower improvements ($< 11\%$) when solving BoundedYield, and the improvement

decreases quickly as the compute load increases. Assuming that LPBound is a good indicator of the optimal, our overall conclusion is that DVFS is at best marginally useful, and that only one intermediate power mode between "on" and "off" is needed.

References

1. Benoit, A., Renaud Goud, P., Robert, Y.: Sharing resources for performance and energy optimization of concurrent streaming applications, http://hal.archives-ouvertes.fr/hal-00457323/PDF/RR-LIP-2010-05.pdf, RR-LIP-2010-05
2. Berral, J.L., Goiri, Í., Nou, R., Julià, F., Guitart, J., Gavaldà, R., Torres, J.: Towards energy-aware scheduling in data centers using machine learning. In: ACM eEnergy. University of Passau, Germany (2010)
3. Bolze, R., Cappello, F., Caron, E., Daydé, M.J., Desprez, F., Jeannot, E., Jégou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quétier, B., Richard, O., Talbi, E.-G., Touche, I.: Grid'5000: A large scale and highly reconfigurable experimental grid testbed. IJHPCA 20(4), 481–494 (2006)
4. Borgetto, D., Casanova, H., Costa, G.D., Pierson, J.M.: Energy-aware service allocation. Tech. Rep. IRIT/RT-2010-7-FR, IRIT (October 2010)
5. Borgetto, D., Da Costa, G., Pierson, J.-M., Sayah, A.: Energy-Aware Resource Allocation. In: Proc. of the Energy Efficient Grids, Clouds and Clusters Workshop (E2GC2). IEEE, Los Alamitos (2009)
6. Carrera, D., Steinder, M., Whalley, I., Torres, J., Ayguadé, E.: Utility-based placement of dynamic web applications with fairness goals. In: IEEE Network Operations and Management Symposium, pp. 9–16 (2008)
7. Da Costa, G., Dias De Assuncao, M., Gelas, J.P., Georgiou, Y., LefËvre, L., Orgerie, A.C., Pierson, J.M., Richard, O., Sayah, A.: Multi-Facet Approach to Reduce Energy Consumption in Clouds and Grids: The GREEN-NET Framework. In: ACM/IEEE International Conference on Energy-Efficient Computing and Networking (e-Energy), Passau, Germany, pp. 95–104. ACM, New York (2010)
8. Doyle, R.P., Chase, J.S., Asad, O.M., Jin, W., Vahdat, A.M.: Model-based resource provisioning in a web service utility. In: Proc. of the USENIX Symposium on Internet Technologies and Systems (2003)
9. Etinski, M., Corbalan, J., Labarta, J., Valero, M.: Utilization driven power-aware parallel job scheduling. Computer Science - Research and Development 25, 207–216 (2010), doi:10.1007/s00450-010-0129-x
10. Etinski, M., Corbalan, J., Labarta, J., Valero, M., Veidenbaum, A.: Power-aware load balancing of large scale mpi applications. In: IPDPS 2009: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing, pp. 1–8. IEEE Computer Society, Washington, DC, USA (2009)
11. Fan, X., Weber, W.D., Barroso, L.A.: Power provisioning for a warehouse-sized computer. In: Proceedings of the 34th Annual International Symposium on Computer Architecture, ISCA 2007, pp. 13–23. ACM, New York (2007)
12. Gandhi, A., Harchol-Balter, M., Das, R., Lefurgy, C.: Optimal power allocation in server farms. In: SIGMETRICS/Performance, pp. 157–168. ACM, New York (2009)

13. Ge, R., Feng, X., Cameron, K.W.: Performance-constrained distributed dvs scheduling for scientific applications on power-aware clusters. In: SC 2005: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, p. 34. IEEE Computer Society, Washington, DC, USA (2005)
14. Gmach, D., Rolia, J., Cherkasova, L., Kemper, A.: Workload Analysis and Demand Prediction of Enterprise Data Center Applications. In: Proc of the 10th IEEE Intnl. Symp. on Workload Characterization, September 2007, pp. 171–180 (2007)
15. Hermenier, F., Lorca, X., Menaud, J.M., Muller, G., Lawall, J.: Entropy: a Consolidation Manager for Clusters. Research Report RR-6639, INRIA (2008)
16. Hoyer, M., Schröder, K., Nebel, W.: Statistical static capacity management in virtualized data centers supporting fine grained QoS specification. In: ACM eEnergy. University of Passau, Germany (2010)
17. Kamitsos, Y., Andrew, L.L.H., Kim, H., Chiang, M.: Optimal Sleep Patterns for Serving Delay Tolerant Jobs. In: ACM eEnergy. University of Passau, Germany (2010), http://netlab.caltech.edu/lachlan/abstract/eEnergySleep.pdf
18. Lawson, B., Smirni, E.: Power-aware resource allocation in high-end systems via online simulation. In: Proceedings of the 19th Annual international Conference on Supercomputing, ICS 2005, pp. 229–238. ACM, New York (2005)
19. Legrand, A., Su, A., Vivien, F.: Minimizing the Stretch when Scheduling Flows of Divisible Requests. Journal of Scheduling 11(5), 381–404 (2008)
20. Niyato, D., Chaisiri, S., Sung, L.B.: Optimal power management for server farm to support green computing. In: CCGRID 2009: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, pp. 84–91. IEEE Computer Society, Washington, DC, USA (2009)
21. Petrucci, V., Loques, O., Mossé, D.: A Dynamic Optimization Model for Power and Performance Management of Virtualized Clusters. In: ACM eEnergy. University of Passau, Germany (2010)
22. Rodero, I., Jamarillo, J., Quiroz, A., Parashar, M., Guim, F., Poole, S.: Energy-efficient application-aware online provisioning for virtualized clouds and data centers. In: First IEEE Sponsored International Green Computing Conference (2010)
23. Stillwell, M., Schanzenbach, D., Vivien, F., Casanova, H.: Resource allocation algorithms for virtualized service hosting platforms. Journal of Parallel and Distributed Computing 70(9), 962–974 (2010)
24. Urgaonkar, B., Shenoy, P., Roscoe, T.: Resource Overbooking and Application Profiling in Shared Hosting Platforms. SIGOPS Oper. Syst. Rev. 36(SI), 239–254 (2002)
25. Wang, Z., Tolia, N., Bash, C.: Opportunities and challenges to unify workload, power, and cooling management in data centers. SIGOPS Oper. Syst. Rev. 44, 41–46 (2010)
26. Zhu, X., Young, D., Watson, B.J., Wang, Z., Rolia, J., Singhal, S., McKee, B., Hyser, C., Gmach, D., Gardner, R., Christian, T., Cherkasova, L.: 1000 Islands: Integrated Capacity and Workload Management for the Next Generation Data Center. In: Proceedings of the International Conference on Autonomic Computing (ICAC 2008), pp. 172–181 (June 2008)

Introduction

Mitsuhisa Sato, Denis Barthou, Pedro C. Diniz, and P. Saddayapan

Topic chairs

This topic deals with architecture design and and compilation for high performance systems. The areas of interest range from microprocessors to large-scale parallel machines; from general-purpose platforms to specialized hardware; and from hardware design to compiler technology. On the compilation side, topics of interest include programmer productivity issues, concurrent and/or sequential language aspects, program analysis, program transformation, automatic discovery and/or management of parallelism at all levels, and the interaction between the compiler and the rest of the system. On the architecture side, the scope spans system architectures, processor micro-architecture, memory hierarchy, and multi-threading, and the impact of emerging trends.

All the papers submitted to this track highlight the growing significance of Chip Multi-Processors (CMP) and scalability issues in performance/power in contemporary high-performance architectures.

The paper "Filtering directory lookups in CMPs with write-through caches" by Ana Bosque, Victor Viñals, Pablo Ibañez and Jose Maria Llaberia proposes an architectural enhancement to reduce the number of directory lookups for a directory-based coherence protocol, in CMP shared caches. The authors describe a hardware filter reducing the associativity of the lookups, eliminating these lookups in some cases and reducing power consumption.

The paper "FELI: HW/SW support for On-Chip Distributed Shared Memory in Multicores" by Carlos Villavieja, Yoav Etsion, Alex Ramirez and Nacho Navarro proposes a set of operating system mechanisms to automatically manage memory on a CMP with on-chip scratchpad memories. The authors describe how the virtual memory paging mechanism is leveraged to achieve this and reduce power consumption.

The paper "Token3D: Reducing Temperature in 3D die-stacked CMPs through Cycle-level Power Control Mechanisms" by Juan M. Cebrián, Juan L. Aragón and Stefanos Kaxiras describes a token-based power management algorithm for multi-core architectures organized as stacks of chips. The paper describes how to take into account temperature and layer information when balancing power, giving higher priority to cool cores over hot ones.

The paper "Unified Locality-sensitive Signatures for Transactional Memory" by Ricardo Quislant, Eladio D Gutierrez, Oscar Plata and Emilio Zapata describes a new design for the hardware support of transactional memory. The authors proposes to combine the use of locality-sensitive signatures with unified hash functions for read and write sets.

Last but not least, two papers address prefetching issues with new hardware prefetching schemes. The paper "Bandwidth Constrained Coordinated HW/SW Prefetching For Multicores" by Sai Prashanth Muralidhara, Mahmut

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 267–268, 2011.

Taylan Kandemir and Yuanrui Zhang presents a hierarchical management and bandwidth-constrained prefetching algorithm for multi-cores. The authors describe a prefetching scheme and the metrics used to adjust dynamically the aggressiveness of the prefetch, handling bandwidth contention and performance. The paper "Using runtime activity to dynamically filter out inefficient data prefetches" by Gamoudi, Nathalie Drach and Karine Heydemann describes a hardware prefetching strategy exploiting runtime activity information and a history-based algorithm to filter out inefficient data prefetches. The paper describe a method to correlate runtime activities with prefetching effects in order to increase prefetching efficiency.

We would like to take this opportunity to thank the authors who submitted a contribution, as well as the Euro-Par Organizing Committee, and the referees with their highly useful comments, whose efforts have made this conference and this topic possible.

Filtering Directory Lookups in CMPs with Write-Through Caches

Ana Bosque[1], Victor Viñals[2], Pablo Ibañez[2], and Jose Maria Llaberia[1]

[1] DAC, UPC, Barcelona, Spain
{abosque,llaberia}@ac.upc.edu
[2] DIIS, University of Zaragoza, Zaragoza, Spain
{victor,imarin}@unizar.es

Abstract. In CMPs, coherence protocols are used to maintain data coherence among the multiple local caches. In this paper, we focus on CMPs using write-through local caches, and a directory-based coherence protocol implemented as a duplicate of the local cache tags. A large fraction of directory lookups is due to stores performed on private data local to the processor performing the store.

We propose to add a filter before the directory in order to either reduce the associativity of the lookups or even eliminate those that are unnecessary. When a block from the shared cache has only one copy in the local caches, the filter identifies the processor and allows for reducing the number of comparisons performed in the corresponding directory lookup. When that is not possible, the filter bits are used to code other situations that can also reduce the number of directory lookups or their associativity.

We evaluate the fillter in a CMP with 8 in-order processors with 4 threads each and a memory hierarchy with local caches and a shared cache. We show that a filter representing 0.7% of the size of the shared cache can avoid, on average, 97% and 93% of all comparisons performed by directory lookups for SPLASH2 and Specweb2005, respectively. Only for SPLASH2, there is a small performance loss of 0.3%. As a result, on average, directory power is reduced 30.8% and 22.4% for SPLASH2 and Specweb2005, respectively.

1 Introduction

Chip-multiprocessors (CMPs) have become the industry choice of design for high-performance processors. Nowadays, most computer manufactures offer CMPs with different number of cores [20,4,13,16,18], where each of them has at least a local cache level. All CMPs support the shared memory programming paradigm. Thus, local caches need to be kept coherent by means of a coherence protocol.

Directory-based protocols keep a directory that stores the state of each block of main memory. All transactions should access this structure in order to determine which coherence actions to perform. A directory can be implemented in two

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 269–281, 2011.

basic ways: by a full-map [8], or by duplicating the local cache tags [31]. Differences between duplicate tag directory and full-map arise in size, lookup method, and retrieved information in a lookup operation. Concerning size, the duplicate tag directory uses the smallest explicit representation of all blocks contained in local caches. Thus, a duplicate tag directory requires less area than a full-map directory. However, by duplicating local cache tags, any directory lookup requires an associative lookup that is expensive in terms of energy consumption. For example, in Niagara 2, a lookup can perform up to 256 comparisons.

The number of directory lookups necessary in a coherence protocol depends on the write policy of the local caches. The commercial CMP Niagara 2 [18] uses write-through local caches and a shared cache. It requires more bandwidth than the Piranha prototype [6], which uses write-back local caches, because all stores must access the shared cache. However, the extra bandwidth consumed by Niagara 2 assures that data is always up-to-date in the shared cache. Thus, an access to shared data is serviced directly from the shared cache without any intervention from the local caches. The drawback of write-through local caches, though, is that private stores are sent both to the shared cache and to the directory, where a (probably useless) lookup needs to be necessarily performed. In a CMP like Niagara 2, any store requires a 96-associative directory lookup.

In this paper, we show that many of the directory lookups done by stores are useless. We propose a mechanism to identify stores to private data in order to avoid many lookups in the directory. Furthermore, the mechanism is extended to deal with other situations in which directory lookups can be avoided.

Our results show that for SPLASH2, just by using a filter whose size is 0.7% the size of the shared cache, we can avoid 97% of the comparisons performed inside the directory with a tiny 0.2% performance loss. For Specweb2005, the number of comparisons performed by directory lookups is reduced by 93%. On average, directory power consumption is reduced by 30.8% and 22.4% for SPLASH2 and Specweb2005, respectively.

The rest of this paper is organized as follows. In Section 2, we motivate our work. Section 3 describes the proposed filter. Section 4 shows our experimental results. Section 5 discusses related work and Section 6 contains the conclusions.

2 Motivation

In a directory-based protocol, both stores that access the shared cache and evictions in an inclusive shared cache require a directory lookup in order to invalidate the copies of the block in the local caches.

In a CMP with write-back local caches, stores access the shared cache either on a miss in the local data cache or to get the block ownership and change the coherence state of the block to *Modified*. However, if local caches are write-through, all the stores must access the shared cache. In our workloads (Section 4.2) we found that only 1 out of 100,000 stores access true shared data. Thus, in a CMP with write-through local caches and a duplicate tag directory, when an associative lookup is performed by a store, it happens that most of the times the only copy of the cache block is located in the processor performing the store. The

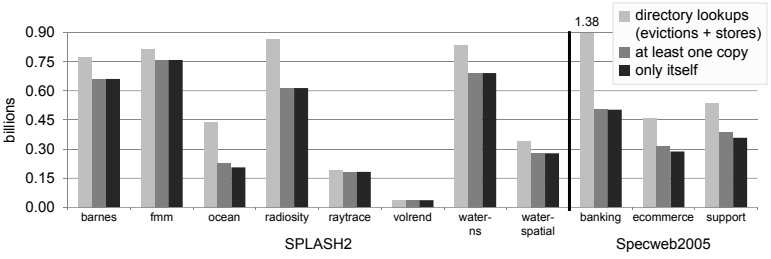


Fig. 1. Billions of directory lookups (`directory lookups`), billions of directory lookups that find at least a copy of the cache block in any local cache (`at least one copy`), and billions of directory lookups performed by stores that only find a copy just in the local cache of the processor that performs the current store (`only itself`).

directory lookups performed by these stores are needless and it is possible to improve directory energy-efficiency by filtering them out.

Figure 1 presents an analysis of directory lookups. Refer to Section 4.1 for the parameters of the simulated CMP model and to Section 4.2 for a description of the workloads used. The difference between the first two bars is the number of times that there are no copies of the shared cache block in any local cache. On average, this difference represents 30% of the directory lookups. The difference between the second and the third bar represents all cases that require to invalidate local cache blocks. These cases are: a) evictions performed over shared cache blocks which have local copies, and b) stores performed over shared cache blocks that are allocated at least in a local cache different from the local cache of the processor performing the store. On average, this difference represents 1% of the directory lookups. The remaining directory lookups (69%) are performed by private stores, i. e. stores that access cache blocks without copies in any other processor's local cache.

The proposed mechanism aims to reduce the number of total directory lookups by filtering out private stores, so that they do not perform expensive and useless directory lookups. Thus, the proposed mechanism reduces the number of directory lookups shown in the third column in Figure 1. Additionally, the filter is enhanced in order to also avoid the 30% of directory lookups that do not find any copy in the local caches, and that are also useless. In Figure 1, these directory lookups are represented by the difference between the first and the second column.

3 Filtering Mechanism

3.1 Overview

We assume a CMP with a shared inclusive L2 cache and multithreaded processors that access local instruction and write-through data caches. A detailed description of the CMP model is in Section 4.1.

As processors are multithreaded, local caches are highly accessed by the processors. A directory organization such as a full-map directory requires a lookup in the local cache tags for every invalidation. As a result, if processor requests and invalidations sent from the directory share the same local cache port, thread execution can be delayed. Thus, the local cache tags require two ports so that thread performance is not diminished. An alternative is to replicate the cache tags [28,9]. This replica is located side-to-side with the local cache tags and it is used by invalidations to set the state bits of the cached blocks.

The replica of the local cache tags can be located in the other side of the interconnection network and be used as a duplicate tag directory. The full-map directory is removed. Now, when an invalidation is sent, the local cache set and way to invalidate is already identified in the message, and a local cache lookup is not needed. As the replica of the local cache tags is located together with the inclusive shared cache, it is possible to keep pointers to the shared cache tags (set index and way) instead of the local cache tags themselves. Consequently, the duplicate tag structure is much smaller [18].

Every directory lookup requires an expensive associative lookup in the duplicate tag structure. However, using a full-map directory, only the tags of the processors effectively having a copy of the block are looked up. Based on program behavior, we propose to use a filter before accessing the duplicate tag directory in order to reduce the lookup associativity. Figure 1 shows that, on average, 69% stores are private. If we identify these cases, the lookup in the duplicate tag directory can be restricted to the duplicate tag of the processor performing the private store (in order to determine which cache way to update in the local cache, see Section 4.1).

The proposed filter manages the same information than a DIR_1NB directory scheme [1], but it is only used as a filter before looking up in the duplicate tag directory. In a DIR_1NB directory, the only processor than can have the copy of a block (owner) is identified. Thus, the proposed filter has as many entries as lines in the shared cache, and each entry has $\log_2 P$ bits plus a valid bit. When the valid bit is zero, the representation of the owner identifier bits is changed to a coarse granularity [14,19]. Consequently, other situations might be identified, for example, whether there are no copies of a block in any local cache. Figure 1 shows that 30% directory lookups are performed under these conditions.

3.2 Filter Operation

Each line in the shared cache has associated one entry in the filter. For every shared cache access or eviction (memory operation from now on), the filter entry is read together with the state bits of the line. Depending on the value stored in the filter, the directory lookup performed by any memory operation accessing that line can be either eliminated or performed over a smaller number of entries in the directory structure.

A filter entry state is updated using only the following information: memory operation type, identifier of the processor performing the memory operation, and previous filter state. We also know the evictions from local caches. Using them

Table 1. Filter states

valid bit	owner identifier	information	filter state
1	xxx	xxx is the only processor that can have a local copy of the block in its local data cache	valid owner
0	000	there are no copies of the block	no copies
0	001	block cached only in the local data caches of processors identified as 0xx	data block (subgroup0)
0	010	block cached only in the local data caches of processors identified as 1xx	data block (subgroup1)
0	011	data block	data block (all)
0	100	unused	
0	101	block cached only in the local instruction caches of processors identified as 0xx	instruction block (subgroup0)
0	110	block cached only in the local instruction caches of processors identified as 1xx	instruction block (subgroup1)
0	111	instruction block	instruction block (all)

the filter information will be precise, but extra directory accesses and costly filter updates will be required. Consequently, we decide to not keep filter information precise all the time, that is, to only know a superset of the copies in the local caches.

3.3 Filter States

The modeled CMP has 8 cores, so a filter entry has 3 owner identifier bits and a valid bit. Table 1 shows how these bits are used to encode different filter states that will reduce directory lookups or directory lookups associativity.

The directory is split in data and instruction directories. Most directory lookups are performed on both directories (only lookups performed to keep instruction/data exclusivity are performed in only one directory (see Section 4.1)). As long as the filter identifies the type of the block (*data or instruction block*), directory lookups are limited to just one directory. Moreover, if the owner (*valid owner*) or the owner's group (*subgroupX*) is identified, the lookup associativity is reduced since only the entries of the owner or its group have to be looked up. Finally, directory lookups are completely avoided if the filter indicates that there are no copies of the block in any local cache (*no copies*).

The filter state *valid owner* is set on three cases: a) local data cache misses that also misses in the shared cache, b) local data cache misses to a block in the shared cache without copies in the local caches (*no copies*), and c) store to a block in the shared cache which may have copies in the local data cache of the processor performing the store (*valid owner* equal to the processor performing the store, *data block (all)*, or *data block (subgroupX)* where 'X' is the subgroup which the processor performing the store belongs to).

The filter state is set to *no copies* in two cases: a) store that misses in the shared cache, and b) store to a cache block in the shared cache which is not present in the local data cache of the processor performing the store (*no copies*, *valid owner* when the owner is different from the processor performing the store, *instruction block*, or *data block (subgroupX)* where 'X' is not the subgroup the processor performing the store belongs to).

Both local instruction and data cache misses modify the filter state to add the processor performing them as one of the processors that can have a copy of the accessed block in its local caches. When a local data cache miss accesses a block whose filter state is *instruction block*, the filter state is not modified.

3.4 Filter Overhead

The filter proposed requires ($1 + \log_2 P$) bits per shared cache line, being P the number of cores in the CMP. For the CMP described in Section 4.1, as it has 8 cores, four extra bits per shared cache line are required. For each 512KB, 64B block-size L2 bank, the filter implementation requires 4KB. This represents 12% of the tag array size (including the state bits in the tag array) and 0.7% of the total bank size (tag array + data array).

4 Evaluation

4.1 Chip Multiprocessor Model

Figure 2 shows the CMP configuration we assume in this work. It is a CMP with 8 in-order multithreaded cores with 4 threads each and a memory hierarchy similar to the one in Niagara 2 [18]. The first cache level is local to each core, and is composed of an instruction cache (L1 I) and a write-through no-write-allocate data cache (L1 D). Each core also has a store buffer (SB) with several entries per thread that contain all outstanding stores. The second-level cache (L2), which is inclusive, is shared among all the cores. It is divided into different banks interleaved by second-level cache blocks. A crossbar communicates the two cache levels. A write-invalidate directory-based protocol is used to maintain the cache coherence among the local caches. The directory is distributed among the second-level cache banks, keeping close to each bank the information about the blocks associated with it. Table 2 collects the specific parameters we chose for the memory hierarchy.

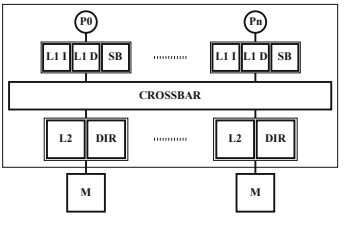


Fig. 2. CMP model

Table 2. Memory hierarchy parameters

L1 D size	8KB	L2 size	4MB
L1 D assoc.	4-way	L2 no. banks	8
L1 D block	16B	L2 assoc.	16-way
L1 I size	16KB	L2 block	64B
L1 I assoc.	8-way	L2 latency	7 cycles
L1 I block	32B	L2 MSHR	8
store buffer	8 entries per thread	Crossbar arb.	3 cycles
		Crossbar lat.	3 cycles
Phys. address	40 bits	Memory lat.	117 cycles

We assume a directory similar to that of Niagara 2 [30], which consists of a copy of the local cache tags. The directory is split into instruction and data directories, replicating the organization of the local caches. The directory gives

Table 3. SPLASH2 benchmarks.

benchmark	dataset	instr (10^9)	cycles (10^9)
barnes	64K particles	4.97	0.62
fmm	64K particles	9.57	1.20
ocean	1026x1026	5.99	0.91
radiosity	-largeroom, -ae 5000 -en 0.050 -bf 0.1	7.45	0.94
raytrace	balls4	5.77	0.79
volrend	head	0.63	0.08
water-ns	2192 particles	13.79	1.72
water-spatial	4096 particles	4.02	0.50

Table 4. Specweb2005 workloads.

workload	simultaneous sessions	web trans.	instr (10^9)	simulation runs
Banking	200	100	15.52	30
Ecommerce	1000	1200	8.07	15
Support	1400	2200	8.07	10

the way or ways of the local caches where the copies of the subblocks are located. Thus, an invalidation message consists of the local cache set and way to invalidate. Stores update local caches when the ack message is received. The ack message includes the way where the copy of the block is located in order to avoid the local cache lookup.

Like in Niagara 2 [30], instruction/data block exclusivity is maintained in the local caches, that is, the same block can not be at once in both instruction and data caches (across all cores). The directory is responsible for ensuring instruction/data exclusivity. The shared cache block size is larger than the block size of the local caches. Thus, copies of different subblocks from the same shared cache block can reside in local caches of different types (instruction/data). The proposed filter has only one entry for every shared cache line. As a result, instruction/data block exclusivity has to be maintained at a shared cache block size granularity to guarantee the correct filter operation.

4.2 Methodology

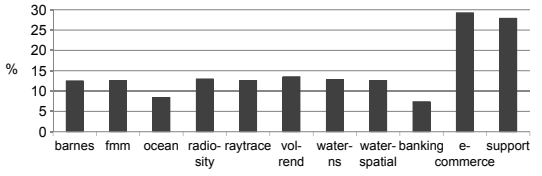
We use a Simics-based simulator. Simics [21] is a full-system multiprocessor simulator capable of running unmodified commercial OSs and applications. We configured Simics to model a SPARC V9 target system with a Total Store Order (TSO) consistency memory model running Solaris 9.

We use the applications from the SPLASH2 benchmark suite [27] and, as non-numerical applications, the three workloads from Specweb2005: Banking, Ecommerce, and Support [15]. In order to adapt the SPLASH2 workloads to our simulated scenario, we scaled the input dataset up as proposed by Monchiero et al. [22] (Table 3). Due to simulation time restrictions, we cannot simulate as many Simics processors in Specweb2005 as in SPLASH2. As a result, Specweb2005 applications are executed in a CMP with 8 non-multithreaded processors.

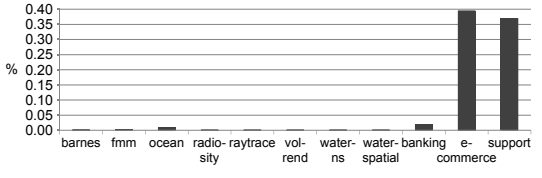
For the three workloads of Specweb2005, we use Apache 2.0.63 web server. Web servers present high time and space variability [3] (Table 4). Conclusions are based on the mean of the simulations and on statistical techniques used by Alameldeen et. al. [3]. To determine the number of web transactions in each workload, we warm the caches for 0.75 billion cycles and then we measure the number of web transactions for 2.25 billion cycles. In the rest of the paper, for Specweb2005 results we show the mean of all simulation runs.

4.3 Filter Coverage

Figure 3 shows the percentage of comparisons performed by the directory lookups in the CMP with the proposed filter with respect to the comparisons performed without filtering. Table 5 shows the number of comparisons performed in the system without filtering.



(a) data directory



(b) instr directory

Fig. 3. Percentage of directory comparisons performed by directory lookups when filtering

Table 5. Billions of directory comparisons without filtering

benchmark	data dir	instr dir
barnes	24.87	57.34
fmm	26.54	61.56
ocean	21.73	53.38
radiosity	28.61	70.45
raytrace	6.23	47.29
volrend	1.28	3.23
water-ns	26.70	60.41
water-spatial	10.82	25.03
banking	45.21	90.09
ecommerce	24.84	43.87
support	28.33	48.59

Figure 3 shows that the number of comparisons performed by directory lookups is reduced, on average, by 93%. The reduction is more important in the instruction directory: more than 99% in the instruction directory for all the benchmarks vs. 88% for SPLASH2 and 81% for Specweb2005 in the data directory.

A data directory lookup is necessary to determine if a block in the local data cache has to be updated and the way in which the copy of the block is located. For this reason, there are several comparisons in the data directory that can not be eliminated.

For Ecommerce and Support, the reduction in data directory comparisons is lower than in the other benchmarks. More than 10% of their stores access cache blocks that are in *data block* filter state, while in the other benchmarks this number is below 1%. That means that the amount of shared data is also larger than in the rest of benchmarks. In this situation, the number of needless directory lookups is smaller, and so the number of comparisons to avoid.

4.4 Performance

The proposed filter modifies the coherence protocol forcing the instruction/data exclusivity at a 64B granularity. Thus, we need to check that the performance

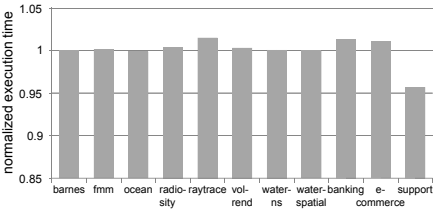


Fig. 4. Normalized execution time

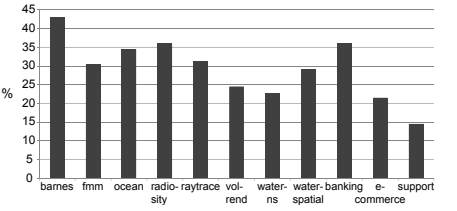


Fig. 5. Percentage of power reduction in the directory

remains unchanged. Figure 4 shows the normalized execution time of the CMP with the filter proposed with respect to the baseline CMP. In SPLASH2 all benchmarks show a performance loss below 0.5%, except raytrace, which has a performance loss of 1.5% due to the increase in the local data cache miss rate. In Specweb2005 we can differentiate two groups: for banking and ecommerce, the mean execution time shows an increase of 1.4% and 1.1%, respectively, in support, the mean shows an execution time decrease of 4.3%. In both groups the confidence interval shows that the execution time is not statistically different.

4.5 Power Consumption

CACTI 6.5 [25] is used to estimate dynamic energy and leakage power for the shared cache tag array and the proposed filter. We modified CACTI to model CAM structures, so that the directory energy consumption, both dynamic and static, can be estimated. All structures were modeled using a 65nm technology with a target frequency of 1.2GHz.

The average dynamic power consumption is computed based on activity statistics of the shared cache, the filter, and the data and instruction directories collected during benchmark execution. The average dynamic power consumption of the directory is 1.5 times the average dynamic power consumed by the tags of the shared cache. However, the leakage power of the tags is 2.2 times the directories leakage since these structures are smaller than the tags of the local caches.

Figure 5 shows the percentage of power reduction in the directory using the filter proposed. It takes into account power reduction in the directory as well as additional consumption due to the filter structure embedded into the shared cache tags. The directory power consumption includes the dynamic power and the leakage power in both data and instruction directories. The proposed filter is placed together with the shared cache tags, so tags and filter state bits are read together in every access to the shared cache. This means that both the energy consumed by the shared cache tag array on any operation and its leakage power increase. These increases affect the energy reduction in the directory. The energy to update the filter state also decreases the dynamic energy reduction in the directory.

On average, the directory power is reduced by 30.8% for SPLASH2 and by 22.42% for Specweb2005. The difference between SPLASH2 and Specweb2005 is due to simulating Specweb2005 in single-thread processors. Ecommerce and Support show a shared cache miss rate higher than the rest of benchmarks. As there is only 1 thread per core, every shared cache miss stalls a core and, as a result, the number of accesses to the shared cache and directory lookups are smaller than in the rest of benchmarks. Thus, the dynamic power reduction in the directory is smaller, but the increase in the leakage power due to the filter remains the same. To prove this argument we simulate SPLASH2 suite in a system with single-threaded cores and we observe a similar reduction in saved power.

Other cache configurations and new generation technologies. The size of the proposed filter is directly proportional to the number of shared cache lines. Moreover, the energy consumed by the directory depends on the number of directory lookups performed which is determined by the shared cache accesses. If the size of the local caches is increased, the shared cache accesses are modified. Thus, we decide to analyze the reduction of power for different cache configurations. We simulate a CMP in which the sizes of the shared cache and the local caches are doubled. The percentage of power reduction is smaller than in the baseline system due to the increase in the power consumption of the proposed filter (bigger shared cache) and the decrease in the number of directory lookups performed (bigger local caches). On average, in the worst case, the percentage of power reduction is 24% for SPLASH2 and 10% for Specweb2005.

Finally, we analyze how the percentage of power reduction is affected for new generation technologies. We model all structures using a 22nm technology with a target frequency of 2.75GHz. On average, the percentage of power reduction is 19.5% for SPLASH2 and 10.5% for Specweb2005.

5 Related Work

This section gathers together several techniques to filter out coherence actions, e.g., local cache lookups or broadcast messages. The filter is either placed together with the local caches or distributed in the on-chip network.

When the filter is placed together with the local cache in snoopy-based protocols in bus-based systems, we can distinguish several ways to reduce the power consumed by coherence actions.

Several proposals try to filter snoop-induced lookups. JETTY [24] adds small structures to SMPs that are accessed before doing the tag cache lookup and Ekman et al. [11] evaluate this proposal on CMPs. Salapura et al. [26] propose a structure that keeps a superset of cached blocks. The Page Sharing Table (PST), proposed by Ekman et al. [12], uses vectors that identify sharing at the page level with precise information.

There is a group of proposals that try to not only filter snoop-induced lookups but to reduce broadcast messages. RegionScout [23] implements several structures per node in a similar way to JETTY [24], but these structures keep global

system information about regions, which are continuous sections of memory. Cantin et al. [7] present an idea similar to RegionScout, but the information kept in the structures is precise and the structures are bigger.

Focusing on logical ring interconnections, Strauss et al. [29] propose using an adaptive filter in each node to skip the snoop-induced lookup when possible and to decide if the lookup should be performed in parallel to sending the request to the next node (to reduce snoop latency) or in sequence (to reduce the number of messages).

Compiler time knowledge can also be used to reduce coherence actions. Information about the behaviour of a program helps determining whether a region of memory is shared or private and limit snoop-induced lookups to shared blocks [10,5].

There are proposals that distribute the filter over the on-chip network for snoopy-based and directory-based protocols. Agarwal et. al. [2] propose adding a region tracker structure in each output port of the routers. This structure indicates which regions are not allocated in the local caches of the processors reached from a specific port, so useless broadcast messages are not sent. Jerger [17], in a coarse-grain like directory-based protocol, adds counting bloom filters to each output port of the routers in order to not broadcast useless invalidation messages addressed to the local caches reached from a specific port.

Unlike previous proposals, the goal of the proposed filter is to reduce energy consumption in a coherence directory implemented as a duplicate tag directory in a CMP with write-through caches. This CMP is similar to Niagara 2 that has a limited number of cores. However, if the number of cores in the system increases significantly (many-cores), cores could be organized in groups or clusters. Every cluster might work like a small CMP with write-through local caches since the coherence protocol inside the cluster is greatly simplified. The shared cache in a cluster would be private for that cluster. It could use a write-back policy to update the last-level cache shared among all the clusters or main memory. Our filtering mechanism would be used inside each cluster. However, such systems are out of the scope of this paper.

6 Conclusions

We have observed that in CMPs with write-through caches, a big fraction of directory lookups is due to stores performed over data that are private to the processor executing the store instruction. In such a situation, a directory lookup is performed but no invalidations are necessary. This needless directory lookup wastes energy. We propose to use a filter before accessing the directory. The filter is able to identify private stores and reduce the number of directory lookups performed or the number of directory entries looked up in a directory lookup.

The proposed filter has an entry for each line in the shared cache. For every shared cache access, a filter entry is read together with the state bits of the block accessed. Every filter entry keeps either the owner of the corresponding block or some useful information to limit the associativity of a directory lookup

performed over the corresponding block. Using this information the number of comparisons in the directory is greatly reduced.

The proposed filter area is 12% the tag array area and 0.7% the total shared cache area, and filtering is performed on every access to the shared cache. Our results show that, on average, the proposed filter reduces the number of comparisons performed by directory lookups by 95%, and reduces the directory power by 28.2% for all the benchmarks.

References

[1] Agarwal, A., Simoni, R., Hennessy, J., Horowitz, M.: An Evaluation of Directory Schemes for Cache Coherence. In: ISCA-15, pp. 280–289 (1988)
[2] Agarwal, N., Peh, L.-S., Jha, N.: In-Network Coherence Filtering: Snoopy coherence without broadcasts, pp. 232–243 (2009)
[3] Alameldeen, A.R., Wood, D.A.: Variability in Architectural Simulations of Multi-Threaded Workloads. In: HPCA-9, p. 7 (2003)
[4] AMD. AMD Multi-Core Technology, http://multicore.amd.com
[5] Ballapuram, C.S., Sharif, A., Lee, H.-H.S.: Exploiting Access Semantics and Program Behavior to Reduce Snoop Power in Chip Multiprocessors. In: ASPLOS XIII, pp. 60–69 (2008)
[6] Barroso, L.A., et al.: Piranha: a Scalable Architecture Based on Single-Chip Multiprocessing. In: ISCA-27, pp. 282–293 (2000)
[7] Cantin, J.F., Lipasti, M.H., Smith, J.E.: Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. In: ISCA-32, pp. 246–257 (June 2005)
[8] Censier, L.M., Feautrier, P.: A New Solution to Coherence Problems in Multicache Systems. IEEE Transactions on Computers C-27(12), 1112–1118 (1978)
[9] Charlesworth, A., Aneshansley, N., Haakmeester, M., Drogichen, D., Gilbert, G., Williams, R., Phelps, A.: The Starfire SMP Interconnect, p. 37 (1997)
[10] Dash, A., Petrov, P.: Energy-Efficient Cache Coherence for Embedded Multi-Processor Systems through Application-Driven Snoop Filtering. In: DSD 2006, pp. 79–82 (2006)
[11] Ekman, M., Dahlgren, F., Stenström, P.: Evaluation of Snoop-Energy Reduction Techniques for Chip-Multiprocessors. In: Workshop on Duplicating, Deconstructing and Debunking, in conjunction with ISCA (May 2002)
[12] Ekman, M., Stenström, P., Dahlgren, F.: TLB and Snoop Energy-Reduction Using Virtual Caches in Low-Power Chip-Multiprocessors. In: ISLPED 2002, pp. 243–246 (2002)
[13] Fujitsu. Fujitsu SPARC64 VII Processor (June 2008)
[14] Gupta, A., dietrich Weber, W., Mowry, T.: Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In: ICPP 1990, pp. 312–321 (1990)
[15] http://www.spec.org/web2005/
[16] Intel. Leading Virtualization Performance and Energy Efficiency in a Multiprocessor Server
[17] Jerger, N.: SigNet: Network-on-chip filtering for coarse vector directories. pp. 1378–1383 (2010)
[18] Johnson, T., Nawathe, U.: An 8-core, 64-thread, 64-bit Power Efficient SPARC SOC (niagara2). In: ISPD 2007, p. 2 (2007)

[19] Laudon, J., Lenoski, D.: The SGI Origin: A ccnuma Highly Scalable Server, pp. 241–251 (1997)
[20] Le, H.Q., et al.: IBM POWER6 microarchitecture. IBM J. Res. Dev. 51(6), 639–662 (2007)
[21] Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A Full System Simulation Platform. Computer 35(2), 50–58 (2002)
[22] Monchiero, M., Ahn, J.H., Falcón, A., Ortega, D., Faraboschi, P.: How to Simulate 1000 Cores. SIGARCH Comput. Archit. News 37(2), 10–19 (2009)
[23] Moshovos, A.: RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In: ISCA-32, pp. 234–245 (June 2005)
[24] Moshovos, A., Memik, G., Falsafi, B., Choudhary, A.: JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers. In: HPCA-7, 2001, pp. 85–96 (2001)
[25] Muralimanohar, N., Balasubramonian, R.: CACTI 6.0: A Tool to Model Large Caches (2009)
[26] Salapura, V., Blumrich, M., Gara, A.: Improving the Accuracy of Snoop Filtering Using Stream Registers. In: MEDEA 2007, pp. 25–32 (2007)
[27] Singh, J.P., Gupta, A., Ohara, M., Torrie, E., Woo, S.C.: The SPLASH-2 Programs: Characterization and Methodological Considerations. In: ISCA-22, p. 24 (1995)
[28] Steinman, M.B., Harris, G.J., Kocev, A., Lamere, V.C., Pannell, R.D.: The AlphaServer 4100 Cached Processor Module Architecture and Design (1996)
[29] Strauss, K., Shen, X., Torrellas, J.: Flexible Snooping: Adaptive Forwarding and Filtering of Snoops in Embedded-Ring Multiprocessors. SIGARCH Comput. Archit. News 34(2), 327–338 (2006)
[30] Sun Microsystems, Inc. OpenSPARC T2 System-On-Chip (SoC) Microarchitecture Specification vol. 1 (May 2008)
[31] Tang, C.K.: Cache System Design in the Tightly Coupled Multiprocessor System. In: AFIPS 1976, pp. 749–753 (1976)

FELI: HW/SW Support for On-Chip Distributed Shared Memory in Multicores

Carlos Villavieja[1,2], Yoav Etsion[2], Alex Ramirez[1,2], and Nacho Navarro[1,2]

[1] Universitat Politecnica de Catalunya, Barcelona, Spain
[2] Barcelona Supercomputing Center, Barcelona, Spain
{first.last}@bsc.es

Abstract. Modern Chip Multiprocessors (CMPs) composed of accelerators and on-chip scratchpad memories are currently emerging as power-efficient architectures. However, these architectures are hard to program because they require efficient data allocation. In addition, when running legacy applications on these architectures, unless their code is adapted to utilize the distributed memory architecture, applications cannot benefit from their high computational power.

In this paper, we propose FELI, a set of operating system mechanisms that allocate application data to on-chip memories without any user intervention. FELI[1], automatically maps data to on-chip memories using the address translation mechanism. It relies on a set of TLB counters, and dynamical migration of pages from off-chip memory to on-chip memory. We also introduce virtually tagged L0 caches to alleviate the address translation overhead. Moreover, we make a comparison in performance and power consumption versus a homogeneous cache-based CMP design.

Our evaluation shows a 50% average improvement in power consumption with the scratchpad-based CMP compared to a cache-based CMP. And a 10% in average memory access time even accounting for the cost of page migrations and TLB invalidations. FELI can automatically allocate on-chip memory to an average of 90% of the applications working set.

Keywords: Chip MultiProcessors, Scratchpad on-chip memories, page migration.

1 Introduction

In recent years, the power wall has led to the emergence of Chip Multiprocessors (CMPs). However, the memory wall still remains a problem on CMPs, as more cores have to be fed with data. Cache-based architectures have mainly been used to alleviate high memory latencies. However, cache memories have unpredictable access time and do not scale well as the number of cores increases in scenarios with a high degree of data sharing. Cache coherency protocols become an issue.

[1] FELI is an acronym for *Fitting Everything Local In*, which is the philosophy of the proposed migration mechanism.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 282–294, 2011.

Scratchpad-based architectures are becoming a promising alternative [2]. They are software managed, have a predictable access time, and they are not as power hungry as cache memories. Digital Signal Processors (DSPs), the Cell/BE or GPU computing platforms already integrate on-chip scratchpad memories like Local Storage (LS) or shared memory areas. Applications use scratchpad memories to fetch specific application data through direct memory access (DMA) and then execute a kernel with all necessary data already present on-chip. After execution, results are usually transferred back to off-chip memory. This allows applications to benefit from lower latencies and predictable access time. Moreover, this memory architecture allows to overlap DMA data transfers with computation which can completely hide memories latencies. In addition, scratchpad memories do not require any coherence protocol and, thus are highly scalable.

Even the emergence of newer parallel programming models, there is still a large number of parallel legacy applications that do not benefit from these architectures. Without code modifications, all application data is located off-chip and all accesses pay the full main memory access latency. These applications were neither designed nor programmed for this kind of physically distributed memory. Therefore, in order to run these applications on scratchpad-based architectures efficiently, novel solutions need to be applied. Memory management in runtime libraries and/or at the Operating System (OS) level are crucial in order to transparently take advantage of these software-managed on-chip memories.

In this paper, we introduce the concept of Local Partition (LP) as a scratchpad memory attached to each core memory management unit(MMU). Under our shared memory physically distributed design, all cores can reference and access any LP inside the CMP and off-chip. The system builds a single global(physical) address space that includes all on-chip and off-chip memories. We show that allocating data through local and remote LP on a chip is beneficial for application performance. In this direction, we introduce FELI, a set of OS mechanisms to transparently perform effective memory allocation and remapping. The OS automatically moves data structures to those scratchpad memories that provide the best memory access times. We describe and evaluate a simple page migration mechanism based on page access counters stored in all entries of the cores Translation Lookaside Buffers (TLBs). Through the memory translation mechanisms data can be mapped to any on-chip/off-chip location. This allows the execution of legacy code in these scratchpad-based CMP architectures with a reasonable average memory access latency. All memory operations require the TLB for address translation and at the same time data mappings to any physical location. To alleviate the power consumption and overhead of the MMU that is introduced by this mechanism, we also introduce a virtually tagged level 0 data cache to avoid most of address translation requests. Finally, to evaluate our proposal, we have also performed a comparison with a traditional cache-based CMP architecture.

Contributions: To the best of our knowledge, this is the first paper that proposes a system that allows to run legacy applications unmodified on a CMP

architecture with scratchpad memories using a single global address space (on-chip Distributed Shared Memory). We make the following contributions:

- We show that legacy (shared-memory) applications can run unmodified in an on-chip Distributed Shared Memory (DSM) architecture without performance penalties.
- We show that a simple page migration policy can move most of the application working set in on-chip memory.
- We introduce a virtually tagged L0 exploring its size and invalidation mechanisms. This avoids TLB lookup and address translation for 70-80% of all memory operations.
- We discuss the characteristics of the OS mechanisms and the hardware (TLB) modifications proposed. These mechanisms enable the implementation of high efficient memory allocation policies in FELI.
- We evaluate FELI on a scratchpad-based CMP and compare the performance and power consumption with an homogeneous cache-based CMP.

2 On-Chip Distributed Shared Memory

2.1 Chip MultiProcessor Architectures

Figures 1 and 2 illustrate the two baseline CMP architectures used in this paper. Both are composed of multiple cores and several on-chip memories connected through an interconnection network to off-chip main memory. Figure 1 shows a cache-based architecture with a private L1 and a partitioned shared L2 cache per core. Figure 2 shows a scratchpad-based architecture with an addressable on-chip memory per core. In this architecture, we propose using a single global physical address space to map together all on-chip and off-chip memories. The on-chip scratchpad memories attached to each core from now on will be called Local Partitions (LPs). Since there is a single global physical address space, any core can access any LP in the CMP with a single load/store instruction or through DMA operations. All cores incorporate a Memory Management Unit (MMU) that includes a Translation Lookaside Buffer (TLB) for address translation, a Network interface Controller (NiC) for packet routing, and a programmable DMA engine to transfer data between local and remote memory (either the LP of another core or the off-chip memory). Indeed, DMA is a key element that allows the application or the Operating System(OS) to program data transfers with the benefit of overlapping program computation and communication. No coherence protocol is implemented for the LPs, so data replication or data migration need to be explicitly performed by software, through DMA or remote read/write operations.

2.2 Single Global Address Space On-Chip

Using a single global address space is essential to transparently manage data transfers. It makes the architecture fully coherent and only one copy of each

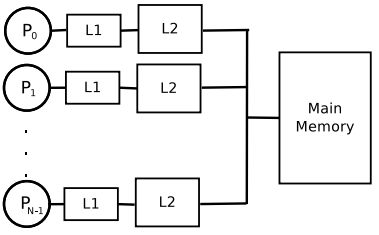


Fig. 1. Cache-based CMP architecture

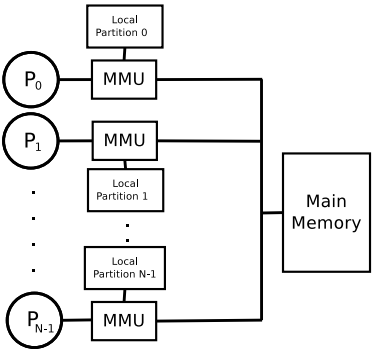


Fig. 2. Scratchpad-based CMP architecture

data is stored. Using address translation, the OS can map any virtual address to any physical location(on-chip or off-chip). In Figure 3, both address spaces are shown. Virtual address space in the top part is split in several parts or areas. Each area named after the letters A to E, represent a virtually contiguous set of pages. The physical address space in the bottom part is split in the physical memory locations. The virtual address areas are mapped as follows: Area A is fully mapped to the off-chip Main Memory. B and C areas are both mapped to the same LP 0. D and E areas are mapped to other LPs. Since we rely on the address translation mechanism to map virtual addresses to physical, the minimal area size is the page size.

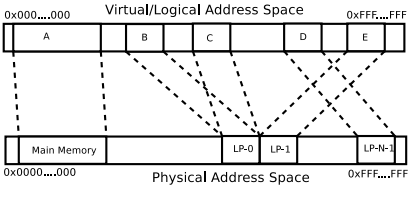


Fig. 3. Example of virtual to physical mapping

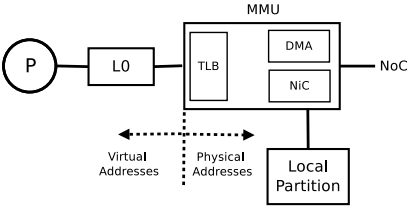


Fig. 4. L0-MMU-core architecture

2.3 FELI - Operating System Support for Locality Management

We have built a set of OS mechanisms to run legacy applications transparently and efficiently on top of the hardware described in Section 2.1. These mechanisms are called FELI.

In a scratchpad-based architecture, an application starts running with all its data loaded at the off-chip memory. A core that needs any data requires an access to the off-chip memory paying the highest access latency. No access to the on-chip memory or LP is done. FELI sets a *page migration* mechanism to alleviate this problem.

On a first data-page access, the requester core's TLB misses for the address translation, and a page-table request is sent to main memory. Next, the page-table request returns the address translation for the page, and the core TLB stores the translation and an access counter. Using the physical address a request is sent to main memory to obtain the data. From this moment on, each subsequent access to that page increases the counter in the TLB entry. The counter is required to monitor when the page is being actively used.

Once the access counter reaches a certain threshold (MMU threshold register), a page migration process is triggered. The threshold value is based on different costs (migration penalty and off-chip main memory accesses) commented in Section 2.5. The page migration allows to have a *zero copy* policy of all data. Each data has a unique location. Using this mechanism, FELI moves data from off-chip memory to the on-chip LP and vice-versa. This way, FELI tries to keep the most used data by each core in its LP. We do not consider page migrations between Local Partitions.

The migration mechanism must ensure memory coherence, therefore the following transaction needs to be performed in order:

1. **Invalidate the TLB entry:** Once the threshold is reached, and we start the page migration, the TLB page entry of the requester core is invalidated.
2. **Invalidate page-table entry:** In order to prevent other cores from requesting the page translation, the page-table entry is invalidated and locked. Translation requests from other cores for that page are stalled.
3. **Invalidate all TLBs:** Once the request to the page table (2) is acknowledged, the core holds the token for the page, and sends a broadcast message to invalidate all TLBs within that entry. This is commonly known as a TLB shootdown process. It includes waiting for all cores acknowledgments.
4. **Victim selection:** We choose a victim page from the LP, and perform a TLB shootdown for the victim page.
5. **Page Transfer:** Once both pages are selected and invalidated, we program the DMA controller to transfer both pages. The victim page from LP to main memory, and the new page from main memory to LP.
6. **DMA End:** Once we receive the DMA finalization signal, we update the page translation entry in the page table and unlock the entry. The next access to the page will miss in the TLB and it will be updated through a page-table request.

2.4 L0 Cache

Using LPs and FELI, we reduce the average memory access time by allocating most of the application's working set on-chip. However, all memory operations pay for the overheads in time and power usage [5] of address translation and request routing at the MMU. Address translation depending on the TLB configuration may take between 2-4 cycles [17]. To minimize the MMU overhead, we have introduced a small virtually tagged cache (L0) attached to each core. Figure 4 shows the L0 cache architecture connected to a core MMU. This L0 cache

only stores data located at the core LP. Therefore, we skip cache coherency with other cores. On a load miss in the L0, the data is requested to the MMU. The data response to the load missed in the L0 is only cached in the L0 if its address is hold at the core's LP. On a store operation, if it hits in the L0, the store is write-through to the LP.

Coherency at L0-LP level is only required for all stores to a LP from a remote core, and for all page migrations. On any of these two operations, the affected address is physical, however the L0 cache is virtually tagged. Since we do not have a reverse translation mechanism because of its high cost, invalidation operations must be performed. For this purpose, we have studied three different invalidation policies:

1. Total L0 invalidation: On an invalidation operation, the entire L0 cache is invalidated.
2. Perfect L0 invalidation: On an invalidation operation, using a reverse mapping we invalidate only the cache lines affected. This technique is easily implemented in our simulator, however, we do not consider it for a viable hardware option due to its real cost.
3. Bit filter L0 invalidation: The MMU holds a bitmap register to monitor all LP pages with some data cached in the L0. Each bit represents a LP page. On a local load request to an address stored in the LP, the bit field for that page is marked. On an invalidation request, we only invalidate pages marked in the bitmap. On a page migration the register is reset. This mechanism minimizes the performance impact of invalidations.

The MMU/TLB monitors requests to the LP to skip synonyms being allocated in the L0 cache. This way L0 can not hold two different virtual addresses pointing to the same data and therefore avoid conflicts. Only data from one page of the possible synonyms can be allocated at the L0. This leaves our L0 cache free of synonym conflicts.

2.5 Discussion on DSM Architecture Parameters

In all system architectures where page migration or page replacement occur, it is necessary to use a page replacement algorithm. In FELI, we have studied three well-known page replacement algorithms: *Random*, *FIFO* and *LRU*. As previously studied [19], the victim selection algorithm becomes irrelevant on configurations with enough space for the application working set. Our baseline 256KB LP can easily allocate application's working set on-chip, while it is a common L2 cache size. We use *random* because it requires less hardware complexity. Previous work from Etsion et al. [8] also confirm using *random* selection is efficient.

We have also quantified the overhead of adding a two bit counter to all TLB entries using CACTI [12]. The total area increase of TLB size is negligible (below 0.1%). In addition, the two bit counter does not affect the TLB access latency.

Another parameter to take into account is the page size used for our evaluation. Initially we chose 4K as the usual size for pages. Several experiments show

that the evaluated benchmarks suffer from internal fragmentation with larger page sizes. Moreover, when using larger sizes, we increase the number of accesses to remote LP increasing the overall memory access time. This translates to a performance degradation as a result of having more accesses to remote LP than LP.

We performed an exploration of the page-counter threshold value used to trigger page migration. Our experiments show that it is necessary to have a small threshold value to detect active pages in the working set of the benchmarks evaluated. For the results shown Section 4, we use a threshold value of two. A small value of the threshold ensures that pages in the LP are the most used ones from the benchmark. This parameter might vary if larger memory page sizes are used.

3 Methodology

Workloads: We use ten benchmarks (blackscholes, bodytrack, dedup, ferret, fluidanimate, rtview, streamluster and swaptions) from PARSEC suite [3] to evaluate the hardware and OS policies presented in this paper. We have selected a set of the most representative benchmarks based on the degree of data sharing between all threads. We used simlarge input set for PARSEC. We have used PIN [11] to obtain all applications traces. All benchmarks are evaluated with configurations of 32 threads. Using a methodology proposed by Casas et al. [4], and validated against SimPoint [9], we selected the most representative part of all benchmarks. Each benchmark trace contains 5×10^9 instructions.

Table 1. Table of the simulation configuration parameters

CMP Size	**32 cores**
TLB	**128 - 4-way - 1 cyc**
L0	**4KB - 4-way - 1 cyc**
LP	**256KB - 3 cyc**
Remote LP	**4 + NoC**
L1	**32KB - 4-way - 2 cyc**
L2	**256KB - 8-way - 7 cyc**
Remote L2	**7 + NoC cyc**
Main Memory	**250 cycs**
NoC	**crossBar**
NoC	**25 cyc - 25.6GB/s**

Table 2. Estimated breakdown of all operation costs involved in a page migration

Operation	**Cost (cyc)**
TLB inv	**50**
PageTable inv	**250-400**
CMP TLB inv	**200-300**
Program DMA	**10**
Transfer	**512**
Migration	**2000**

Simulator: For the evaluation of the memory architecture, we have used TaskSim [15], a trace-driven multicore simulator. TaskSim targets the simulation of parallel applications. It allows performing detailed simulations of all components in the CMP architectures shown in Figures 1 and 2. All memory accesses behave the CMP architectures defined in Section 2.1. TaskSim simulates all latencies for all components in the architecture, including memory controllers and

the interconnection network. All memory configurations and latencies in Table 1 have been obtained using CACTI [12]. Table 2 shows an approximated cost for each operation concerning the implemented page migration mechanism. Based on real measurements we have estimated the total cost of a page migration to 2000 cycles.

Metrics: To evaluate the CMP system performance, we measure the average access time per memory location(AMAT). We have also evaluated the power consumption of both CMP architectures. We account for static and dynamic power of all memories on both CMPs architectures.

4 Experimental Evaluation

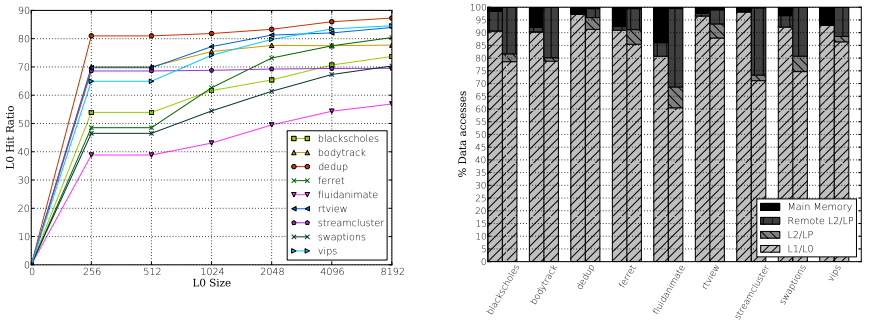
Both of the CMP architectures evaluated are similar in terms of on-chip memory capacity, however, memory access latency and power consumption are quite different as we can observe in this Section.

All the scratchpad-based CMP configurations use a page migration threshold of two. Discussion on choosing this value can be found in Section 2.5. For the L0 coherency, we use the Bit filter invalidation policy described in Section 2.4 because it is high efficient at a reasonable performance cost compared to perfect L0 invalidation which requires a reverse-mapping mechanism, or invalidating the whole L0 cache which reduces the L0 performance dramatically.

First, we analyze the performance of the new L0 cache introduced to overcome the overhead on address translation. Figure 5(a) shows the L0 cache hit ratio as we increase its size. The X-axis represents different L0 cache sizes, and the Y-axis shows the hit ratio. Each line of the figure represents a single benchmark. As it can be observed, at size 4KB, the L0 cache size hit ratio stabilizes around 70%. We use 4KB size and not larger to maintain a small 1 cycle latency. In addition, using a small L0 cache allows to avoid huge performance degradation and power consumption when invalidating the L0 because of page migrations or remote stores.

Figure 5(b) shows the percentage of memory accesses (Y-axis) that hit in the different types of memories in each CMP architecture. For all benchmarks shown in the X-axis, the left bar represents the cache-based and the right bar represents the scratchpad-based results. It can be observed that the L0 hit ratio for almost all benchmarks is above 75%. Although the L0 cache size is much smaller (4KB to 32KB) than the L1 cache, the L0 hit ratio is in average just a 10% lower. Moreover, for most applications the hit ratio for remote LP is higher than L2 cache. Page migration on other cores allows to allocate most of the application working set on-chip minimizing accesses to off-chip main memory. These results demonstrate that combining the L0 cache with the page migration mechanism to LP is a very efficient allocation policy. However, cache line replacement outperforms on high irregular memory access pattern applications (p.e: streamcluster).

In order to compare the two CMP architectures, we have evaluated the average memory access time (AMAT) and the power consumption for the cache-based



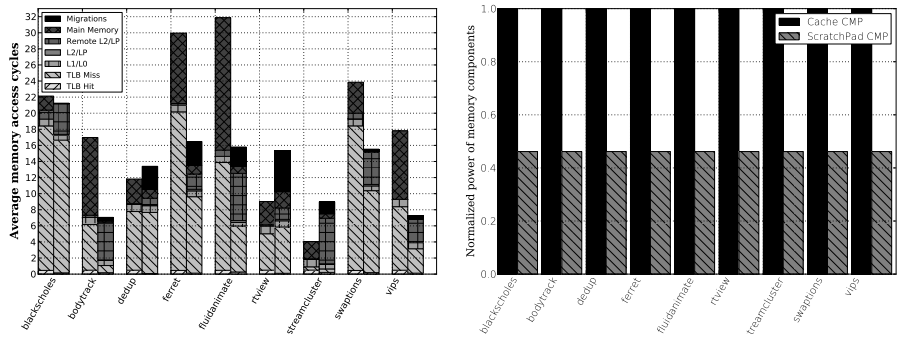
(a) Evaluation of the L0 Hit Ratio/Size for the LP CMP architecture.

(b) Applications Data Layout comparing cache/scratchpad-based CMP.

Fig. 5. In the left graph, we observe an evaluation of the L0 Hit Ratio/Size for the LP CMP architecture. In the right graph, we observe the Applications Data Layout comparing cache/scratchpad-based CMP.

and scratchpad-based CMP. Figure 6(a) shows the AMAT which represents in terms of memory latency, which memories take most of accesses time over the application execution. The X-axis of the graph shows the benchmarks. For all applications two bars represent the cache and the scratchpad-based respectively. The Y-axis shows the AMAT in cycles. The average access time for scratchpad-based is around 13 cycles compared to 14 for the cache-based CMP. It can be observed, in the bottom part of each bar, for scratchpad-based CMP the percentage of time for TLB Hits is negligible because of the high hit ratio in L0 cache. Cache-based CMP benefit by the performance of a higher L1 but with a higher cache latency and the addition of the address translation latency (1 cycle). Moreover, applications with less regular access pattern (bodytrack, fluidanimate, ferret, vips) have a lower L0 cache hit ratio. However, in these applications the remote LP hit ratio is more effective than the L2 cache. This is because using page migration more not yet used data is prefetched than on a L2 miss. For the cache-based case, we observe a high percentage of accesses to the off-chip memory.

Figure 6(b) shows an estimated power consumption comparison of both architectures. We used CACTI [12] to obtain the dynamic and static power consumption of all memory components of both architectures. The X-axis of the graph shows the benchmarks evaluated. As previous graphs, cache and scratchpad-based CMP are shown. The Y-axis shows the power consumption normalized to the architecture that consumes most. For all applications we can observe the power consumption of the scratchpad-based architectures is around 50-55% more efficient for all PARSEC benchmarks. The first reason why scratchpad-based architecture is much more efficient is the high hit ratio of the L0 cache. This allows to skip many lookups in the TLB, and since TLBs are power-hungry caches, the overall power consumption is highly reduced. Moreover, even if the L0 has a similar consumption to the L1 cache, the power consumption of the LP is much



(a) Average Memory Access Time (b) Power Consumption Comparison

Fig. 6. Average Memory Access Time graph and Power Consumption of all benchmarks for cache and scratchpad-based CMP architectures

better than that of an L2 cache. All benchmarks evaluated allocate most of their working set on-chip, and therefore the main source of dynamic power consumption comes from LP in the scratchpad-based CMP, while in the cache-based the main source for the dynamic power consumption are the L1 and TLB caches.

5 Related Work

Distributed shared memory (DSM) machines have largely been used in the past. However, as far as we know, FELI is the first to evaluate DSM in CMPs. DSM related work mainly explores page migration techniques. In [10] several techniques are described deeply dealing with memory management in local/remote memory nodes. Holliday et al. [10] refers to Aging as an effective page migration technique. Corbalan et al. [7] demonstrate than page migration is relevant to achieve high performance ratios in OpenMP parallel applications. However, they also demonstrate that process scheduling and affinity are key when page migration is considered. Since we are not considering OS context switch neither scheduling threads, thread affinity does not affect our results. Introducing thread scheduling would require remote LP migration to transfer remote pages to a new core LP when threads migrate. Dimitrios et al. [14] also uses page reference information to support page migration. They achieve a performance speedup of 264% for OpenMP parallel applications. As Corbalan et al. [7] they combine page migration with scheduling. In contrast to our solution, they consider compiler support to identify hot memory areas. In our initial experiments, we evaluated first touch and profile based technique to feed the OS with profiling information, however, our results using FELI obtained finer granularity over hot memory areas without the use of profiling. FELI is based on the TLB trigger and hence when page access reaches the threshold value, it obtains the application's hot memory areas dynamically at reasonable cost. Similarly Scheurich et al. [16] and Jeun et al. [20] present the pivot mechanism as an alternative

access reference based page migration algorithm. They use access information from each processor as a threshold and as the direction to migrate memory pages to other processors. These previous work has been done for Symmetric Multi-Processors(SMPs). Our work is concentrated in Chip MultiProcessors (CMPs) where latency and a high bandwidth is provided because of on-chip integration. Our work is still not comparable since we do not migrate pages between LPs.

Chaundri [6] presents PageNUCA, a set of OS-assisted locality management policies for large CMP NUCAS. It applies page-migration mechanisms for the last level cache of a CMP achieving a 12% of performance and energy improvement. This solution is closest to our scheme but it is designed for a cache-based architecture and it is only applied for shared L2 cache. We evaluated FELI over all on-chip memory.

Several work in scratchpad-based architectures are focused on compiler support to efficiently allocate data. Avissar et al. [1] and Nguyen at al. [13] present an optimal scheme to allocate data on scratchpad-based architectures in embedded applications. They use a compiler strategy that automatically partition application data among the memory units. Other solutions to improve memory allocation in scratchpad-based architectures can be found based on compile time techniques [18].

6 Conclusions

In this paper, we propose FELI, a set of Hardware/Software mechanisms to run legacy applications in a scratchpad-based CMP architecture. FELI uses a page migration mechanism to dynamically allocate data into on-chip addressable memories or Local Partitions (LPs). This is the first paper where DSM is used for scratch-pad based CMPs. Our mechanisms use address translation to map any virtual address to any physical address. In FELI, the OS manages the LPs as caches of 4KB lines, exploiting the predictable access time and the power efficiency of scratchpad memories. In addition, we have added a virtually tagged L0 cache to improve the overall memory latency. FELI automatically allocates around 90% of application data on-chip. On a performance and power consumption comparison with a cache-based CMP, FELI achieves an average reduction of 10% in memory access time and a reduction of 50% in power consumption.

Acknowledgments. This research is supported by the Consolider program (contract No. TIN2007-60625) from the Ministry of Science and Innovation of Spain, the TERAFLUX project (ICT-FP7-248647), and the European Network of Excellence HIPEAC-2 (ICT-FP7-249013). Y. Etsion is supported by a Juan de la Cierva Fellowship from Ministry of Science and Innovation of Spain. Special thanks to the members of the Heterogeneous Architecture group at BSC and the anonymous reviewers for their comments and suggestions.

References

1. Avissar, O., Barua, R., Stewart, D.: An optimal memory allocation scheme for scratch-pad-based embedded systems. ACM Trans. Embed. Comput. Syst. 1(1), 6–26 (2002)
2. Banakar, R., Steinke, S., Lee, B.S., Balakrishnan, M., Marwedel, P.: Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In: CODES 2002: Proceedings of the Tenth International Symposium on Hardware/Software Codesign, pp. 73–78 (2002)
3. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: Characterization and architectural implications (October 2008)
4. Casas, M., Badia, R.M., Labarta, J.: Automatic structure extraction from MPI applications tracefiles. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 3–12. Springer, Heidelberg (2007)
5. Chang, Y.-J., Lan, M.-F.: Two new techniques integrated for energy-efficient tlb design. IEEE Trans. Very Large Scale Integr. Syst. 15, 13–23 (2007)
6. Chaudhuri, M.: Pagenuca: Selected policies for page-grain locality management in large shared chipmultiprocessor caches. In: In Proceedings of HPCA-15 (2009)
7. Corbalan, J., Martorell, X., Labarta, J.: Evaluation of the memory page migration influence in the system performance: the case of the SGI o2000. In: ICS 2003: Proceedings of the 17th annual international conference on Supercomputing, pp. 121–129 (2003)
8. Etsion, Y., Feitelson, D.G.: L1 cache filtering through random selection of memory references. In: PACT, pp. 235–244 (2007)
9. Hamerly, G., Perelman, E., Calder, B.: How to use simpoint to pick simulation points. SIGMETRICS Perform. Eval. Rev. 31, 25–30 (2004)
10. Holliday, M.A.: Reference history, page size, and migration daemons in local/remote architectures. SIGARCH Comput. Archit. News 17(2), 104–112 (1989)
11. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation, pp. 190–200 (2005)
12. Muralimanohar, N., Balasubramonian, R., Jouppi, N.P.: Architecting efficient interconnects for large caches with cacti 6.0. IEEE Micro 28(1), 69–79 (2008)
13. Nguyen, N., Dominguez, A., Barua, R.: Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. ACM Trans. Embed. Comput. Syst. 8(3), 1–32 (2009)
14. Nikolopoulos, D.S., Papatheodorou, T.S., Polychronopoulos, C.D., Labarta, J., Ayguadé, E.: User-level dynamic page migration for multiprogrammed shared-memory multiprocessors. In: ICPP 2000: Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing, p. 95 (2000)
15. Rico, A., Cabarcas, F., Quesada, A., Pavlovic, M., Vega, A.J., Villavieja, C., Etsion, Y., Ramirez, A.: Scalable simulation of decoupled accelerator architectures. Tech. Rep. UPC-DAC-RR-2010-14, Universitat Politècnica de Catalunya (June 2010)
16. Scheurich, C., Dubois, M.: Dynamic page migration in multiprocessors with distributed global memory. IEEE Transactions on Computers 38, 1154–1163 (1989)

17. Swaminathan, S., Patel, S.B., Dieffenderfer, J., Silberman, J.: Reducing power consumption during tlb lookups in a powerpc" embedded processor. In: Proceedings of the 6th International Symposium on Quality of Electronic Design, ISQED 2005, pp. 54–58 (2005)
18. Udayakumaran, S., Dominguez, A., Barua, R.: Dynamic allocation for scratch-pad memory using compile-time decisions. ACM Trans. Embed. Comput. Syst. 5(2), 472–511 (2006)
19. Villavieja, C., Ramirez, A., Navarro, N.: On-chip distributed shared memory. Tech. Rep. UPC-DAC-RR-CAP-2011, Universitat Politècnica de Catalunya (February 2011)
20. Jeun, W.-C., Kee, Y.-S., Ha, S.: Improving performance of openMP for SMP clusters through overlapped page migrations. In: Mueller, M.S., Chapman, B.M., de Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP 2005 and IWOMP 2006. LNCS, vol. 4315, pp. 242–252. Springer, Heidelberg (2008)

Token3D: Reducing Temperature in 3D Die-Stacked CMPs through Cycle-Level Power Control Mechanisms

Juan M. Cebrián[1], Juan L. Aragón[1], and Stefanos Kaxiras[2]

[1] University of Murcia, Spain
{jcebrian,jlaragon}@ditec.um.es
[2] University of Uppsala, Sweden
kaxiras@it.uu.se

Abstract. Nowadays, chip multiprocessors (CMPs) are the new standard design for a wide range of microprocessors: mobile devices (in the near future almost every smartphone will be governed by a CMP), desktop computers, laptop, servers, GPUs, APUs, etc. This new way of increasing performance by exploiting parallelism has two major drawbacks: off-chip bandwidth and communication latency between cores. 3D die-stacked processors are a recent design trend aimed at overcoming these drawbacks by stacking multiple device layers. However, the increase in packing density also leads to an increase in power density, which translates into thermal problems. Different proposals can be found in the literature to face these thermal problems such as dynamic thermal management (DTM), dynamic voltage and frequency scaling (DVFS), thread migration, etc. In this paper we propose the use of microarchitectural power budget techniques to reduce peak temperature. In particular, we first introduce Token3D, a new power balancing policy that takes into account temperature and layout information to balance the available per core power along other power optimizations for 3D designs. And second, we analyze a wide range of floorplans looking for the optimal temperature configuration. Experimental results show a reduction of the peak temperature of 2-26ºC depending on the selected floorplan.

Keywords: Power budget, power tokens, DVFS, power balancing.

1 Introduction

With the global market dominated by chip multiprocessors and the GHz race over, designers look for ways to increase productivity by increasing the number of available processing cores inside the CMP. The shrinking of transistor's feature size allows the integration of more cores, as the per-core power consumption decreases with each new generation. However, interconnects have not followed the same scaling trend as transistors, becoming a limiting factor in both performance and power consumption. One intuitive solution to reduce wirelength of the interconnection network is to stack structures on top of each other, instead of using a traditional planar distribution.

Introduced by Souri *et al.* in [22], 3D architectures stack together multiple device layers (i.e., cores, memory) with direct vertical interconnects through them

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 295–309, 2011.

(inter-wafer vias or die-to-die vias). A direct consequence of this design is the reduction on the communication delays and power costs between different cores, as well as an increase in packing density that depends on the number of available layers. However, despite of the great benefits of 3D integration, there are several challenges that designers have to face. First, the increase in packing density also leads to an increase in power density that eventually translates into thermal problems. Second, a deeper design space exploration of different floorplan configurations is essential to take advantage of these emerging 3D technologies. Third, chip verification complexity increases with the number of layers.

To face the first challenge there are several proposals that come from the 2D field:

- *Dynamic Voltage and Frequency Scaling* (DVFS) to reduce power consumption, and thus temperature. DVFS-based approaches can be applied either to the whole 3D chip or only to cores that show thermal problems (usually cores away from the edges of the 3D chip) [1][13][20].
- Task/thread migration to move execution threads from internal to external cores whenever possible, or reschedule memory intensive threads to internal cores and CPU intensive threads to external cores [6][24][7].

These mechanisms are usually triggered by a *Dynamic Thermal Management* (DTM) scheme, so whenever a core exceeds a certain temperature, power control or task migration mechanisms take place inside the CMP. However, these mechanisms are not perfect. DVFS is a coarse-grain mechanism usually triggered by the operating system with very long transition times between power modes that leads to a high variability in temperature. On the other hand, task migration, despite the fact that it can be applied at a finer granularity (i.e., faster) than DVFS, has the additional overhead of warming up both the cache and the pipeline of the target core. Moreover, none of these mechanisms affects leakage power consumption. Leakage (or static power) is something that many studies do not take into consideration when dealing with temperature, but it cannot be ignored. For current technologies (32nm and below), even with gate leakage under control by using *high-k* dielectrics, subthreshold leakage has a great impact in the total power consumed by processors. Furthermore, leakage depends on temperature, so it is crucial to add a leakage-temperature loop to update leakage consumption in real time depending on the core/structure's temperature.

Therefore, in order to accurately control peak temperature, which is of special interest in 3D-stacked processors as this integration technology exasperates thermal problems, a much tighter control is necessary to restraint the power consumption of the different cores. Recently, Cebrian *et al.* proposed the use of a hybrid mechanism to match a predefined power budget [4][5]. This mechanism accurately matches a power budget and ensures minimal deviation from the target power and the corresponding temperature, by first using DVFS to lower the average power consumption towards the power budget and then removing power spikes by using microarchitectural mechanisms (e.g., pipeline throttling, confidence estimation on branches, critical path prediction, etc).

In this paper we make three major contributions. First, we analyze the effects of cycle-level accurate power control mechanisms to control peak temperature in 3D die-stacked processors. Based on this analysis we propose *Token3D*, a novel power

balancing mechanism that takes into account temperature and layout information when balancing power among cores and layers. Second, we analyze a wide range of floorplan configurations looking for the optimal temperature configuration, taking into account both dynamic and leakage power (as well as the leakage-temperature loop). And third, we include some specific power control mechanisms for vertical 3D floorplans. Experimental results show a reduction of the peak temperature of 2-26ºC depending on the selected floorplan when including cycle-level power control mechanisms into the 3D die-stacked design. Summarizing, the main contributions of the present work are the following:

- Reducing the peak temperature through power control mechanisms:
  - Implementation and analysis of power balancing mechanisms on 3D die-stacked architectures to minimize hotspots.
  - Introduction of a new policy to balance power among cores, *Token3D*. This policy will use layout and temperature information to distribute the available power among the different cores and layers, giving more work to cool cores and cores close to edges than to internal cores.
- Temperature analysis of the main 3D design choices:
  - Analysis of different 3D floorplan designs using accurate area, power (both static and dynamic) and heatsink information.
  - Analysis of the effects of ROB resizing [18] on temperature for vertical designs.
  - Temperature analysis when using ALUs with different physical properties (energy-efficient *vs.* low latency ALUs) on the same layout.
  - Implementation and analysis of a hybrid floorplan design (vertical+ horizontal).

The rest of this paper is organized as follows. Section 2 provides some background on power-saving techniques for CMPs and 3D die-stacked multicores. Section 3 describes the proposed Token3D approach. Section 4 describes our simulation methodology and shows the main experimental results. Finally, section 5 shows our concluding remarks.

2 Background and Related Work

In this section we will introduce the main power and thermal control mechanisms as well as an overview on 3D die-stacked processors along with the different floorplan design choices.

2.1 Power and Thermal Control in Microprocessors

2.1.1 Dynamic Voltage Frequency Scaling (DVFS)

Dynamic Voltage and Frequency Scaling (DVFS) has been, for the past 20 years, one of the most common mechanisms to reduce power consumption in microprocessors. Introduced in [13], DVFS takes advantage of transistor quadratical dependence on supply voltage and linear dependence on frequency ($P = V_{DD}^2 \times f$) and downscales

both voltage and frequency to save power. However, as the process technology scales down, the margin between V_{DD} (supply voltage) and V_T (threshold voltage) is reduced, decreasing the processor's reliability among other undesirable effects. Furthermore, the transistor's delay (or switching speed) depends on $\delta \approx 1/ (V_{DD} - V_T)^{\alpha}$, with $\alpha > 1$. That means that V_{DD} can lowered as long as the margin between V_{DD} and V_T is kept constant (i.e., V_T must be lowered accordingly). However, the counterpart of reducing V_T is twofold: a) leakage power increases as it exponentially depends on V_T [8]; and b) processor reliability is further reduced.

In the CMP field, Isci *et al.* [1] and later Sartori *et al.* [20] proposed DVFS-based power control mechanisms specifically designed for single-threaded applications. These proposals switch between different DVFS power modes trying to maximize throughput under certain power constraints. Unfortunately, as they rely on the use of performance counters and/or time estimation, these proposals only work properly for multiprogrammed or single-threaded applications, because in parallel applications synchronization points may increase global execution time although local core performance counters show a performance increase (due to spinning).

2.1.2 Dynamic Thermal Management (DTM)

As mentioned before, temperature is the main drawback in 3D die-stacked designs. In 2001, Brooks and Martonosi [3] introduced *Dynamic Thermal Management* (DTM) mechanisms in microprocessors. In that work they explore performance trade-offs between different DTM mechanisms trying to tune up the thermal profile at runtime. Thread migration [21], fetch throttling [6], clock gating or distributed dynamic voltage scaling [9] are techniques that can be used by DTM mechanisms. For the thermal management of 3D die-stacked processors, most of the prior work has addressed design stage optimization, such as thermal-aware floorplanning (as in [10]). In [24], the authors evaluate several policies for task migration and DVS specifically designed for 3D architectures. Something similar is done in [7], where the authors explore a wide range of different floorplan configurations using clock gating, DVFS and task migration to lower peak temperature.

However, both thread migration and DVFS-based approaches exhibit really low accuracy when matching a target power budget, and thus a high deviation from the target temperature. So the designers have two choices, either to increase the power constraint to ensure the target temperature or to use a more accurate way to match the desired (if needed) power budget and temperature. In order to do this we first need a way to measure power accurately, because up to now power was estimated by using performance counters, although the new Intel Sandy Bridge processors include some MSRs (machine specific registers) that can be used to retrieve power monitoring information from different processor structures.

2.1.3 Measuring Power in Real-Time

Power tokens were introduced in 2009 [4] as a way to approximate the power being consumed by the processor at a cycle level. The dynamic power consumed by an instruction can be estimated at commit stage by adding, to the base power consumption of the instruction (i.e., all regular accesses to structures done by that instruction which are known *a priori*), a variable component that depends on the time it spends in the pipeline. A *power token* unit is defined as the joules consumed by one

instruction staying in the instruction window for one cycle. The number of *power tokens* consumed by an instruction will be calculated as the addition of its base *power tokens* plus the number of cycles it spends in the instruction window. As in [4][5], the implementation of the *Power Token* approach is done by means of an 8K-entry history table (Power Token History Table – PTHT), accessed by PC, which stores the power cost (in tokens) of each instruction's last execution. The PTHT is updated with the current number of *power tokens* consumed when an instruction commits. Hence, the overall processor power consumption in a given cycle can be easily estimated based on the instructions that are traversing the pipeline without using performance counters just by accumulating the *power tokens* (provided by the PTHT) of each instruction being fetched.

2.1.4 Hybrid Power Control Approaches

Along with *power tokens*, in [4] we introduced a two-level approach that firstly applies DVFS as a coarse-grain approach to reduce power consumption towards a predefined power budget, and secondly chooses between different microarchitectural techniques to remove the remaining and numerous power spikes. The second-level mechanism depends on how far the processor is over the power budget in order to select the most appropriate microarchitectural technique.

However, previous approaches failed to match the target power budget when considering the execution of parallel workloads in a CMP processor. Very recently, we have proposed *Power Token Balancing* (PTB) [5]. This mechanism will balance the power between the different cores of a 2D CMP to ensure a given power constraint (or budget) with minimal energy and performance degradation. Based in *power token* accounting, this proposal uses a PTB *load-balancer* as a centralized structure that receives and sends power information (measured as *power tokens*) from cores under the power budget to cores over the power budget. Tokens are used as a currency to account for power, so it is important to note that they are neither sent nor received, cores just send the number of spare tokens. PTB will benefit from any power unbalance between cores. Note that task migration mechanisms are orthogonal to PTB and can be applied together for further temperature reductions.

2.2 Building a 3D Die-Stacked Processor

In order to build a 3D die-stacked processor we need to decide two things: how we build and put together the different layers and how we establish the communication between them. There are two main approaches to build the layers: the bottom-up and the top-down approaches. The first approach involves a sequential device process. The frontend processing is repeated on a single wafer to build multiple active layers before creating interconnects among them. The second approach processes each layer separately (wafer-to-wafer), using conventional techniques, and then assembles them using wafer-bonding technology. Once we have built the different layers we need to establish communications between them. There are various vertical interconnect technologies that have been explored, including wire bonded, microbump, contactless (capacitive or inductive), and through-via vertical interconnect. A comparison in terms of vertical density and practical limits can be found in [23][24].

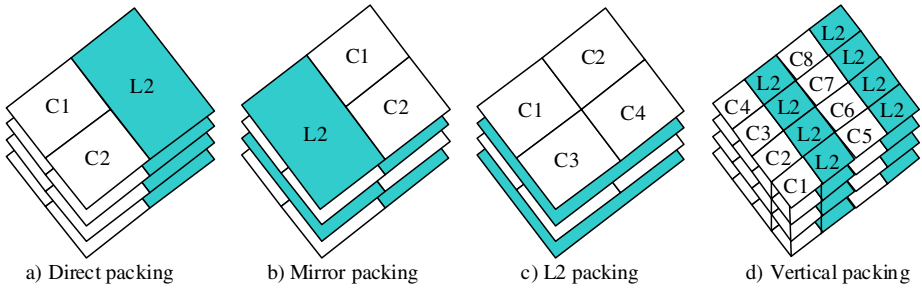


Fig. 1. Core distribution along the layers

2.3 3D Integration Technology

From the previously introduced technologies, wafer-to-wafer bonding appears to be the most promising approach [2] and there are many recent publications that have chosen this type of 3D stacking technology [12][14][16]. Therefore, this is the integration approach we are going to follow in this paper.

Now there are multiple choices on how cores are distributed along the different layers, which are shown in Figure 1. We can clearly indentify two trends; either build the cores vertical or horizontal. Horizontal distributions (a-c) are the most common choices in literature, as they are easier to implement and validate. On the other hand, vertical designs (Figure 1-d), introduced by Puttaswamy *et al.* in [19], offer improved latency and power reduction compared to horizontal designs. However, they supposed an inter-layer communication latency to be in the order of one FO4, and current technologies can do 9-12 FO4 in one cycle. Therefore, in their proposal inter-layer communication could be done in less than one cycle while other papers claim that inter-layer communication takes as long as an off-chip memory access [23]. Furthermore, vertical designs require really accurate layer alignment to match a structure split in different layers, and that is far from the current technology status. However, as a possible future implementation of 3D die-stacked processors we also evaluate these floorplans in this paper, and for comparative purposes, we also assume one FO4 interconnection delay for our evaluation of vertical designs (10µm length wires between layers).

3 Thermal Control in 3D Die-Stacked Processors

3.1 *Token3D*: Balancing Temperature on 3D-Staked Designs

As cited before, *Power Token Balancing* (PTB) is a global balancing mechanism to restrain power consumption up to a preset power budget [5]. One of the main goals of this paper is to analyze the effects of the original PTB approach in 3D die-stacked architectures. We will also propose a novel policy, *Token3D*, aimed at distributing the power among cores and/or dies that are over their local power budget. *Token3D* will give priority to cooler cores, usually located close to the edges/surface of the 3D stack. By prioritizing those cores, *Token3D* balances not only power but also

temperature, as cool cores will work more than the rest of cores, balancing the global CMP temperature. Once a cool core gets to a synchronization point or to a low computation phase (i.e., low IPC due to a misprediction event) it will naturally cool down again, acting like a heatsink to hotter cores located beneath it in the 3D stack.

3.2 *Token3D* Implementation Details

Token3D is a new policy on how PTB splits the available *power tokens*, given by cores under the power budget to the PTB load-balancer, among the cores that are over the power budget (details about *power tokens* and the PTB approach are covered in sections 2.1.3 and 2.1.4). Basically, *Token3D* will create N buckets, where N represents the amount of layers of our 3D die-stacked processor. Then the PTB load-balancer will place the coolest core in bucket *one* and will distribute the rest of the cores between the available buckets in increments of 5% in temperature. So, cores that have a difference between 0 and 5% in temperature with respect to the coolest core will be placed in the same bucket; cores between 5% and 10% will be placed on

Table 1. Simulated CMP configuration

Processor Core	
Process Technology:	32 nanometres
Frequency:	3000 MHz
VDD:	0.9 V
Instruction Window:	128 entries + 64 LsQ
Decode Width:	4 inst/cycle
Issue Width:	4 inst/cycle
Functional Units:	6 Int Alu; 2 Int Mult 4 FP Alu; 4 FP Mult
Pipeline:	14 stages
Branch Predictor:	64KB, 16 bit Gshare
Memory Hierarchy	
Coherence Protocol:	MOESI
Memory Latency:	300 Cycles
L1 I-cache:	64KB, 2-way, 1 cycle lat.
L1 D-cache:	64KB, 2-way, 1 cycle lat.
L2 cache:	2MB/core, 4-way, unified, 12 cycles latency
Network Parameters	
Topology:	2D mesh
Link Latency:	4 cycles
Flit size:	4 bytes
Link Bandwidth:	1 flit / cycle

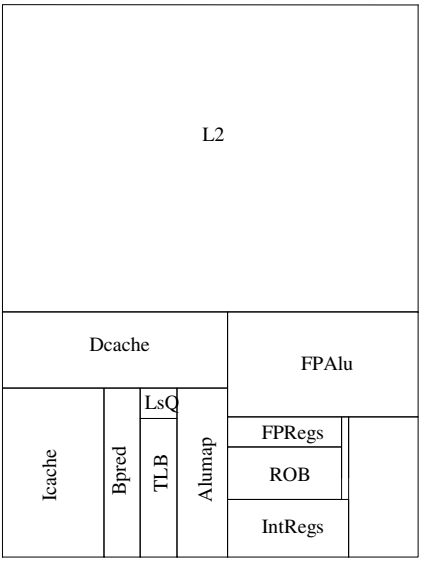


Fig. 2. Core floorplan

Table 2. Evaluated benchmarks and input working sets

	Benchmark	**Size**	**Benchmark**	**Size**
SPLASH-2	Barnes	8192 bodies, 4 time steps	Raytrace	Teapot
	Cholesky	tk16.0	Water-NSQ	512 molecules, 4 time steps
	FFT	256K complex doubles	Water-SP	512 molecules, 4 time steps
	Ocean	258x258 ocean	Tomcatv	256 elements, 5 iterations
	Radix	1M keys, 1024 radix	Unstructured	Mesh.2K, 5 time steps
PARSEC	Blackscholes	simsmall	Swaptions	Simsmall
	Fluidanimate	simsmall	x264	Simsmall

the next bucket; and so on until N. Note that this process does not need to be done at a cycle level, as temperature does not change so quickly. In our case, this process is performed every 100K-cycles. For example, in a four layer 3D-stacked processor, if the coolest core has an average temperature of 70ºC, bucket *one* will hold cores with temperatures between 70ºC and 73.5ºC, bucket *two* will hold cores with temperate between 73.5ºC and 77ºC, bucket *three* 77ºC to 80.5ºC and bucket *four* any core over 80.5ºC.

Once we have identified the cores that are over the power budget (those that did not provide any tokens to the PTB load-balancer), the load balancer will distribute the *power tokens* between the active buckets (i.e., the buckets that have cores over the power budget) in an iterative way, giving extra tokens depending on the bucket the core is in. For a 4-layer design, the bucket that holds the hottest core will have a *x1* multiplier on the number of received tokens, while the coolest bucket will have a *x4* multiplier on the amount of received tokens. For example, if buckets 1, 2 and 3 are active (being 1 the one that holds the coolest cores), all the cores will receive one token, cores in buckets 2 and 1 will receive a second token and, finally, cores in bucket 1 will receive a third token. If there are any *power tokens* left, we repeat the process.

4 Experimental Results

In this section we will evaluate both the original PTB and the novel *Token3D* approaches as mechanisms to control temperature in a 3D die-stacked CMP. In addition, we will evaluate some specific optimizations for a vertical design that uses a custom floorplan where hotspot structures have been placed in the upper (cooler) layers whereas cooler structures are placed in lower layers. We will also analyze the different floorplan organizations in order to minimize peak temperature in the 3D die-stacked architecture. For our evaluation the selected power budget is 50% of the original power consumption of the processor.

4.1 Simulation Environment

For evaluating the proposed approache we have used the Virtutech Simics platform extended with Wisconsin GEMS v2.1 [17]. GEMS provides both detailed memory simulation through a module called Ruby and a cycle-level pipeline simulation through a module called Opal. We have extended both Opal and Ruby with all the studied mechanisms that will be explained next. The simulated system is a homogeneous CMP consisting of a number of replicated cores connected by a switched 2D-mesh direct network. Table 1 shows the most relevant parameters of the simulated system. Power scaling factors for a 32nm technology were obtained from McPAT [13]. To evaluate the performance and power consumption of the different mechanisms we used scientific applications from the SPLASH-2 benchmark suite in addition to some PARSEC applications (the ones that finished execution in less than 5 days in our cluster). Results have been extracted from the parallel phase of each benchmark. Benchmark sizes are specified in Table 2.

3D thermal modeling can be accomplished using an automated model that forms the RC circuit for given grid dimensions. For this work we have ported HotSpot 5.0 [21] thermal models into Opal and have built our tiled CMP by replicating N times our customized floorplan, where N is the number of cores. Figure 2 shows the base floorplan design we have chosen. This floorplan was obtained from Hotfloorplaner (provided by the Hotspot 5.0). Our resulting CMP will be composed of a varying number of these cores (from 2 to 16). As cited before, we will assume an interconnection delay between layers of one FO4 (10μm length wires, as in [19]).

Moreover, thermal hotspots increase cooling costs and have a negative impact on reliability and performance. The significant increase in cooling costs requires designs for temperature margins lower than the worst-case. Leakage power is exponentially dependent on temperature, and an incremental feedback loop exists between temperature and leakage, which may turn small structures into hotspots and potentially damage the circuit. High temperatures also adversely affect performance, as the effective operating speed of transistors decreases as they heat up. In this paper we model both leakage (through McPAT) and the leakage/temperature loop in Opal, so leakage will be updated on every Hotspot exploration window (10K cycles). Leakage power is translated into power tokens and updated according to the formula $L_{new} = L_{Base}\ x\ e^{Leak\_Beta\ x\ (TCurrent\ -\ TBase)}$ where *Leak_Beta* depends on technology scaling factor and is provided by HotSpot 5.0, L_{new} is the updated leakage, L_{Base} is the base leakage (obtained using McPAT), *TCurrent* is the current temperature and *TBase* is the base temperature. Once leakage is updated, it is translated back to *power tokens*.

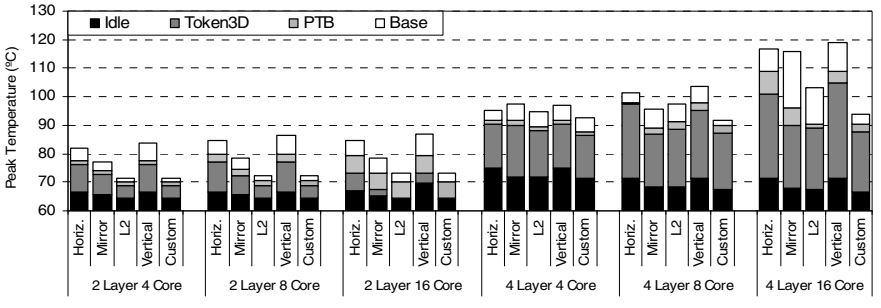


Fig. 3. Peak temperature for PTB, Token3D and the base case for different floorplans and core configurations

Another important parameter is the cooling system. The regular thermal resistance of a cooling system ranges from 0.25 K/W for the all-copper fan model at the highest speed setting (very good), to 0.33 K/W for the copper/aluminum variety at the lowest setting. In this work we model a real-world Zalman CNPS7700-Cu heatsink with 0.25 K/W thermal resistance and an area of 3.268 cm^2 (136mm side).

4.2 Effects of *Token3D* on Peak Temperature

Figure 3 shows the peak temperature for different floorplan configurations and a varying number of cores (from 4 to 16) using stacked bars. The reported *idle*

temperature corresponds to the average idle temperature of the cores[1]. The studied floorplans are: Horizontal (Figure 1.a), Mirror (Figure 1.b), L2 (Figure 1.c), Vertical (Figure 1.d) and Custom. As cited before, this last floorplan corresponds to a new configuration that places hotspots into upper layers of the 3D stack, giving more chances for them to cool down, and will be further discussed later in the next subsection. In Figure 3 we can clearly see that both L2 and Custom are the best designs to reduce peak temperature of the processor. This is due to the fact that both designs place the L2 in lower layers, and, as it can be seen in Figure 4, the L2 is the coolest structure within a core, even though we are accounting for leakage to calculate temperature. This placement leaves hotspots close to the surface and hot structures can cool down easily. We can also see that even a simple change in the floorplan such as mirroring between layers gives substantial temperature reduction (5-6ºC) compared to the horizontal design.

When considering the vertical design we can observe a higher peak temperature than the horizontal one. This vertical design was introduced in [19] by Puttaswamy *et al.* along with a dynamic power saving mechanism, *Thermal Herding*, that disables layers at runtime, depending on the number of bits used by the different instructions. This vertical design assumes each structure is vertically implemented across all layers. In our evaluation of this vertical design, the area occupied by each structure and its power consumption is divided by the number of available layers, but we do not disable any layer, to isolate our proposed power control mechanisms from the benefits obtained by *Thermal Herding*. For instance, in a 4-layer vertical design the implemented thermal model calculates the temperature of a structure in layer *i* by considering one fourth of its original power and area, however, the fraction of that structure is stacked on top of another equal portion of the same structure, with all portions simultaneously accessed, and therefore, increasing temperature. Note, however, that the use of *thermal herding* and its ability to disable unused layers for the vertical design is orthogonal to the use of our proposed PTB and *Token3D* approaches.

When it comes to the studied power control mechanisms both the original PTB and *Token3D* are able to reduce peak temperature by 2-26ºC depending on the floorplan configuration. *Token3D* is always 1-3% better than the original PTB balancing mechanism. We must also note that, as we get closer to the idle temperature, any temperature reduction comes at a higher performance degradation.

Figure 4-left shows a more detailed analysis on the effects of both PTB and *Token3D* in the peak temperature of the different core structures. We selected the most probable configuration for 3D die-stacked cores (Mirror, Figure 1.b) and a 4-layer 16-core CMP for this *per structure* temperature analysis. As cited before, PTB and *Token3D* are evaluated with a preset power budget of 50% of the original average power consumption. For comparison purposes we also evaluate DVFS trying to match the same target power budget of 50%. Figure 4-left helps us to locate our design hotspots (I-cache, TLB, Branch predictor, Load store queue) and see how both cycle-level power control mechanisms are able to reduce peak temperature by 20-36%. For example, the I-cache goes from 150ºC down to 110ºC, 30ºC less than DVFS. We can

[1] We define "idle" temperature as the temperature of the whole CMP in idle state (i.e., only the operating system is running).

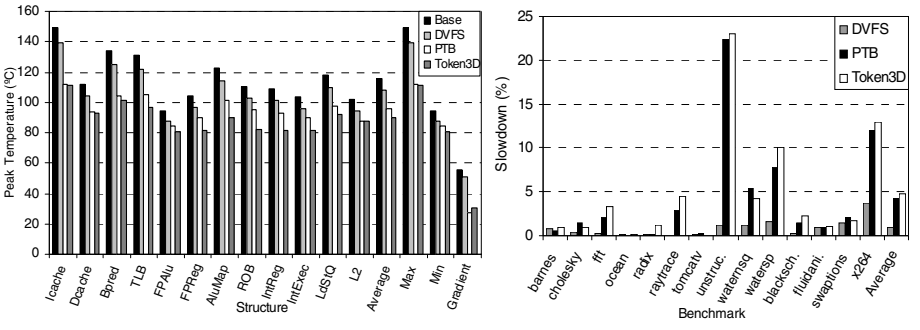


Fig. 4. Peak temperature (left) and performance (right) of a 4-layer 16-core CMP using the mirror floorplan

also see that, on average for this selected design, *Token3D* is 5-6ºC better than regular PTB. It is also important to note that our cycle-level mechanisms are able to reduce all hotspots peak temperature and put them close to the average core temperature. This last result is specially interesting on 3D architectures, as they exacerbate thermal problems and a much tighter power control is necessary. This is the benefit we expected from the highly accurate power budget matching our mechanisms provide, that ensures minimal deviation from the target power budget and, therefore, temperature. In Figure 4-left we also show the *spatial gradient* (temperature difference between the hottest and coolest structure of the core). Reducing spatial gradients is important because they can cause clock skew and impact circuit delay [1]. In particular, both PTB and *Token3D* are able to reduce this gradient by more than 50%, from 50ºC to 22ºC.

In terms of performance degradation (Figure 4-right), regular PTB behaves slightly better than *Token3D*, as power is equally divided between all cores and they can get to the next synchronization point more evenly, while *Token3D* will unbalance cores and make them wait at the synchronization point more time.

4.3 Further Temperature Optimizations

In addition to the PTB temperature analysis and the introduction of *Token3D* we also wanted to perform some optimizations for the vertical 3D die-stacked layout. More specifically, we will analyze the effects on peak temperature of MLP-based instruction window (ROB) resizing [18] and ALU selection based on instruction criticality (from ALUs placed on different layers) while varying the number of cores.

Figure 5 shows the effects on peak temperature of different instruction window (IW) sizes for a 4-layer vertical core design (Figure 1.d). Each core has a 128-entry IW that is equally distributed across the different layers in the vertical design (as we are working with 4 layers, each layer has 32 entries). Entries are disabled by layer, so we disable entries in groups of 32. In order to decide the current IW size we use a dynamic MLP-based IW resizing mechanism as proposed in [18]. In Figure 5-left, we also show the distribution of the average IW size for different benchmark suites (represented with lines). This average window size highly varies between

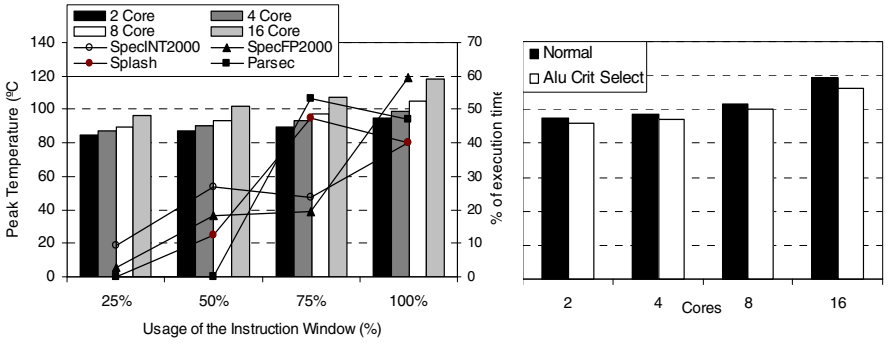


Fig. 5. Peak temperature of the instruction window (left) and ALUs (right) for a varying number of cores

benchmarks, as memory-bound benchmarks require many IW entries to bring more data simultaneously from memory, while CPU-bound applications do not need that many entries. Therefore, instead of just showing the peak temperature reduction of the average benchmarks (bars in Figure 5-left) we decided to do a design exploration of the peak temperature based on the IW size. For example, Parsec benchmarks use 0% of the time 25% and 50% of the IW, 55% of the time use a 75% of the IW (12ºC reduction) and 45% of the time use the whole IW.

When working with vertical designs we can think of having different types of ALUs placed into different layers: fast (and hot) ALUs placed on upper layers for critical instructions plus slower power-saving ALUs placed in lower layers. As our core design includes an instruction criticality predictor we can use this information to decide where we want to send a specific instruction. Figure 5-right shows the effect on peak core temperature having half of the ALUs placed in layers 2-3 (upper layers) and half of the ALUs placed in layers 0-1 (lower layers). The ALUs in the lower layers consume 25% of the original power consumption but are also 25% slower than the original ALUs. Results show a peak temperature reduction of 3-5ºC. This small temperature reduction is due to the fact that in our core design ALUs are not a hotspot (as it can be seen in Figure 4-left: IntExec and FPAlu structures) for the studied benchmarks, and thus, their temperature contribution has almost no impact on the average peak temperature of the processor. However, we can expect better results with other CPU-bound applications where ALUs become a hotspot.

Finally, we want to introduce a custom floorplan design that merges both vertical and horizontal designs. This design is an extension of the L2 design (Figure 1.c) for a 4-layer core. Based on the information provided by Figure 4-left we can separate cool from hot structures and place them in different layers. Hot structures are placed in the top layer (Bpred, Icache, Alumap, TLB, LdStQ, IntReg and ROB), which is the closest to the heatsink. The second layer consists of the rest of structures except the L2, and the last two layers hold the L2 cache and memory controllers. This custom design has the additional advantage of reducing inter-layer communication when bringing data from memory, as memory controllers and the L2 are placed close to the socket. As we can see in Figure 3 (last bar on each group), this design is able to reduce peak temperature by almost 12ºC for a 4-layer 16-core processor.

5 Conclusions

3D die-stacked integration offers a great promise to increase scalability of CMPs by reducing both bandwidth and communication latency problems. However, the increase on core density leads to an increase in temperature and hotspots in these designs. Moreover, as building process scales down below 32nm, leakage becomes an important source of power consumption and, as it increases exponentially with temperature, causes a power/temperature loop that negatively affects to 3D die-stacked processors. To control temperature, regular DTM mechanisms detect overheating in any of the temperature sensors and trigger a power control mechanism to limit power consumption and cool the processor down. However, neither DVFS nor task migration (the most frequently used mechanisms) offer accurate ways to match this target power budget.

Power tokens and *Power Token Balancing* (PTB) were introduced by Cebrian *et al.* as an accurate way to account for power and match a power constraint with minimal performance degradation by balancing power among the different cores of a 2D CMP. In this paper we evaluate these mechanisms in a new design scenario, 3D die-stacked processors. In this scenario PTB is able to reduce average peak temperature by 2-20ºC depending on the selected floorplan. For specific hotspot structures (i.e., instruction cache) PTB can reduce peak temperature by almost 40% in a 4-layer 16-core CMP. In addition, we have proposed *Token3D*, a novel policy that takes into account temperature and layout information when balancing power, giving priority to cool cores over hot ones. This new policy enhances PTB by providing an additional 3% temperature reduction over the original PTB approach. Also note that task migration is orthogonal to PTB and can be applied simultaneously for further temperature reductions.

To conclude this work we have also extended 3D die-stacked vertical designs with additional power control mechanisms. First, we enabled instruction window resizing based on MLP. CPU-intensive applications are highly dependent on cache, but do not show performance degradation if the instruction window is reduced. On the other hand, memory-intensive applications require big instruction windows to locate loads and stores and take advantage of MLP. Based on these properties we extended previous vertical designs with adaptive instruction window resizing. Second, we split ALUs in different groups, low latency and high latency ALUs. Low latency ALUs consume more power and should be placed in upper layers of the 3D design, on the other hand, high latency ALUs are more energy-friendly and can be placed in lower layers of the 3D stack, lowering the chances of becoming a potential hotspot. An instruction criticality predictor was used to decide where an instruction should be placed, either in a fast but expensive or in a slow but efficient unit.

Finally, we explored a custom 3D design that merges both vertical and horizontal designs trying to minimize hotspots. In this design hot processor structures are placed in upper layers while cool structures are placed in lower layers. The design is able to reduce peak temperature by an additional 10% / 85% over the best horizontal / vertical designs.

Acknowledgements. This work was supported by the Spanish MEC, MICINN and EU Commission FEDER funds under Grants CSD2006-00046 and TIN2009-14475-C04. Also by the EU-FP7 ICT Project "Embedded Reconfigurable Architecture (ERA)", contract No. 249059. Finally, the EU-FP7 HiPEAC funded an internship of J.M. Cebrián at U. Uppsala.

References

[1] Ajami, A.H., Banerjee, K., Pedram, M.: Modeling and analysis of nonuniform substrate temperature effects on global ULSI interconnects. IEEE Trans. on CAD 24(6), 849–861 (2005)

[2] Black, B., Annavaram, M., Brekelbaum, N., DeVale, J., Jiang, L., Loh, G.H., McCaule, D., Morrow, P., Nelson, D.W., Pantuso, D., Reed, P., Rupley, J., Shankar, S., Shen, J., Webb, C.: Die Stacking (3D) Microarchitecture. In: Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 469–479 (December 2006)

[3] Brooks, D., Martonosi, M.: Dynamic thermal management for high-performance microprocessors. In: Proc. of the 7th Int. Symposium on High Performance Computer Architecture, HPCA (2001)

[4] Cebrián, J.M., Aragón, J.L., García, J.M., Petoumenos, P., Kaxiras, S.: Efficient Microarchitecture Policies for Accurately Adapting to Power Constraints. In: Proc. of the 23rd Int. Parallel and Distributed Processing Symposium, IPDPS (2009)

[5] Cebrián, J.M., Aragón, J.L., Kaxiras, S.: Power Token Balancing: Adapting CMPs to Power Constraints for Parallel Multithreaded Workloads. To appear in Proc. of the 25rd Int. Parallel and Distributed Processing Symposium (May 2011)

[6] Coskun, A.K., Rosing, T.S., Whisnant, K.A., Gross, K.C.: Static and dynamic temperature-aware scheduling for multiprocessor SOCS. IEEE Trans. on VLSI 16(9), 1127–1140 (2008)

[7] Coskun, A., Ayala, J., Atienza, D., Rosing, T., Leblebici, Y.: Dynamic Thermal Management in 3D Multicore Architectures. In: Proc. of the Int. Conf. on Design, Automation and Test in Europe (2009)

[8] Flynn, M.J., Hung, P.: Microprocessor Design Issues: Thoughts on the Road Ahead. IEEE Micro 25(3) (2005)

[9] Gomaa, M., Powell, M.D., Vijaykumar, T.N.: Heat-and-Run: leveraging SMT and CMP to manage power density through the operating system. In: Proc. of the 10th Int. Conf. on Architectural Support for Programming Languages and Operating Systems ASPLOS (2004)

[10] Healy, M., et al.: Multiobjective microarchitectural floorplanning for 2-d and 3-d ICs. IEEE Transactions on CAD 26(1) (2007)

[11] Isci, C., Buyuktosunoglu, A., Cher, C., Bose, P., Martonosi, M.: An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In: Proc. of the 39th Int. Symposium on Microarchitecture, MICRO (2006)

[12] Kgil, T., D'Souza, S., Saidi, A., Binkert, N., Dreslinski, R., Mudge, T., Reinhardt, S., Flautner, K.: PicoServer: using 3D stacking technology to enable a compact energy efficient chip multiprocessor. In: Proc. of the 12th Int. Conf. on Arch. Support for Programming Languages and Operating Systems (2006)

[13] Li, S., Ahn, J.H., Strong, R.D., Brockman, J.B., Tullsen, D.M., Jouppi, N.P.: McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In: Proc. of the 42nd Int. Symposium on Microarchitecture, MICRO (2009)
[14] Loi, G.L., Agrawal, B., Srivastava, N., Lin, S.C., Sherwood, T., Banerjee, K.: A thermally-aware performance analysis of vertically integrated (3-D) processor-memory hierarchy. In: Proc. of the 43rd Int. Conference on Design Automation (July 2006)
[15] Macken, P., Degrauwe, M., Paemel, V., Oguey, H.: A voltage reduction technique for digital systems. In: Proc. of the IEEE Int. Solid-State Circuits Conf. (February 1990)
[16] Madan, N., Balasubramonian, R.: Leveraging 3D Technology for Improved Reliability. In: Proc. of the 40th Annual IEEE/ACM Int. Symposium on Microarchitecture (December 2007)
[17] Martin, M.M.K., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D., Wood, D.A.: Multifacet's general execution-driven multiprocessor simulator (gems) toolset. SIGARCH Comput. Archit. News 33(4), 92–99 (2005)
[18] Petoumenos, P., Psychou, G., Kaxiras, S., Cebrian, J.M., Aragon, J.L.: MLP-aware Instruction Queue Resizing: The Key to Power-Efficient Performance. In: Proc. of the 23rd Int. Conf. on Architecture of Computing Systems (ARCS) (February 2010)
[19] Puttaswamy, K., Loh, G.H.: Thermal Herding: Microarchitecture Techniques for Controlling Hotspots in High-Performance 3D-Integrated Processors. In: Proc. of the 13th Int. Symposium on High Performance Computer Architecture (HPCA), pp. 193–204 (2007)
[20] Sartori, J., Kumar, R.: Distributed Peak Power Management for Many-core Architectures. In: Proc. of the Int. Conference on Design, Automation and Test in Europe, DATE (2009)
[21] Skadron, K., Stan, M., Huang, W., Velusamy, S., Sankaranarayanan, K., Tarjan, D.: Temperature-aware microarchitecture. In: Proc. of the Int. Symposium on Computer Architecture, ISCA (2003)
[22] Souri, S.J., Banerjee, K., Mehrotra, A., Saraswat, K.C.: Multiple Si layer ICs: motivation, performance analysis, and design implications. In: Proc. of the Int. Conf. on Design Automation (2000)
[23] Xie, Y., Loh, G.H., Black, B., Bernstein, K.: Design space exploration for 3D architectures. J. Emerg. Technol. Comput. Syst. 2(2), 65–103 (2006)
[24] Zhu, C., Gu, Z., Shang, L., Dick, R.P., Joseph, R.: Three-dimensional chip-multiprocessor run-time thermal management. IEEE Transactions on CAD 27(8), 1479–1492 (2008)

Bandwidth Constrained Coordinated HW/SW Prefetching for Multicores

Sai Prashanth Muralidhara, Mahmut Kandemir, and Yuanrui Zhang

Department of Computer Science and Engineering
Pennsylvania State University, University Park, PA 16802, USA
{smuralid,kandemir,yuazhang}@cse.psu.edu

Abstract. Prefetching is a highly effective latency hiding technique that can greatly improve application performance. However, aggressive prefetching can potentially stress the off-chip bandwidth. The resulting bandwidth stalls can potentially negate the performance gain due to prefetching. In this paper, focusing on a multicore environment, we first study the comparative benefits of hardware and software prefetching and analyze if the two are complimentary or redundant. This analysis also evaluates different aggressiveness levels of hardware prefetching. Secondly, we weigh the positive performance benefits of prefetching against the negative performance effects of bandwidth stalls. Thirdly, we propose a hierarchical prefetch management scheme for multicores that controls the prefetch levels such that the overall performance gain is improved. Lastly, we show that our proposed off-chip bandwidth aware prefetch management scheme is very effective in practice, leading to performance gains of upto about 10% in system throughput over a bandwidth agnostic prefetching scheme.

1 Introduction

Prefetching is a well-known memory latency hiding technique, which predicts future memory accesses and proactively fetches the corresponding memory elements to the cache ahead of time in order to hide memory access latencies during execution [7] [8] [9] [10]. Prefetching can either be implemented at the hardware level [7] [8] [10] [9] or by the software [20] [18]. The effectiveness of a prefetching scheme is directly dependent on the predictability of memory accesses, which is an application characteristic. In a multicore system, each core prefetches data elements independently into the cache. The benefits due to prefetching can potentially be different for different cores depending on the application characteristics. Further, each core/application can potentially be involved in both hardware and software prefetching. There have been previous techniques proposed to throttle inaccurate prefetchers and increase aggressiveness levels on more accurate ones [28]. Also, when the last level cache is shared, aggressive prefetching can worsen the cache interference problem, especially when it is inaccurate and/or inefficient. In such cases, it is helpful to throttle the prefetches that are inaccurate and cause high interference in the shared cache space [11].

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 310–325, 2011.

In this paper, we first study the comparative accuracies and benefits of software prefetching and different levels of hardware prefetching. We then study and analyze the impact of prefetching on the off-chip memory bandwidth performance. Prefetching can lead to increased off-chip bus traffic, and can potentially increase the pressure on the off-chip bandwidth. This can cause extensive bandwidth stalls.We explore the tradeoff between extensive aggressive prefetching and bandwidth stalls. Further, we study if the performance degradation due to bandwidth stalls wipe away the performance gains achieved as a result of prefetching.

We propose a hierarchical bandwidth-aware coordinated prefetching scheme that manages the prefetch aggressiveness levels of different cores such that the performance gains due to prefetching are improved, while the performance losses due to bandwidth stalls are reduced. This prefetch management scheme operates dynamically and decisions are made at the end of each execution interval. More specifically, a *global prefetch manager* considers the overall bandwidth delay and the prefetch effectiveness of each core during each execution interval, and then decides to increase or decrease the prefetch aggressiveness levels of the cores. This decision to change the prefetch levels of the cores is made such that the performance improvement due to prefetching is higher than the stall time due to limited bandwidth and contention. It then directs the individual core-level prefetch managers to change the prefetch levels correspondingly. At each core, a *core-level prefetch manager* manages and enforces the prefetch aggressiveness levels. This prefetch manager not only issues hardware prefetch requests but also handles the software prefetch instructions. It decides whether to allow software prefetching or hardware prefetching or both and also at what aggressiveness levels. It is to be noted that prefetching on a core can be termed very aggressive if both hardware prefetching at the highest aggressiveness level and software prefetching is enabled. Aggressiveness can be downgraded by reducing the aggressiveness of hardware or software prefetching or both. Overall, the main goal of our approach is to reward useful prefetchers and punish the ones that hurt bandwidth availability without any performance benefit. Lastly, we evaluate our proposed scheme on set of workloads comprising of applications from the SPEC 2006 benchmark suite [1] on a simulation based setup, and show that our scheme yields average system throughput benefits of about 8%, and up to about 10% over an off-chip bandwidth unaware scheme. To summarize, we make the following contributions in this paper:

- We evaluate the performance benefit of both hardware (different levels) and software prefetching schemes. We later compare the performance improvement due to prefetching against the performance degradation due to the extra pressure it exerts on the off-chip bandwidth.
- We propose a *hierarchical prefetch management scheme* that tries to dynamically change the prefetch levels of the individual cores such that the performance degradation due to bandwidth contention is reduced and the performance improvement due to prefetching is improved.

- We present an extensive experimental evaluation of the proposed hierarchical prefetch management. Our results show that the proposed scheme is very effective in practice and improves the system throughput by up to 10%, and by an average of 8%.

2 Background and Methodology

2.1 Prefetching

Prefetching is a widely employed technique intended to improve on-chip cache performance [7] [8] [9] [10] [20] [18]. Prefetching, however, is not always beneficial. Some fraction of the predicted memory requests are never accessed. This is not the only instance of wasted prefetching. A future memory request prediction can turn out to be true but before the prefetched memory element is accessed, it might be evicted from the cache. Also, a prefetched request may kick out a useful data element from the cache. In these instances, prefetching increases the off-chip bus traffic and possibly cause bandwidth stalls without any significant benefit. Therefore, prefetch accuracy, which is an application characteristic determines the overall performance benefit from prefetching.

Hardware Prefetching. In the case of hardware prefetching, the future memory access prediction and the process of initiating requests to prefetch those elements are carried out by the hardware at runtime. Due to costs and limits on delay, hardware prefetchers generally implement a simple stride based prefetching or a stream based prefetching. A very aggressive hardware prefetcher would typically predict a large number of future memory requests and prefetch them. In comparison, a prefetcher with a lower aggressiveness level would be more conservative, predicting and issuing fewer prefetches. In this paper, we refer to and implement a stream prefetcher [28] [6] [24]. Aggressiveness level of a stream prefetcher is defined by two parameters: prefetch distance and prefetch degree [28] [6] [24]. *Prefetch Distance* dictates how far ahead of the demand access stream the prefetcher can issue prefetch requests, and *Prefetch Degree* determines how many cache blocks to prefetch when there is a cache block access to a monitored memory region.

Software Prefetching. In this case, the future memory access prediction is made statically, at compile time or at the coding time, and specific instructions are inserted into the code body to prefetch those predicted elements at the time of execution. Some applications render themselves to easy compile time prediction in which case the software prefetching is very effective [20] [18] [19]. Software prefetching also has the ability to employ complex and time consuming prefetching algorithms since the process is done apriori at compile time. Hardware prefetching, on the other hand, employs simpler prediction mechanisms but does well where software prefetching fails to analyze the code, e.g., as in the case of pointer-based applications.

2.2 Experimental Setup

Core architecture	UltraSparc 3, 3.1 GHz
Operating system	Sun Solaris 9
L1 caches	private, 3 cycle latency, direct-mapped
L2 cache	shared, 15 cycle latency, 16 way associative
Memory latency	260 cycles
Hardware Prefetcher	64 stream prefetcher per core, 4 prefetch levels
DRAM controller	demand-prefetch equal priorities, on-chip, 128 entry req buffer, FR-FCFS
DRAM chip	refer to Micron DDR2-800 [2]

Fig. 1. Default system parameters used

We model the off-chip memory bandwidth and implement the prefetching infrastructure for multicores using a Simics [3] based in-house module. The base system architecture simulated in our evaluations is a four-core multicore machine with a shared L2 cache and a shared off-chip memory bandwidth. The shared L2 cache is assumed to be a partitioned cache (i.e., its cache ways are distributed evenly across applications though in principle we could use any partitioning strategy). The cores simulated in this system are based on the UltraSparc 3 architecture [5]. The main architectural details of the simulated system are shown in the table given in Figure 1. In the evaluation of the proposed dynamic scheme later, we employ execution intervals of 10 million instructions. The hardware prefetcher used in this paper is a stream prefetcher [28] [6] [24] with 64 streams per prefetcher.

Benchmarks. For all the motivational and evaluation purposes, we use the applications from the SPEC 2006 benchmark suite [1], and construct our workloads from the subsets of these applications. To enable software prefetching on the applications, they are compiled on a SUN compiler with the highest optimization flag set.

Terminology. In this paper, by "prefetch level", we mean the "aggressiveness level" of the prefetcher. All types of prefetching mentioned in this paper are implemented for the last level of cache in a multicore. Whenever we refer to "software prefetching" in this paper, we mean the handling of the software-inserted prefetch instructions in the hardware. We do not propose or implement a new software prefetching algorithm. We compile the applications using a software prefetch enabled compiler that inserts prefetch instructions into the executable. We only refer to the way these instructions are handled in the hardware.

3 Empirical Motivation

3.1 Prefetching Benefits

The goal of this section is to compare the performance of various prefetching techniques with different aggressiveness levels across different applications.

Hardware Prefetching. Figure 2 plots the performance of our applications when different levels of prefetching are enabled compared to the case of no prefetching. We experimented with four different prefetch levels: *no prefetching, level 1, level 2* and *level 3*. *Level 1* prefetching has a prefetch distance of 4 and prefetch degree of 1. Prefetch distance and prefetch degree of *level 2* are

16 and 2 respectively, and those of *level 3* are 64 and 4. In this set of experiments, software prefetching is disabled, which means the prefetch instructions are ignored as no-ops. Since we are first interested in studying the performance benefits of prefetching in isolation, the performance effects due to bandwidth constraints are not considered in these experiments. From Figure 2, we can infer that while some applications are prefetch sensitive and, therefore benefit from more aggressive levels of prefetching, others do not exhibit large performance gains as prefetch levels are increased. In the above scenario, the prefetch levels can be reduced on applications that are not very prefetch-sensitive without a high performance penalty. On the flip side, increasing the prefetch levels on prefetch-sensitive applications can be very beneficial.

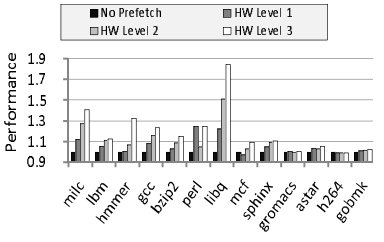


Fig. 2. Performance comparisons of different levels of hardware prefetching. The performance values are normalized to that of the no prefetching case.

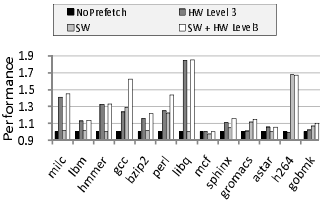


Fig. 3. Performance comparisons of software prefetching, hardware level 3 prefetching, and both with the case of no prefetching. The performance values are normalized to that of the no prefetching case.

Software Prefetching. Figure 3 compares software prefetching, hardware level 3 prefetching, combined software-hardware prefetching against the no-prefetching case. One can see from this plot that, for some applications, hardware prefetching does much better than software prefetching, whereas for some others, it is the other way around. More interestingly, in some cases, enabling both hardware and software prefetching is much better than enabling just one of them, as in the case of *gcc* and *perl*. In some other cases, although effective individually, enabling both does not do any better than enabling only one of them, as in the case of *astar* and *h264*. Therefore, in a multicore system, some applications perform better when both hardware and software prefetching are enabled, while some others perform equally well with just one of them enabled.

3.2 Off-Chip Bandwidth Effects

In this section, we study the effect of prefetching on off-chip bandwidth pressure. We employ an off-chip bandwidth of 6.4 GB/s in these experiments. For this purpose, we selected a workload of four applications: *lbm, mcf, libquantum,* and *milc*. These four applications are executed on a four-core processor (one application per core) with a shared, partitioned cache, and a shared off-chip bandwidth.

One core prefetching. In the first run, we enabled prefetching only on the first core which executed *lbm*, while disabling prefetching on all other cores. We

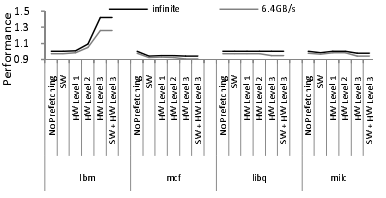


Fig. 4. Performance comparisons of different prefetching schemes with both the infinite bandwidth case and a bandwidth of 6.4 GB/s, when prefetching is enabled only on core 1 (*lbm*) and disabled for all others

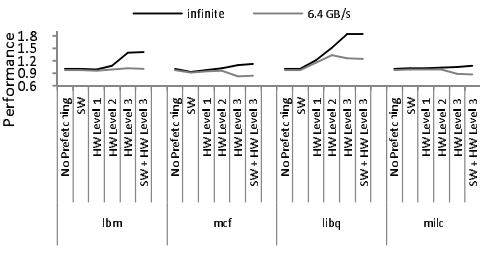


Fig. 5. Performance comparisons of different prefetching schemes with both the infinite bandwidth case and a bandwidth of 6.4 GB/s, when prefetching is enabled on all cores

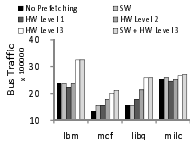


Fig. 6. Contributions to the bus traffic by different applications

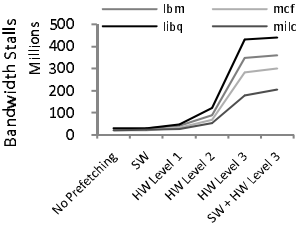


Fig. 7. Bandwidth stalls (in cycles) suffered by applications as the prefetching level is increased

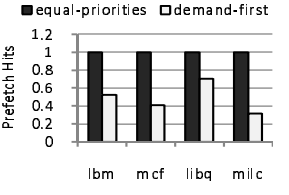


Fig. 8. Comparison of equal priorities for prefetch and demand requests versus a scheme where demand requests are prioritized over prefetch requests in terms of the number of useful prefetches

experimented with software prefetching, three levels of hardware prefetching, and a combined hardware-software prefetching scheme. The results in Figure 4 show that *lbm*, which executed on core 1, achieves a performance benefit when compared to the case of no prefetching. However, its benefits are reduced due to the limited bandwidth constraint. Further, it degrades the performance of the other applications due to the additional requests (prefetch requests from core 1) and the resulting bandwidth stalls. When using the most aggressive prefetching, the performance degradations on the other cores are significant. We also repeated this by enabling prefetching on core 2, core 3 and core 4 alone, and observed similar results. Therefore, prefetching can have different degrees of performance degradation due to bandwidth constraints. Further, *aggressive prefetching by one core can adversely impact the performance of other applications due to bandwidth contention and the resulting delays.*

All cores prefetching. We also considered a more realistic execution scenario, where different applications prefetch memory elements individually and the cores share the available off-chip bandwidth. In this case, prefetching is enabled on all

the cores. In Figure 6, we plot the comparative contributions to the bus traffic by the applications when prefetching is enabled on all the cores. The bus traffic increases rapidly when the prefetch level is increased for some applications, while for others, the increase is not that steep (for instance *milc*). Figure 7 shows how this increase in bus traffic translates into stalls due to limited bandwidth. Note that, even if the bus traffic increase is small, bandwidth stalls can be significant. The above two graphs plot absolute values of bus traffic increase and bandwidth stall cycles. Figure 5, on the other hand, illustrates how these factors affect the performance of applications when different prefetch variants are enabled for both the infinite bandwidth case and a more realistic case of 6.4 GB/s bandwidth. In the limited bandwidth case, prefetching aggressively in a bandwidth unaware manner on all the cores results in some performance improvement only on core 3 (*libq*). In all other applications/cores, performance degradation due to limited bandwidth completely wipes out all the benefits from prefetching and in some cases results in a net performance degradation. This effect increases with increasing prefetch levels. Also, for some applications, while absolute values of bandwidth stalls in Figure 7 increase sharply with prefetch levels, performance degradation is not that steep. Therefore, some applications are more bandwidth-stall resistant (tolerant). In modeling the performance effects later in Section 4.3, we take this into account. We do not just consider prefetch accuracies and the resulting bus traffic as the basis as done previously [28] [11] but also consider the bandwidth stalls and the actual impact of bandwidth stalls on application performance as the basis.

To summarize, while prefetching aggressively can improve performance, it can also hurt the performance due to bandwidth constraints. Therefore, it is important to enable prefetching without increasing bandwidth delays extensively.

3.3 Prefetch Request Priority

Increase in bandwidth delays due to prefetching typically occurs only if prefetch requests are treated on par with demand memory requests. If normal load/store (demand) memory requests have a higher priority than the prefetching requests, then additional bus traffic due to prefetch requests may not lead to any additional bandwidth delay. It is to be noted here that bandwidth delays might still be present in the system but those delays are due to the normal (demand) memory requests, and will be present irrespective of whether prefetching is turned on or not. Prioritizing demand requests and prefetching requests equally leads to increased performance improvement from prefetching as can be seen in Figure 8. This is due to the fact that if the prefetch requests have a lower priority than the demand requests, then the prefetch requests can get delayed inordinately and these increased bandwidth delays can render most of prefetch requests useless (since prefetched data would be brought into the cache late). This leads to decreased prefetch efficiency and, therefore, decreased positive performance impact of prefetching [16]. Therefore, our proposed scheme employs equal priorities, and tries to keep the number of useful prefetches high, while at the same time, mitigating the additional bandwidth stalls due to prefetch requests.

4 Bandwidth Aware Prefetching

Figure 9 summarizes the operation of our proposed scheme. A *global prefetch manager* makes decisions on whether to increase or decrease the prefetching levels on the individual cores and the decisions are communicated to the *core-level prefetch manager*.

The details on how these decisions are made are presented in Section 4.3. After the global manager directs a core-level prefetch manager to either increase or decrease the prefetch level of the core, the core-level manager applies the prefetch-level changes locally (i.e., to the core it is attached to), as described in Section 4.1. This prefetch management scheme works dynamically, making decisions on prefetch level changes and applying those changes at the end of each execution interval. This scheme is also history based, in the sense that all the relevant statistics, which include the total bandwidth stall-time and the prefetch efficiency counters of individual cores, collected during an execution interval are used to make decisions for the next execution interval.

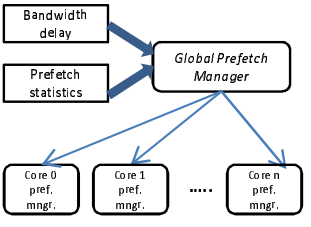


Fig. 9. Hierarchical bandwidth aware prefetching scheme that includes a *global prefetch manager* and a set of *core-level prefetch managers*

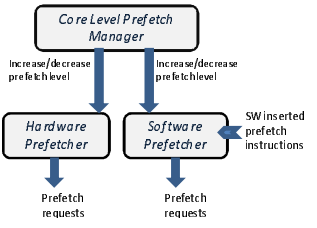


Fig. 10. Details of a core-level prefetch manager, which controls the prefetch levels of both hardware and software prefetchers of a core

Implementation. Hardware support is needed to maintain the performance counters. The prefetch management scheme itself is implemented in the runtime system/OS, which reads these hardware performance counters.

4.1 Core-Level Prefetch Manager

The core-level prefetch manager sets and enforces the prefetch aggressiveness level at the core level. It can either increase or decrease the prefetch level based on the directions from the global prefetch manager.

The core-level prefetch manager handles the changes in prefetch levels of the hardware prefetcher similar to that proposed in [11]. In addition to the hardware prefetcher, our proposed prefetch manager also employs a *software prefetcher*, which is an engine that handles all the software prefetch instructions issued by the core (compiler-inserted or programmer inserted). A prefetch instruction, when issued, results in a prefetch request. All such

```
increase_prefetch_level()
begin
  accuracy_HW = prefhits_SW / prefetches_HW
  accuracy_SW = Prefhits_SW / prefetches_SW
  if accuracy_HW > accuracy_SW
    //Increase HW prefetch level
    increase prefetch_distance_HW
    increase prefetch_degree_HW
  else
    //Increase SW prefetch level
    increase prefetch_distance_SW
    increase prefetch_degree_SW
end
```

Fig. 11. Prefetch level increase function

prefetch requests are routed through our proposed software prefetcher. When the global prefetch manager directs the core-level manager to either increase or decrease the prefetch level, the core-level manager can increase or decrease the prefetch level of either the hardware prefetcher or the software prefetcher. What we mean by "prefetch levels" in hardware and software prefetchers is explained later in detail. The role of the core-level prefetch manager in controlling the prefetch levels of both hardware and software prefetches is illustrated in Figure 10. The global prefetch manager either increases or decreases prefetch level, and does not set absolute values. The decision of whether to change the prefetch level of the hardware prefetcher or the software prefetcher is determined by calculating the corresponding *prefetch accuracies*. More accurate prefetcher is always preferred. This way, we prioritize either hardware or software prefetching based on their accuracies (the prefetch increase function is shown in Figure 11, prefetch decrease function is on similar lines).

4.2 Prefetch Levels

Hardware Prefetch Levels. We implement a stream prefetcher for hardware prefetching [6]. As mentioned earlier, the aggressiveness level of a stream prefetcher is defined by two parameters: *prefetch distance* and *prefetch degree*. Our hardware prefetcher design is similar to that implemented in [28] and further details on implementation can be found in [28] [6] [24]. In essence, the prefetch distance determines how far ahead of the memory stream the prefetch requests are issued and the prefetch degree determines how many prefetch requests are issued each time. We implement four prefetch levels in this work: *no prefetch*, *low prefetch*, *medium prefetch*, and *high prefetch*. No prefetch level performs no prefetching. Low prefetch level performs prefetching with a prefetch distance of 4 and prefetch degree of 1. Medium prefetch performs prefetching with a prefetch distance of 16 and a prefetch degree of 2, while the high prefetch level has prefetch distance of 64 and prefetch degree of 4.

Software Prefetch Levels. The software prefetcher implements the software prefetch levels by filtering the prefetch requests. As mentioned before, the software prefetcher receives all the prefetch requests that are issued by the software (compiler inserted or programmer inserted) instructions. The four aggressiveness levels of software prefetching are: *no prefetch*, *low prefetch*, *medium prefetch*, and *high prefetch*. When the level is set to no prefetch, all the prefetch requests are dropped. In the case of low prefetch level, two in every four prefetch requests are dropped, while only one in every four is dropped in the case of medium prefetch level. When the level is set to high prefetch, all prefetch requests coming from the software inserted prefetch instructions are issued by the software prefetcher without dropping any of them.

4.3 Global Prefetch Manager

As shown in Figure 9, the two main inputs to the *global prefetch manager* are the total bandwidth stall-time and the prefetch statistics.

Bandwidth stall-time. A demand request stalls in the memory controller queue if there are other requests ahead which are being serviced or waiting to be serviced. While the prefetch requests may also wait, they do not contribute to performance degradation (a higher wait-time for prefetch requests can of course limit the benefits due to prefetching). We define "*bandwidth_stall*" as the total stall-time (in cycles) experienced by the demand requests in a given execution interval. It is the sum of all individual demand request stall-times in that execution interval. Observe that "stall-time" in this context refers to wait-time in the queue due to bandwidth constraint. It does not include the time for a demand request to get serviced (to perform the memory operation). We compute *bandwidth_stall* using a simple counter in the memory controller. Since the off-chip bandwidth is a single resource shared across all the cores, *bandwidth_stall* is a single value, which is the sum of bandwidth stalls of all requests of all cores serviced by the off-chip bandwidth during the given execution interval.

Prefetch Statistics. As described in Section 4.1, each core has a hardware prefetcher and a software prefetcher associated with it. We define "$prefetches_i$" to be the total number of prefetches issued by core i. It is the sum of the number of prefetches issued by the hardware prefetcher and those issued by the software prefetcher. The metric "$prefhits_i$" is defined as the total number of prefetch requests (both hardware and software) that turned out to be hits for core i. These values are calculated using the prefetch bit of the cache line and by employing counters in the prefetchers.

Benefit Estimation. The performance improvement on core i due to prefetching is quantified by a parameter called "$benefit_i$". This improvement is specifically due to the avoidance of a fraction of core i cache misses. The metric $benefit_i$ is computed for each core i using the prefetch statistics collected during the execution interval as follows: $benefit_i = \frac{Reduction\_in\_cache\_miss\_stall\_time_i}{instructions_i}$. Therefore, accounting for this reduction in cache misses, we obtain: $benefit_i = \frac{(misses\_old_i - misses\_new_i) \times avg\_miss\_penalty}{instructions_i} = \frac{prefhits_i \times avg\_miss\_penalty}{instructions_i}$, where $instructions_i$ is the number of instructions executed in the current execution interval, $misses\_old_i$ is the estimated number of cache misses if prefetching was not enabled, $misses\_new_i$ is the number of cache misses with prefetching, and $avg\_miss\_penalty$ is the average cache miss penalty in cycles.

Cost Estimation. Prefetching leads to additional memory requests (in addition to the normal load/store demand requests). The measure of performance degradation suffered by core i due to memory bandwidth stall-time resulting from these prefetch requests it issues is quantified by the metric $cost_i$. Due to the fact that memory bandwidth is shared, the additional prefetches issued by core i can cause bandwidth stalls for not only core i but also for all other cores as well. As a result, $cost_i$ should take all these stalls into account. Firstly, the

total bandwidth stall caused by the prefetches issued by all the cores can be estimated as below: $total\_prefetch\_stall = \frac{\Sigma_{i=0}^{n} prefetches_i}{total\_requests} \times bandwidth\_stall$, where $\Sigma_{i=0}^{n} prefetches_i$ is the sum of prefetches issued by all the cores during the interval, $total\_requests$ is the total number of requests that reached the memory controller during the execution interval (i.e., sum of the demand and prefetch requests), and $bandwidth\_stall$ is the total bandwidth stall time as defined earlier. We can now estimate the stall caused by core i (due to the prefetches issued by core i) as follows: $prefetch\_stall_i = \frac{prefetches_i}{\Sigma_{i=0}^{n} prefetches_i} \times total\_prefetch\_stall$. For each core i, we now have $prefetch\_stall_i$, which is the estimated absolute bandwidth stall-time caused by the prefetch requests issued by core i. Since the off-chip bandwidth is a shared resource, $prefetch\_stall_i$, caused by core i can affect demand requests of any of the cores. We define $band\_stall_{i,j}$ as the bandwidth stall caused by the prefetches from core i on the performance of core j (on the demand requests of core j). This value estimates the fraction of the bandwidth stall of core j, due to the prefetch requests issued by core i. We can estimate $band\_stall_{i,j}$ as follows: $band\_stall_{i,j} = \frac{demand_j}{\Sigma_{i=0}^{n} demand_k} \times prefetch\_stall_i$, where $demand_j$ is the total number of demand requests issued by core j, which in this case is approximately equal to the number of L2 cache misses on core j, $\Sigma_{i=0}^{n} demand_k$ is the total number of demand requests issued by all cores.These $band\_stall_{i,j}$ values estimated above are the absolute stall times in cycles and not the impact on performance. Therefore, we now estimate $cost_i$, which is a measure of the total performance degradation caused by the prefetches issued by core i on the performance of all cores including core i. Note that performance degradation considered above is just the effect of bandwidth stalls. The value of $cost_i$ can be estimated as follows: $cost_i = \Sigma_{j=0}^{n} \frac{band\_stall_{i,j}}{instructions_j}$. It is important to note that, we do not consider prefetch accuracies or the absolute bandwidth stalls in our estimation of $benefit_i$ and $cost_i$ values. We estimate both these values in terms of the net effect on the application performance.

Algorithm. The global prefetch manager manages the prefetch levels for each core with the goal of improving the overall performance gains due to prefetching. In order to do so, global manager employs a cost/benefit analysis based scheme.

A prediction based dynamic scheme is employed by the global manager, i.e., the algorithm works by computing and making prefetch level changes for cores at the end of each execution interval. To begin with, all cores prefetch at the highest aggressiveness levels. The $benefit_i$ and $cost_i$ values are estimated for every core i at the end of each interval after

```
global_prefetch_manager()
begin
  for each execution interval:
    read bandwidth_stall
    for each i from 0 to num_cores:
      read instructions_i, prefetches_i and prefhits_i
      compute benefit_i and cost_i
      if (benefit_i − cost_i) >= cost_i × α then
        //increase the prefetch level of core i
        core_level_manager_i.increase_prefetch_level()
      else if (benefit_i − cost_i > 0 and
      benefit_i − cost_i < cost_i × α)
        //do not change the prefetch level of core i
      else (benefit_i − cost_i) <= 0 then
        //decrease the prefetch level of core i
        core_level_manager_i.decrease_prefetch_level()
  end for
end
```

Fig. 12. The algorithm executed by the global prefetch manager

reading the relevant performance counter values. For each core i, the prefetch level is increased if the $benefit_i - cost_i$ is greater than the $cost_i \times \alpha$ (i.e., if $benefit_i$ is greater than $cost_i$ by α percentage). If, on the other hand, the $benefit_i - cost_i$ is lower than the $cost_i \times \alpha$ but greater than zero, then the prefetch level is left unchanged. Finally, if $benefit_i$ is less than the $cost_i$ value, then the prefetch level is decreased for core i. The global prefetch manager enforces the prefetch level change for a given core i by directing the core-level manager of the corresponding core. The reason for reducing the prefetch level for a given core is obvious since the estimated benefit is lower than the estimated cost. On the other hand, increasing the prefetch level is more nuanced. The level is increased only if the estimated benefit is greater than the cost by a *pre-defined threshold value* (α).If the benefit is not greater than the cost by α percentage, the prefetch level is left unchanged. This algorithm can reduce the prefetch level of a core i gradually to zero (which means no prefetches are issued) when $benefit_i$ continues to be lesser than $cost_i$ after continuous prefetch level decrements. In this case, when the prefetch level is zero, $benefit_i$ will always be zero and the prefetch level will potentially be stuck at zero without being increased. To avoid this scenario, the core-level prefetch manager increments the prefetch level of a core to level 1 if the prefetch level is stuck at zero for more than two execution intervals. In this algorithm, since we consider benefit and cost values in terms of estimated changes in application performance, the goal is always to improve the performance of applications and improve the overall system throughput.

α values. The α values are tunable to make the prefetching scheme more conservative or more aggressive. We experimented with a lot of α values and finally determined that a value of 0.2 is reasonable. Therefore, in our implementation, if the benefit exceeds the cost by 20%, we increase the prefetch level.

5 Experimental Evaluation

Our evaluation setup is described in Section 2. A four-core machine with a shared, partitioned L2 cache was modeled as the underlying multicore architecture. We built several workloads that consist of four applications, each from the SPEC 2006 suite [1]. In all our evaluations, we collect results and data for a period of 1 billion cycles. Cache is however warmed up for a period of 500 million instructions prior to collecting results. We consider execution intervals of 10 million instructions. Our proposed prefetching scheme is called *Dyn_Band* throughout the experimental section.

Average Throughput. Figure 13 presents the throughput gain acheived by our proposed scheme (*Dyn_Band*) over other prior prefetching schemes when averaged over 10 different workloads we experimented with. Different workloads might benefit differently from the prior prefetching schemes. Our proposed scheme recognizes this and enables only those prefetching schemes and levels that benefits the workloads, also taking into account the bandwidth pressure exerted by the extra prefetch memory requests. Our proposed scheme yeilds an average

system throughput gain of about 8% over the best of the previous prefetching schemes.

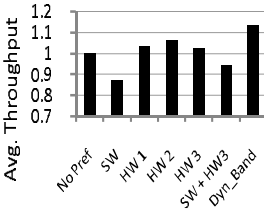


Fig. 13. Comparison of workload throughput averaged across multiple workloads

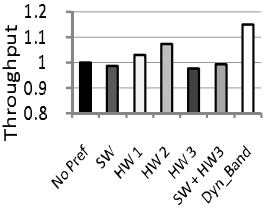


Fig. 14. Throughput comparison for the workload (*lbm*, *mcf*, *libquantum*, and *milc*).

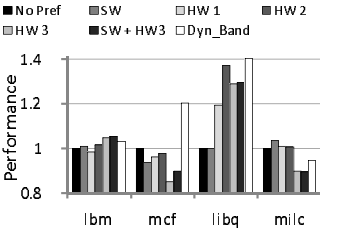


Fig. 15. Performance comparisons of the applications in the workload (*lbm*, *mcf*, *libquantum*, and *milc*).

Workload Instance. In order to understand our proposed scheme in more detail, we now present the results for a single workload instance that consists of *lbm*, *mcf*, *libquantum*, and *milc*. The corresponding throughput results are shown in Figure 14. In this case, our proposed dynamic bandwidth-aware prefetching scheme improves throughput by 15% over the no prefetching scheme. Among the other prefetching schemes, hardware level 2 prefetching does better than others because of lower pressure on off-chip bandwidth. Our dynamic bandwidth-aware scheme has a throughput gain of about 8% over this hardware level 2 prefetching. Figure 15 shows the individual application performance values. We observe that the application *milc* gains about 40% in performance over the no prefetching scheme and *mcf* gains about 20%.

Dynamics of the system. In order to analyze the working of our proposed scheme, we consider the execution of a workload comprising of *bzip2, libq, sphinx* and *gromacs* applications, and focus on the performances of *libq* and *gromacs*. We track how our scheme works dynamically, and adjusts the prefetch levels of these two applications based on their $benefit$ and $cost$ values (note here that our scheme works and adjusts the prefetch levels of all four applications; we focus on just two for clarity).

Figures 16 and 17 plot the observed $benefit$ and $cost$ values for these two applications for 11 execution intervals, when our scheme is used. In the case of *libq*, the $benefit$ value is consistently higher than the $cost$ value, while in the case of *gromacs*, the values are very close together. In order to study how our scheme dynamically changes the prefetch levels in accordance with the above values, we plot the $\frac{benefit-cost}{cost}$ values for the two applications for the same 11 execution intervals in Figure 18. Recall that, in the global prefetch management algorithm presented earlier in Figure 12, the equation $benefit_i - cost_i > cost_i \times \alpha$ is used to decide whether to increase the prefetch level or not. If the value $\frac{benefit-cost}{cost}$ is greater than α (0.2), then the prefetch level is increased and so on. Figure 19 plots the prefetch level changes made by our proposed scheme for both the applications. Note that, at execution interval 3, the prefetch level of *gromacs*

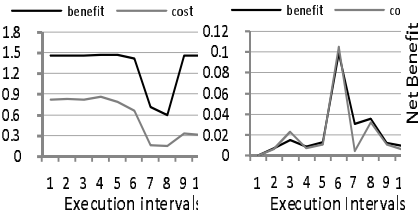


Fig. 16. Benefit and cost values of *libq* during execution

Fig. 17. Benefit and cost values of *gromacs* during execution

Fig. 18. Net benefit values of *libquantum* and *gromacs* during execution

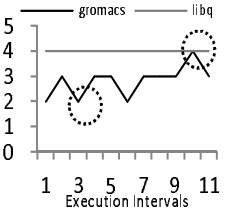


Fig. 19. Prefetch levels of *libquantum* and *gromacs* during execution

is reduced to 2 because the $benefit - cost$ value is less than zero (circled in Figures 18 and 19). Also, at execution interval 10, when $\frac{benefit-cost}{cost}$ becomes greater than 0.2 for gromacs, the prefetch level is increased to 4. However, it is reverted back because it was not highly benefitial. On the flipside, the prefetch level of *libq* is maintained at 4 since its $\frac{benefit-cost}{cost}$ values are consistently greater than 0.2.

Sensitivity analysis. We increased the memory bandwidth from 6.4 GB/s to 12.8 GB/s and executed the workloads. An average throughput improvement of about 7% over the best other prefetching scheme was observed. Therefore, even with higher bandwidth, our scheme achieves significant throughput improvement. We also experimented with different α values and found that a value of 0.2 provides the right balance.

6 Related Work

Hardware Prefetching. Hardware-controlled prefetching is an efficient way to implement prefetching [15] [7] [8] that tries to mitigate the negative effect of cold misses. Sequential prefetching automatically prefetches several consecutive data blocks into the cache upon a miss in the cache [9] [10]. Palacharla and Kessler investigate advanced stream buffers and filtering techniques to enhance the prefetching efficiency [24]. Hur and Lin discuss a dynamic stream detection technique that adapts the aggressiveness levels of prefetching in order to improve prefetching performance [13].

Software Prefetching. Seminal work related to software prefetching was authored by Mowry et al in [20], where they propose to use software controlled prefetch instruction insertion to enable prefetching. Other software prefetching schemes include [18] [20].

Prefetch Control. Srinath et al propose to use feedback control to improve the positive impact of prefetching and mitigate the adverse impact of harmful prefetches [28]. In [11], Ebrahimi et al investigate a control mechanism that can dynamically adjust the prefetch aggressiveness levels.

Off-Chip Bandwidth Studies. Rixner et al [27] introduce a scheduling policy that favors requests that hit in the row buffer over other requests. Nesbit et al suggest to prioritize memory requests of applications in accordance to their QoS requirements [23]. Rafique et al propose to adaptively change the fraction of memory bandwidth allocation for each thread [25]. In [14], Ipek et al study a machine learning approach in which a reinforcement learning based scheme is used to dynamically adapt scheduling decisions in the memory controller. Mutlu and Moscibroda proposed a stall time fair memory access scheduling in [21] and a parallelism-aware batch scheduling scheme in [22]. Liu et al study the effects of memory bandwidth partitioning on system performance [17].

Prefetching and Off-Chip Bandwidth. Lee et al propose to dynamically increase and decrease the priorities of prefetch requests at the memory controller in order to improve the benefits due to prefetching and decrease the penalties of inaccurate prefetchers [16]. In [12], Ebrahimi et al introduce a cooperative hardware/sofwtare approach to prefetch linked date structures in a bandwidth-efficient way.

In this paper, we considered the off-chip bandwidth as an important constraint, based on which, the prefetching levels of different cores are adjusted such that the prefetch benefits are improved. We considered the off-chip bandwidth stalls instead of the inter-core interferences [11] as the constraint. We did so because inter-core interferences are not prefetch specific and can result from demand accesses as well. We also modeled the benefits and costs of prefetching in terms of performance changes in this work, which makes our scheme throughput driven, and evaluated the comparative benefits of hardware and software prefetching.

7 Concluding Remarks

In this paper, we proposed a smart prefetch management scheme that exploits the performance benefits of prefetching while mitigating the performance degradation due to bandwidth stalls. Our proposed scheme is very effective in practice yielding a performance benefit of up to 8% in throughput over a bandwidth unaware prefetching strategy.

References

1. http://www.spec.org/spec2006
2. Micron: 1GB DDR2 SDRAM component: MT47H128M8HQ-25, http://download.micron.com/pdf/datasheets/dram/ddr2/1GbDDr2.pdf
3. Magnusson, P.S., et al.: Simics: A full system simulation platform. Computer 35(2), 50–58 (2002)
4. Xie, Y., Loh, G.H.: Dynamic Classification of Program Memory Behaviors in CMPs. In: CMP-MSI (2008)
5. Hetherington, R.: The UltraSparc T1 processor. SUN (2005)

6. Tendler, J., et al.: Power4 System Microarchitecture. IBM Technical White Paper (October 2001)
7. Baer, J.-L., Chen, T.-F.: An effective on-chip preloading scheme to reduce data access penalty. In: Proc. SC (1991)
8. Charney, M.J., Puzak, T.R.: Profetching and memory system behavior of the spec95 benchmark suite. IBM J. Res. Dev. (1997)
9. Dahlgren, F., et al.: Fixed and adaptive sequential prefetching in shared memory multiprocessors. In: Proc. ICPP (1993)
10. Dahlgren, F., et al.: Sequential hardware prefetching in shared-memory multiprocessors. IEEE Trans. Parallel Distrib. Syst. (1995)
11. Ebrahimi, E., et al.: Coordinated control of multiple prefetchers in multi-core systems. In: Proc. MICRO (2009)
12. Ebrahimi, E., et al.: Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In: Proc. HPCA (2009)
13. Hur, I., Lin, C.: Memory prefetching using adaptive stream detection. In: Proc. MICRO (2006)
14. Ipek, E., et al.: Self-optimizing memory controllers: A reinforcement learning approach. In: Proc. ISCA (2008)
15. Jouppi, N.P.: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. SIGARCH Comput. Archit. News (1990)
16. Lee, C.J., et al.: Prefetch-aware dram controllers. In: Proc. MICRO (2008)
17. Liu, F., et al.: Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. In: Proc. HPCA (2010)
18. Mowry, T., Gupta, A.: Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. J. Parallel Distrib. Comput. (1991)
19. Vanderwiel, S., Lilja, D.: Data Prefetch Mechanisms. ACM Computing Surveys, CSUR (2000)
20. Mowry, T.C., et al.: Design and evaluation of a compiler algorithm for prefetching. In: Proc. ASPLOS (1992)
21. Mutlu, O., Moscibroda, T.: Stall-time fair memory access scheduling for chip multiprocessors. In: Proc. MICRO (2007)
22. Mutlu, O., Moscibroda, T.: Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In: Proc. ISCA (2008)
23. Nesbit, K.J., et al.: Fair queuing memory systems. In: Proc. MICRO (2006)
24. Palacharla, S., Kessler, R.E.: Evaluating stream buffers as a secondary cache replacement. In: Proc. ISCA (1994)
25. Ebrahimi, E., et al.: Fairness via source throttling: a configurable and high-performance fairness substrate for multicore memory systems. In: Proc. ASPLOS (2010)
26. Rafique, N., et al.: Effective management of dram bandwidth in multicore processors. In: Proc. PACT (2007)
27. Rixner, S., et al.: Memory access scheduling. In: Proc. ISCA (2000)
28. Srinath, S., et al.: Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In: Proc. HPCA (2007)

Unified Locality-Sensitive Signatures for Transactional Memory

Ricardo Quislant, Eladio D. Gutierrez, Oscar Plata, and Emilio L. Zapata

Department of Computer Architecture, University of Málaga,
ETSI Informática, Campus Teatinos, Málaga, E 29071, Spain
{quislant,eladio,oplata,zapata}@uma.es

Abstract. Transactional memory systems coordinate the execution of concurrent transactions by committing non-conflicting ones. Transaction conflicts are detected by recording on-the-fly the memory locations issued by the threads. Some implementations use two per-thread Bloom filters (signatures), one for reads and another for writes, for that purpose. Signatures summarize sets of memory addresses accessed inside a transaction in bounded hardware. However, fixed-sized hardware introduces the address aliasing problem that results in false positives during the conflict checking process.

It is known that the false positive rate increases with the size of the transactions, which has a strong negative impact in the performance of their concurrent execution. In a previous work, authors developed a technique with the aim of reducing the probability of false positives by exploiting spatial locality. In this paper we propose a new technique based on joining the two Bloom filters into a single one and partially sharing the hash function mappings for reads and writes. This unification technique is combined with the locality-sensitive one and it is proved that the false positive rate is further reduced.

This paper proves that unified locality-sensitive signatures improve the execution performance of large concurrent transactions in most tested codes compared to separate signatures, without increasing significantly the required hardware area and with a small increment of power consumption.

Keywords: Hardware transactional memory, memory locality, signatures, Bloom filters.

1 Introduction

Transactional Memory (TM) [8,7] emerges as an alternative to the conventional multithreaded programming to ease the writing of concurrent programs. TM introduces the concept of transaction, a block of computations which appears to be executed with atomicity and isolation. Transactions replace a pessimistic lock-based model by an optimistic one which solves the abstraction and composition problems.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 326–337, 2011.

TM systems coordinate the execution of concurrent transactions by committing non-conflicting ones. A conflict occurs when concurrent transactions access the same memory location and, at least, one of the accesses is a write. Transaction conflicts are detected by recording on-the-fly the memory locations issued by the threads. Some TM implementations use two per-thread Bloom filters [1] (signatures), one for reads and another for writes, for that purpose. Signatures summarize sets of memory addresses accessed inside a transaction in bounded hardware. However, fixed-sized hardware introduces the address aliasing problem (different memory addresses with the same signature representation) that results in false positives during the conflict checking process. Examples of systems that use signatures are BulkSC [4], LogTM-SE [19], SigTM [12], FlexTM [17], and STMlite [11].

It is known that the false positive rate increases with the size of the transactions, and this have a strong negative impact in the performance of their concurrent execution. In a previous work [14], authors developed a technique with the aim of reducing the probability of false positives. This technique defines new hash function mappings so that nearby located addresses share some bits in the Bloom filters, that is, it exploits spatial locality. In this paper we propose a new technique based on joining the two Bloom filters into a single one and partially sharing the hash function mappings for reads and writes without adding significant hardware complexity. The rationale behind this technique is the uneven cardinality that transactional read/write sets exhibit, where read sets are usually larger than write sets. As a result, the signature for reads populates much more than the one for writes and, consequently, the false positive rate for the read signature may be high while, at the same time, the write filter has still a low occupation, with negligible false positive rate. This unification technique is combined with the locality-sensitive one and it is proved that the false positive rate is further reduced.

We use the Wisconsin GEMS LogTM-SE simulator [10] to implement and evaluate the performance of the proposed unified locality-sensitive signatures. Besides, we use CACTI [18] to evaluate the hardware area and power requirements. Experimental results show that the proposed approach is able to reduce the false positive rate and improve the execution performance in most of the benchmark codes, with an insignificant increase in hardware area and a slight increase in power consumption.

The rest of the paper is organized as follows. Next section presents a background on signatures, describing how they are usually designed and implemented, and a brief review of the related work. In Section 3 we introduce and discuss our proposed unified signature design, including their implementation, and a comparison with the separate signature design. Section 4 analyzes the hardware area and power requirements of our signature designs. In section 5 we show an analysis of our proposed signatures and we determine the false positive rate in different cases. Section 6 presents the implementation of unified signatures on the GEMS simulator, and discusses how our novel signature design may improve the execution performance. Finally, Section 7 concludes the paper.

2 Background and Related Work

In the context of TM, each concurrent thread uses its signatures to record all the memory locations issued when executing inside a transaction. These locations are sorted out into a read set (RS) and a write set (WS). Thus, each thread needs a pair of private signatures. As they are used for conflict detection amongst concurrent transactions, signatures do not tolerate false negatives (undetected true conflicts) but may assume a limited amount of false positives (false conflicts). On the other hand, the RS and WS sizes are unknown in advance, therefore, signatures should not limit the number of addresses to be tracked. In addition, test and insertion of an address should be fast operations.

Fulfilling the requirements above, Ceze et al. [5] proposed a signature implementation with per-thread Bloom filters. These filters were devised to test whether an element is a member of a set in a time and space-efficient way. The Bloom filter comprises a bit array and k different hash functions that map elements into k randomly distributed bits of the array. At first, all the array bits are set to 0. Inserting an element into the Bloom filter consists in setting to 1 the k bits given by the hash functions. Test for membership consists in checking that those k bits are asserted.

Bloom filters are also known as true or regular Bloom filters. Sanchez et al. [16] proposed the parallel Bloom filter as a hardware-efficient implementation of regular Bloom filters. Whereas the regular filter is implemented as a k-ported SRAM, the parallel one consists of k 1-ported SRAMs, yielding the same or better false positives rate. The same work concludes that Bloom filters should include the H3 class of hash functions [3], instead of bit-selection ones [15], since they are closer to random distribution. However, H3 is more hardware expensive than bit-selection as it needs an XOR tree per hash bit.

An alternative hardware-efficient implementation of hash functions, Page-Block-XOR hashing (PBX), has been proposed in [20]. They use the concept of entropy to find the input bits to the hash functions with high randomness, allowing to reduce the hardware complexity of those functions. Notary also proposes a technique to reduce the number of asserted bits in the signature, based on segregating addresses into private and shared sets. Then, only the shared addresses are recorded in the signature. This solution requires support at the compiler, runtime/library and operating system levels. In addition, the programmer must define which objects are private or shared.

Recently, Choi et al. [6] proposed adaptive grain signatures, that keep the history of transaction aborts and dynamically changes the input bit range to the hash functions based on the abort history. The aim of this design is to reduce the number of false positives that harm the execution performance.

3 Unified Signature Design

Parallel Bloom filters have been proved to yield similar or better performance than regular ones and they require less hardware [16] [14]. Consequently, regular implementation will not be taken into account in this paper.

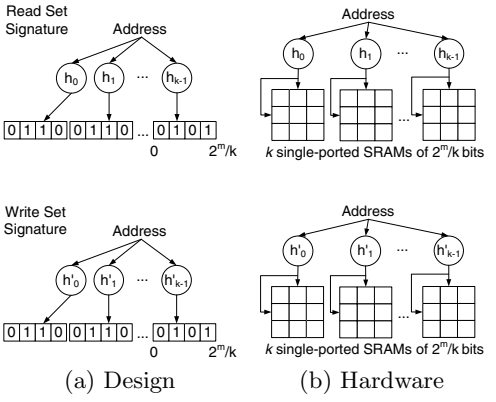


Fig. 1. Parallel Separate Signatures

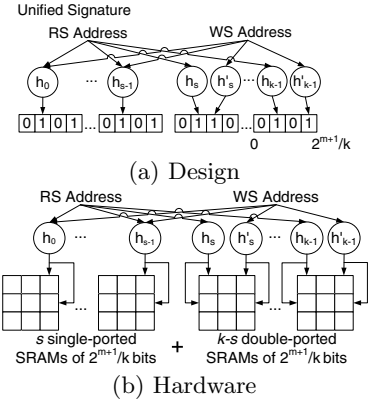


Fig. 2. Parallel Unified Signatures

Parallel Bloom filters comprise k arrays of $2^m/k$ bits, each of which is only indexed by its own hash function. Figure 1 shows the design and implementation of parallel Bloom signatures. They consist of two separate parallel filters to record the read set and write set addresses. Parallel filters can be implemented as single-ported SRAMs, thus saving in hardware area with respect to regular filters which are implemented as multi-ported SRAMs.

The unified counterpart for the parallel separate signature is depicted in Figure 2. In this case, the bit array is also partitioned into k smaller arrays but $(2^{m+1}/k)$-bit length. Each array is indexed by two hash functions, one for the read set, $h_{[0,k-1]}$, and the other one for the write set, $h'_{[0,k-1]}$. Consequently, parallel unified filters need 2-ported SRAMs instead of single-ported ones taking about twice the area of parallel separate filters. To alleviate this problem, s SRAMs can be made single-ported as Figure 2b shows. This way, an address inserted as a read address is also inserted as a write and vice-versa.

The motivation behind unified signatures come from Table 1 which shows the percentage of addresses that have been both read and written inside transactions for each benchmark (a description of the simulation environment can be found in Section 6.1) with respect to the total number of addresses (without repetition). About 50% of locations are both read and written for Bayes, Kmeans and Yada. Overall, about 30% of total locations addressed by each benchmark has been both read and written.

In order to work out the value of s a trade off between hardware requirements and signature performance has to be carried out. On the one hand, if s is set to k, the unified signature implements k single-ported SRAMs. Thus, such a signature requires the same hardware than the parallel separate signature but it is unable to discriminate between read and written addresses and it could degrade the performance. On the other hand, if s is set to 0, the unified signature implements k double-ported SRAMs increasing the hardware requirements but maximizing the probabilities of discrimination between read and written addresses. Section 6.2 explores every possible scenario.

Table 1. Percentage of addresses that have been both read and written inside transactions

Bench	%	Bench	%
Bayes	51.0	Labyrinth	15.3
Genome	16.0	SSCA2	25.0
Intruder	7.1	Vacation	8.4
Kmeans	48.6	Yada	45.0

Table 2. Area (mm^2) and dynamic energy per access (nJ) requirements of parallel separate and parallel unified signatures. 32nm technology. $k = 4$.

	Area		Energy	
Filter size (2^m)	**4Kbit**	**16Kbit**	**4Kbit**	**16Kbit**
Separate	0.0084	0.0292	0.0020	0.0047
Unified $s = 0$	0.0191	0.0640	0.0030	0.0081
Unified $s = 3$	0.0098	0.0331	0.0026	0.0068

Finally, hash functions are implemented as H3 XOR functions [3] that only comprise a set of XOR gate trees per function. XOR gate trees do not require significant area and, moreover, they can be replaced by a single line of XOR gates by using PBX hashing [20].

4 Hardware Evaluation

Table 2 compares the area required by unified and separate signatures for several filter sizes. "Filter size" row is the size of one set filter, i.e. 4Kbit means two filters of 4Kbit (for RS and WS) for separate signatures and one filter of 8Kbit for unified ones. We used CACTI 6.5 [13] to model the SRAMs using the 32nm technology node. Parallel separate signatures comprise eight single-ported SRAMs (4 for the RS and 4 for the WS) as $k = 4$, while parallel unified $s = 0$ signatures have four double-ported SRAMs. Separate read/write ports are used. Parallel unified $s = 3$ signatures have three single-ported SRAMs and only one double-ported SRAM. Ports are dual-ended which means that two lines are required per bitline.

Table 2 shows that parallel separate signatures yield the best area and energy numbers. Regarding the parallel unified $s = 0$ signature, it is about twice larger than the parallel separate signature due to its double-ported SRAMs. The parallel unified $s = 3$ configuration, is the closest to the parallel separate one in terms of area. It is only a 13% larger because of the double-ported SRAM. However, parallel unified $s = 3$ signatures outperforms parallel separate ones as seen in Section 6.3. Regarding energy, Table 2 shows a 30% increment in dynamic energy consumption for parallel unified $s = 3$ signatures.

Concerning the hashing logic area, Sanchez et al. [16] worked out one-fifth of the SRAM area for 4 XOR hash functions. This area can be halved using PBX hashing [20] without impact in the performance.

5 False Positive Analysis

Let A be a sequence of addresses, to be inserted in a single Bloom filter of 2^m bits with k hash functions, whose cardinality is $n = Card(A)$. The false positive probability is commonly calculated [16] [14] as:

$$p_{\text{FP}}(m, k, n) = \left(1 - \left(1 - \tfrac{1}{2^m}\right)^{nk}\right)^k. \quad (1)$$

Eq. (1) can be adapted to the locality-sensitive signature scheme of [14] by considering two supplementary parameters: f which is the probability of an address to be *local*, that is, near to another one in the sequence, and b which measures the average number of bits asserted by a *local* reference with respect to its closest neighbor in the sequence. The value of f will depend on the spatial locality of the program. The value of b can be estimated as $b = \frac{1}{2}+2\cdot\frac{1}{4}+3\cdot\frac{1}{8}+4\cdot\frac{1}{8}$ for the locality-sensitive signatures defined in [14] with $k = 4$ hash functions. For such signatures the false positive probability is given now by:

$$p_{\text{FP LOC}}(m,k,n,f) = \left(1-\left(1-\tfrac{1}{2^m}\right)^{n(1-f)k+nfb}\right)^k. \tag{2}$$

First, consider separate filters, where the read and write sets are stored separately, in order to compare their false positive rates to those of the unified filter. Let us define $p_R = \frac{Card(R-R\cap W)}{Card(R\cup W)}$ and $p_W = \frac{Card(W-R\cap W)}{Card(R\cup W)}$ as the probability of an address of the sequence being only read or written, respectively, using the cardinality function of the read (R) and write (W) sets. Consequently, $n = Card(R\cup W)$. Also an address in the sequence can be both read and written with probability $p_{RW} = \frac{Card(R\cap W)}{Card(R\cup W)}$. Therefore, the false positive probability in each filter, assuming locality-sensitive signatures, can be expressed as:

$$p^{read}_{\text{FP LOC}} = \left(1-\left(1-\tfrac{1}{2^m}\right)^{n(p_R+p_{RW})\bar{k}}\right)^k, p^{write}_{\text{FP LOC}} = \left(1-\left(1-\tfrac{1}{2^m}\right)^{n(p_W+p_{RW})\bar{k}}\right)^k, \tag{3}$$

where $\bar{k} = (1-f)k + fb \leq k$ is the average number of hash insertions in the locality-sensitive scheme.

The effective false positive rate will finally depend on how many checks take place on each separate filter. This way, a mathematical expectation of the false positive rate for the separated locality-sensitive signatures can be expressed as:

$$E[p^{\text{SEPARATE}}_{\text{FP LOC}}(m,k,n,f)] = c_R p^{read}_{\text{FP LOC}} + c_W p^{write}_{\text{FP LOC}}. \tag{4}$$

Here c_R and c_W denote the probability of each filter being checked during the sequence of references. This checking pattern is directly linked to the way in which the threads inspect the potential data dependencies. It remains unknown until run-time, being very dependent on the parallelization strategy and the input data. Other important issues having influence on the checking pattern are the coherence protocol and the abort/resume policy of transactions.

Regarding unified filters, Eq. (2) is still valid as long as the k hashing functions used by reads and writes are disjoint. To make a fair comparison, the size of the unified locality-sensitive filter must be the sum of the sizes of the separate filters. Thus, the false positive probability for this unified locality-sensitive filter is given by

$$p^{\text{UNIFIED}}_{\text{FP LOC}}(m,k,n,f) = p_{\text{FP LOC}}(m+1,k,n(1+p_{RW}),f). \tag{5}$$

In Table 3 several scenarios are shown for different values of the parameters defined above. Eqs. (4) and (5) have been evaluated with high and low values for the given parameters: locality (f), only read addresses (p_R), read and written addresses (p_{RW}), and number of checks in the read filter (c_R). Note that $p_R + p_{RW} + p_W = 1$ and $c_R + c_W = 1$. Labels in the table point out the scheme (separate or unified) with the lowest false positive rate according to equations. In the 66% of the explored scenarios the unified scheme beats the separate one. Nevertheless, the scenario which is closer to real workloads is $c_R = 0.5$, i.e. read and write filters are evenly checked, because the TM system assures strong atomicity [8] and data requested to main memory (out of the bounds of TM) must be checked in both filters. Notice that, in this case, the unified scheme yields better false positive rates until the filter gets filled in about $\frac{2}{3}$ of its total capacity. With high locality such a limit shifts to $\frac{3}{4}$ or even disappears.

Table 3. Signature scheme, separate (SEP) or unified (UNI), with the lowest false positive rate according to Eqs. (4) and (5) for several values of the given parameters (Bloom filters with $m = 10$ and $k = 4$)

		$p_R = 0.15$						$p_R = 0.25$						$p_R = 0.5$					
		$p_{RW} = 0.2$			$p_{RW} = 0.5$			$p_{RW} = 0.2$			$p_{RW} = 0.5$			$p_{RW} = 0.2$			$p_{RW} = 0.5$		
f	n \ c_R	0.2	0.5	0.8	0.2	0.5	0.8	0.2	0.5	0.8	0.2	0.5	0.8	0.2	0.5	0.8	0.2	0.5	0.8
0.2	128	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI
	256	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI
	512	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI
	768	UNI	SEP	SEP	UNI	SEP	SEP	UNI	SEP	SEP	UNI	UNI	UNI	SEP	SEP	UNI	SEP	SEP	UNI
	1024	UNI	SEP	SEP	UNI	SEP	SEP	UNI	SEP	SEP	UNI	UNI	UNI	SEP	SEP	UNI	SEP	SEP	UNI
0.8	128	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI
	256	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI
	512	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI
	768	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI	UNI	SEP	UNI	UNI	SEP	UNI	UNI
	1024	UNI	UNI	SEP	UNI	SEP	SEP	UNI	UNI	SEP	UNI	UNI	UNI	SEP	UNI	UNI	SEP	SEP	UNI

6 Evaluation

6.1 Methodology

To evaluate the performance of our unified locality-sensitive signatures we used Simics [9] full system execution-driven simulator along with the TM module GEMS [10] from the Wisconsin Multifacet Project. Simics simulates the SPARC architecture and it is able to run an unmodified copy of a Solaris operating system. Solaris 10 was installed on the simulated machine and all workloads run on top of it. GEMS' Ruby module implements the LogTM-SE TM [19] and also includes a detailed timing model for the memory system. Ruby was modified to include the proposed unified signature design described in Section 3.

The base CMP system consists of 16 in-order, single-issue cores with a 32KB split, 4-way associative, 64B block private L1 cache each. L2 cache is unified,

Table 4. Workloads: Input parameters and TM characteristics

Bench	Input	#xact	Time in xact	avg $\|RS\|$	avg $\|WS\|$	max $\|RS\|$	max $\|WS\|$
Bayes	-v32 -r1024 -n2 -p20 -s0 -i2 -e2	523	94%	76.9	40.9	2067	1613
Genome	-g512 -s64 -n8192	30304	86%	12.1	4.2	400	156
Intruder	-a10 -l128 -n128 -s1	12123	96%	19.1	2.5	267	20
Kmeans	-m40 -n40 -t0.05 -i rand-n1024-d1024-c16	1380	6%	99.7	48.5	134	65
Labyrinth	-i rand-x32-y32-z3-n64	158	100%	76.5	62.9	278	257
SSCA2	-s13 -i1.0 -u1.0 -l3 -p3	47295	19%	2.9	1.9	3	2
Vacation	-n4 -q60 -u90 -r16384 -t4096	24722	97%	19.7	3.6	90	30
Yada	-a20 -i 633.2	5384	100%	62.7	38.4	776	510

8MB, 16-bank, 8-way associative, and 64B block size. A packet-switched interconnect with 64B links connects the cores and cache banks. Cache coherence implements the MESI protocol and maintains an on-chip directory which holds a bit vector of sharers. Main memory is 4GB.

Simulation experiments use perfect signatures (no false positives, hardware unimplementable) as the reference. Filter size ranges from 64 bits, which matches the word length in SPARC architecture, to 8K bits length, which matches the performance of perfect signatures for the simulated workloads. All filters use 4 hash functions of the H3 family [3]. Same H3 matrices of Ruby were used.

The benchmarks belong to the Stanford's STAMP suite [2] which is designed for TM research and includes a wide range of applications with emphasis on large read and write sets. STAMP benchmarks have been adapted to GEMS by applying Luke Yen's patches from the University of Wisconsin, Madison. Table 4 summarizes the input parameters and main transactional characteristics of the benchmarks.

6.2 Unified Signature Results

Unified signature motivation and design are described in Section 3. Table 1 shows that the percentage of addresses both read and written inside transactions is substantial, so we conducted the experiments to find out the number of hash functions that can be shared by read and write filters without losing performance. For that purpose, shared functions range from $s = 0$, all SRAMs are double-ported, to $s = 4$, all SRAMs are single-ported which means that every insertion into the read set is also an insertion into the write set and vice-versa.

Figure 3 shows the execution time of unified signatures. The more read set and write set hash functions are shared ($s > 0$) the better results are obtained for all the benchmarks. In fact, the best results are obtained for $s = 4$ in every benchmark except Bayes and Genome, which execution is slowed down about 1.25× with respect to separate filters for 8Kbit signatures. Therefore, unified $s = 3$ signatures should be used instead of $s = 4$ ones, as these benchmarks are not pretty sensitive to read and write discrimination but other might be.

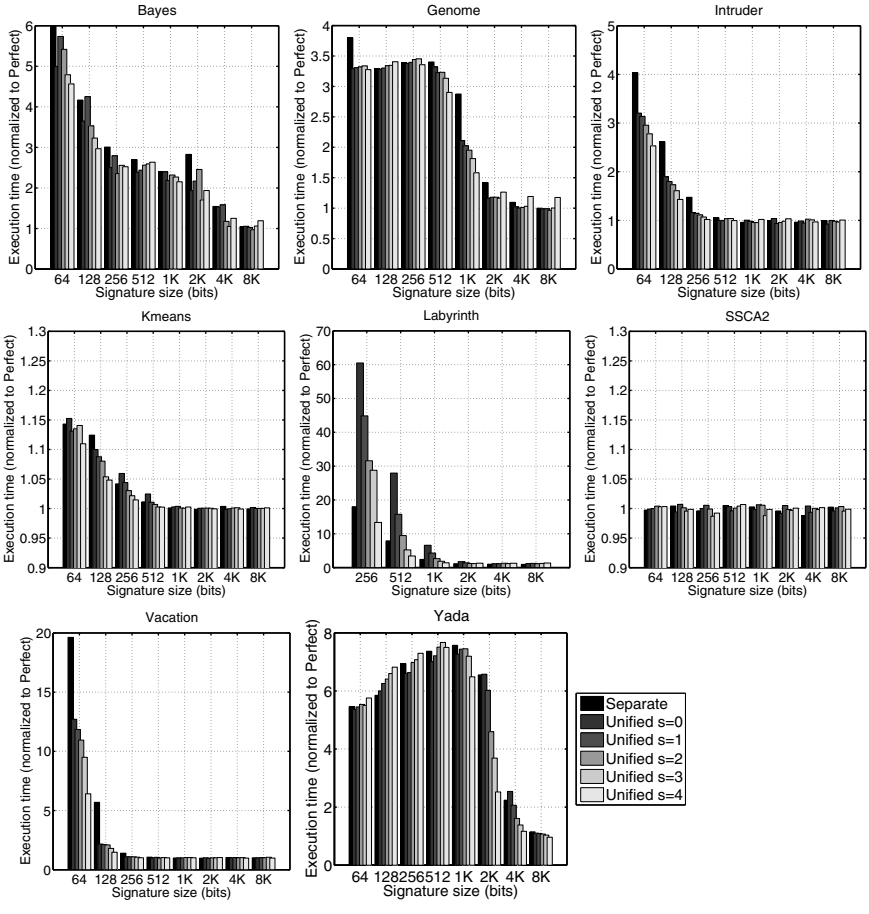


Fig. 3. Execution time normalized to perfect signature comparing separate to unified signatures. Parameter s varies from 0 (2-ported SRAMS) to 4 (1-ported).

6.3 Unified Locality-Sensitive Signature Results

Locality-sensitive hashing [14] takes advantage of locality of reference to store an address stream more concisely in a Bloom filter. Locality-sensitive hash functions store nearby locations sharing some bits of the bit array, thus lowering the occupancy of the filter. For contiguous addresses, the number of hashing outputs with different values is 1. Addresses with distance 2 are different in no more than 2 hashing outputs and, addresses with distance greater than $2^{k-1} - 1$ may have no hashing outputs in common.

Figure 4 shows the results of unified $s = 3$ locality-sensitive signatures. Two different possibilities are shown:

– L1: This scheme makes that the hash functions h_3 and h'_3 assert less bits in their filter. This reduces the false positive rate because of low occupancy,

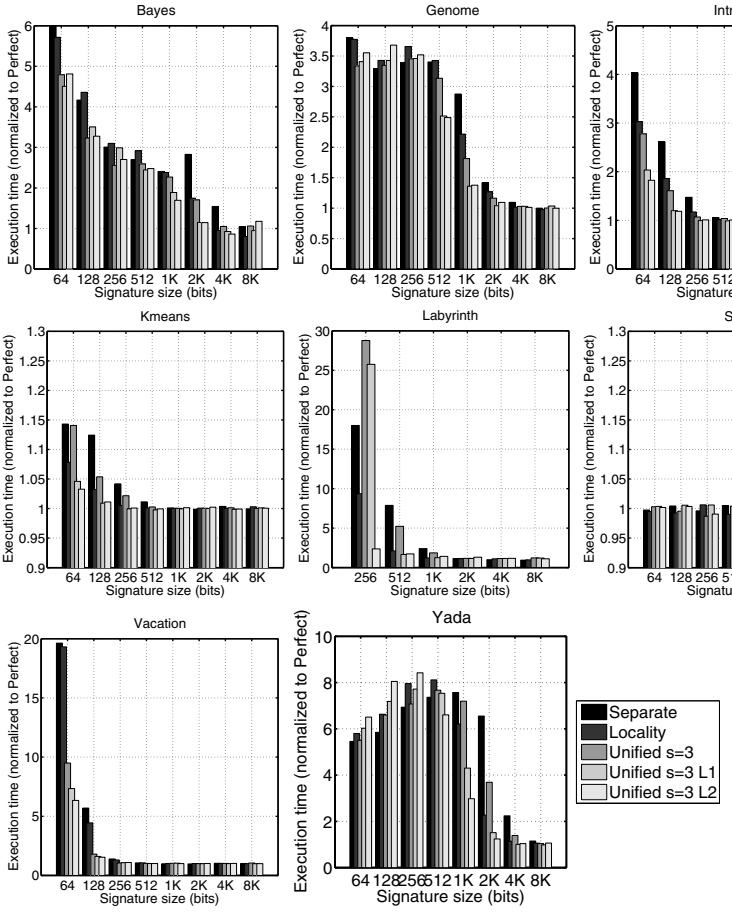


Fig. 4. Execution time normalized to perfect signatures comparing separate, separate locality and unified $s = 3$ signatures enhanced with locality hashing (L1 and L2)

but the filter may fail to discriminate reads/writes from nearby located reads/writes.

- L2: This scheme is the opposite to L1. In this case, h_3 and h'_3 behaves as normal but the others assert less bits. The filter not sharing the hash functions stay the same as in $s = 3$ configuration, discriminating between locations read and written, and the other filters get the locality improvement.

Figure 4 shows similar results for L1 and L2 schemes for all benchmarks except Labyrinth, Genome and Yada. Labyrinth behaves better with L2 for small signatures and, Genome and Yada get slightly worse results for small signatures and L2. Unified locality-sensitive signatures outperform separate ones for the majority of the tested codes.

7 Conclusions

We propose a unified signature design in the context of transactional memory which keeps track of both the read and write sets in the same filter without adding significant hardware complexity. Several configurations of unified signatures are analyzed and evaluated. Additionally, unified signatures are enhanced using locality-sensitive hashing, proposed by the authors in a previous work.

We used the Wisconsin GEMS to implement and evaluate the performance of the proposed unified locality-sensitive signatures. Besides, we used CACTI to evaluate the hardware area and power requirements. Experimental results show that the proposed approach improves the execution performance in most of the benchmark codes, with an insignificant increase in hardware area and a slight increase in power consumption, making of it a good alternative to separate signatures.

Acknowledgment

We would like to thank Dr. Luke Yen (AMD) for providing his patches to adapt STAMP workloads to GEMS simulator. This work has been supported by the Ministry of Education of Spain with project CICYT TIN2006-01078 and by the Junta de Andalucia with project P08-TIC-04341.

References

1. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (1970)
2. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing. In: IEEE Int'l Symp. on Workload Characterization (IISWC 2008), pp. 35–46 (2008)
3. Carter, L., Wegman, M.: Universal classes of hash functions. J. Computer and System Sciences 18(2), 143–154 (1979)
4. Ceze, L., Tuck, J., Montesinos, P., Torrellas, J.: BulkSC: Bulk enforcement of sequential consistency. In: 34th Ann. Int'l. Symp. on Computer Architecture (ISCA 2007), pp. 278–289 (2007)
5. Ceze, L., Tuck, J., Torrellas, J., Cascaval, C.: Bulk disambiguation of speculative threads in multiprocessors. In: 33th Ann. Int'l. Symp. on Computer Architecture (ISCA 2006), pp. 227–238 (2006)
6. Choi, W., Draper, J.: Locality-aware adaptive grain signatures for transactional memories. In: IEEE Int'l. Symp. on Parallel and Distributed Processing (IPDPS 2010), pp. 1–10 (2010)
7. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: 20th Ann. Int'l. Symp. on Computer Architecture (ISCA 1993), pp. 289–300 (1993)
8. Larus, J., Rajwar, R.: Transactional Memory. Morgan & Claypool Pub. (2007)
9. Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B., Werner, B.: Simics: A full system simulation platform. IEEE Computer 35(2), 50–58 (2002)

10. Martin, M.M.K., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D., Wood, D.A.: Multifacet's general execution-driven multiprocessor simulator GEMS toolset. ACM SIGARCH Comput. Archit. News 33(4), 92–99 (2005)
11. Mehrara, M., Hao, J., Hsu, P.-C., Mahlke, S.: Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In: ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2009), pp. 166–176 (2009)
12. Minh, C.C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., Olukotun, K.: An effective hybrid transactional memory system with strong isolation guarantees. In: 34th Ann. Int'l. Symp. on Computer Architecture (ISCA 2007), pp. 69–80 (2007)
13. Muralimanohar, N., Balasubramonian, R., Jouppi, N.: CACTI 6.0: A tool to model large caches. Tech. Rep. HPL-2009-85, HP Laboratories (2009)
14. Quislant, R., Gutierrez, E., Plata, O., Zapata, E.: Improving signatures by locality exploitation for transactional memory. In: Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT 2009), pp. 303–312 (2009)
15. Ramakrishna, M.V., Fu, E., Bahcekapili, E.: Efficient hardware hashing functions for high performance computers. IEEE Trans. on Computers 46(12), 1378–1381 (1997)
16. Sanchez, D., Yen, L., Hill, M., Sankaralingam, K.: Implementing signatures for transactional memory. In: 40th Ann. IEEE/ACM Int'l Symp. on Microarchitecture (MICRO 2007), pp. 123–133 (2007)
17. Shriraman, A., Dwarkadas, S., Scott, M.L.: Flexible decoupled transactional memory support. In: 35th Ann. Int'l. Symp. on Computer Architecture (ISCA 2008), pp. 139–150 (2008)
18. Wilton, S.J.E., Jouppi, N.P.: CACTI: an enhanced cache access and cycle time model. IEEE Journal of Solid-State Circuits 31(5), 677 (1996)
19. Yen, L., Bobba, J., Marty, M.R., Moore, K.E., Volos, H., Hill, M.D., Swift, M.M., Wood, D.A.: LogTM-SE: Decoupling hardware transactional memory from caches. In: 13th Int'l. Symp. on High-Performance Computer Architecture (HPCA 2007), pp. 261–272 (2007)
20. Yen, L., Draper, S.C., Hill, M.D.: Notary: Hardware techniques to enhance signatures. In: 41st Ann. IEEE/ACM Int'l Symp. on Microarchitecture (MICRO 2008), pp. 234–245 (2008)

Using Runtime Activity to Dynamically Filter Out Inefficient Data Prefetches

Oussama Gamoudi, Nathalie Drach, and Karine Heydemann

UPMC/LIP6, Paris, France
{oussama.gamoudi,nathalie.drach,karine.heydemann}@lip6.fr

Abstract. Data prefetching is an effective way to bridge the increasing performance gap between processor and memory. Prefetching can improve performance but it has some side effects which may lead to no performance improvement while increasing memory pressure or to performance degradation. Adaptive prefetching aims at reducing negative effects of prefetching while keeping its advantages. This paper proposes an adaptive prefetching method based on runtime activity, which corresponds to the processor and memory activities retrieved by hardware counters, to predict the prefetch efficiency. Our approach highlights and relies on the correlation between the prefetch effects and runtime activity. Our method learns all along the execution this correlation to predict the prefetch efficiency in order to filter out predicted inefficient prefetches. Experimental results show that the proposed filter is able to cancel the negative impact of prefetching when it is unprofitable while keeping the performance improvement due to prefetching when it is beneficial. Our filter works similarly well when several threads are running simultaneously which shows that runtime activity enables an efficient adaptation of prefetch by providing information on running-applications behaviors and interactions.

1 Introduction

Data prefetching, either hardware-based or compiler-assisted, is an effective way to bridge the increasing performance gap between processor and memory. Prefetching has the potential to improve performance if the memory access patterns are correctly predicted and if prefetching requests are initiated early enough before the program accesses the predicted memory addresses. However, when the prediction is wrong, when prefetches are issued too early or when the access demands are numerous, prefetching may lead to cache pollution by occupying cache space, by evicting useful data and uselessly generate a higher pressure on memory. In some cases, prefetching may even degrade performance.

Hardware prefetching can be effective for some applications whereas any configuration of a prefetching scheme may not be beneficial to some other applications [12]. As an example, for the next sequence prefetcher (NSP) [11], the configuration determines by how far in advance data are preteched, named prefetch distance, and the amount of prefetched data, named prefetch degree and is also

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 338–350, 2011.

called prefetcher aggressiveness. Figure 1 shows the performance improvement for different levels of aggressiveness of a next sequence prefetcher (the aggressiveness of the prefetcher varies from not aggressive to very aggressive). Figure 1 groups benchmarks into two classes: the *prefetch friendly benchmarks* for whom at least one aggressive prefetching improves performance by more than 20%, and the *prefetch unfriendly benchmarks* for whom none of the prefetching schemes increases performance, or for whom prefetching tends to degrade performance. Prefetch unfriendly benchmarks have irregular behaviors: irregular control flow and/or irregular memory accesses. These behaviors are specific to an application, in addition prefetching effects vary according to application execution phases in which the number and the patterns of data accesses change [8]. Moreover, some applications, either prefetch friendly or unfriendly, can be negatively impacted by prefetching when their execution is disturbed by applications running simultaneously. In order to limit any performance degradation and useless memory bandwith usage, so to reduce the number of unsuitable prefetches, it is necessary to dynamically adapt prefetch to match it to all running programs behaviors and phases.

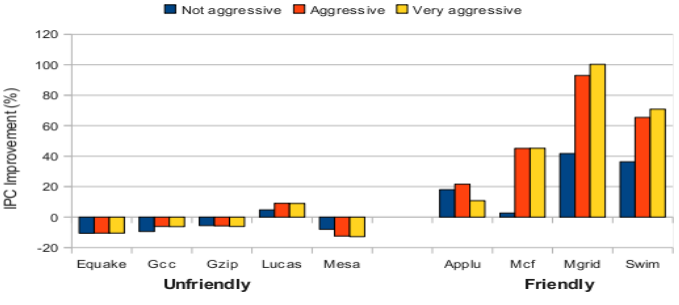


Fig. 1. IPC (instructions per cycle) improvement for different aggressivenesses of a next sequence prefetcher

Previous works [12,14,13] on adaptive prefetching techniques are mostly based on prefetch efficiency metrics, such as prefetch accuracy, to adapt the prefetch policy. In this paper, we propose an adaptive prefetching technique that uses in addition to a prefetch efficiency metric runtime information, we call *runtime activity*, to filter prefetching requests. Runtime activity gives some insight on the running applications behaviors (irregular/regular control, high memory activity,...). A runtime activity is a vector of events (number of L1/L2 cache misses, of executed load/store, of executed branches, ...) recorded during a *sampling interval* or *interval* thanks to hardware counters present in current hardware architectures. As shown in this paper, there exists a correlation between the runtime activity and the prefetch efficiency. Our approach continuously learns on-line this correlation to predict the outcome of prefetches. Thus, predicted ineffective prefetches are dynamically filtered out (cancelled).

Our experimental results show that our adaptive method enables to filter out predicted inefficient prefetches, and so to reduce the negative effects of prefetching while keeping the high performance improvement for prefetch friendly applications. Indeed, our filter is able to cancel 90% of the inefficient prefetches for prefetch unfriendly benchmarks and even to improve performance in presence of prefetching up to 18% for some of them. We also show that our filter works similarly well when several threads are running simultanaously: more than 70% of the inefficient prefetches are eliminated, which shows the relevancy of using runtime information.

The rest of this paper is organized as follows. The section 2 describes related work. Section 3 discusses the correlation between runtime activity and prefetch efficiency. Section 4 describes our adaptive prefetching method. Section 5 presents experimental results. Finally, conclusions are presented in section 6.

2 Related Works

Adaptive prefetch has been investigated to improve prefetch efficiency and performance. Adaptive prefetching techniques can be classified into two categories, *the adaptive dependent techniques* and *the adaptive independent techniques*.

The first category adapts dynamically the prefetch algorithm (parameter values). S. Srinath et al. [12] proposed to reduce the negative performance of prefetching by dynamically adjusting the aggressiveness (prefetch distance and/or degree) of the hardware prefetcher with rules using prefetch accuracy, prefetch lateness and an estimation of cache pollution. Saavedra et al. [10] proposed an adaptive scheme for software prefetching: prefetching instructions are inserted at compile time. At runtime, the prefetch distance is increased or reduced depending on the prefetch accuracy and memory latency periodically estimated. Nesbit et al. [8] proposed to detect program phase changes by using an instruction working set signature. Then, when a stable phase is detected, prefetcher's parameters are tuned according to the average number of instructions per cycle (IPC). Dahlgren et al. [3] proposed to increment or decrement prefetch degree for the next block prefetching scheme with simple threshold rules using prefetch accuracy.

Adaptive dependent methods reduce the negative effects of prefetching by adjusting the prefetcher parameters or aggressiveness [12,10]. However, with the usage of a new prefetcher, these methods need to be redesigned to consider the new prefetcher's parameters. Hence, the second category, the adaptive independent techniques which do not act on the prefetch algorithm, has the advantage to be completely transparent to the prefetching mechanism. Moreover, in presence of more than one prefetcher as implemented in some recent processors [1], an adaptive independent approach enables to control over them. Some adaptive independent techniques try to filter prefetches by predicting if a prefetch request will be inefficient. X. Zhuang et al. [14] proposed a hardware-based prediction mechanism of prefetches efficiency using a history-based table

similar to a two-level branch predictor. Two table indexing schemes have been proposed, an address-based one (PA-based) and a program counter based one (PC-based). These schemes reduce ineffective data prefetches by more than 90% when combined with a Next Sequence Prefetcher (NSP) and Shadow Directory Prefetcher (SDP). However, their schemes seem not robust to encountered inefficient prefetch instructions or prefetched data addresses that turn out to be efficient afterwards during execution. Some other works addressed the issue of cache pollution due to prefetching independently of the prefetcher by adapting the cache replacement policy either with rules based on cache pollution, prefetch accuracy and lateness [12], either by predicting dead-blocks [6,5] or by using a metric of usefulness of prefetched blocks [7]. These techniques could be profitably combined with our method, they are complementary.

3 Correlation between Runtime Activity and Prefetch Efficiency

As said in the introduction, prefetching efficiency varies according to program phases during execution and depends on running applications behaviors and their interactions. In this section, we show that runtime information able to express these characteristics involved in prefetch efficiency can be used to predict prefetching outcome.

Runtime information, we call runtime activity, is a vector of events (number of L1/L2 cache misses, of executed load/store, of executed branches, ...) recorded during a sampling interval. Many events can be recorded and various combinations of them are related to prefetch efficiency. However, it is necessary to keep the number of events as low as possible for implementation considerations. We empirically reduced the set of events related to runtime activity involved in prefetch efficiency such as irregular/regular control, speculation, high memory activity, etc. to five events: 1) the number of executed load/store 2) the number of executed branches 3) number of hit-predicted branches, 4) the number of L1 cache misses and 5) the number of L2 cache misses.

Runtime activity can be used to predict the prefetch efficiency if it is different before/during phases where prefetching turns out to be efficient and during/before phases where prefetching turns out to be inefficient. As runtime activity is recorded during an interval, an interval duration able to record a significant activity and able to discriminate prefetch efficiency must be determined. To do so, we simulated different benchmarks and recorded runtime activity and the prefetch efficiency for various interval durations.

As adaption decision computation requires few cycles (see section 4) and for hardware implementation reasons, a filtering decision could not be taken at each prefetch request. Thus, a filtering decision is to be computed periodically and taken for a small while. Therefore, execution is divided into sampling intervals during which runtime activity and prefetch efficiency are recorded. The prefetching intervals are classified into two classes depending on the prefetch efficiency during them : the *good prefetching intervals* correspond to intervals

Table 1. Percentage of similar runtime activities assigned to both good and bad prefetching intervals depending on the sampling interval duration given in cycles

interval duration	10	1000	2500	5000	7500	10000	20000	50000	100000
similar activities	65.5%	12.72%	3.81%	1.03%	1.04%	1.02%	2.02%	77.98%	96.37%

where there are more pending and useful prefetches than the evinced ones, and the *bad prefetching intervals* to the opposite case. We assigned to a runtime activity recorded during an interval T the prefetching interval efficiency retrieved at the end of the next interval T + 1.

Table 1 shows the percentage of similar runtime activities that correspond to both good and bad prefetching intervals for increasing interval durations. A low percentage means that runtime activity is different before good and bad prefetching intervals. Therefore, the runtime activity is relevant to predict its efficiency. Results show that this percentage is less than 4% for interval durations ranging 2,500 cycles to 10,000 cycles. So, there exists a correlation between runtime activity and prefetching intervals efficiency for such interval durations. Indeed, in our experiments, the prefetching intervals of these durations are in average composed of 87.3% of the same kind of prefetches (good or bad). Such interval durations are then smaller than the duration of program phases where prefetching is either efficient or inefficient. When increasing the duration of the sampling interval (more than 20,000 cycles), the number of good and bad prefetches included in an interval becomes more and more close. Thus, interval duration must not be too large compared to phases duration. Also, a too small interval is not able to record significant activity.

In the remainder, a sampling interval of 5,000 cycles is considered in the experiments. This interval duration allows: 1) several prefetches to be issued, 2) to discriminate both classes of prefetching effects, 3) a more reactive adaptation by taking an adaptation decision more frequently than with a larger interval, and 4) to compute the prediction and take the adaptation decision.

4 Adaptive Prefetching Method Based on Runtime Activity

In this section, we present our adaptive approach which uses the correlation between runtime activity and the prefetch efficiency intervals to filter prefetches. A filtering decision is taken for an interval duration and consists in cancelling all prefetches during the corresponding interval or left them being issued.

Principle. Execution time is divided into sampling intervals during which runtime activity as well as the prefetch efficiency, when prefetches are not cancelled, are recorded. At the end of each sampling interval, the A_{cur} runtime activity recorded during this interval is used to compute a filtering decision thanks to a predictor. The prediction of prefetch efficiency uses a n-entry activity table

AT of runtime activities and their corresponding prefetch efficiency. The filtering decision is taken by computing the distance between the current activity and all activities recorded in the activity table. The closest activity is selected. If it is close enough, the filtering decision for the next interval depends on the prefetch efficiency class associated with it: if the interval is a bad one, prefetches are cancelled during the next interval, in the opposite, prefetches are issued as if no filter was present. Otherwise, the current activity is recorded in the activity table, prefetches are launched during the next interval and the prefetch efficiency recorded during the next interval is assigned to the newly recorded activity. Thus, prefetching is either enabled (not filtered out) or disabled (filtered out) during the next interval.

The predictor uses a n-entry activity table AT whose each entry contains a runtime activity A (the counter values recorded during one sampling interval T) and the prefetch efficiency assigned to it (retrieved at the end of the sampling interval T+1). An activity A recorded in the table is viewed as the center of a region in the events space (space of events values or of activities) leading to the same prefetching effect. It is necessary to learn a right size for each recorded activity. Thus, each entry also contains a distance D which defines the size of the zone in the events space centered in A for which A can give a filtering decision: a current activity is said close enough to A if the distance $d(A, A_{cur}) <= D$. By default, D is set to a predefined value $Dmax$ and D is decremented each time the entry has led to a misprediction. Moreover, as some good and bad recorded runtime activities may be close to each others in the events space, using only the distance D for prediction is not robust enough. Hence, a training phase is used to encounter several times activities close to each others in the events space and leading to the same prefetching effect. Therefore, a confidence level C is associated to each recorded activity to indicate if the entry is mature enough to predict or is still in the learning phase. While the confidence level is greater than zero the entry cannot be used for prediction. The confidence level is set to a predefined value $Cmax$ at the insertion of an activity in the table and is decremented each time it is the nearest activity of the current activity and the prefetching efficiency of the following interval is the same as the one already recorded. Otherwise, the distance D of the immature entry is decremented. A counter used by the replacement policy and needed at each new entry insertion while the table is full is also associated with each entry.

To sum up, with our approach prefetching is enabled either because all activities in the table are too far, because the nearest activity enables prefetching or is not mature yet.

Parameters and distance function. We studied empirically the parameters $Dmax$ and $Cmax$ of our mechanism. Since this requires an exponential number of simulations in terms of combinations of parameters, we studied the effects of parameters one by one. We empirically selected the value 20 for $Dmax$. We set $Cmax$ equal to 3: an higher value requires a too long learning phase leading to

too few filtering decisions and a lower one is not robust enough. We also empirically compared the Euclidean and the Manhattan distance functions for the computation of the distance between the current activity and those recorded in the activity table. Our study showed that both distance functions are equivalent in terms of prediction rate. So, as the Euclidean distance is more costly in terms of hardware implementation and CPU-cycles, we choose the Manhattan distance as the distance function.

Filter decision computation time. The determination of the nearest activity requires n distance computations (between the current activity and the n activities stored in the n-entry activity table) and $n - 1$ comparisons to find the minimal distance. However, comparisons can be done in parallel with the distance computation. Assuming that a comparison requires in one cycle and distance computation p CPU-cycles, then around $n * p$ cycles are needed to determine the nearest activity. It is then important to keep $n * p$ small compared to the sampling interval duration. The size n of the activity table is discussed in section 5.2.

Hardware cost. In order to collect feedback on the prefetch efficiency, it is possible to use only one counter to record the difference in good and bad prefetches as follows: one control bit is affiliated to each prefetched cache line. This bit is used to mark if a line was prefetched (1 for prefetched data, 0 otherwise). When a cache line whose control bit is set is referenced by the processor, the prefetch counter is incremented and the control bit is reset. If a cache line with a set control bit is replaced, the prefetch counter is decremented and the control bit is reset. Hence, at the end of a sampling interval, if the prefetch counter is positive, then the runtime activity observed in the previous interval is marked as good. Otherwise, the runtime activity is marked as bad. The prefetch counter is reset at the end of each sampling interval.

The activity table AT is used to keep relevant activities. Each entry of AT requires 2 bytes per event counter so 10 bytes for the runtime activity, 1 bit for the class membership (1 for good, and 0 for bad), $\lceil log2(Cmax) \rceil$ that is 2 bits for the confidence level of an activity : any entry has to be encountered three times with the same prefetch outcomes to be considered as mature i.e. to be used for prediction, $\lceil log2(Dmax) \rceil$ that is 5 bits for the distance threshold, and $log_2(n)$ bits for the slot dedicated to the replacement policy (LRU), giving the relevance of the entry, where n is the size of the activity table.

To model activity, our approach needs 5 hardware counters to record the 5 events involved in the prediction. Also, a register is needed to keep a pointer to the nearest or to a newly added activity. This pointer enables to update the corresponding entry of the table at the end of the next interval, once the prefetch efficiency is known. Moreover, an additionner and a comparator are needed to compute distances, to determine the minimal one and also for the replacement policy.

5 Experimental Evaluation

5.1 Experimental Environment

Simulation environment. For the experiments, we used the M5 simulator [2] configured as a SMT processor with a 2GHz clock. The simulated SMT processor settings are specified in Table 2. The hardware prefetch generator can be triggered by data accesses either to the L1 or to the L2 cache, in our work the hardware prefetcher is triggered by L1 cache accesses. As in the evaluation framework MicroLib [9], the Stride Prefetcher (SP) [4] is among those that have the best performance, low cost and power consumption, we added it to the simulation tools. Moreover, we also considered the Next Sequence Prefetcher (NSP) since it is the best among those that prefetch into L1 cache as shown in [9]. We selected a fixed prefetch distance and a fixed degree that improves the IPC performance for each prefetch friendly benchmark, and that shows the least worst IPC performance for each prefetch unfriendly benchmark. The M5 simulator was also added with our adaptive prefetch filter that executes concurrently with the main processor simulator.

Simulation methodology. We use the SPEC CPU2000 benchmarks for our experimental evaluation. As discussed in Section 1, the considered benchmarks are classified into two groups: prefetch friendly benchmarks and prefetch unfriendly benchmarks. Each benchmark was compiled using GCC with the *-O4* option. We ran each benchmark with the reference input set up to 500-millions instructions when running alone or with other applications. Each benchmark was run: 1) alone as in a single-thread wide-issue processor 2) with a benchmark from the same benchmark group and 3) with a benchmark from a different benchmark group. The results present a subset of all benchmark pairs that covers the behaviors encountered among all pairs.

Table 2. Microarchitecture parameter values

CPU frequency	2GHz
Fetch, decode, issue width	4 instructions per cycle
Instruction windows	128-RUU, 128-LSQ
BTB entries	4096
SMT commit policy	round-robin
L1 Icache	32KB, 4-way, 32 bytes, 1 cycle hit latency, 1 read port
L1 Dcache	32KB, 4-way, 64 bytes, 2 cycle hit latency, LRU, writeback, 4 read ports, 1 write port
L1 data MSHRs	8
Unified L2 cache	1MB, 4-way, 64 bytes, 12 cycles hit latency, LRU, writeback, 1 read port, 1 write port
L2 MSHRs	8
Bus frequency	600 MHz, width 64 bytes
Memory latency	150 cycles
Stride prefetcher	512 PC-entries
Prefetch request queue	64 entries - access demand have an higher priority

5.2 Experimental Results

In order to evaluate our approach, we considered different metrics presented in Tables 4 and 3. We considered the IPC improvement with and without our method relatively to no prefetch (third and fourth columns of tables), the IPC improvement with our method relatively to without filtering (fifth column of tables). We also computed the percentage of reduction of bad prefetch requests and good prefetch requests (sixth and seventh columns respectively). Table 3 shows all these results when a NSP prefetcher is used and Table 4 corresponds to a SP prefetcher.

Reduction of good and bad prefetches. For prefetch unfriendly benchmarks running alone, the bad prefetches are, in average for both prefetchers, reduced by 90% with our method and can be up to 98%. The reduction is in average 55% for other benchmarks when they are running alone. In the case of SMT execution, the reduction of bad prefetches is on average 75%, 87% and 62% for respectively friendly + friendly, unfriendly + unfriendly, and friendly + unfriendly benchmarks. On average 50% and 9% of good prefetches are removed for prefetch unfriendly benchmarks and prefetch friendly benchmarks when they are running alone. In a SMT context, the reduction varies from less than 10% to 95%. Thus, despite the potential of our method to cancel bad prefetches, it can not cancel all bad prefetches, and in some cases it disables many good prefetches. Indeed, when an interval is a good (resp. bad) one, it does not exclude

Table 3. Experimental results using a NSP prefetcher

Benchmarks	Type	$\frac{IPC_{pref}}{IPC_{wo\_pref}}$%	$\frac{IPC_{filter}}{IPC_{wo\_pref}}$%	$\frac{IPC_{filter}}{IPC_{pref}}$%	Bad pref. reduction	Good pref. reduction
Equake	U	-10.5	1.2	13.1	98.4	95.1
Gcc	U	-5.5	0.8	6.7	85.6	27.5
Gzip	U	-5.7	0.4	6.5	90.2	83.1
Lucas	U	9	9	-0.04	96.1	5
Mesa	U	-12.4	3.4	18.1	99.1	89.5
Average	**U**	**-5**	**3**	**8.4**	**93.9**	**60**
Applu	F	21.6	21.7	0.02	79	20.4
Mcf	F	45.1	44.8	-0.23	86.2	14.0
Mgrid	F	100.3	99.2	-0.51	53.7	1
Swim	F	70.8	70.6	-0.11	23.6	7
Average	**F**	**59.4**	**59.1**	**-0.24**	**60.6**	**10.6**
Applu + Swim	F + F	48,6	48,6	0	34.1	9.1
Mgrid + Applu	F + F	73	70	-1.72	94.1	43.3
Swim + Mgrid	F + F	94.9	93.5	-0.7	90.3	55.4
Mcf + Applu	F + F	09.1	19.4	9.5	94.1	25.3
Average	**F + F**	**56.4**	**57.9**	**0.96**	**78.2**	**33.3**
Mgrid + Equake	F + U	35.6	37.4	1.33	39.3	8.2
Swim + Lucas	F + U	60.4	61	0.36	48.9	11.2
Mgrid + Gzip	F + U	37.3	39.9	1.86	56.2	17.7
Average	**F + U**	**44.4**	**46.1**	**1.14**	**48.1**	**12.4**
Lucas + Gzip	U + U	25.8	25	-0.6	41.8	12.9
Gcc + Equake	U + U	-14.2	0.8	17.4	98.4	92.1
Gcc + Mesa	U + U	-6	1.5	8	98.8	73.4
Mesa + Lucas	U + U	13	17	3.5	95.4	25.8
Average	**U + U**	**4.7**	**11.1**	**5.6**	**83.6**	**51.1**

the presence of bad (resp. good) prefetches. For some benchmarks like Gcc + Gzip or Equake most of the good prefetches are filtered due to the low number of good prefetches and their inclusion in the bad prefetching intervals. Thus, the heterogeneity of intervals causes the privation of some good prefetches or the achievement of some bad prefetches whatever the accuracy of the prediction. Moreover, during the training phase bad prefetches are issued.

Nevertheless, these results show that our adaptive prefetch based on runtime activity is able to dynamically highly reduce the number of bad prefetches when one or several benchmarks are running simultaneously.

Performance impact. The third and fourth columns of Tables 3 and 4 show the IPC improvement with and without filter with respect to no prefetching.

For prefetch unfriendly benchmarks running alone, whatever the prefetcher used, our filter avoids any IPC degradation compared to no prefetching. In the case of a NSP prefetcher, our filter improves IPC resulting from prefetching up to 18% (on *Mesa*) and by 8.4% on average. Also, our filter improves IPC resulting from prefetching by 9.5% on average when a SP prefetcher is considered. This performance improvement is due to the removal of negative effects of prefetching thanks to the filtering of a large number of bad prefetches. For prefetch friendly benchmarks, the same or almost the same IPC values are observed when our method is enabled or disabled: our approach enables to keep the benefits of prefetching by rightly predicting the good intervals and improves the

Table 4. Experimental results using a SP prefetcher

Benchmarks	Type	$\frac{IPC_{pref}}{IPC_{wo\_pref}}$%	$\frac{IPC_{filter}}{IPC_{wo\_pref}}$%	$\frac{IPC_{filter}}{IPC_{pref}}$%	Bad pref. reduction	Good pref. reduction
Equake	U	-11.2	0.11	12.7	93.7	74.5
Gcc	U	-4.6	1.05	5.9	97.3	83
Gzip	U	-7.7	0.3	8.7	86.6	43
Lucas	U	-4.6	3.4	8.4	76.1	2.5
Mesa	U	-3.7	1.2	16.8	81.4	66.3
Mcf	U	-4.8	0.02	5.1	75.2	9.4
Average	U	**-7.7**	**1.01**	**9.5**	**85**	**46.5**
Applu	F	23,3	23.2	-0.1	87.6	19.9
Mgrid	F	51.3	50.4	-0.6	16	0.4
Swim	F	46.9	45.7	-0.8	33.4	0.02
Average	**F**	**29.2**	**29.8**	**0.5**	**53**	**7.4**
Applu + Swim	F + F	45.7	44.2	-1.04	88.7	33.2
Mgrid + Applu	F + F	51.3	51	-0.2	32.6	16.2
Swim + Mgrid	F + F	71.8	70.1	-1	72.1	35.8
Average	**F + F**	**39.9**	**42.2**	**1.6**	**72.9**	**43.3**
Mgrid + Equake	F + U	19.5	21.8	1.9	72.1	12.3
Swim + Lucas	F + U	45.8	47.3	1	71.3	29.9
Mgrid + Gzip	F + U	23.9	25.2	1.1	62	10.1
Applu + Mcf	F + U	-9.1	3.4	13.7	98.2	88.1
Average	**F + U**	**20.04**	**24.4**	**3.65**	**75.9**	**35.1**
Lucas + Gzip	U + U	16.1	14.3	-1.6	97.1	72.3
Gcc + Equake	U + U	-0.9	2.1	3	96.2	88.6
Mcf + Lucas	U + U	28.8	27.1	-1.33	68.9	32.7
Mesa + Lucas	U + U	0	0.9	0.9	96.8	95.2
Average	**U + U**	**11**	**11.1**	**0.65**	**89.8**	**72.2**

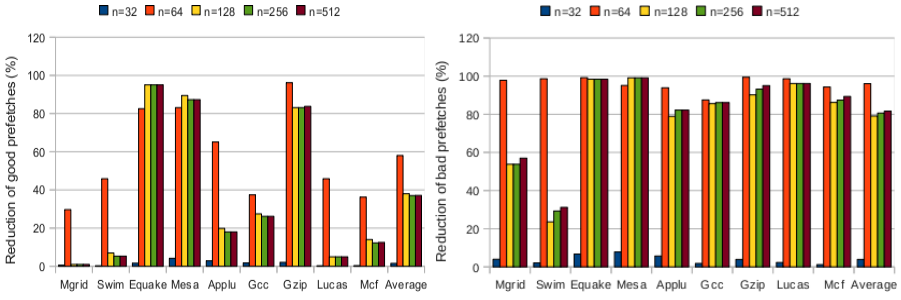


Fig. 2. Reduction of good and bad prefetches for different sizes of the activity table

usage of memory bandwidth due to prefetching thanks to the reduction of inefficient prefetches. In the context of SMT-execution, the results show that some benchmarks derived from prefetch unfriendly benchmarks groups favor prefetching when they are running simultaneously (e.g the IPC improvement for Lucas + Gzip is over 20%). Indeed, as the execution of instructions belonging to several threads are interleaved, the distance between demand accesses may be increased allowing more prefetches to be issued and thus to transform bad prefetches in the single-threaded context into good ones in the SMT-context. To the opposite, some prefetch friendly benchmarks combinations do not profit from prefetching (e.g the IPC improvement for Mcf + Applu is less than 20%) in a SMT context. Indeed, cache sharing may increase the number of evinced prefetched cache lines. Also, the higher memory traffic in a SMT-context can provoke the cancellation of prefetches in the prefetch queue. Therefore, less prefetches may be issued in a SMT-context. As a consequence, the average performance improvement is lower in a SMT-context: IPC is improved less than 3% in average. However, any application performance is no more degraded by prefetching thanks to our filter and the benefit can be up to 17.4% (Gcc + Equake). The runtime activity reflecting the events of all simultaneously running applications enables an efficient filtering of prefetch in such a context.

In summary, these experimental results show that the proposed filter is able to cancel the negative impact of prefetching when it is unprofitable while keeping the performance improvement due to prefetching when it is beneficial. Our filter works similarly well when several threads are running simultaneously which shows that runtime activity enables to efficiently adapt prefetch by providing information on running-applications behaviors and interactions.

Size of the table. As we mentioned in section 4, at the beginning of each interval the current activity is compared to all activities in the activity table AT. AT is a n-entry table. Figure 2 shows the reduction of bad prefetches for different table sizes. The results show that when the size of the table is too small, the reduction of bad prefetches is low: most of activities have not the time to become mature; they are evinced to record activities encountered afterwards. A large table is useful since it allows to keep relevant activities and avoid too many training phases, hence the bad prefetching intervals will be rapidly predicted and

filtered out. Nevertheless, a large table AT is very costly in computation time since all entries must be parsed and it requires more hardware. As shown by the results, a value of n higher than 128 enables a high reduction of bad prefetches. As a table of 128 entries is not expensive both in hardware and filtering decision computation time, we choose this value in our experiments.

6 Conclusion

This paper has proposed a hardware independent adaptive prefetching method based on runtime activity which corresponds to the processor and memory activities retrieved by hardware counters to predict the prefetch efficiency. The method highlights and relies on the correlation between the prefetch effects and runtime activity, which expresses the applications behaviors and their interactions. All along the execution, this correlation is learnt in order to predict the prefetch efficiency to filter predicted inefficient prefetches. Experimental results have shown that the proposed filter is able to cancel the negative impact of prefetching when it is unprofitable while keeping the performance improvement due to prefetching when it is beneficial. The reduction of the bad prefetches is in average of 90% which, in some cases, can improve performance up to 18%. Our filter works similarly well when several threads are running simultaneously which shows that runtime activity enables an efficient adaptation of prefetch by providing information on running-applications behaviors and interactions.

References

1. Intel 64 and IA-32 Architectures Optimization Reference Manual. Intel Corporation (2009)
2. Binkert, N.L., Dreslinski, R.G., Hsu, L.R., Lim, K.T., Saidi, A.G., Reinhardt, S.K.: The m5 simulator: Modeling networked systems. IEEE Micro 25(1), 52–60 (2006)
3. Dahlgren, F., Dubois, M., Stenstrm, P.: Sequential hardware prefetching in shared-memory multiprocessors. IEEE Trans. Parallel Distrib. Syst. 25(1), 733–746 (1995)
4. Fu, J.W.C., Patel, J.H., Janssens, B.L.: Stride directed prefetching in scalar processors. In: MICRO 25: Proceedings of the 25th Annual International Symposium on Microarchitecture, pp. 102–110. IEEE Computer Society Press, Los Alamitos (1992)
5. Hu, Z., Kaxiras, S., Martonosi, M.: Timekeeping in the memory system: predicting and optimizing memory behavior. In: ISCA 2002: Proceedings of the 29th Annual International Symposium on Computer Architecture, pp. 209–220. IEEE Computer Society, Washington, DC, USA (2002)
6. Lai, A.-C., Fide, C., Falsafi, B.: Dead-block prediction & dead-block correlating prefetchers. In: ISCA 2001: Proceedings of the 28th Annual International Symposium on Computer Architecture, pp. 144–154. ACM, New York (2001)
7. Mutlu, O., Kim, H., Armstrong, D.N., Patt, Y.N.: Cache filtering techniques to reduce the negative impact of useless speculative memory references on processor performance. In: SBAC-PAD 2004: Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing, pp. 2–9. IEEE Computer Society Press, Los Alamitos (2004)

8. Nesbit, K.J., Dhodapkar, D., Smith, J.E.: Ac/dc: An adaptive data cache prefetcher. In: PACT 2004: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, pp. 135–145. IEEE Computer Society, New York (2004)
9. Perez, D.G., Gilles, M., Temam, O.: Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In: MICRO 37: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 43–54. IEEE Computer Society, New York (2004)
10. Saavedra, R.H., Park, D.: Improving the effectiveness of software prefetching with adaptive execution. In: PACT 1996: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques, p. 68. IEEE Computer Society, Washington, DC, USA (1996)
11. Smith, A.J.: Cache memories. Computing Surveys 14(3) (1982)
12. Srinath, S., Mutlu, O., Kim, H., Patt, Y.N.: Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In: HPCA 2007: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture, pp. 63–74. IEEE Computer Society, Washington, DC, USA (2007)
13. Srinivasan, V., Davidson, E.S., Tyson, G.S.: A prefetch taxonomy. IEEE Trans. Comput. 53(2), 126–140 (2004)
14. Zhuang, X., Lee, H.-H.S.: Reducing cache pollution via dynamic data prefetch filtering. IEEE Trans. Comput. 56(1), 18–31 (2007)

Introduction

Salvatore Orlando, Gabriel Antoniu, Amol Ghoting, and Maria S. Perez

Topic chairs

The manipulation and handling of an ever increasing volume of data by current data-intensive applications require novel techniques for efficient data management. Despite recent advances in every aspect of data management (storage, access, querying, analysis, mining), future applications are expected to scale to even higher degrees, not only in terms of volumes of data handled but also in terms of users and resources, often making use of multiple, pre-existing autonomous, distributed or heterogeneous resources. The notion of parallelism and concurrent execution at all levels remains a key element in achieving scalability and managing efficiently such data-intensive applications, but the changing nature of the underlying environments requires new solutions to cope with such changes. In this context, this topic sought papers in all aspects of data management (including databases and data-intensive applications) that focus on some form of parallelism and concurrency. Each paper was reviewed by four reviewers and, after discussion, we were able to select four regular papers.

The accepted papers address relevant issues on various topics such as effective data compression, GPU-based data indexing, distributed collaborative data filtering and parallel query processing.

The paper entitled "Compressing the Incompressible with ISABELA: In-situ Reduction of Spatio-Temporal Data" by S. Lakshminarasimhan et al. proposes an effective method for In-situ Sort-And-B-spline Error-bounded Lossy Abatement (ISABELA) of scientific data that is widely regarded as effectively incompressible. ISABELA achieves an accurate fitting model that guarantees a ¿= 0.99 correlation with the original data and leverages temporal patterns in scientific data to compress data by 85%, while introducing only a negligible overhead on simulations in terms of runtime. The authors demonstrate that the proposed method outperforms existing lossy compression methods, such as Wavelet compression.

The second paper, entitled "kNN Query Processing in Metric Spaces using GPUs" by R. Barrientos addresses the idea of using GPUs to accelerate brute-force searching algorithms for metric-space databases. It shows how to improve existing GPU implementations and explores the viability of using GPUs in this context. The paper discusses the performance of both brute-force and indexing-based algorithms that take into account the intrinsic dimensionality of the elements of the database.

The third paper, entitled "An Evaluation of Fault-Tolerant Query Processing for Web Search Engines" by M. Marin et al. addresses strategies to perform parallel query processing in large scale Web search engines. The paper studies

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 351–352, 2011.

the suitability of such strategies for the case where processor replication is used to improve query throughput and to support fault-tolerance.

The forth paper, entitled "Performance Optimizations for Distributed Collaborative Filtering" by A. Narang et al. focuses on the usage of collaborative-filtering-based recommender systems by Internet-oriented companies for automatic predictions about user interests: the idea is to infer data from information about like-minded users. The paper presents a distributed algorithm that uses collaborative filtering for soft real-time distributed co-clustering. The proposed algorithm is optimized for multi-core cluster architectures.

We take this opportunity to thank the authors who submitted a contribution, the Euro-Par Organizing Committee, as well as the referees whose relevant comments and efforts substantially contributed to the effectiveness of the evaluation process and to the quality of the resulted program for this topic.

Distributed Scalable Collaborative Filtering Algorithm

Ankur Narang, Abhinav Srivastava, and Naga Praveen Kumar Katta

IBM India Research Laboratory, New Delhi
{annarang,abhin122,nagapkat}@in.ibm.com

Abstract. Collaborative filtering (CF) based recommender systems have gained wide popularity in Internet companies like Amazon, Netflix, Google News, and others. These systems make automatic predictions about the interests of a user by inferring from information about like-minded users. Real-time CF on highly sparse massive datasets, while achieving a high prediction accuracy, is a computationally challenging problem. In this paper, we present a novel design for soft real-time (less than 10 *sec.*) distributed co-clustering based Collaborative Filtering algorithm. Our distributed algorithm has been optimized for multi-core cluster architectures using pipelined parallelism, computation communication overlap and communication optimizations. Theoretical parallel time complexity analysis of our algorithm proves the efficacy of our approach. Using the Netflix dataset (100M ratings), we demonstrate the performance and scalability of our algorithm on 1024-node Blue Gene/P system. Our distributed algorithm (implemented using OpenMP with MPI) delivered training time of around $6s$ on the full Netflix dataset and prediction time of $2.5s$ on $1.4M$ ratings ($1.78\mu s$ per rating prediction). Our training time is around $20\times$ (more than one order of magnitude) better than the best known parallel training time, along with high accuracy (0.87 ± 0.02 RMSE). To the best of our knowledge, this is the best known parallel performance for collaborative filtering on Netflix data at such high accuracy and also the first such implementation on multi-core cluster architectures such as Blue Gene/P.

1 Introduction

Collaborative filtering (CF) is a subfield of machine learning that aims at creating algorithms to predict user preferences based on past user behavior in purchasing or rating of items [13], [15]. Here, the input is a set of known item preferences per user, typically in the form of a user-item ratings matrix. This $(user * item)$ ratings matrix is typically very sparse. The Collaborative Filtering problem is to find the unknown preferences of a user for a specific item, i.e. an unknown entry in the ratings matrix, using the underlying collaborative behavior of the user-item preferences. Collaborative Filtering based recommender systems are very important in e-commerce applications. They help people find more easily, items that they would like to purchase [16]. This enhances the user experience which typically leads to improvements in sales and revenue. Such systems are also increasingly important in dealing with information overload since they can lead users to information that others like them have found useful. With massive data rates in telecom, finance and other industries, there is a strong need to deliver soft real-time training for CF as it will lead to further increase in customer experience and revenue generation. Hence, soft real-time CF (with less than 10 *sec.*) based recommender systems are very useful.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 353–365, 2011.

Typical approaches for CF include matrix factorization based techniques, correlation based techniques , co-clustering based techniques and concept decomposition based techniques. Matrix factorization [17] and correlation [5] based techniques are computationally expensive hence cannot deliver soft real-time CF. Further, in matrix factorization based approaches, updates to the input ratings matrix leads to non-local changes which leads to higher computational cost for online CF. Concept Decomposition based technique [1] perform spherical k-means followed by least-squares based approximation of the original matrix. This work presents only sequential performance of 13.5 minutes for training of the full Netflix dataset which is far from being considered soft real-time. Co-clustering based techniques [8], [6] have better scalability but have not been optimized to deliver high throughput on massive data sets. [6] presented dataflow parallelism based co-clustering implementation which did not scale beyond 8 cores due to cache miss and in-memory lookup overheads. Moreover, CF over highly sparse data sets leads to lower compute utilization. Further, for large scale distributed / cluster environment (256 nodes and beyond), communication cost can dominate the overall performance and the communication cost becomes worse with increasing size of the cluster, leading to performance degradation. Thus, high computational demand, low parallel efficiency (due to cache overheads and low compute utilization) and high communication cost are the key challenges to achieving high throughput distributed Collaborative Filtering on highly sparse data sets.

In order to optimize the parallel performance, achieve high parallel efficiency and give near real time guarantees, we optimized our distributed algorithm using pipelined parallelism, compute communication overlap and communication optimizations (including topology mapping, steiner node for communication time reduction) for massively parallel multi-core cluster architectures such as Blue Gene/P [1]. In order to maintain high parallel efficiency, our algorithm makes compute vs. communication tradeoffs at various phases of the algorithm. Analytical parallel time complexity analysis proves the scalability provided by our performance optimizations as compared to the naive MPI based approach that has been used in all prior implementations. We evaluated our parallel CF algorithm on the prestigious Netflix Prize data set [3]. Netflix provides around $100M$ ratings (on a scale from 1 to 5 integral stars) along with their dates from 480189 randomly-chosen, anonymous subscribers on 17770 movie titles. On this dataset, we test the hybrid(MPI+OMP) parallel version of our optimized algorithm. We demonstrate around $20\times$ CF performance (including training time) over the full Netflix dataset as compared to the best prior parallel approaches.

This paper makes the following key contributions:

- We present the design of a novel distributed co-clustering based Collaborative Filtering algorithm for soft real-time (less than 10 *sec.*) performance over highly sparse massive data sets on multi-core cluster architectures. Our algorithm involves performance optimizations such as pipelined parallelism, computation communication overlap and communication optimizations (including topology mapping and steiner nodes for communication cost reduction).
- Analytical parallel time complexity analysis, theoretically establishes the improvement in performance and scalability using our algorithm.

[1] www.research.ibm.com/bluegene

– We demonstrate soft real-time distributed CF using the Netflix Prize dataset on a 1024-node Blue Gene/P system. We achieved a training time of around $6s$ with the full Netflix dataset and prediction time of $2.5s$ on $1.4M$ ratings with RMSE (Root Mean Square Error) of 0.87 ± 0.02. This is around $20\times$ (more than one order of magnitude) better than the best known parallel training time [6] along with high accuracy. To the best of our knowledge, this is the highest known distributed performance at such high accuracy. Our algorithm also demonstrates high scalability for large number of nodes on MPP architectures.

2 Related Work

Typical CF techniques are based on correlation criteria [5] and matrix factorization [17]. The correlation-based techniques use similarity measures such as Pearson correlation and cosine similarity to determine a neighborhood of like-minded users for each user and then predict the users rating for a product as a weighted average of ratings of the neighbors. Correlation-based techniques are computationally very expensive as the correlation between every pair of users needs to be computed during the training phase. Further, they have much reduced coverage since they cannot detect item synonymy. The matrix factorization approaches include Singular Value Decomposition (SVD [14]) and Non-Negative Matrix Factorization (NNMF) based [17] filtering techniques. They predict the unknown ratings based on a low rank approximation of the original ratings matrix. The missing values in the original matrix are filled using average values of the rows or columns. Unlike correlation-based methods, the matrix factorization techniques treat the users and items symmetrically and hence, handle item synonymy and sparsity in a better fashion. However, the training component of these techniques is computationally intensive, which makes them impractical to have frequent re-training. Incremental versions of SVD based on folding-in and exact rank-1 updates [4] partially alleviate this problem. But, since the effects of small updates are not localized, the update operations are not very efficient.

[8] studies a special case of the weighted Bregman co-clustering algorithm. The co-clustering problem is formulated as a matrix approximation problem with non-uniform weights on the input matrix elements. Both the users and the items are clustered so that item synonymy ceases to be a problem. As in the case of SVD and NNMF, the co-clustering algorithm also optimizes the approximation error of a low parameter reconstruction of the ratings matrix. However, unlike SVD and NNMF, the effects of changes in the ratings matrix are localized which makes it possible to have efficient incremental updates. [8] presents parallel algorithm design based on co-clustering. It compares the performance of the algorithm against matrix factorization and correlation based approaches on the MovieLens[2] and BookCrossing dataset [19] (269392 explicit rating(1-10) from 47034 users on 133438 books). We consider soft real-time (around 10 *sec.*) CF framework using hierarchical parallel co-clustering optimized for multi-core clusters using pipelined parallelism and computation communication overlap. We deliver scalable performance over 100M ratings of the Netflix data using 1024 nodes

[2] http://www.grouplens.org/data/. 100K ratings(1-5) 943 users, 1682 movies.

of Blue Gene/P with 4 cores at each node. [6] uses a dataflow parallelism based framework (in Java) to study performance vs. accuracy trade-offs of co-clustering based CF. However, it doesn't consider re-training time for incremental input changes. Further, the parallel implementation does not scale well beyond 8 cores due to cache miss and in-memory lookup overheads. We demonstrate parallel scalable performance on 1024 nodes of Blue Gene/P and $20\times$ better training time and better prediction time along with high prediction accuracy (0.87 ± 0.02 RMSE).

[18] presents a parallel algorithm based on Alternating-Least-Squares with Weighted-λ-Regularization (ALS-WR) for the Netflix Prize dataset. Their solution, using parallel Matlab on a Linux cluster, takes 2.5 hrs for training (30 ALS iterations) and with RMSE value around 0.9 on 1000 hidden features. We address the matrix approximation problem using a novel distributed co-clustering algorithm that incorporates performance optimizations to achieve highly scalable performance with the record training time of $5.9s$ on the full Netflix dataset and high accuracy. [10] studies IO scalable co-clustering by mapping a significant fraction of computations performed by the Bregman co-clustering algorithm to an on-line analytical processing (OLAP) engine. [12] studies the scalability of basic MPI based implementation of co-clustering. We deliver more than one order of magnitude higher performance compared to this work, by performing communication optimizations for multi-core cluster based MPPs such as Blue Gene/P. [1] presents results of collaborative filtering using *Concept decomposition* based approach. Concept decomposition is a matrix approximation scheme that solves a least-squares problem after clustering. It has been empirically established [7] that the approximation power (when measured using the Frobenius norm) of concept decompositions is comparable to the best possible approximations by truncated SVDs [9]. However, [1] presents the results of a sequential concept decomposition based algorithm that takes 13.5mins. training time for the full Netflix data, which is very high when looking at soft real-time performance. We achieve around $138\times$ better performance using an optimized distributed algorithm designed for multi-core cluster architectures.

3 Background and Notation

In this paper, we deal with partitional co-clustering where all the rows and columns are partitioned into disjoint row and column clusters respectively. We consider a general framework for addressing this problem that considerably expands the scope and applicability of the co-clustering methodology. As part of this generalization, we view partitional co-clustering as a lossy data compression problem [2] where, given a specified number of rows and column clusters, one attempts to retain as much information as possible about the original data matrix in terms of statistics based on the co-clustering [11]. The main idea is that a reconstruction based on co-clustering should result in the same set of user-specified statistics as the original matrix.

A $k * l$ partitional co-clustering is defined as a pair of functions:
$\rho : 1, \ldots, m \longmapsto 1, ..., k$; and, $\gamma : 1, \ldots, n \longmapsto 1, ..., l$. Let $\hat{U}$ and $\hat{V}$ be random variables that take values in $1, ..., k$ and $1, ..., l$ such that $\hat{U} = \rho(U)$ and $\hat{V} = \gamma(V)$. Let, $\hat{Z} = [\hat{z}_{uv}] \in S^{m \times n}$ be an approximation of the data matrix Z such that $\hat{Z}$ *depends only upon* a given co-clustering (ρ, γ) and certain summary statistics derived from

co-clustering. Let $\hat{Z}$ be a (U,V)-measurable random variable that takes values in this approximate matrix $\hat{Z}$ following w, i.e., $p(\hat{Z}(U,V) = \hat{z}_{uv}) = w_{uv}$. Then, the goodness of the underlying co-clustering can be measured in terms of the expected distortion between Z and $\hat{Z}$, that is,

$$E[d_\phi(Z,\hat{Z})] = \sum_{u=1}^{m}\sum_{v=1}^{n} w_{uv} d_\phi(z_{uv},\hat{z}_{uv}) = d_{\Phi_w}(Z,\hat{Z}) \quad (1)$$

where $\Phi_w : S^{m\times n} \longmapsto \mathbb{R}$ is a is a separable convex function induced on the matrices such that the Bregman divergence ($d_\Phi()$) between any pair of matrices is the weighted sum of the element-wise Bregman divergences corresponding to the convex function ϕ. From the matrix approximation viewpoint, the above quantity is simply the weighted element-wise distortion between the given matrix Z and the approximation $\hat{Z}$. The co-clustering problem is then to find (ρ,γ) such that (1) is minimized.

Now we consider two important convex functions that satisfy the Bregman divergence criteria and are hence studied in this paper.

(I-Divergence) : Given $z \in \mathbb{R}_+$, let $\phi(z) = zlogz - z$. For $z1, z2 \in \mathbb{R}$, $d_\phi(z_1,z_2) = z_1 log(z_1/z_2) - (z_1 - z_2)$.

(Squared Euclidean distance) : Given $z \in \mathbb{R}$, let $\phi(z) = z^2$. For $z1, z2 \in \mathbb{R}$, $d_\phi(z_1,z_2) = (z_1 - z_2)^2$.

Given a co-clustering (ρ,γ), Modha et al. discuss six co-clustering bases where each co-clustering basis preserves certain summary statistics on the original matrix. It also proves that the possible co-clustering bases $(C1 \ldots C6)$ form a hierarchical order in the number of cluster summary statistics they preserve. The co-clustering basis $C6$ preserves all the summaries preserved by the other co-clustering bases and hence is considered the most general among the bases. In this paper we discuss the partitioning co-cluster algorithms for the basis $C6$. For co-clustering basis $C6$ and Euclidean-divergence objective, the matrix approximation is given by: $\hat{A}_{ij} = A^{COC}_{gh} + (A^{CC}_{ih} - A^{RC}_{gj})$, where, $A^{RC}_{gj} = \frac{S^{RC}_{gj}}{W^{RC}_{gj}} = \frac{\sum_{i'|\rho(i')=g} A_{i'j}}{\sum_{i'|\rho(i')=g} W_{i'j}}$; $A^{CC}_{ih} = \frac{S^{CC}_{ih}}{W^{CC}_{ih}} = \frac{\sum_{j'|\gamma(j')=h} A_{ij'}}{\sum_{j'|\gamma(j')=h} W_{ij'}}$;

Algorithm 1. Sequential Static Training via Co-Clustering

Input: Ratings Matrix A, Non-zeros matrix W, No. of row clusters l, No. of column clusters k.
Output: Locally optimal co-clustering (ρ,γ) and averages A^{COC},A^{RC},A^{CC},A^{R} and A^{C}.
Method:
1. Randomly initialize (ρ,γ)
while *RMSE value is converging* **do**
  2a. Compute averages A^{COC},A^{RC}_{gj},A^{CC}_{ih},A^{R} and A^{C}.
  2b. Update row cluster assignments
  $\rho(i) = \underset{1\le g\le k}{\textbf{argmin}} \sum_{j=1}^{n} W_{ij} d_\phi(A_{ij},\hat{A}_{ij}), 1 \le i \le m$
  2c. Update column cluster assignments
  $\gamma(i) = \underset{1\le h\le l}{\textbf{argmin}} \sum_{i=1}^{m} W_{ij} d_\phi(A_{ij},\hat{A}_{ij}), 1 \le j \le n$
end

$A^{COC}_{gh} = \frac{S^{COC}_{gh}}{W^{COC}_{gh}} = \frac{\sum_{i'|\rho(i')=g} \sum_{j'|\gamma(j')=h} A_{i'j'}}{\sum_{i'|\rho(i')=g} \sum_{j'|\gamma(j')=h} W_{i'j'}}$. The sequential update algorithm for the basis $C6$ is as shown in Algorithm 1 where the approximation matrix $\hat{A}$ for various co-clustering bases can be obtained from [2]. For Euclidean divergence, Step 2b. and 2c. of Algorithm 1 use $d_\phi(A_{ij}, \hat{A}_{ij}) = (A_{ij} - \hat{A}_{ij})^2$. For I-divergence, Step 2b. and 2c. of Algorithm 1 use $d_\phi(A_{ij}, \hat{A}_{ij}) = A_{ij} * \log(\hat{A}_{ij}/A_{ij}) - A_{ij} + \hat{A}_{ij}$

In the above sequential algorithm (Algorithm 1), we notice two important steps - a) Calculating the matrix averages, and, b) updating the row and column cluster assignments. Further, given the matrix averages, row and column cluster updates can be done independently, and row updates themselves can be done in parallel.

4 Optimized Distributed Co-clustering Algorithm

For multi-core cluster architectures, one can utilize the available intra-node parallelism along with inter-node parallelism to get highly scalable distributed co-clustering algorithm. Let, c be the number of cores (threads) per node in the distributed architecture, referred to as $T_1 \ldots T_c$. These cores (threads) per node can be used to obtain computation communication overlap as well pipelining across the iterations in the distributed algorithm. This can significantly reduce the communication bottlenecks of the algorithm. Algorithm 2 presents the distributed algorithm with these performance optimizations. The *while loop* executes iterations until the RMSE value converges to within a given error bound. Within each iteration the following steps (Step5..Step10) get executed. In **Step 5.**, threads $(T_2 \ldots T_c)$ compute the partial contribution to row-cluster averages, A^{RC}_{gj}; while simultaneously, thread T_1, performs *MPI_Allgather* to get the column-cluster membership (γ). Thus, (intra-iteration) computation communication overlap is achieved which leads to improved performance. Similarly, computation communication overlap is achieved in the following steps. In **Step 6.**, threads $(T_2 \ldots T_c)$ compute the partial contribution to column-cluster averages, A^{CC}_{ih}; while simultaneously, thread T_1, performs *MPI_Allreduce* to compute the row-cluster averages A^{RC}_{gj}. In **Step 7.**, threads $(T_2 \ldots T_c)$ compute the partial contribution to co-cluster averages, A^{COC}_{gh}; while simultaneously, thread T_1, performs *MPI_Allreduce* to compute the column-cluster averages A^{CC}_{ih}. In **Step 8.**, threads $(T_2 \ldots T_c)$ compute the partial $\hat{A}_{ij}$ values using A^{CC}_{ih} and A^{RC}_{gj}; while simultaneously, thread T_1, performs *MPI_Allreduce* to compute the co-cluster averages A^{COC}_{gh}. In **Step 9.**, all threads $(T_1 \ldots T_c)$ in a node, compute final row-cluster memberships for all the rows that are owned by that node. In **Step 10.**, threads $(T_2 \ldots T_c)$ compute final column-cluster memberships while simultaneously, thread T_1, performs *MPI_Allgather* to get the row-cluster memberships from all other nodes.

In order to reduce the communication cost, the nodes are divided into groups, each group having the same number of nodes. A small constant number of nodes in each group act as *Steiner nodes* and help in inter-group communication. So, each communication step in Algorithm 2 is broken into two phases: (a) Intra-group communication, followed by (b) Inter-group communication using the Steiner nodes. Since, the communication group sizes are significantly reduced by using this grouping strategy, the communication cost goes down thus improving the scalability of the distributed algorithm. Further, to ensure non-overlap, across any two groups, of their intra-group

communication, we use topology mapping to map each group onto a plane in the 3D Torus Interconnect architecture of Blue Gene/P. This leads to further decrease in the communication time.

5 Parallel Time Complexity Analysis

In this section, we establish theoretically, the performance and scalability advantage of our optimized distributed algorithm. Refer notation given in Table 5.

The distributed algorithm described in section 2 takes a certain number of iterations, say I. In each iteration the rows are assigned to row clusters and columns are assigned to column clusters. Each iteration has multiple steps. In *Step 5.*, the thread T_1 of all nodes communicate using all-gather operation to aggregate column to column-cluster mapping information. This communication time is given by: $O(S_0 + (n/B_0) * log(P_0))$. Simultaneously, threads $T_2 \ldots T_c$ of each node compute partial contributions of each node towards A^{RC}_{gj}. This computation time is $O(mn/(P_0.c))$. The overall time for *Step 5.* is given by $\max(O(S_0 + (n/B_0) * log(P_0)), mn/(P_0.c))$. Assuming, that compute time dominates, the time complexity for *Step 5.* can be approximated by $O(mn/(P_0 * c))$.

In *Step 6.*, the thread T_1 of all nodes communicate using all-reduce operation to compute the row-cluster averages A^{RC}_{gj}. This communication time is given by: $O(S_0 + (mn/B_0) * log(P_0))$. Simultaneously, threads $T_2 \ldots T_c$ of each node compute partial contributions of each node towards A^{CC}_{ih}. This computation time is $O(mn/(P_0.c))$. Thus, the overall time for *Step 6.* is given by $\max(O(S_0 + (mn/B_0) * log(P_0)), mn/(P_0.c))$. Assuming, that the communication time dominates, the time complexity for *Step 6.* can be approximated by $O(S_0 + (mn/B_0) * log(P_0))$. Similarly, the time complexity for *Step 7.* can be approximated by $O(S_0 + (mn/B_0) * log(P_0))$.

In *Step 8.*, the thread T_1 of all nodes communicate using all-reduce operation to compute the co-cluster averages A^{COC}_{gh}. This communication time is given by: $O(S_0 + (kl/B_0) * log(P_0))$. Simultaneously, threads $T_2 \ldots T_c$ of each node compute partial values for assignment of each row (and column) to k possible row-clusters (and l possible column-clusters). This computation time is $O(mns * (k+l)/(P_0.c))$. Thus, the overall time for *Step 8.* is given by $\max(O(S_0 + (kl/B_0) * log(P_0)), mns * (k+l)/(P_0.c))$. Assuming that the compute time dominates, the time complexity for *Step 8.* can be approximated by $O(mns * (k+l)/(P_0.c))$. In a similar fashion, the compute time for *Step 9.* is $O(mns * (k+l)/(P_0 * c))$. Assuming that the compute time dominates *Step 10.*, its time complexity can be approximated by $O(mns * (k+l)/(P_0 * c))$.

Thus, the overall time complexity for the hybrid distributed co-clustering algorithm , per iteration, is given by:

$$T_h(m, n, P_0) = O((mn/P_0 * c) + S_0 + (mn/B_0) * log(P_0) + mns * (k+l)/(P_0 * c)) \quad (2)$$

One can observe, that an MPI only (algorithm referred to as *base* algorithm), which does not have computation communication overlap, has run time around $c\times$ higher as compared to the hybrid Algorithm 2. This is so, because the hybrid algorithm achieves effective overlap between computation and communication in most of the steps of the algorithm and utilizes c available cores per node to get higher performance, while the

Algorithm 2. Distributed (Hybrid - MPI+OMP) Static Training via Co-Clustering

Input: Ratings Matrix (A), Non-zeros matrix (W), No. of row clusters (l), No. of column clusters (k).
Output: Locally optimal co-clustering (ρ,γ) and averages A_{gh}^{COC},A_{gj}^{RC},A_{ih}^{CC}.
Data Distribution: (Each node has total c threads - $\{T_1 \dots T_c\}$)
1. Each node p gets $m_p = m/P_0$ rows and $n_p = n/P_0$ columns.
2. Further, threads $T_1 \dots T_c$ of each node p, each get $m_{p'}$ rows (i.e, a $m_{p'} \times n$ submatrix) and $n_{p'}$ columns (i.e, a $m \times n_{p'}$ submatrix, where $m'_p = \frac{m_p}{T_c}$ and $n'_p = \frac{n_p}{T_c}$.

Method:
1. Each $T_i, i \in [1..c]$: Randomly initialize (ρ_i^p,γ_i^p)
2. T_1 : Gather all the row and column sums/weights S_i^R, S_j^C, W_i^R, W_j^C $\forall i, j$ from the other nodes using MPI_Allgather.
3. $T_1 \dots T_c$: Calculate **all** row and column averages $A_i^R = \frac{S_i^R}{W_i^R}$ and A_j^C.
(Note that Step 2 and Step 3 can be executed in parallel)
4. T_1: Gather the **global** Row-cluster membership (ρ) by concatenating (ρ^p) using MPI_Allgather.

while *RMSE value has not converged,* ***Each thread*** *in a node does the following* **do**

5. $T_2 \dots T_c$: Calculate the partial contributions to Row-Cluster Averages A_{gj}^{RC}
T_1 : Gather the **global** Column-cluster membership (γ) by concatenating (γ^p) using MPI_Allgather.

6. $T_2 \dots T_c$: Calculate the partial contribution to Column-Cluster Averages A_{ih}^{CC}
T_1: Do MPI_AllReduce to compute the global row-cluster averages A_{gj}^{RC}
7. $T_2 \dots T_c$: Calculate the contribution of the local rows and columns to the co-cluster sums/weights i.e, S_p^{COC} and W_p^{COC}.

T_1: Do MPI_AllReduce to compute the global col-cluster averages A_{ih}^{RC}
8. T_1:Do an MPI_AllReduce on above contributions and get the **global** co-cluster sums/weights S^{COC}, W^{COC} and calculate A^{COC}.
$T_2 \dots T_c$: Partially compute $\hat{A}^R(i, j, g)$, $\hat{A}^C(i, j, h)$ the local row cluster and column cluster assignment steps for each choice of assignment g,h

9. $T_1 \dots T_c$: Update all the local row cluster assignments ρ^p by first updating $\hat{A}^R(i, j, g)$ with the co-cluster averages to generate $\hat{A}_{ij}$
$\rho^p(i) = \underset{1 \le g \le k}{\textbf{\textit{argmin}}} \sum_{j=1}^{n} w_{ij} d_\phi(A_{ij}, \hat{A}_{ij})$, i : rows owned by node p

10. T_0 : Gather the **global** Row-cluster membership (ρ) by concatenating (ρ^p) using MPI_Allgather.
$T_1 \dots T_3$: Update all the local column cluster assignments γ^p by first updating $\hat{A}^C(i, j, h)$ with the cocluster averages to generate $\hat{A}_{ij}$

$\gamma^p(i) = \underset{1 \le h \le l}{\textbf{\textit{argmin}}} \sum_{i=1}^{m} w_{ij} d_\phi(A_{ij}, \hat{A}_{ij})$, j : cols owned by node p

end

Table 1. Notation

Symbol	Definition
P_0	Total number of nodes for computation
c	Number of threads (cores) per node
(m, n)	Number of rows and columns in the input matrix
s	Sparsity factor of the matrix
(k, l)	Number of row and column clusters respectively
(m/k)	Average number of rows per row cluster
n/l	Average number of columns per column cluster
B_0	Interconnect Bandwidth for AllReduce / Allgather operation
S_0	Setup cost for AllReduce / Allgather operation

MPI only (base) algorithm performs all the computation and communication steps sequentially. Further, the load-imbalance across the nodes is reduced, by factor c, in the hybrid algorithm as compared to the base algorithm. Assuming, γ as the load-imbalance factor for the base algorithm, the training time for the base algorithm gets magnified by the factor, $(1 + \gamma)$. For the same data distribution, the hybrid algorithm will have load imbalance of $(1 + \gamma/c)$ and hence a lower magnification factor in its training time. Thus, in the best case, this leads to $O(c^2 * \frac{(1+\gamma)}{(c+\gamma)})$ performance gain of hybrid vs. base algorithm.

5.1 Optimum Thread Distribution

In a general case, one can optimize the communication by providing more than one thread for communication. We study this general communication optimization technique in this section and determine the optimum number of threads to achieve best performance.

Let r be the number of threads (cores) that are devoted to computation per step, while the remaining $(c-r)$ threads (cores), perform communication per step. When, multiple threads are used for communication, we assume that it takes x steps to complete one communication task across all nodes. In this case, the time complexity of the hybrid distributed co-clustering algorithm is given by:

$$T_h(m, n, r, P_0) = O((mn/P_0 * r) + (S_0 + (mn/B_0) * log(P_0)) * (2x/(c - r)) + 3mns * (k + l)/(P_0 * r)) \tag{3}$$

Differentiating the above expression for $T_h(m, n, r, P_0)$ with respect to r, and setting it to zero, we can determine the optimum number of threads to be used for computation per node. We get the following quadratic equation to determine the optimum r:

$$2xP_0 * (S_0 + mn/B_0 * \log(P_0)) * r^2 = (mn + 3mns * (k + l)) * (c^2 + r^2 - 2cr) \tag{4}$$

Solving, the optimum value of r is given by:

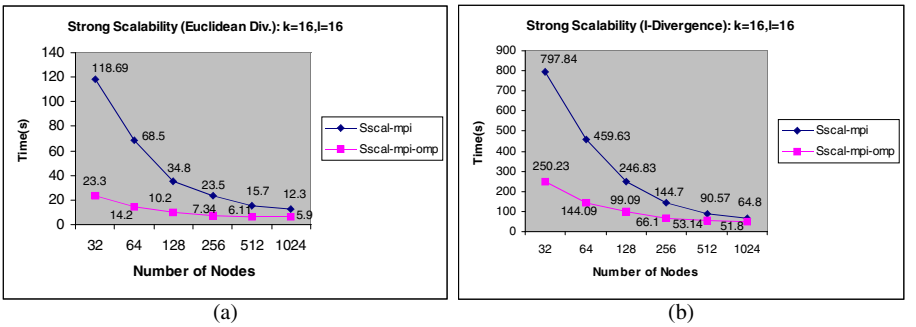
$$r^* = \frac{\sqrt{(4c^2Z^2 + 8c^2xP_0Y)} - 2cZ}{2 * (2xP_0Y - Z)}, \text{where,} \\ Y = S_0 + mn/B_0 * \log(P_0), \text{and, } Z = mn + 3mns(k+l) \tag{5}$$

6 Results and Analysis

The hybrid distributed algorithm was implemented using MPI and OpenMP, while the base distributed algorithm was implemented using only MPI. The Netflix Prize dataset was used to evaluate and compare the performance and scalability of these distributed co-clustering algorithms. The experiments were performed on the Blue gene/P (MPP) architecture. Each node in Blue Gene/P is a quad-core chip with frequency of 850 MHz having 2 GB of DRAM and 32 KB of L1 cache per core. Blue Gene/P has the following major interconnects:(a) 3D-Torus interconnect which provides 3.4 Gbps per link on each of the 12 links per node (total 5.1 GBps per node), and, (b) Collective Network that provides 6.8 Gbps per link. MPI was used across the nodes for communication, while within each node OpenMP was used to parallelize the computation and communication amongst the four cores. For all the experiments, we obtained RMSE in the range 0.87 ± 0.02 on the data. Below, k refers to the number of row clusters generated while l refers to the number of column clusters generated. Netflix data was used for evaluation of the distributed algorithms. For Netflix, the number of rows, m, is around $480K$; the number of columns, n, is $17,770$, and the sparsity factor, s is around 85. We present the strong, weak and data scalability analysis of the training phase for both Euclidean divergence and I-divergence based co-clustering.

6.1 Strong Scalability

For strong scalability, we used the full Netflix data for each experiment, while increasing the number of nodes, from 64 to 1024. Fig. 1(a) illustrates that the hybrid algorithm (for Euclidean divergence) has consistently better performance over the base algorithm: $5.1\times$ better than the base when $P_0 = 32$ and $2.1\times$ better at $P_0 = 1024$. Here, the hybrid algorithm has more than $(c = 4)\times$ better performance than the base algorithm due to reduction in load-imbalance as explained in Section 5. In the hybrid algorithm, as the number of nodes increases from 32 to 1024, the compute time decreases by $26\times$ while the communication time remains almost the same, this leads to overall $4\times$ decrease in total training time with $32\times$ increase in the number of nodes(P_0). Fig. 1(b) illustrates the performance gain of the hybrid algorithm over the base algorithm for I-divergence. Here, the performance gain of hybrid vs base decreases from $3.2\times$ for $P_0 = 32$ nodes to $1.25\times$ for $P_0 = 1024$ nodes. By using more efficient load balancing techniques, the performance of the hybrid (MPI+OpenMP) algorithm can be improved further. Moreover, by using the optimum number of cores for communication using the formula specified in the Section 5.1, one can get better overall performance. Further, for I-divergence, the gain for the hybrid algorithm from the decrease in inter-node load-imbalance is offset by the loss from intra-node load-imbalance amongst the threads. Hence, in case of I-divergence the gain of the hybrid algorithm over the base algorithm is not as large as in the Euclidean divergence.



(a) (b)

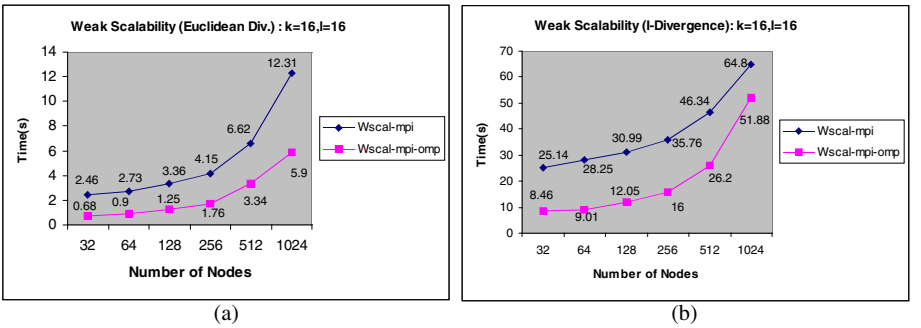
Fig. 1. Strong Scalability: (a) Euclidean divergence. (b) I-divergence

6.2 Weak Scalability

Fig. 2(a) displays the weak scalability for Euclidean distance based co-clustering as the number of nodes (P_0) increases from 32 to 1024 and the training data increases from 3.125% to 100% of the full Netflix dataset (with $k = 16$, $l = 16$). Here, the hybrid algorithm performs consistently better compared to the base algorithm: $3.61\times$ better at $P_0 = 32$ and $2.1\times$ better at $P_0 = 1024$. The total time for the hybrid algorithm increases by $8.67\times$ as the number of nodes increase from 32 to 1024. This is due to the compute time increase by $2.91\times$ and also increase in load imbalance. Fig. 2(b) illustrates the weak scalability of the hybrid algorithm for I-divergence: with $32\times$ increase in the data and number of nodes, the training time only increases by $6.13\times$. Further, the hybrid algorithm performs consistently better than the base algorithm.

6.3 Data Scalability

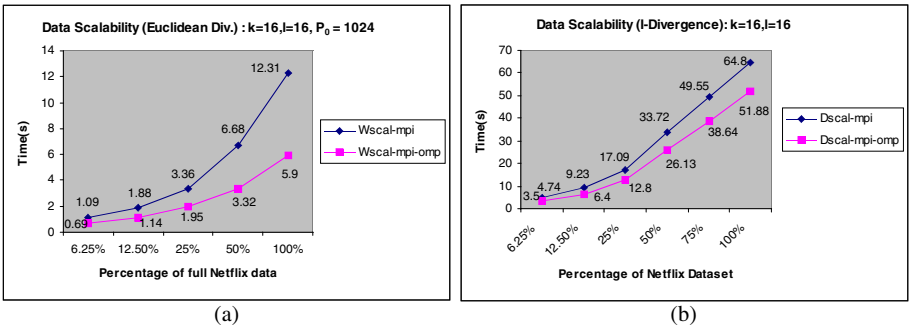
Fig. 3(a) displays the data scalability for Euclidean distance based co-clustering as the training data increases from 6.25% to 100% of the full Netflix dataset, while $P_0 = 1024$.



(a) (b)

Fig. 2. Weak Scalability: (a) Euclidean divergence. (b) I-divergence

The training time for the hybrid algorithm increases by $8.55\times$ with $16\times$ increase in data, while that for the base algorithm increases by $11.3\times$. Thus, the hybrid algorithm shows better than linear data scalability and also better data scalability as compared to the base algorithm. The hybrid algorithm also performs better than the base by $1.58\times$ at $P_0 = 32$ and $2.1\times$ better at $P_0 = 1024$. Fig. 3(b) illustrates the data scalability for the hybrid algorithm with I-divergence as the training time increases only by $14.8\times$ with $16\times$ increase in data, while the number of nodes is kept constant at $P_0 = 1024$.



(a) (b)

Fig. 3. Data Scalability: (a)Euclidean divergence. (b) I-divergence

7 Conclusions and Future Work

Real-time collaborative filtering with high prediction accuracy is a computationally challenging problem. We have presented the design of a novel distributed co-clustering based Collaborative Filtering algorithm. Our algorithm demonstrates soft real-time (less than 10 *sec.*) performance over highly sparse massive data sets. Using pipelined parallelism and compute communication overlap optimizations our hybrid (MPI+OpenMP) algorithm outperforms all known prior results for CF while maintaining high accuracy. Theoretical time complexity analysis proves the scalability of our algorithm. We demonstrated soft real-time parallel CF using the Netflix Prize dataset on Blue Gene/P architecture. We delivered the best known training time of around $6s$ for the full Netflix dataset and the best known prediction of $1.78us$ per prediction (rating) for $1.4M$ ratings with high prediction accuracy (RMSE value of 0.87 ± 0.02). This training time is $20\times$ (more than one order of magnitude) better than the best known parallel training time. We also demonstrated strong, weak and data scalability for multi-core cluster architectures. In future, we intend to investigate performance analysis using queuing theoretic models for large scale systems.

References

1. Ampazis, N.: Collaborative filtering via concept decomposition on the netflix dataset. In: ECAI, pp. 143–175 (2008)
2. Banerjee, A., Dhillon, I., Ghosh, J., Merugu, S., Modha, D.S.: A generalized maximum entropy approach to bregman co-clustering and matrix approximation. Journal of Machine Learning Research 8(1), 1919–1986 (2007)

3. Bennett, J., Lanning, S.: The netflix prize. In: KDD-Cup and Workshop at the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2007)
4. Brand, M.: Fast online svd revisions for lightweight recommender systems. In: SIAM International Conference on Data Mining, pp. 37–48 (2003)
5. Breese, J.S., Heckerman, D., Kadie, C.: Empirical analysis of predictive algorithms for collaborative filtering. In: Fourteenth International Conference on Uncertainty in Artificial Intelligence, pp. 43–52 (1998)
6. Daruru, S., Marin, N.M., Walker, M., Ghosh, J.: Pervasive parallelism in data mining: dataflow solution to co-clustering large and sparse netflix data. In: 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1115–1124 (2009)
7. Dhillon, I.S., Modha, D.S.: Concept decompositions for large sparse text data using clustering. In: Machine Learning, pp. 143–175 (1999)
8. George, T., Merugu, S.: A scalable collaborative filtering framework based on co-clustering. In: Fifth International Conference on Data Mining, pp. 625–628 (2005)
9. Golub, G.H., Loan, C.F.V.: Matrix computations. The Johns Hopkins University Press, Baltimore (1996)
10. Hsu, K.-W., Banerjee, A., Srivastava, J.: I/o scalable bregman co-clustering. In: Proceedings of the 12th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (2008)
11. Mallela, I.D.S., Modha, D.: Information-theoretic co-clustering. In: Proceedings of the 9th International Conference on Knowledge Discovery and Data Mining, pp. 89–98 (2003)
12. Kwon, B., Cho, H.: Scalable co-clustering algorithms. In: Hsu, C.-H., Yang, L.T., Park, J.H., Yeo, S.-S. (eds.) ICA3PP 2010. LNCS, vol. 6081, pp. 32–43. Springer, Heidelberg (2010)
13. Resnick, P., Varian, H.R.: Recommender systems - introduction to special section. Comm. ACM 40(3), 56–58 (1997)
14. Sarwar, B., Karypis, G., Konstan, J., Reidl, J.: Application of dimensionality reduction in recommender systems: a case study. In: WebKDD Workshop (2000)
15. Sarwar, B.M., Karypis, G., Konstan, J.A., Riedl, J.: Analysis of recommendation algorithms for e-commerce. In: ACM Conference on Electronic Commerce, pp. 158–167 (2000)
16. Schafer, J.B., Konstan, J.A., Riedi, J.: Recommender systems in e-commerce. In: ACM Conference on Electronic Commerce, pp. 158–166 (1999)
17. Srebro, N., Jaakkola, T.: Weighted low rank approximation. In: Twentieth International Conference on Machine Learning, pp. 720–728 (2003)
18. Zhou, Y., Wilkinson, D., Schreiber, R., Pan, R.: Large scale parallel collaborative filtering for the netflix prize. In: Fourth International Conference on Algorithmic Aspects in Information and Management, pp. 337–348 (2008)
19. Ziegler, C.N., McNee, S.M., Konstan, J.A., Lausen, G.: Improving recommendation lists through topic diversification. In: Fourteenth International World Wide Web Conference (2005)

Compressing the Incompressible with ISABELA: In-situ Reduction of Spatio-temporal Data

Sriram Lakshminarasimhan[1,2], Neil Shah[1], Stephane Ethier[3], Scott Klasky[2], Rob Latham[4], Rob Ross[4], and Nagiza F. Samatova[1,2,⋆]

[1] North Carolina State University, Raleigh, NC 27695, USA
[2] Oak Ridge National Laboratory, Oak Ridge, TN 37830, USA
[3] Princeton Plasma Physics Laboratory, Princeton, NJ 08543, USA
[4] Argonne National Laboratory, Argonne, IL 60439, USA
samatova@csc.ncsu.edu

Abstract. Modern large-scale scientific simulations running on HPC systems generate data in the order of terabytes during a single run. To lessen the I/O load during a simulation run, scientists are forced to capture data infrequently, thereby making data collection an inherently *lossy* process. Yet, *lossless* compression techniques are hardly suitable for scientific data due to its inherently random nature; for the applications used here, they offer less than 10% compression rate. They also impose significant overhead during decompression, making them unsuitable for data analysis and visualization that require repeated data access.

To address this problem, we propose an effective method for In-situ Sort-And-B-spline Error-bounded Lossy Abatement (**ISABELA**) of scientific data that is widely regarded as effectively incompressible. With ISABELA, we apply *a preconditioner* to seemingly random and noisy data along spatial resolution to achieve an accurate fitting model that guarantees a ≥ 0.99 correlation with the original data. We further take advantage of temporal patterns in scientific data to compress data by $\approx 85\%$, while introducing only a negligible overhead on simulations in terms of runtime. ISABELA significantly outperforms existing lossy compression methods, such as Wavelet compression. Moreover, besides being a communication-free and scalable compression technique, ISABELA is an inherently local decompression method, namely it does not decode the entire data, making it attractive for random access.

Keywords: Lossy Compression, B-spline, In-situ Processing, Data-intensive Application, High Performance Computing.

1 Introduction

Spatio-temporal data produced by large-scale scientific simulations easily reaches terabytes per run. Such data volume poses an I/O bottleneck—both while writing the data into the storage system during simulation and while reading the data back during analysis and visualization. To alleviate this bottleneck, scientists have to resort to subsampling, such as capturing the data every s^{th} timestep.

⋆ Corresponding author.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 366–379, 2011.

This process leads to an inherently *lossy* data reduction.

In-situ data processing—or processing the data in-tandem with the simulation by utilizing either the same compute nodes or the staging nodes—is emerging as a promising approach to address the I/O bottleneck [12]. To complement existing approaches, we propose an effective method for In-situ Sort-And-B-spline Error-bounded Lossy Abatement (ISABELA) of scientific data.

ISABELA is particularly designed for compressing spatio-temporal scientific data that is characterized as being inherently noisy and random-like, and thus commonly believed to be uncompressible [16]. In fact, any *lossless* compression technique [3,13] is capable of reducing such data by no more than a 10% of its original size, besides being computationally intensive and, therefore, hardly suitable for *in-situ* processing (see Section 3).

The intuition behind ISABELA stems from the following three observations. First, while being almost random and noisy in its natural form—when sorted—scientific data exhibits a very strong signal-to-noise ratio due to its monotonic and smooth behavior in its sorted form. Second, prior work done in curve fitting [7,17] have shown that monotone curve fitting, such as monotone B-splines, can offer some attractive features for data reduction, including, but not limited to, their goodness of fit with significantly fewer coefficients to store. Finally, the monotonicity property of the sorted data gets preserved in most of its positions with respect to adjacent time steps. Hence, this property of monotonic inheritance across temporal resolution offers yet another venue for improvement of the overall data compression ratio.

While intuitively simple, ISABELA has addressed a number of technical challenges imposed by end-user's requirements. One of the most important factors for the user's adoption of any lossy data reduction technology is the assurance that the user-acceptable error-bounds are respected. Since curve fitting accuracy is often data-dependent, ISABELA must be robust in its approximation. While curve fitting operations are traditionally time consuming, performing the compression *in-situ*, mandates ISABELA to be fast. Finally, while data sorting—as a pre-conditioner for data reduction—is "a blessing," it is "a curse" at the same time; reordering the data requires keeping track of the new position indices to associate the decoded data with its original ordering. While management of spline coefficients could be viewed as a light-weight task, the heavy-weight index management forces ISABELA to make some non-trivial decisions between the data compression rates and the data accuracy.

2 A Motivating Example

Much of the work for *in-situ* data reduction in this paper stems from a Gyrokinetic Tokamak Simulation (GTS) [15] for studying plasma micro-turbulence in the core of magnetically confined fusion plasmas of toroidal devices in nuclear reactors. On current petaflop systems, such as NCCS/ORNL Jaguarpf, the GTS code, utilizing ADIOS [11] for its intensive I/O, has demonstrated weak scaling for up to 65,536 cores on the 8-core per node configuration.

The entire GTS data set can be broadly divided into: (1) checkpoint data to restart the simulation in case of an execution failure (C&R); (2) analysis (A) data, such as density and potential fluctuations, for performing various post-processing physics analyses, and (3) diagnostics data used, for example, for code validation and verification (V&V) (see Table 1).

Table 1. Summary of GTS output data by different categories

Category	Write Frequency	Read Access	Size/Write	Total Size
C&R	Every 1-2 hours	Once or never	A few TBs	≈TBs
A	Every 10^{th} time step	Many times	A few GBs	≈TBs
V&V	Every 2^{nd} time step	A few times	A few MBs	≈GBs

Unlike C&R data that requires lossless compression, analysis (A) data is inherently lossy, and as such, it can tolerate some error-bounded loss in its accuracy. What is more important is that it is the analysis data that is being accessed many times by different scientists using various analysis and visualization tools or Matlab physics analysis codes. Therefore, aggressive data compression that could enable interactive analytical data exploration is of paramount concern, and is, therefore, the main focus of ISABELA. For illustrative purposes, throughout the paper, we will use temporal snapshots of the GTS analysis data consisting of one-dimensional 64-bit double precision floating point arrays of 172,111 values each for *Potential* and *Density* fluctuations.

3 Problem Statement

The inherent complexity of scientific spatio-temporal data drastically limits the applicability of both lossless and lossy data compression techniques and presents a number of challenges for new method development. Such data not only consists of floating-point values, but also exhibits randomness without any distinct repetitive bit and/or byte patterns (also known as high entropy data, and hence, uncompressible [5,14]). Thus, applying standard *lossless* compression methods does not result in an appreciable data reduction.

Table 2 illustrates the compression rates achieved and the time required to compress and decode 12,836KB of GTS analysis data by state-of-the-art methods. In addition, scientific data often exhibits a large degree of fluctuations in values across even directly adjacent locations in the array. These fluctuations render *lossy* multi-resolution compression approaches like Wavelets [6] ineffective.

The compression ratio $CR_M(D)$ of a compression method M for data D of size $|D|$ reduced to size $|D_M|$ is defined by Eq. 1:

$$CR_M(D) = \frac{|D| - |D_M|}{|D|} \times 100\%. \quad (1)$$

The accuracy of lossy encoding techniques is measured using Pearson's correlation coefficient (ρ) and Normalized Root Mean Square Error between an

$N-$dimensional original data vector $D = (d_0, d_1, \ldots, d_{N-1})$ and decompressed data vector $D' = (d'_0, d'_1, \ldots, d'_{N-1})$ defined by Eq. 2:

$$NRMSE_M(D) = \frac{RMSE_M(D, D')}{Range(D)} = \frac{\sqrt{\sum_{i=0}^{N-1} (d_i - d'_i)^2}}{max(D) - min(D)}. \quad (2)$$

Table 2. Performance of examplar lossless and lossy data compression methods

Metric	**FPC**	**LZMA**	**ZIP**	**BZ2**	**ISABELA**	**Wavelets**	**B–splines**
Lossless?	Yes	Yes	Yes	Yes	No	No	No
CR_M (%)	3.12	2.72	1.13	1.11	**81.44***	22.51*	0*
Compression (sec.)	0.58	7.01	1.03	3.96	0.93	0.62	0.78
Decompression (sec.)	0.56	1.38	0.49	1.18	1.05	0.58	0.82

*CR achieved by lossy models for 0.99 correlation and 0.01 NRMSE fixed accuracy. All runs are performed on an Intel Core 2 Duo 2.2 GHz processor with 4 GB RAM, openSUSE Linux v11.3.

4 Theory and Methodology

Existing multi-resolution compression methods often work well on image data or time-varying signal data. For scientific data-intensive simulations, however, data compression across the *temporal resolution* requires data for many timesteps be buffered in memory that is, obviously, not a viable option. Applying lossy compression techniques on this data across the *spatial resolution* requires a significant tradeoff between the *compression ratio* and the *accuracy*. Hence, to extract the best results out of the existing approximation techniques, a transformation of this data layout becomes necessary.

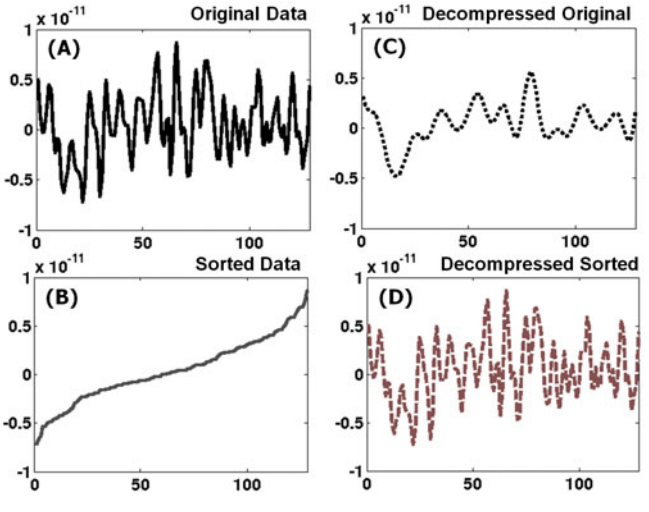


Fig. 1. A slice of GTS Potential: (A) original; (B) sorted; (C) decoded after B–splines fitting to original; and (D) decoded after B–splines fitting to sorted

4.1 Sorting-Based Data Transformation

Sorting changes the data distribution in the spatial domain, from a highly irregular signal (Fig. 1, **A**) to a smooth and monotonous curve (Fig. 1, **B**). The rationale behind sorting—as a pre-conditioner for a compression method—is that fitting on a monotonic curve can provide a model that is more accurate than the one on unordered and randomly distributed data. Figure 1 illustrates the significant contrast in how closely (**D**) or poorly (**C**) the decompressed data approximates the original data when the $B-$splines curve fitting [2] operates on sorted versus unsorted data, respectively.

4.2 Cubic $B-$Splines Fitting

Sorting the data in an increasing order provides a sequence of values whose rate of change is guaranteed to be the slowest. Although this sequence resembles a smooth curve, performing curve fitting using non-linear polynomial interpolation becomes difficult for complex shape curves. Computing interpolation constants for higher-order polynomials in order to fit these complex curves is computationally intensive for *in-situ* processing.

A more effective technique is by using $B-$splines curve fitting. A $B-$splines curve is a sequence of piecewise lower order parametric curves joined together via knots. Cubic $B-$splines are composed of polynomial functions of degree $d = 3$, which have faster interpolation time and produce "smooth" curves (i.e., second-order differentiable) at the knot locations. The shape of the $B-$splines curve is determined by a knot sequence that describes the span of the piecewise segments, and a set of basis functions that influences the segments of the curve. Because of this property splines can control the local shape of the curve without affecting the shape of the curve globally. This also implies that both curve fitting and interpolation are efficient operations and can provide location-specific data decoding without decompressing all the data. While sorting rearranges the points, location-specific decoding is still possible by performing an additional single level translation of data location from the original to the sorted vector, and then retrieving the interpolated value on the sorted $B-$spline curve.

4.3 Maximizing Compression Ratio via Window Splitting

In this section, we look at approaches to maximizing the compression ratio, while maintaining an accurate approximation model. Let us assume that the original data D is a vector of size N, namely $D = (d_0, d_1, \ldots, d_{N-1})$. This way we can associate a value d_i with each index value $i \in I = \{0, 1, \ldots, N-1\}$. Let us also assume that each vector element, $d_i \in \mathbb{R}$, is stored as a 64-bit double-precision value. Therefore, storing the original data requires $|D| = N \times 64$ bits.

Assuming that D is a discrete approximation of some curve, its $B-$splines interpolation D_B requires storing only $B-$splines constants—the knot vector and the basis coefficients—in order to reconstruct the curve. Let C denote the

number of such 64-bit double-precision constants. Then storing the compressed data after B-splines curve fitting requires $|D_B| = C \times 64$ bits.

The random-like nature of D (see Fig. 1, (A)) requires $C \sim N$ to provide accurate lossy compression, and hence, leads to a poor compression rate (see Table 2, last column). However, applying B-splines interpolation after sorting D requires only a few constants, $C = O(1) << N$, in order to provide high decompression accuracy (see Fig. 1, (D)).

While significantly reducing the number of $B-$splines constants C, sorting D will reorder the vector elements via some permutation π of its indices, namely $I \xrightarrow{\pi} I_\pi = \{i_1, i_2, \ldots, i_N\}$, such that $d_{i_j} \leq d_{i_{j+1}}, \forall i_j \in I_\pi$. As a result, we need to keep track of the new index I_π so that we could associate the decompressed sorted vector D_π back to the original vector D by using its correct index I. Since each index value i_j requires $log_2 N$ bits, the total storage requirement for I_π is thus $|I_\pi| = N \times log_2 N$ bits. Therefore, the vector length N is the only factor that determines the storage requirements for the index I_π.

One way to optimize the overall compression ratio, CR_{ISABELA}, is to first split the entire vector D into fixed-sized windows of size W_0, or $D = \bigcup D^k$, $D^i \cap D^j = \emptyset$, $I^k = \{(k-1)W_0, (k-1)W_0 + 1, \ldots, kW_0\}$, $i, j, k \in \overline{1, N_W}$, $i \neq j$, and $N_W = \left\lceil \frac{N}{W_0} \right\rceil$. Then, the $B-$splines interpolation is applied to each window D^k separately.

With this strategy, ISABELA's storage requirement for the compressed data is defined by Eq. 3:

$$\begin{aligned} |D_{\text{ISABELA}}| &= \sum_{k=1}^{N_W} (|D_B^k| + |I_\pi^k|), \\ &= N_W \times (C \times 64 + W_0 \times log_2 W_0) \end{aligned} \tag{3}$$

Substituting Eq. 3 into Eq. 1 and simplifying the resulting equation, we obtain the following compression ratio for ISABELA defined by Eq. 4:

$$CR_{\text{ISABELA}}(D) = (1 - \frac{log_2(W_0)}{64} - \frac{C}{W_0}) \times 100\% \tag{4}$$

From Eq. 4, we can analytically deduce the trade-off between the window size W_0 and the number of $B-$splines constants C that give the best compression ratio. For example, for $W_0 > 65,536$, the size of the index alone would consume more than 25% of the original data. We found that $C = 30$ and $W_0 = 1024$ allows ISABELA to achieve both > 0.99 correlation and < 0.05 NRMSE between the original and decompressed GTS data. Also, fixing $W_0 = 1024$ balances the cost of storing both the index and the fitting coefficients giving an overall compression rate of 81.4% per time step.

4.4 Error Quantization for Guaranteed Point-by-Point Accuracy

The above sorting-based curve fitting model ensures accurate approximation only on a *per window* basis and not on a *per point* basis. As a result, in certain

locations, the $B-$splines estimated data deviates from the actual by a margin exceeding a defined tolerance. For example, almost 95% of the approximated GTS Potential values average a 2% relative error, where the percentage of the relative error (ϵ) at each index i between $D = (d_0, d_1, ..., d_{N-1})$ and $D_{ISABELA} = (d'_0, d'_1, ..., d'_{N-1})$ is defined as $\epsilon_i = \frac{d_i - d'_i}{d_i} \times 100\%$. While the number of such location points is reasonably low due to accurate fitting achieved by $B-$splines on monotonic data, ISABELA guarantees that a user-specified point-by-point error is respected by utilizing an *error quantization* strategy.

Storing relative errors between estimated and actual values enables us to reconstruct the data with high accuracy. Quantization of these errors into 32-bit integers results in a large degree of repetition, where majority of the values lie between [-2, 2]. These integer values lend themselves to high compression rates ($75\% - 90\%$) with standard lossless compression libraries. These compressed relative errors are stored along with the index during encoding. Upon decoding, applying these relative errors ensures decompressed values to be within a user-defined threshold τ_ϵ for per point relative error.

4.5 Exploiting Δ–Encoding for Temporal Index Compression

To a large extent, the ordering of the sorted data values is similar between adjacent timesteps, i.e., the monotonicity property of the data extends to index integer values. Hence, we apply a differential encoding scheme to the index vector I_π before compressing the index using standard lossless compression libraries. [Note that subsequent scheme is applied to each individual data window D^n.]

Suppose that at timestep t_0, we first build the index $I_\pi(t_0)$ consisting of no redundant values, essentially, incompressible. Hence, this index is stored as is. But, at the next timestep $t_0 + 1$, the difference in index values $\Delta I_{+1} = I_\pi(t_0 + 1) - I_\pi(t_0)$ is small (see Fig. 2) due to monotonicity of the original data values D^n and, hence, the sorted values across adjacent timesteps.

Thus, instead of storing the index values at each timestep, we store the index values at t_0, denoted as the *reference* index, along with the compressed pairwise index differences ΔI_{+1} between adjacent timesteps. But, in order to recover the

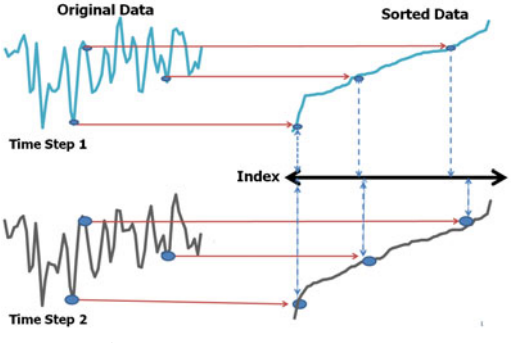
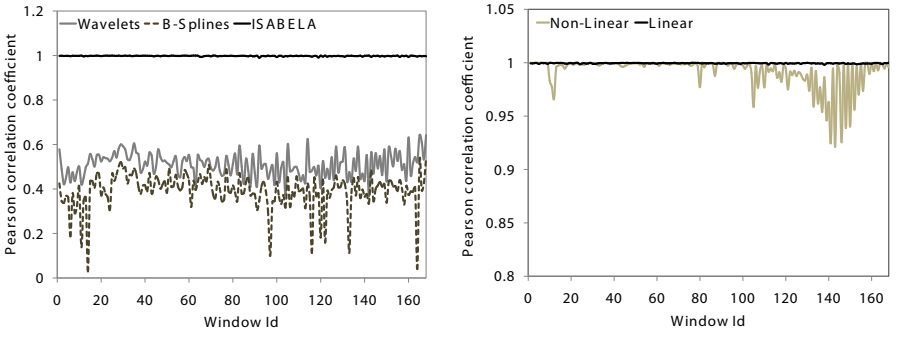


Fig. 2. Illustration of Δ–encoding of the index across temporal resolution

data at time $t_0 + \delta t$, we must read in both the reference index $I_\pi(t_0)$ and all the *first-order* differences ΔI_{+1} between adjacent timesteps in the time window $(t_0, t_0+\delta t)$. Therefore, the higher value of δt will adversely affect reading time. To address this problem, we instead store and compress a *higher-order* difference, $\Delta I_{+j} = I_\pi(t_0+j) - I_\pi(t_0)$, where $j \in (1, \delta t)$, for the growing value of δt until the size of the compressed index crosses a user-defined threshold. Once the threshold is crossed, the index for the current timestep is stored as is, and is considered as the new reference index.

5 Results

Evaluation of a lossy compression algorithm primarily depends on the accuracy of the fitting model and the compression ratio (CR) achieved. As this compression is performed *in-situ*, analysis of the time taken to perform the compression assumes significance as well. Here, we evaluate ISABELA with emphasis on the aforementioned factors, using normalized root mean standard error ($NRMSE$) and Pearson correlation (ρ) between the original and decompressed data as accuracy metrics. [Note that achieving $NRMSE \sim 0$, $\rho \sim 1$, and $CR \sim 100\%$ would indicate excellent performance.]



(a) (b)

Fig. 3. Accuracy (ρ): (a) Per window correlation for Wavelets, $B-$splines, and ISABELA with fixed $CR = 81\%$ for GTS Density. (b) Per window correlation for GTS linear and non-linear stage Potential decompressed by ISABELA.

5.1 Per Window Accuracy

In this section, we compare the Pearson correlation (ρ) between the original and decompressed data using Wavelets and B-Splines on original data and using ISABELA. The following parameters are fixed in this experiment: $CR = 81\%$ for Density, $W_0 = 1024$, $C_{B-spline} = 150$, and $C_{\text{ISABELA}} = 30$. Wavelet coefficients are thresholded to achieve the same compression rate. Figure 3(a) illustrates that ISABELA performs exceptionally well even for much smaller C values due to

the monotonic nature of the sorted data. In fact, $\rho > 0.99$ for almost all the windows. However, both Wavelets and $B-$splines exhibit a large degree of variation and poor accuracy across different windows. This translates to NRMSE values that are one-to-two orders of magnitude larger than the average 0.005 NRMSE value produced by ISABELA.

ISABELA performs exceptionally well on data from the linear stages of the GTS simulation (first few thousand timesteps), as shown in Fig. 3(b). Yet, the performance for the non-linear stages (timestep $\approx 10,000$), where the simulation is characterised by a large degree of turbulence, is of particular importance to scientists. Figure 3(b), with intentionally magnified correlation values, shows that accuracy for the non-linear stages across windows drops indeed. Unlike Wavelets (Fig. 3(a)), however, this correlation does not drop below 0.92.

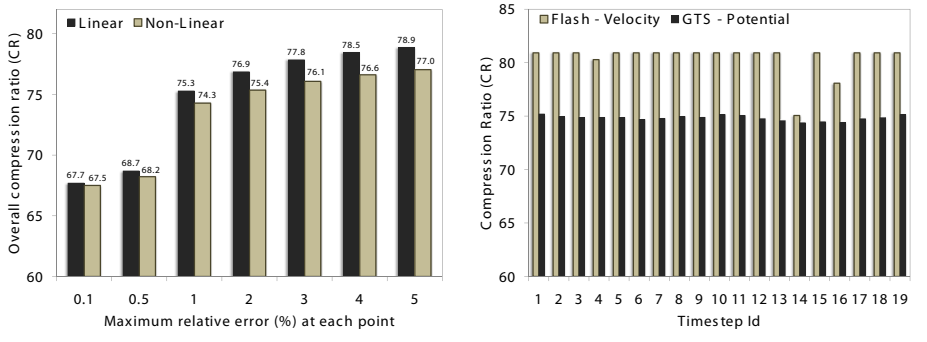


Fig. 4. Compression ratio (CR) performance: (a) For various per point relative error thresholds (τ_ϵ) in GTS Potential during linear and non-linear stages of the simulation. (b) For various timesteps with $\tau_\epsilon = 1\%$ at each point (for GTS Potential: $t_1 = 1,000$, $\Delta t = 1,500$; for Velocity in Flash: $t_1 = 3,000$, $\Delta t = 3,500$.

5.2 Trade-Off between Compression and Per Point Accuracy

To alleviate the aforementioned problem, we apply error quantization, as described in Sec. 4.4. In both linear and non-linear stages of GTS simulation, the compression ratios are similar when the per point relative error (τ_ϵ) is fixed (see Fig. 4(a)). This is because the relative error in consecutive locations for the sorted data tends to be similar. This property lends well to encoding schemes. Thus, even when the error tends to be higher in the non-linear stage, compared with the linear stage, the compression rates are highly similar. For $\tau_\epsilon = 0.1\%$ at each point, the CR lowers to an around 67.6%. This implies that by capturing 99.9% of the original values, the data from the simulation is reduced to less than one-third of its total size.

Figure 4(b) shows the compression ratio (with $\tau_\epsilon = 1\%$) over the entire simulation run using the GTS fusion simulation and Flash astrophysics simulation codes. For GTS Potential data, the compression ratio remains almost the same

across all stages of the simulation. With Flash, after error quantization, most relative errors are 0's. Compressing these values results in negligible storage overhead, and hence CR remains at 80% for the majority of timesteps.

5.3 Effect of Δ–encoding on Index Compression

In this section, we show that compressing along the time dimension further improves ISABELA's overall compression of spatio-temporal scientific datasets by up to 2%—5%. Table 3 show the compression rates achieved for different orders of $\Delta I_{+j}, j = 1, 2, 3$. While increasing W_0 improves spatial compression to a certain extent, it severely diminishes the reduction of the index along the temporal resolution. This is due to the fact that with larger windows and a larger δt between timesteps, the difference in index values lacks the repetitiveness necessary to be compressed well by standard lossless compression libraries.

Table 3. Impact of Δ–encoding on CR for Potential (Density)

W_0	Without Δ–encoding	ΔI_{+1}	ΔI_{+2}	ΔI_{+3}
512	80.08 (80.08)	81.83 (84.14)	81.87 (85.09)	81.68 (85.36)
1024	**81.44 (81.44)**	**83.14 (85.65)**	**83.21 (86.57)**	**82.98 (86.76)**
2,048	81.34 (81.34)	83.03 (85.56)	83.07 (86.44)	82.88 (86.66)
4,096	80.51 (80.51)	82.14 (84.64)	82.21 (85.51)	82.03 (85.76)
8,192	79.32 (79.32)	80.99 (83.38)	81.04 (84.24)	80.83 (84.46)

5.4 Compression Time

The overhead induced on the runtime of the simulation due to *in-situ* data compression is the net sum of the times taken to sort D, build I_π, and perform cubic $B-$spline fitting. However, for a fixed window size W_0, sorting and building the index is computationally less expensive compared to $B-$spline fitting. When executed in serial, ISABELA compresses data at ≈12 MB/s rate, the same as gzip, compression level 6, as shown in Table 2. Within the context of the running simulation, each core is expected to generate around 10 MB of data every 10 seconds that can be reduced to ≈2 MB in 0.83 seconds using ISABELA. Additionaly, to further reduce the impact of *in-situ* compression on the main computation, ISABELA can be executed at the I/O nodes rather than at the compute nodes [1,18].

In the case of compression, parallelization is achieved by compressing each window independently. However, a more fine-grain parallelization can be applied to decompression, as each point in the $B-$spline curve can be reconstructed independently. To evaluate the scalability and parallelization characteristics of ISABELA decompression, we evaluate the time taken for decompression against serial, OpenMP and GPU-based implementations in a single node environment. Figure 5 shows the performance of decompression of all three implementations. The serial implementation is faster when decompressing less than 1,000 points,

but as the number of decompressed points increases, both GPU and OpenMP versions offer the advantage in terms of computational time. This is especially true with a GPU-based implementation, which is better suited for fine-grain parallel computation.

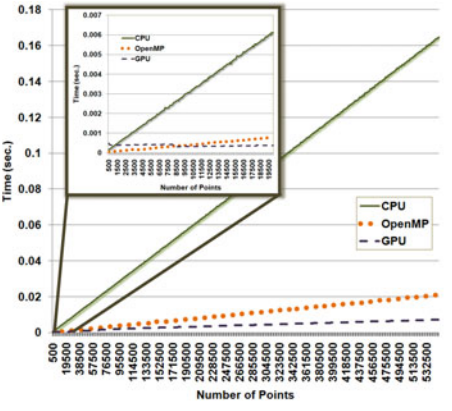


Fig. 5. Computational time for serial, CPU-parallelized, and GPU-parallelized versions of ISABELA's $B-$spline reconstruction part. 8 OpenMP threads were used in this particular plot, corresponding to two quad core Intel Xeon X5355 processors.

5.5 Performance for Fixed Compression

In this section, we evaluate the performance of ISABELA and Wavetlets on 13 public scientific datasets (from numerical simulations, observations, and parallel messages) [3] for the fixed $CR = 81\%$. We compare the averages of ρ_a and $NRMSE_a$ of ISABELA and Wavelets across 400 windows (see Table 4). Out of the 13 datasets, eleven (three) datasets exhibit $\rho_a = 0.98$ with ISABELA (Wavelets). The $NRMSE_a$ values for Wavelets are consistently an order of magnitude higher than for ISABELA. Wavelets outperform ISABELA on `obs_spitzer` consisting of a large number of piecewise linear segments for most of its windows. Cubic $B-$splines do not estimate well when segments are linear.

6 Related Work

Lossy compression methods based spline fitting or Wavelets have been primarily used in the field of visualization, geometric modeling, and signal processing. Very few studies applied such techniques when neither spatial nor temporal correlation of data can be directly exploited. Chou et al. [4] and Lee et al. [9] explored spline fitting for random data to optimize the location of control points to reduce approximation error. In contrast, we aim to transform the data to take advantage of the accuracy and easily-expressible qualities of splines.

Table 4. ISABELA vs. Wavelets for fixed $CR = 81\%$ and $W_0 = 1,024$.

	ρ_a		$NRMSE_a$	
	Wavelets	ISABELA	Wavelets	ISABELA
msg_sppm	0.400 ± 0.287	**0.982 ± 0.017**	0.203 ± 0.142	**0.051 ± 0.015**
msg_bt	0.754 ± 0.371	**0.981 ± 0.054**	0.112 ± 0.151	**0.038 ± 0.024**
msg_lu	0.079 ± 0.187	**0.985 ± 0.031**	0.422 ± 0.103	**0.048 ± 0.015**
msg_sp	0.392 ± 0.440	**0.967 ± 0.051**	0.307 ± 0.243	**0.064 ± 0.033**
msg_sweep3d	0.952 ± 0.070	**0.998 ± 0.006**	0.075 ± 0.036	**0.004 ± 0.003**
num_brain	**0.994 ± 0.008**	0.983 ± 0.028	**0.010 ± 0.011**	0.011 ± 0.005
num_comet	0.988 ± 0.018	**0.994 ± 0.025**	0.020 ± 0.020	**0.010 ± 0.006**
num_control	0.614 ± 0.219	**0.993 ± 0.017**	0.083 ± 0.037	**0.009 ± 0.002**
num_plasma	0.605 ± 0.062	**0.994 ± 0.004**	0.277 ± 0.038	**0.033 ± 0.004**
obs_error	0.278 ± 0.203	**0.994 ± 0.004**	0.303 ± 0.091	**0.024 ± 0.009**
obs_info	0.717 ± 0.136	**0.993 ± 0.006**	0.181 ± 0.078	**0.026 ± 0.016**
obs_spitzer	**0.992 ± 0.001**	0.742 ± 0.004	**0.005 ± 0.000**	0.030 ± 0.000
obs_temp	0.611 ± 0.114	**0.994 ± 0.011**	0.096 ± 0.025	**0.009 ± 0.003**

Lossless compression techniques [3,8,10,13] have been recently applied to floating point data. Unlike most lossless compression algorithms, the techniques presented in [3,10] are specifically designed for fast online compression of data. Lindstrom and Isenberg [10] introduced a method for compressing floating-point values of 2D and 3D grids that functions by predicting each floating point value in the grid and recording the difference between the predictive estimator and the actual data value. They also provide the option of discarding least significant bits of the delta and making the compresion lossy. However, the number of significant precision bits that can be saved is limited to 16, 32, 48, or 64 for double precision data. When applied to a one-dimensional data from GTS simulation, storing only 16 significant bits provided a compression of 82%, which is comparable with ISABELA's, but more than 75% of points had per-point relative error of over 1%. By storing 32 bits, the per-point relative error was found to be within 0.1%, but the compression rate achieved (58.2%) was 13% less than ISABELA's. Morever, like other lossless algorithms, location- specific decoding is not possible.

Most lossy compression techniques either use Wavelets or some form of data quantization to compress the data sets that are fed as input to visualization tools. However, visualization community focuses on providing multi-resolution view-dependent level of detail. The error rate tolerated with lossy compression techniques on data used for visualization tend to be higher when compared to the data used for analysis. Hence, very little work exists that accurately compresses non-image or seemingly random data, even outside the scientific community. In fact, to the best of our knowledge, ISABELA is the first approach to use $B-$spline fitting in the context of reduction for data that is essentially random.

7 Summary

This paper describes ISABELA, an effective *in-situ* method designed to compress spatio-temporal scientific data. ISABELA compresses data over both the spatial and temporal resolutions. For the former, it essentially applies data sorting, as a pre-conditioner, that significantly improves the efficacy of cubic B-spline spatial compression. For the latter, it uses Δ–encoding of the higher-order differences in index values to further reduce index storage requirements. By capturing the relative per point errors and applying error quantization, ISABELA provides over 75% compression on data from GTS, while ensuring 99% accuracy on all values. On 13 other scientific datasets ISABELA provides excellent approximation and reduction, consistently outperforming extensively used Wavelets compression.

Acknowledgements. This work was supported in part by the U.S. Department of Energy, Office of Science (SciDAC SDM Center, DE-AC02-06CH11357, DE-FC02-10ER26002/DE-SC0004935, DE-FOA-0000256, DE-FOA-0000257) and the U.S. National Science Foundation (CCF-1029711 (Expeditions in Computing)). Oak Ridge National Laboratory is managed by UT-Battelle for the LLC U.S. D.O.E. under contract no. DEAC05-00OR22725.

References

1. Abbasi, H., Lofstead, J., Zheng, F., Klasky, S., Schwan, K., Wolf, M.: Extending I/O through high performance data services. In: Cluster Computing, Austin, TX, IEEE International (September 2007)
2. De Boor, C.: A Practical Guide to Splines. Springer, Heidelberg (1978)
3. Burtscher, M., Ratanaworabhan, P.: FPC: A high-speed compressor for double-precision floating-point data (2009), http://www.csl.cornell.edu/~burtscher/research/FPC/
4. Chou, J., Piegl, L.: Data reduction using cubic rational B-splines. IEEE Comput. Graph. Appl. 12, 60–68 (1992)
5. Cover, T.M., Thomas, J.: Elements of information theory. Wiley-Interscience, New York (1991)
6. Frazier, M.W.: An introduction to Wavelets through linear algebra, p. 501. Springer, Heidelberg (1999)
7. He, X., Shi, P.: Monotone B-spline smoothing. Journal of the American Statistical Association 93(442), 643–650 (1998)
8. Isenburg, M., Lindstrom, P., Snoeyink, J.: Lossless compression of predicted floating-point geometry. Computer-Aided Design 37(8), 869–877 (2005); CAD 2004 Special Issue: Modelling and Geometry Representations for CAD
9. Lee, S., Wolberg, G., Shin, S.Y.: Scattered data interpolation with multilevel B-splines. IEEE Trans. on Viz. and Comp. Graphics 3(3), 228–244 (1997)
10. Lindstrom, P., Isenburg, M.: Fast and efficient compression of floating-point data. IEEE Trans. on Viz. and Comp. Graphics 12(5), 1245–1250 (2006)
11. Lofstead, J., Zheng, F., Klasky, S., Schwan, K.: Adaptable, metadata rich IO methods for portable high performance IO. In: IPDPS 2009, Rome, Italy (May 2009)

12. Ma, K., Wang, C., Yu, H., Tikhonova, A.: In-situ processing and visualization for ultrascale simulations. Journal of Physics: Conference Series 78(1), 012043 (2007)
13. Ratanaworabhan, P., Ke, J., Burtscher, M.: Fast lossless compression of scientific floating-point data. In: Proc. of the DCC (2006)
14. Sayood, K.: Introduction to data compression. Morgan Kaufmann Publishers Inc., San Francisco (1996)
15. Wang, W.X., al, e.: Gyro-kinetic simulation of global turbulent transport properties in Tokamak experiments. Physics of Plasmas 13(9), 092505 (2006)
16. Welch, T.A.: A technique for high-performance data compression. Computer 17, 8–19 (1984)
17. Wold, S.: Spline functions in data analysis. American Statistical Association and American Society for Quality 16(1), 1–11 (1974)
18. Zheng, F., al, e.: PreDatA–preparatory data analytics on peta-scale machines. In: IPDPS, Atlanta, GA (April 2010)

kNN Query Processing in Metric Spaces Using GPUs

Ricardo J. Barrientos[1], José I. Gómez[1], Christian Tenllado[1], Manuel Prieto Matias[1], and Mauricio Marin[2]

[1] Architecture Department of Computers and Automatic, ArTeCS Group, Complutense University of Madrid, Madrid, España
ribarrie@fdi.ucm.es
[2] Yahoo! Research Latin America, Santiago, Chile
mmarin@yahoo-inc.com

Abstract. Information retrieval from large databases is becoming crucial for many applications in different fields such as content searching in multimedia objects, text retrieval or computational biology. These databases are usually indexed off-line to enable an acceleration of on-line searches. Furthermore, the available parallelism has been exploited using clusters to improve query throughput. Recently some authors have proposed the use of Graphic Processing Units (GPUs) to accelerate brute-force searching algorithms for metric-space databases. In this work we improve existing GPU brute-force implementations and explore the viability of GPUs to accelerate indexing techniques. This exploration includes an interesting discussion about the performance of both brute-force and indexing-based algorithms that takes into account the *intrinsic dimensionality* of the element of the database.

1 Introduction

Similarity search has been widely studied in recent years and it is becoming more and more relevant due to its applicability in many important areas. Efficient kNN search, namely k nearest-neighbors search, is useful in multimedia information retrieval, data mining or pattern recognition problems. In general, when similarity search is undertaken by using metric-space database techniques, this problem is often featured by a large database whose objects are represented as high-dimensional vectors. A distance function operates on those vectors to determine how similar the objects are to a given kNN query object. The distance between any given pair of objects (i.e. high-dimensional vectors) is known to be an expensive operation to compute and thereby the use of parallel computation techniques can be an effective way to reduce running times to practical values in large databases.

In this paper we propose and evaluate efficient metric-space techniques to solve kNN search on GPUs. Obtaining efficient performance from this hardware can be particularly difficult in our application domain since metric-space solutions developed for traditional shared memory multiprocessors and distributed systems [16] cannot be implemented efficiently on GPUs.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 380–392, 2011.

Our focus is on search systems devised to solve large streams of queries. Conventional parallel implementations for clusters and multicore systems that exploit corse-grained inter-query parallelism are able to improve query throughput by employing index data structures constructed off-line upon the database objects. On GPUs we are able to exploit fine-grained parallelism and it can be more efficient to just resort to brute-force algorithms, especially for high-dimensional metric-spaces. The interesting problem to solve in this case is how to reduce the amount of work required to keep track of the current objects making into the kNN set. We propose a realization of this approach that outperforms alternative approaches to brute-force based on global ordering of the distances of database objects to the query. Our proposal keeps a partial ordering of objects which results from applying a novel strategy based on parallel priority queues.

We also experimented with a couple of metric-space index data structures that we have found amenable for GPU parallelization as they allow matrix-like computations. Finding a way of mapping these indexes onto GPUs resulted quite complex and tricky (the main difficulty is to exploit the available memory bandwidth), and thereby a second contribution of this paper is the proposal of a GPU based metric-space index data structure for similarity search.

The remaining of this paper is organized as follows. Section 2 gives some background information on similarity search and metric-space databases. Section 3 describes the main features of our computing platform and summarizes some previous related work. In Section 4 and 5 we introduce our proposals and discuss the most important finding from a performance evaluation against baseline strategies. We conclude in Section 6 highlighting our main contributions.

2 Similarity Search Background and Related Work

A *metric space* (X, d) is composed of an universe of valid objects $\mathbb{X}$ and a *distance function* $d : \mathbb{X} \times \mathbb{X} \rightarrow \mathbb{R}^+$ defined among them. The distance function determines the similarity between two given objects and holds several properties such as strict positiveness, symmetry, and the triangle inequality The finite subset $\mathbb{U} \subset \mathbb{X}$ with size $n = |\mathbb{U}|$, is called the database and represents the collection of objects of the search space.

There are two main queries of interest: ***Range Search*** [6] and ***The k nearest neighbors (kNN)*** [1,8]. In the former, the goal is to retrieve all the objects $u \in \mathbb{U}$ within a radius r of the query q (i.e. $(q, r)_d = \{u \in \mathbb{U} / d(q, u) \leq r\}$), whereas in the latter, the goal is to retrieve the set $kNN(q) \subseteq \mathbb{U}$ such that $|kNN(q)| = k$ and $\forall u \in kNN(q), v \in \mathbb{U} - kNN(q), d(q, u) \leq d(q, v)$.

For solving both kind of queries and to avoid as many distance computations as possible, many indexing approaches have been proposed. We have focused on the *List of Clusters* (LC) [5] and *SSS-Index* [2] strategies since (1) they are two of the most popular non-tree structures that are able to prune the search space efficiently and (2) they hold their indexes on dense matrices which are very convenient data structures for mapping algorithms onto GPUs [9].

In the following subsections we explain the construction of both indexes and describe how range queries are solved using them (range searches are simpler than kNN, but many kNN searches are built on them).

2.1 List of Clusters (LC)

This index [5] is built by choosing a set of centers $c \in U$ with radius r_c where each center maintains a bucket that keeps tracks of the objects contained within the ball (c, r_c). Each bucket holds the closest k-elements to c. Thus the radius r_c is the maximum distance between the center c and its k-nearest neighbor.

The buckets are filled up sequentially as the centers are created and thereby a given element i located in the intersection of two or more center balls remains assigned to the first bucket that hold it. The first center is randomly chosen from the set of objects. The next ones are selected so that they maximize the sum of the distances to all previous centers.

A range query q with radius r is solved by scanning the centers in order of creation. For each center $d(q, c)$ is computed and only if $d(q, c) \leq r_c + r$, it is necessary to compare the query against the objects of the associated bucket. This process ends up either at the first center that holds $d(q, c) < r_c - r$, meaning that the query ball (q, r) is totally contained in the center ball (c, r_c), or when all centers have been considered.

2.2 Sparse Spatial Selection (SSS-Index)

During construction, this pivot-based index [2] selects some objects as *pivots* from the collection and then computes the distance between these pivots and the rest of the database. The result is a table of distances where columns are the pivots and rows the objects. Each cell in the table contains the distance between the object and the respective pivot. These distances are used to solve queries as follows. For a range query (q, r) the distances between the query and all pivots are computed. An object x from the collection can be discarded if there exists a pivot p_i for which the condition $|d(p_i, x)d(p_i, q)| > r$ does hold. The objects that pass this test are considered as potential members of the final set of objects that form part of the solution for the query and therefore they are directly compared against the query by applying the condition $d(x, q) \leq r$. The gain in performance comes from the fact that it is much cheaper to effect the calculations for discarding objects using the table than computing the distance between the candidate objects and the query.

A key issue in this index is the method that calculates the pivots, which must be good enough to drastically reduce total number of distance computations between the objects and the query. An effective method is as follows. Let $(\mathbb{X}, d)$ be a metric space, $\mathbb{U} \subset \mathbb{X}$ an object collection, and M the maximum distance between any pair of objects, $M = max\{d(x, y)/x, y \in \mathbb{U}\}$. The set of pivots contains initially only the first object of the collection. Then, for each element $x_i \in \mathbb{U}$, x_i is chosen as a new pivot if its distance to every pivot in the current set of pivots is equal or greater than αM, being α a constant parameter. Therefore,

an object in the collection becomes a new pivot if it is located at more than a fraction of the maximum distance with respect to all the current pivots.

3 Graphic Processing Units (GPU)

GPUs have emerged as a powerful cost-efficient many-core architecture. They integrate a large number of functional units following a SIMT model. We develop all our implementations using NVIDIA graphic cards and its CUDA programming model ([9]). A CUDA *kernel* executes a sequential code on a large number of threads in parallel. Those threads are grouped into fixed size sets called *warps*[1]. Threads within a *warp* proceed in a lock step execution. Every cycle, the hardware scheduler of each GPU multiprocessor chooses the next warp to execute (i.e. no individual threads but warps are swapped in and out). If the threads in a warp execute different code paths, only those that follow the same path can be executed simultaneously and a penalty is incurred.

Warps are further organized into a grid of *CUDA Blocks*: threads within a block can cooperate with each other by (1) efficiently sharing data through a shared low latency local memory and (2) synchronizing their execution via barriers. In contrast, threads from different blocks can only coordinate their execution via accesses to a high latency global memory. Within certain restrictions, the programmer specifies how many blocks and how many threads per block are assigned to the execution of a given kernel. When a kernel is launched, threads are created by hardware and dispatched to the GPU cores.

According to NVIDIA the most significant factor affecting performance is the bandwidth usage. Although the GPU takes advantage of multithreading to hide memory access latencies, having hundreds of threads simultaneously accessing the global memory introduces a high pressure on the memory bus bandwidth. The memory hierarchy includes a large register file (statically partitioned per thread) and a software controlled low latency shared memory (per multiprocessor). Therefore, reducing global memory accesses by using local shared memory to exploit inter thread locality and data reuse largely improves kernel execution time. In addition, improving memory access patterns is important to allow coalescing of warp loads and to avoid bank conflicts on shared memory accesses.

4 GPU Mapping of *k*-Nearest Neighbor Algorithms

In this section we describe the mapping of three k-NN algorithms onto CUDA-enabled GPUs: a brute-force approach and two index-based search methods.

All of them exploit two different levels of parallelism. As in some previous papers [11] [15] [7] we assume a high frequency of incoming queries and exploit coarse-grained inter-query parallelism. However, we also exploit the fine-grained parallelism available when solving a single query. Overall, each query is processed by a different CUDA Block that contains hundreds of threads (from 128 to 512,

[1] Currently, there are 32 threads per *warp*.

depending of the specific implementation) that efficiently cooperate to solve it. Communication and synchronization costs between threads within the same CUDA Block are rather low, so this choice looks optimal to fully exploit the enormous parallelism present in k-NN algorithms.

Another common point of our three implementations is the usage of priority queues (implemented using a heap [12]) to keep track of the potential candidates found by the threads. This avoids the sorting of the full vector of distances at the cost of a final reduction stage (described in subsection 4.1), as well as increasing the irregularity of the accesses. Nevertheless, data locality is optimized as much as possible holding queries and heaps in shared memory whenever possible.

4.1 Exhaustive Search Algorithm

k-NN is typically implemented on GPUs using brute force methods applying a two-stage scheme. First, all the distances from the target query to the elements on the database are evaluated and then a second stage sorts these distances to obtain the nearest elements. In [10], the final stage is implemented with a modified parallel insertion sort in order to just get the k closest elements, whereas in [13] authors used an improved Radix-sort. In both cases, full GPU resources are employed to solve a single query. As mentioned above, we assume a high frequency of incoming queries and also exploit a coarser level of parallelism by solving several queries simultaneously. In this case, the first stage becomes a matrix multiplication ($Q * U$, where Q rows hold the queries and U columns store the database), which can be implemented very efficiently on GPUs [19].

Our implementation of the *sorting* phase is based on a *logarithmic reduction* algorithm that consists of three steps. First, the distance vector is evenly distributed across the CUDA Block threads. To fulfill CUDA alignment and coalescing requirements, elements are assigned in a round-robin fashion such that concurrent accesses of threads within a warp are performed to consecutive memory addresses. Each thread keeps its own private heap to store its partial K results (i.e. the K minimal values found in its part of the vector). For real size problems, the size of hte distance vector is higher than K; thus, the time to fill each heap is almost negligible. In the *steady state*, each thread must compare the new distance with the top of the heap. Just in case this distance is lower than the current top, a heap insertion is performed. As computation evolves, is less and less likely to find smaller elements. At that point, memory accesses become very regular and threads within a warp almost never diverge.

The other two stages implement the reduction of the local heaps and are common to all our implementations as mentioned above. The second step starts once all the threads in the CUDA Block have finished its assigned computations. A synchronization barrier is needed to ensure that all threads have finished before starting this step. The input to this stage is a set of *tpb* heaps, each filled with K elements (*tpb* stands for the number of threads per CUDA Block). Now, just one *warp* from the whole CUDA Block becomes active. At this point, we sacrifice parallelism to guarantee a better memory exploitation, which reveals to be much more relevant for final performance. Heaps are statically assigned, again

in a round-robin fashion, to each individual thread within the *warp*. Each thread will traverse its set of heaps, keeping the K minimal values found. Once again, a heap is used to store temporal results and the output of this step consists of 32 (the warp size) heaps filled with K elements. For the investigated values of K, these heaps can be allocated on the shared memory.

In the last step, a single thread performs the reduction of the K smaller values out of the 32 heaps. There is no need of explicit synchronization between steps in this case, due to the locked-step execution of threads in the same warp during the second step. The final results are also stored on a heap, allocated in shared memory (in order to keep the memory footprint as low as possible, in-place mapping is performed if the set of 32 heaps from step 2 were also allocated in shared memory). Both in the second and third step we exploit the fact that input elements are already organized in heaps, which allows us to heavily reduce the number of comparisons and insertions in the new output heap.

4.2 LC

The data structure that holds the LC index consists of 3 arrays denoted as *CENTER*, *RC* and *CLUSTERS*. *CENTER* is a $D \times N_{cen}$ matrix (D is the dimension of the elements[2] and N_{cen} is the number of centers), where each column represents the center of a cluster, *RC* is an array that stores the covering radius of each cluster, and *CLUSTERS* is a $D \times (B_{size} \cdot N_{clu})$ matrix (N_{clu} is the number of clusters and B_{size} is the number of elements per clusters) that holds the elements of each cluster. Index information is stored column-wise to favor coalesce memory accesses.

kNN queries are solved with LC indexes using auxiliary range queries with increasing or decreasing radius. The former starts performing a range query with a given initial range R_{ini} and repeat those range queries increasing the radius by Δ until the nearest K-elements are found[3]. The latter starts with $R_{ini} = \infty$ and performs a single iteration in which the radius is reduced dynamically. Sequential implementations usually employ the decreasing alternative since it provides better performance [6]. However the increasing radius strategy exhibits higher inherent parallelism and we have analyzed both of them. In both cases, parallelism comes from the distribution of distance evaluations.

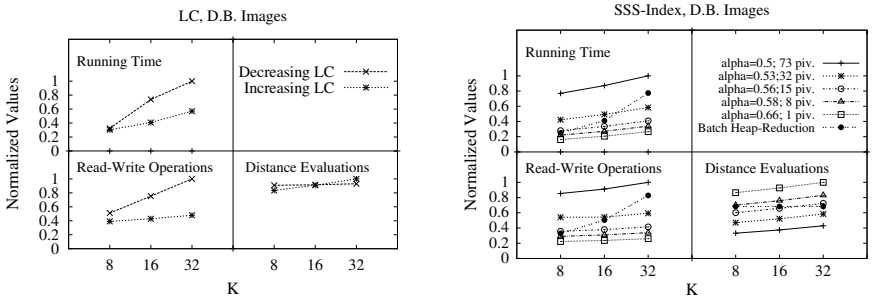
In our implementations, all threads evaluate first the distance between the assigned query q and a subset of the elements of *CENTER* following a Round-Robin distribution of this matrix. Based on these distances some centers and their respective clusters are discarded using triangle inequality. For clusters cannot be discarded, more distances are evaluated in parallel following again Round-Robin distribution of the *CLUSTERS* matrix. As in the exhaustive approach, each thread holds potential candidates in a local heap, which are finally reduced using the same logarithmic strategy. For the decreasing radius strategy we start

[2] For the *Spanish* database, D is the maximum size of a word.

[3] Both parameters R_{ini} and Δ are usually set empirically with an off-line analysis of the database using a small sample of its elements. In our experiments we have used less than 1% of their elements to set them.

with $R_{ini} = \infty$ and perform some initial updates of this radius when threads fill their local heaps for the first time. At these events, the radius is adjusted with the longest distance processed so far by the thread (i.e. the root of its local heap) using CUDA `atomicMin`($radius_{current}$, $local_{longest-distance}$) function. Later on, the radius is adjusted similarly whenever a new element is inserted on a local heap. The *current radius* is always used to test if the elements of a given cluster can be discarded using the triangle inequality.

As shown in Figure 1(a), although both methods are able to discard a similar proportion of the database collection, the increasing strategy outperforms the decreasing counterpart. Note that in a sequential setting, the decreasing approach scans the database elements in order and may adjusts the current radius after processing every single element. However, in a parallel setting these updates are performed at a higher granularity since many elements are processed in parallel. Therefore, the current radius decreases *more slowly* and less clusters are discarded. Furthermore, warps divergences are more expensive in the decreasing strategy. They occur when new elements are inserted into local heaps and in the decreasing method involve an additional atomic instruction in order to adjust the radius. As a consequence of these penalties, the decreasing method exhibits a worse memory behavior since costly divergences prevent coalesce memory accesses.



(a) (b)

Fig. 1. Performance of kNN queries using a) *LC* with both increasing and decreasing radius range queries and b) *SSS-Index* with different number of pivots.

4.3 SSS-Index

We have used 3 matrices to implement *SSS-Index*: *PIVOTS*, *DISTANCES* and *DB*. *PIVOTS* is a $D \times N_{piv}$ matrix (D is the dimension of the elements and N_{piv} is the number of pivots) where each column represents a pivot. *DISTANCES* is a $N_{piv} \times N_{DB}$ matrix (N_{DB} = number of elements of the database) where each row holds the distance vector between pivots and an element of the database. *DB* is the reference database. As in the *LC*, the index information is stored column-wise to favor coalesce memory accesses.

As centers in LC, pivots are distributed across threads following a round-robin distribution to evaluate their distance with the query. On a later stage, the rows of *DISTANCES* are distributed across threads, that test if their respective elements of the database can be discarded. For every non discarded element, a distance evaluation is performed and a set of heaps is filled accordingly (as in previous implementations).

In [2], authors have found empirically that $\alpha = 0.4$ yields the minimal number of distance evaluations. Our own experiments on GPUs confirm this behavior: the more pivots are used (up to a certain threshold), the less distance evaluations are performed. However, as shown in Figure 1(b), the best performance is obtained with just one pivot. Indeed the more pivots used, the worst the execution time becomes. *Irregularity* explains this apparent contradiction: when using more pivots, threads within a warp are more likely to diverge. Moreover, memory access pattern becomes more irregular and hardware cannot coalesced them. This leads to the observed increase in the number of Read/Write operations. Summarizing, less distance evaluations do not pay off due to the overheads caused by warp divergences and irregular access patterns. Overall, just one pivot provides the optimal performance for many of our reference databases.

5 Experimental Results

As computing platforms we have used a NVIDIA GeForce GTX 280 GPU (30 multiprocessors, 8 cores per multiprocessor, 16K of shared memory) equipped with 4GB of device memory and an Intel's Clovertown processor with 16 GB of RAM. We have use three different reference databases (described below) and the parameter K (the number of nearest neighbors) has been set to 8, 16 and 32. Similar values have been also used in previous papers [13] [10] [3].

***Spanish*:** A Spanish dictionary with 51,589 words and we used the *edit distance* [14] to measure similarity. On this metric-space we processed 40,000 queries selected from a sample of the Chilean Web which was taken from the TODOCL search engine. This can be considered a low dimensional metric space.

***Images*:** We took a collection of images from a NASA database containing 40,700 images vectors, and we used them as an empirical probability distribution from which we generated a large collection of random image objects containing 120,000 objects. We built each index with the 80% of the objects and the remaining 20% objects were used as queries. In this collection we used the *euclidean distance* to measure the similarity between two objects. Intrinsic dimensionality of this space higher than in the previous database, but it is still considered low.

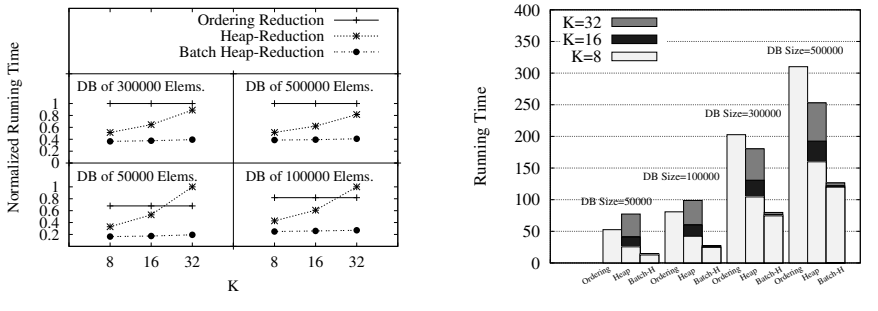
***Faces*:** This database was created from a collection of 8480 face images obtained from Face Recognition Grand Challenge [17]. We apply the *Eigen Face Method* [18] to obtain a projection matrix, that can be used to generate a feature vector from any face image. We used this collection as an empirical probability distribution from which we generated a large collection of random face image objects containing 120,000 objects. We used the *euclidean distance* to measure the similarity between two objects. Each *eigenface* consist in a vector of 254 elements.

Even if the intrinsic dimensionality of the space may be lower than 254, it is large enough to ruin traditional indexing benefits.

Figure 2 compares different exhaustive search methods. *Ordering reduction* stands for the state-of-the-art solution: all the distance evaluations are performed first. Next, the whole GPU is devoted to sort the obtained distances per query (i.e., queries are solved one at a time, and full resources are employed to sort a single vector of distances). A very efficient parallel version of the *quicksort* algorithm is employed at that step [4]. *Batch-Heap Reduction* corresponds with the technique explained in Section 4.1: one CUDA Block solves a single query and multiple queries are solved in parallel. Finally, we include a third version labeled *Heap-Reduction* that follows our implementation but solves just one query at a time. Figure 2(a) compares normalized running times for different reference database size and different values of K. The points are normalized to the largest value of the experiment. Figure 2(b) shows the absolute running time of the same set of experiments.

Our proposals are able to outperform the *Ordering* counterpart in most experiments. Even if we solve one query at a time (*Heap-Reduction* in the Figure) we outperform sorting-based algorithms for large databases and small values of K due to a better memory management. When we exploit the full strength of GPU launching as many CUDA Blocks as queries the difference increases and, more relevant, our implementation becomes much less sensitive to K. Note the performance of any sorting-based implementation is independent of K, since they sort the full distance vector).

We now turn our attention to the proposed indexing algorithms. For LC, the preliminary analysis presented in Section 4.2 motivated us to pick 64 as the number of elements per cluster in the high-dimensional database, while lowering it to 32 in the *Spanish* database. Similarly, we restrict ourselves to the increasing radius approach since it always perform better than the decreasing counterpart. Regarding *SSS-Index*, and following the conclusions drawn in Section 4.3, we

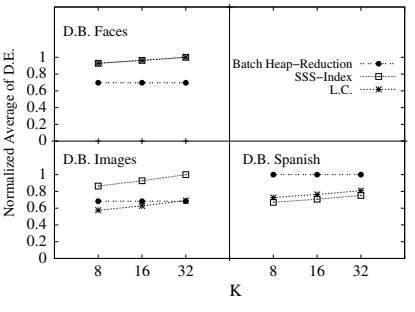


(a) (b)

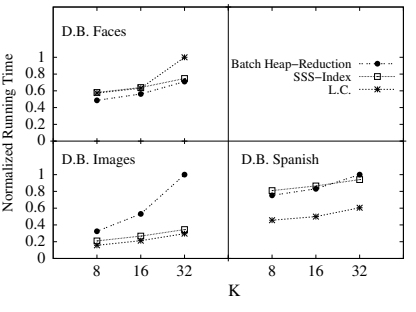
Fig. 2. Normalized (a) and absolute (b) running times of the investigated exhaustive search algorithms for different K and number of elements using *Faces* database

use just a single pivot ($\alpha = 0.66$) for vector high-dimensional databases and 68 pivots ($\alpha = 0.5$) for *Spanish* database.

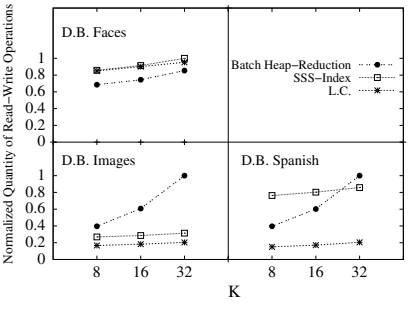
Figure 3 shows different set of results to illustrate several important findings of our three implementations. We first place our attention on the total number of distance evaluations (Figure 3(a)). The *Spanish* database behaves as expected: indexing mechanisms do significantly decrease the number of distance evaluations when compared to the exhaustive search method. However, as space dimensionality increases, that is no longer the case. Indeed the opposite behavior is observed: indexing mechanisms perform more distance evaluations than the exhaustive algorithm. This is specially true for *SSS-Index* with just one pivot. Obviously, this fact implies that some evaluations are performed more than once with the indexing mechanism, which is possible due to the increasing radius approach. It must be noted that we intentionally decided not to reuse distance evaluations to avoid repeated computations since it introduces an enormous source of irregularity. On GPUs, decreasing the amount of work in this way, does not pay off.



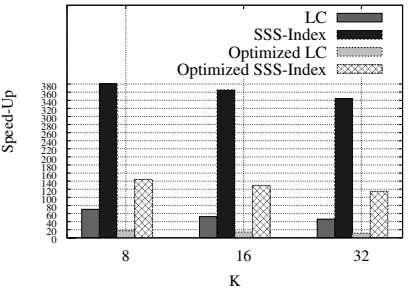
(a) Number of Distance Evaluations



(b) Running time



(c) Number of read/write operations



(d) Speed-Up of *LC* and *SSS-Index*

Fig. 3. Normalized a) Distance evaluations per query (average) b) Running time and c) Read-write Operations (of 32, 64 o 128 bytes) to *device memory*. d) Speed-Up of *LC* and *SSS-Index* over sequential counterparts with DB *Images*.

One would expect that running times mimic the trend exhibited by the distance evaluations but results in Figure 3(b) partially contradicts this intuition: the exhaustive search algorithm behaves worse than expected, specially for the *Images* database. Figure 3(c) has the clue: memory access pattern, which heavily influences performance on current GPUs, behaves better for the indexing mechanism, specially for *LC*. As stated in Section 3, when a warp launches misaligned or non-consecutive memory accesses, hardware is not able to coalesce it and a single reference may become up to 32 separate accesses. In all our implementations, heap insertions usually imply warp divergences and lack of locality, thus increasing the number of read/write operations. Indexed algorithms performs more distance evaluations but, since many distance evaluations are evaluated several times, the number of heap insertions is significantly reduced. However, as dimensionality increases the higher cost of these evaluations starts to tradeoff the difference in heap insertions. There, indexed mechanisms perform poorly and our exhaustive-search implementation outperforms both of them. The *Faces* database, the one with largest dimensionality in our experiments, illustrates this situation (see Figure 3(b)).

Finally, Figure 3(d) shows the performance speed-ups of our indexed implementations over its corresponding sequential implementations. For each of the two algorithms, we implemented a naive *out-of-the-box* version and a heavily tuned and compiler optimized one. Results are very impressive for *SSS-Index* due to the poor CPU performance of this indexing mechanisms. Regarding the optimized sequential version, we obtain up to 144x speedup for $k = 8$. But even for the lighter index (*List of Clusters*) our implementation achieves reasonable speedups of a 17x. Our experiments show that this speed-up is not easy to attend with OpenMP versions running on medium-sized clusters.

6 Conclusions

In this paper we have presented efficient implementations of typical indexing mechanisms together with an exhaustive-search version which are mapped on CUDA based GPUs.

We may highlight the following findings after our exploration: 1) when performing *kNN* search based on *range searches* in parallel, an *increasing radius* strategy becomes more efficient than the traditional *decreasing radius*. This is specially true for GPUs given their memory system restrictions. 2) Optimal parameters for both, *List of Clusters* and *SSS-Index* metrics are extremely different than those found on distributed implementations. In particular, the best GPU implementation found for *SSS-Index* uses a single pivot to prune the search space, which is highly inefficient since this pivot is found randomly. 3) In such a context, considering running time, our exhaustive-search proposal outperforms the indexed versions whereas for the lower dimension datasets our index strategies outperform exhaustive-search.

Note that from sequential computing literature we can learn that in metric-spaces with very high dimensions, indexing strategies are no longer useful as they

lose their ability to reduce running time. Here, the only option is to compare the query against the whole set of database objects. For that case, the exhaustive search strategy proposed in this paper is clearly the most efficient alternative for GPU based metric-space query processing.

References

1. Aha, D.W., Kibler, D.: Instance-based learning algorithms. In: Machine Learning, pp. 37–66 (1991)
2. Brisaboa, N.R., Fariña, A., Pedreira, O., Reyes, N.: Similarity search using sparse pivots for efficient multimedia information retrieval. In: ISM, pp. 881–888 (2006)
3. Bustos, B., Deussen, O., Hiller, S., Keim, D.A.: A graphics hardware accelerated algorithm for nearest neighbor search. In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2006. LNCS, vol. 3994, pp. 196–199. Springer, Heidelberg (2006)
4. Cederman, D., Tsigas, P.: Gpu-quicksort: A practical quicksort algorithm for graphics processors. J. Exp. Algorithmics 14, 1.4–1.24 (2009)
5. Chavéz, E., Navarro, G.: An effective clustering algorithm to index high dimensional metric spaces. In: The 7th International Symposium on String Processing and Information Retrieval (SPIRE 2000), pp. 75–86. IEEE CS Press, Los Alamitos (2000)
6. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.L.: Searching in metric spaces. In: ACM Computing Surveys, pp. 273–321 (September 2001)
7. Costa, V.G., Barrientos, R.J., Marín, M., Bonacic, C.: Scheduling metric-space queries processing on multi-core processors. In: Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2010), pp. 187–194. IEEE Computer Society, Pisa (2010)
8. Cover, T., Hart, P.: Nearest neighbor pattern classification. IEEE Transactions on Information Theory 13(1), 21–27 (1967), `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1053964`
9. CUDA: Compute Unified Device Architecture. ©2007 NVIDIA Corporation, `http://developer.nvidia.com/object/cuda.html`
10. Garcia, V., Debreuve, E., Barlaud, M.: Fast k nearest neighbor search using gpu. In: Computer Vision and Pattern Recognition Workshop, pp. 1–6 (2008)
11. Gil-Costa, V., Marin, M., Reyes, N.: Parallel query processing on distributed clustering indexes. Journal of Discrete Algorithms 7(1), 3–17 (2009)
12. Knuth, D.E.: The Art of Computer Programming, vol. 3. Addison-Wesley, Reading (1973)
13. Kuang, Q., Zhao, L.: A practical gpu based knn algorithm, Huangshan, China, pp. 151–155 (2009)
14. Levenshtein, V.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 10, 707–710 (1966)
15. Marin, M., Gil-Costa, V., Bonacic, C.: A search engine index for multimedia content. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 866–875. Springer, Heidelberg (2008)

16. Marin, M., Ferrarotti, F., Gil-Costa, V.: Distributing a metric-space search index onto processors. In: 39th International Conference on Parallel Processing, ICPP 2010, San Diego, California, pp. 13–16 (2010)
17. Phillips, P.J., Flynn, P.J., Scruggs, T., W., K., Bowyer, J.C., Hoffman, K., J., Marques, J.M., Worek, W.: Overview of the face recognition grand challenge. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2005, vol. 1, pp. 947–954 (June 2005)
18. Turk, M., Pentland, A.: Eigenfaces for recognition. Journal of Cognitive Neuroscience 3(1), 71–86 (1991)
19. Volkov, V., Demmel, J.W.: Benchmarking gpus to tune dense linear algebra. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC 2008, pp. 31:1–31:11. IEEE Press, Piscataway (2008), http://portal.acm.org/citation.cfm?id=1413370.1413402

An Evaluation of Fault-Tolerant Query Processing for Web Search Engines

Carlos Gomez-Pantoja[1,2], Mauricio Marin[1,3], Veronica Gil-Costa[1,4], and Carolina Bonacic[1]

[1] Yahoo! Research Latin America, Santiago, Chile
[2] DCC, University of Chile, Chile
[3] DIINF, University of Santiago of Chile, Chile
[4] CONICET, University of San Luis, Argentina

Abstract. A number of strategies to perform parallel query processing in large scale Web search engines have been proposed in recent years. Their design assume that computers never fail. However, in actual data centers supporting Web search engines, individual cluster processors can enter or leave service dynamically due to transient and/or permanent faults. This paper studies the suitability of efficient query processing strategies under a standard setting where processor replication is used to improve query throughput and support fault-tolerance.

1 Introduction

Search engines index very large samples of the Web in order to quickly answer user queries. The data structure used for this purpose is the so-called inverted file [2]. In order to support parallel query processing the inverted file is evenly distributed among P processors forming a cluster of computers. Here either a *local* or *global* approach to indexing can be employed. A number of parallel query processing strategies can be applied to process queries under the local and global indexing approaches. They are based on what we call below *distributed* and *centralized* ranking. Each one has its own advantages under an idealized setting in which processors never fail.

The research question in this paper is what combination of indexing and ranking is the most suitable one under a real-life setting in which processors can leave and re-enter service at unpredictable time instants. We focus on the case in which full quality of service is required, namely the final outcome is not affected by processor failures. More specifically, responding approximated answers to user queries by ignoring the contribution of temporarily out-of-service processors is not an option. This implies that queries (or part of them) hit by failures must be re-executed trying to keep as low as possible their response times.

Note that the complete processing of a query goes through several major steps where usually each step is executed in a different set of cluster processors. The system is dimensioned to make each query last for a very small fraction of a second in each cluster and thereby query re-execution from scratch upon a failure

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 393–404, 2011.

is feasible. At the same time, query response time per step cannot be too high since the cumulative sum of response times must not overcome a given upper bound for the latency experienced by users. In this paper we focus on the most costly step for a query, that is, the determination of the top-R documents IDs that best fit the query. In this scenario, our aim is to know how the response time of the individual queries that were active at the instant of a failure is affected and how a failure affects the stability of the whole system in aspects such as overall load balance and query throughput.

To compare the approaches under exactly the same conditions we developed discrete-event simulators validated against actual implementations, which model the essentials of the query processing tasks and uses traces from actual executions to generate the work-load associated with each ranking and indexing strategy.

The remaining of this paper is organized as follows. Section 2 describes the indexing and ranking strategies. Section 3 describes the experimental testbed used to obtain the results presented in Section 4. Conclusions are in Section 5.

2 Indexing and Ranking

User queries are received by a broker machine that distributes them among processors (nodes) forming a cluster of P machines. The processors/nodes work cooperatively to produce query answers and pass the results back to the broker. Support for fault-tolerance is made by introducing replication. To this end, and assuming a P-processor search cluster, $D-1$ copies for each of the P processors are introduced. The system can be seen as a $P \times D$ matrix in which each row maintains an identical copy of the respective partition (column) of the data.

Local and global indexing: Each column keeps a $1/P$ fraction of an index data structure, called inverted file [2], which is used to speed up the processing of queries. In order to support parallel query processing the inverted file is evenly distributed among the P columns and replicated $D-1$ times along each column. An inverted file is composed of a vocabulary table and a set of posting lists. The vocabulary table contains the set of distinct relevant terms found in the document collection. Each of these terms is associated with a posting list that contains the document identifiers where the term appears in the collection along with additional data used for ranking purposes.

The two dominant approaches to distributing an inverted file [1,9,10,11,15,17,7] among P processors are: (**a**) the document partitioning strategy (also called *local indexing*), in which the documents are evenly distributed among the processors and an inverted index is constructed in each processor using the respective subset of documents, and (**b**) the term partitioning strategy (called *global indexing*), in which a single inverted file is built from the whole text collection to then evenly distribute the terms and their respective posting lists among the processors.

Distributed and centralized ranking: Trying to bring into the same comparison context the strategies reported in the literature is quite involved since almost each research group uses different methods for document ranking, query

processing upon the term, and document partitioned inverted files [1,9,10,15,17]. Some use either intersection (AND) or union (OR) queries [6,8,18]. Others perform exhaustive traversal of posting lists while others do list pruning which can be made on posting lists sorted by term frequency or document ID.

In this work, the ranking of documents is performed using the list pruning method described in [16]. This method generates a workload onto processors that is representative of other alternative methods [2,4,5]. In this case, posting lists are kept sorted by in-document term frequency. The method works by pushing forward a barrier $S_{\max}$ [16]. A given posting list pair (*id_doc*, *freq*) is skipped if the term frequency *freq* in the document *id_doc* is not large enough to overcome the barrier $S_{\max}$. For either local or global indexing, one can apply this method for either distributed or centralized ranking.

For local indexing under distributed ranking, the broker sends a given query to a processor selected in a circular manner. We say this processor becomes the *merger* for this query. A merger receives a query from the broker and broadcasts it to all processors. Then all processors execute the document ranking method and update their barrier $S_{\max}$ using the local posting lists associated with the query terms. After these calculations, each processor sends to the merger its local top-R results to let the merger calculate the global top-R documents and send them to the broker machine. These documents are then sent to a second cluster of computers, which contain the actual text of documents, in order to produce the final answer web-page presented to the user.

For global indexing under distributed ranking, the broker sends the query to one of the processors that contain the query terms (the least loaded processor is selected). This processor becomes the merger for the query. In the case that the query terms are not located all in the same processor, the merger processor sends the query to all other processors containing query terms. Then those processors compute their estimation of the top-R results. In this case, however, the processors that do not contain co-resident terms can only compute an approximation of the ranking because they do not have information about the remote posting lists to increase their respective barriers $S_{\max}$.

In [1], it is suggested that processors should send to the merger at least $6\,R$ results for global indexing. The merger must re-compute the document scores by using information of all terms and then produce the global top-R results. Note that with probability $1/P$ two query terms are co-resident. This probability can be increased by re-distributing terms among processors so that terms appearing very frequently in the same queries tend to be stored in the same processors. This can be made by obtaining term correlation information from actual queries submitted in the recent past. We call this approach *clustered* global indexing.

Centralized ranking [3,12] is different in the sense that now pieces of posting lists are sent to processors to be globally ranked rather than performing local ranking on them in their respective processors. The rationale is that the granularity of ranking is much larger than the cost of communication in current cluster technology and that by performing global ranking the barrier $S_{\max}$ goes up much faster which significantly reduces the overall fraction of posting lists

scanned during ranking. Centralized ranking stops computations earlier than distributed ranking. In practice, this means that each query requires less processor time to be solved.

For local indexing under centralized ranking the broker sends the query to one of the processors selected circularly. This processor becomes the ranker for the query. Then the ranker broadcasts the query to all processors and they reply with a piece of posting list of size K/P for each query term (with K fairly larger than R, say $K = 2\,R$).

The ranker merges these pieces of posting lists and execute the ranking method sequentially. After this, for all terms in which the pruning method consumed all the pairs (*id_doc*, *freq*), a new block of K/P pairs per term is requested to the P processors (ranker included) and the same is repeated. The ranking ends up after completing a certain number of these iterations of size K per term.

For global indexing under centralized ranking the broker sends the query to one of the processors circularly selected. This processor becomes the ranker for the query and it might not contain any of the query terms. The reason for this is to favor the load balance of the ranking process (at the expense of communication) which is the most costly part of the solution of a query. Upon reception of a query, the ranker processor sends a request for the first K-sized piece of posting list for each query term to each processor holding them. The arriving pairs (*id_doc*, *freq*) are merged and passed through the ranking method sequentially as in the local indexing and centralized ranking case.

The disadvantage of the global indexing approach is the above mentioned AND queries in which a large amount of communication is triggered each time it is necessary to calculate the intersection of two posting lists not located in the same processor. Another difficulty in the global index is the huge extra communication required to construct the index and distribute it among the processors. A practical solution to this problem has been proposed in [11].

Note that in the local indexing approach it is possible to reduce the average number of processors involved in the solution of queries as proposed in [13] by using a location cache. This comes from the fact that one can re-order documents stored in the processors in such a way that they form clusters to which queries can be directed. At search time, when there is not enough information to determine which of the clusters can significantly contribute to the global top-R results, the query is sent to all processors as in the conventional approach. In this paper we study this approach as well. We call it *clustered* local indexing.

Overall, in this paper we investigate how efficient are the different query processing strategies listed in Table 1 under fails of randomly selected processors.

3 Experimental Framework

The processing of a query can be basically decomposed into a sequence of very well defined operations [12]. For instance, for local indexing and centralized ranking, the work-load generated by the processing of an one-term query requiring r iterations to be completed and arriving to processor p_i can be represented by the sequence,

Table 1. The different strategies for parallel query processing investigated in this paper and the code names used to label them in the figures of Section 4

Code	*Query Processing Strategy*
GIDR	Global Indexing with Distributed Ranking
CGIDR	Clustered Global Indexing with Distributed Ranking
LIDR	Local Indexing with Distributed Ranking
CLIDR	Clustered Local Indexing with Distributed Ranking
GICR	Global Indexing with Centralized Ranking
CGICR	Clustered Global Indexing with Centralized Ranking
LICR	Local Indexing with Centralized Ranking
CLICR	Clustered Local Indexing with Centralized Ranking

Table 2. Boldface letters in the expressions stand for the primitive operations Broadcast (B), Fetch (F), Rank (R), Send (S) and Merge (E). Also $M \leq P$ and we use "+" to indicate one or more repetitions of the same sequence of operations.

Code	*Most-likely Sequence of Primitive Operations for Two-Terms Queries*
GIDR	$[\,\mathbf{F}(K) \,\|\|\, 2 \rightarrow \mathbf{R}(K) \,\|\|\, 2\,]^{+} \rightarrow \mathbf{S}(6R) \rightarrow \mathbf{E}(12R)^{\langle p_i \rangle}$ (just one term in p_i)
CGIDR	$[\,2\,\mathbf{F}(K) \rightarrow \mathbf{R}(2K)\,]^{+}$ (both terms are in p_i with high probability)
LIDR	$\mathbf{B}(2t)_{P}^{\langle p_i \rangle} \rightarrow [\,2\,\mathbf{F}(K/P) \,\|\|\, P \rightarrow \mathbf{R}(2K/P) \,\|\|\, P\,]^{+} \rightarrow \mathbf{S}(R) \,\|\|\, P \rightarrow \mathbf{E}(PR)^{\langle p_i \rangle}$
CLIDR	$\mathbf{B}(2t)_{M}^{\langle p_i \rangle} \rightarrow [\,2\,\mathbf{F}(K/M) \,\|\|\, M \rightarrow \mathbf{R}(2K/M) \,\|\|\, M]^{+} \rightarrow \mathbf{S}(R) \,\|\|\, M \rightarrow \mathbf{E}(MR)^{\langle p_i \rangle}$
GICR	$[\,\mathbf{F}(K) \,\|\|\, 2 \rightarrow \mathbf{S}(K) \rightarrow \mathbf{R}(2K)^{\langle p_i \rangle}\,]^{+}$ (one term in p_i, probability $1-1/P$)
CGICR	$[\,2\,\mathbf{F}(K) \rightarrow \mathbf{R}(2K)\,]^{+}$ (both terms are in p_i with high probability)
LICR	$\mathbf{B}(2t)_{P}^{\langle p_i \rangle} \rightarrow [\,2\,\mathbf{F}(K/P) \,\|\|\, P \rightarrow 2\,\mathbf{S}(K/P) \,\|\|\, P \rightarrow \mathbf{R}(2K)^{\langle p_i \rangle}\,]^{+}$
CLICR	$\mathbf{B}(2t)_{M}^{\langle p_i \rangle} \rightarrow [\,2\,\mathbf{F}(K/M) \,\|\|\, M \rightarrow 2\,\mathbf{S}(K/M) \,\|\|\, M \rightarrow \mathbf{R}(2K)^{\langle p_i \rangle}\,]^{+}$

$$\text{Broadcast}(t)^{\langle p_i \rangle} \rightarrow \Big[\,\text{Fetch}(K/P) \,\|\, P \rightarrow \text{Send}(K/P) \,\|\, P \rightarrow \text{Rank}(K)^{\langle p_i \rangle}\,\Big]^{r}$$

The broadcast operation represents the part in which the ranker processor p_i sends the term t to all processors. Then, in parallel ($\|$), all P processors perform the fetching, possibly from secondary memory, of the K/P sized pieces of posting lists. Also in parallel they send those pieces to the ranker p_i. The ranker merges them and performs a ranking of size K repeated until completing the r iterations. Similar workload representations based on the same primitives Broadcast, Fetch, Send, and Rank can be formulated for the combinations of Table 1. The exact order depends on the particular indexing and ranking methods, and basically all of them execute the same primitive operations but with **(i)** different distribution among the processors, **(ii)** different timing as their use of resources is directly proportional to their input parameter (e.g., Send(K) or Send(R)$\|\ P$), and **(iii)** different number of repetitions r. For the sake of a fair comparison, we perform distributed ranking partitioned in quanta of size K for global indexing and K/P for local indexing (this does not imply an extra cost, it just fixes the way threads compete and use the hardware resources in the simulator). We ignore the cost of merging because it is comparatively too small with

respect to ranking. In Table 2 we show the sequences of primitive operations for the strategies described in Table 1.

For each query we obtained the exact trace or sequence of primitive calls (including repetitions) from executions using query logs on actual implementations of the different combinations for ranking and indexing. Those programs also allow the determination of the relative cost of each primitive with respect to each other. We included this relative cost into the simulator. Apart from allowing a comparison under exactly the same scenario, the simulator with its primitive operations competing for resources, has the advantage of providing an implementation independent view of the problem since results are not influenced by interferences such as programming decisions and hardware among others.

3.1 Process-Oriented Discrete-Event Simulator

The query traces are executed in a process-oriented discrete-event simulator which keeps a set of $P \times D$ concurrent objects of class Processor. The simulator keeps at any simulation time instant a set of $q \cdot P$ active queries, each at a different stage of execution as its respective trace dictates. An additional concurrent object is used to declare out of service a processor selected uniformly at random, and another object is used to re-start those processors. These events take place at random time intervals with negative exponential distribution.

For each active query there is one fetcher/ranker thread per processor and one ranker/merger thread for centralized/distributed ranking. To cause cost, the simulator executes query quanta for both centralized and distributed ranking (Table 2). The type of indexing determines the size of the quanta, either K or K/P, to be used by each primitive operation and also the level of parallelism, either P, number of non-resident terms or none. The quanta can be K/M, with $M \leq P$, when clustered local indexing is used. Here the level of parallelism is M, where M is the number of processors where the query is sent based on a prediction of how "good" are document clusters stored in those processors for the query (for each query, M is determined from actual execution traces).

The asynchronous threads are simulated by concurrent objects of class Thread attached to each object of class Processor. Competition for resources among the simulated threads is modeled with P concurrent queuing objects of class CPU and *ram-cached* Disk which are attached to each Processor object respectively, and one concurrent object of class Network that simulates communication among processors. Each time a thread must execute a Rank(x) operation it sends a request for a quanta of size x to the CPU object. This object serves requirements in a round-robin manner. In the same way a Fetch(x) request is served by the respective Disk object in a FCFS fashion. These objects, CPU and Disk, execute the SIMULA like `Hold(time_interval)` operation to simulate the period of time in which those resources are servicing the requests, and the requesting Thread objects "sleep" until their requests are served.

The concurrent object Network simulates an all-to-all communication topology among processors. For our simulation study the particular topology is not really relevant as long as all strategies are compared under the same set of

resources CPU, Disk and Network, and their behavior is not influenced by the indexing and ranking strategy. The Network object contains a queuing communication channel between all pairs of processors. The average rate of channel data transfer between any two processors is determined empirically by using benchmarks on the hardware described in Section 4.

Also the average service rate per unit of quanta in ranking and fetching is determined empirically using the same hardware, which provides a precise estimation of the relative speed among the different resources. We further refined this by calibrated the simulator to achieve a similar query throughput to the one achieved by the actual hardware. This allows us to evaluate the consequences of a processor failure under a fairly realistic setting with respect to its effects in the throughput and average response time of queries.

The simulator has been implemented in `C++` and the concurrent objects are simulated using `LibCppSim` library [14]. To obtain the exact sequence of primitive operations (Table 2) performed by each strategy (Table 1), we executed `MPI-C++` implementations of each strategy on a cluster of computers using $P=32$ processors and an actual query log. We indexed in various forms a 1.5TB sample of the UK Web and queries were taken from an one year log containing queries submitted by actual users to www.yahoo.co.uk.

3.2 Simulating Failures

During the simulation of *centralized* ranking and just before a failure, all processors are performing their roles of rankers and fetchers for different queries. Upon failure of processor p_i, all queries for which p_i was acting as ranker must be re-executed from scratch. This is mainly so because the current (potentially large) set of candidate documents to be part of the global top-R results is irreversibly lost for each query in which p_i was their ranker. In this case a new ranker processor for those queries must be selected from one of the $D-1$ running replicas of processor p_i and re-execution is started off. Nevertheless, all pieces of posting lists already sent by the fetchers are at this point cached in their respective processors so the disk accesses are avoided and the extra cost comes only from the additional communication. For other queries, the processor p_i was acting as a fetcher (and sender) of pieces of posting lists. The respective rankers detect inactivity of p_i and send their following requests for pieces of posting lists to one of the $D-1$ running replicas. In both cases (i.e., p_i ranker and fetcher) the replicas are selected uniformly at random.

For *distributed* ranking and just before a failure, the processors are performing merging of local top-R results to produce the global top-R ones for a subset of the active queries. They are also performing local ranking to provide their local top-R results to the mergers of other subset of queries. Upon failure of processor p_i, a new replica p_i is selected uniformly at random for each query for which p_i is a merger. The other processors performing local-ranking for those queries send their local top-R results to those replicas of p_i and the replicas selected as mergers must re-execute the local ranking for those queries to finally obtain the global top-R results for the affected queries. The processor p_i was also acting as

local ranker for mergers located in other processors. In that case, the randomly selected replicas p_i recalculate the local top-R for the affected queries and send them to their mergers.

3.3 Simulator Validation

Proper tuning of the simulator cost parameters is critical to the comparative evaluation of the strategies. We set the values of these parameters from actual implementation runs of the query processing strategies. These implementations do not support fault tolerance and thereby are not able to exactly reproduce the same conditions before, during and after processor failures. Such an objective would be impossible in practice given the hardware and system software available for experimentation. Even so, this would require us to modify the MPI library protocols to make it able to detect failures without aborting the program in all processors and to selectively re-start queries hit by failures. These protocols have to be application specific as it is a firm requirement that response time of queries must be below a fraction of a second. This excludes MPI realizations prone to scientific and grid computing applications that provide support for processor failures. This also excludes systems such as map-reduce (e.g., Hadoop) which are also intended to perform off-line processing in terms of what is understood as on-line processing in Web search engines.

Nevertheless, the simulator allows us to achieve the objectives of evaluating different failure scenarios. These scenarios start from a situation in which the simulator mimics, in a very precise manner, the steady state regime of the actual program running the respective query processing strategy. From that point onwards processor failures are injected, there is an increase in load for surviving processors, the underlying network must cope with lost connections, and signal re-execution of queries. The cost parameters of the concurrent objects do not change in this case and these objects properly simulate saturation when some sections of the network and processors are overloaded.

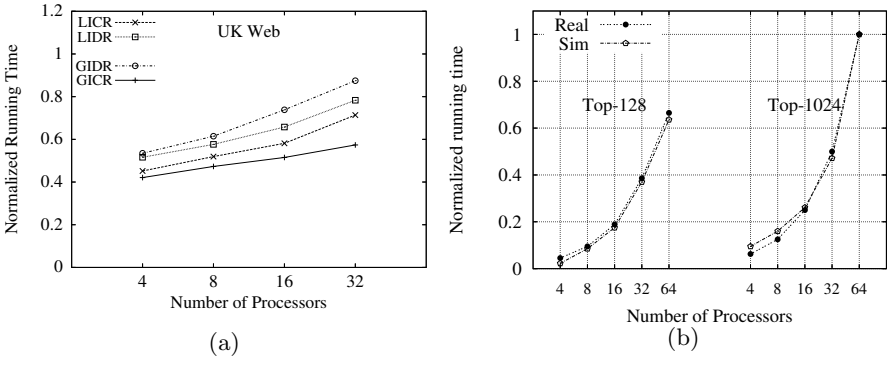


Fig. 1. Validation of simulations against actual implementations

Figure 1 shows results for overall running time achieved with actual implementations of the query processing strategies. We injected the same total number of queries in each processor. Thereby, running times are expected to grow with the number of processors. The results from the simulator (omitted in Figure 1.a, differences below 1%) overlap each curve very closely. Figure 1.b shows results both for real execution time and simulation time for the LIDR strategy. These results are for an implementation of the LIDR strategy independent to ours. They were obtained with the Zettair Search Engine (www.seg.rmit.edu.au/zettair) and we adjusted our simulator parameters to simulate the performance of Zettair. We installed Zettair in the 32 processors of our cluster and broadcast queries to all processors using TCP/IP sockets to measure query throughput. Again the simulator is capable of almost overlapping each point in the curves of Figure 1.b. The two sets of curves show results for K= 128 and 1024 respectively.

4 Comparative Evaluation

In the following, we show results normalized to 1 in order to better understand and illustrate a comparative analysis in terms of percentage differences among the strategies. We assume that main memory of processors is large enough to make negligible disk accesses as it indeed occurs in current search engines.

Figures 2.a, 2.c, 2.e and 2.g show results for centralized ranking (CR) upon global and local indexing (GI and LI respectively) for $P = 32$ and $D = 4$. Figure 2.a shows the average number of queries that are affected by failures for different rates of failures. The x-axis values indicate a range of rates of failures from to very high to very low; the value 10 indicates a very high rate whereas 80 indicates a very low rate. Figure 2.c shows the effect in the response time of queries that must be re-executed from scratch upon a failure. Figure 2.e shows the effect in all of the queries processed during the experiments. Figure 2.e shows overall query throughput.

Firstly, these results show that overall queries are not significantly affected by failures in the $P \times D$ arrangement of processors. Secondly, these results indicate that the strategies that aim at reducing the number of processors involved in the solution of individual queries are more effective. This is counter intuitive since as they use less processors, the amount of calculations that must be re-executed is larger than the strategies that use more processors to solve individual queries. However, these strategies are fast enough to let re-executions be performed with a more quiet effect in the surrounding queries not hit by the failure.

The same trend is observed in the strategies that use distributed ranking (DR) upon the global and local indexes (GI and LI). This can be seen in the Figures 2.b, 2.d, 2.f and 2.h. The performance of these strategies is quite behind the centralized ranking strategies. On average they performed at least 5 more iterations during query processing making that the impact of failures were more significant in terms of query throughput degradation. This is reflected in Figures 3.a and 3.b which show the average response time for all queries (3.a) and for queries involved in a failure only (3.b).

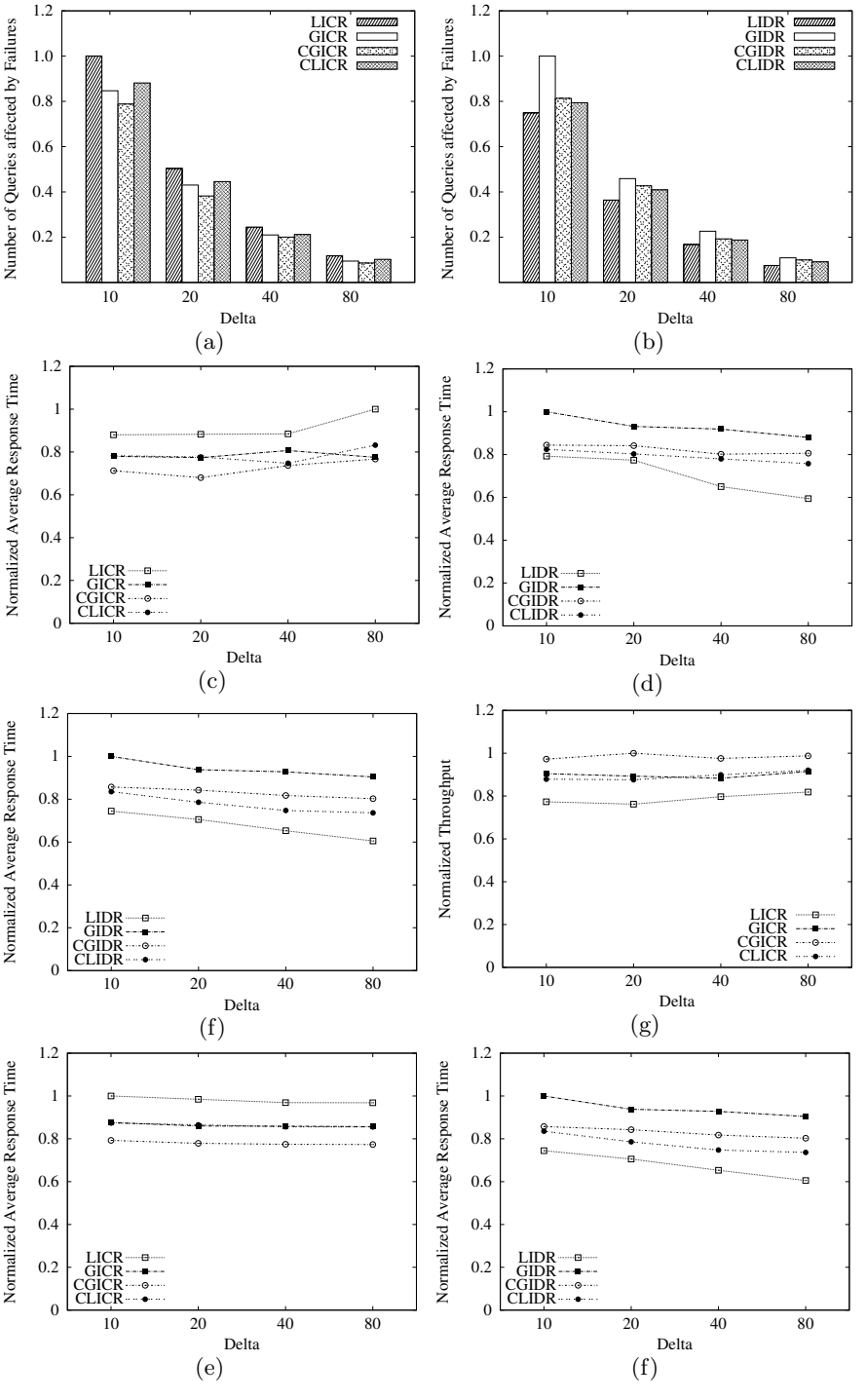


Fig. 2. Experiments for centralized and distributed ranking

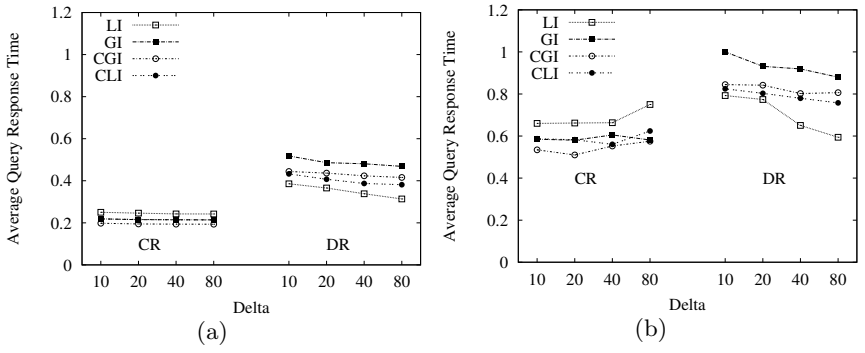


Fig. 3. Normalized average response times for all queries (a) and for queries involved in failures (b). Both figures are in the same normalized scale.

5 Concluding Remarks

In this paper we have evaluated different strategies for indexing and ranking in Web search engines under a fault-tolerant scenario. We resorted to simulators to re-produce exactly the same conditions for each tested strategy. The simulators were properly tuned with actual implementations of the strategies. Simulations and actual executions agreed within a small difference. For each strategy, traces were collected from the execution of the actual implementations and injected in the respective simulators to cause cost in the simulated time and react upon failures by re-executing the respective traces of affected queries. The failure arrival rate was kept independent of the query processing strategy being simulated.

The results from a set of relevant performance metrics, clearly show that centralized ranking strategies behave better than distributed ranking strategies upon failures. This holds for both local and global indexing. At first glance this appears counter intuitive since distributed ranking gets more processors involved in the document ranking process and thereby upon a failure it is only necessary to re-execute $1/P$-th of the computations, with P being the number of involved processors. In contrast, centralized ranking assigns just one processor to the document ranking process and thereby the query must be re-executed from scratch in another processor. The key point is that centralized ranking is much faster than distributed ranking and this makes the difference in a environment prone to failures since the system quickly gets back into steady state.

In addition, the simulations show that global indexing achieves better performance than local indexing for list-pruning ranking methods under processor failures. This kind of methods have become current practice in major vertical search engines since they reduce the amount of hardware devoted to process each single query. Our results show that global indexing in combination with centralized ranking is able to significantly reduce hardware utilization for disjunctive queries. Conjunctive queries are better adapted to local indexing but in both cases, centralized ranking is a better alternative than distributed ranking.

A combination of distributed and centralized ranking is also possible: a first round can be made using the centralized approach to quickly increase the global $S_{\max}$ barrier for the query, and then the global barrier is communicated to the involved processors so that they can use it to perform distributed ranking from this point onwards. This combination, which reduces overall communication, is expected to be useful in cases in which the ranking method is not able to aggressively prune posting list traversal for all query terms. We plan to study this alternative in the near future.

References

1. Badue, C., Baeza-Yates, R., Ribeiro, B., Ziviani, N.: Distributed query processing using partitioned inverted files. In: SPIRE, pp. 10–20 (November 2001)
2. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley, Reading (1999)
3. Bonacic, C., Garcia, C., Marin, M., Prieto, M.E., Tirado, F.: Exploiting Hybrid Parallelism in Web Search Engines. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 414–423. Springer, Heidelberg (2008)
4. Broder, A.Z., Carmel, D., Herscovici, M., Soffer, A., Zien, J.Y.: Efficient query evaluation using a two-level retrieval process. In: CIKM, pp. 426–434 (2003)
5. Broder, A.Z., Ciccolo, P., Fontoura, M., Gabrilovich, E., Josifovski, V., Riedel, L.: Search advertising using web relevance feedback. In: CIKM, pp. 1013–1022 (2008)
6. Chaudhuri, S., Church, K., Christian König, A.: Liying Sui. Heavy-tailed distributions and multi-keyword queries. In: SIGIR, pp. 663–670 (2007)
7. Ding, S., Attenberg, J., Baeza-Yates, R.A., Suel, T.: Batch query processing for Web search engines. In: WSDM, pp. 137–146 (2011)
8. Falchi, F., Gennaro, C., Rabitti, F., Zezula, P.: Mining query logs to optimize index partitioning in parallel web search engines. In: INFOSCALE, p. 43 (2007)
9. Jeong, B.S., Omiecinski, E.: Inverted file partitioning schemes in multiple disk systems. TPDS 16(2), 142–153 (1995)
10. MacFarlane, A.A., McCann, J.A., Robertson, S.E.: Parallel search using partitioned inverted files. In: SPIRE, pp. 209–220 (2000)
11. Marin, M., Gil-Costa, V.: High-performance distributed inverted files. In: CIKM 2007, pp. 935–938 (2007)
12. Marin, M., Gil-Costa, V., Bonacic, C., Baeza-Yates, R.A., Scherson, I.D.: Sync/async parallel search for the efficient design and construction of web search engines. Parallel Computing 36(4), 153–168 (2010)
13. Marin, M., Gil-Costa, V., Gomez-Pantoja, C.: New caching techniques for web search engines. In: HPDC, pp. 215–226 (2010)
14. Marzolla, M.: Libcppsim: a Simula-like, portable process-oriented simulation library in C++. In: ESM, pp. 222–227. SCS (2004)
15. Moffat, A., Webber, W., Zobel, J., Baeza-Yates, R.: A pipelined architecture for distributed text query evaluation. Information Retrieval (August 2007)
16. Persin, M., Zobel, J., Sacks-Davis, R.: Filtered document retrieval with frequency-sorted indexes. JASIS 47(10), 749–764 (1996)
17. Xi, W., Sornil, O., Luo, M., Fox, E.A.: Hybrid partition inverted files: Experimental validation. In: Agosti, M., Thanos, C. (eds.) ECDL 2002. LNCS, vol. 2458, pp. 422–431. Springer, Heidelberg (2002)
18. Zhang, J., Suel, T.: Optimized inverted list assignment in distributed search engine architectures. In: IPDPS (2007)

Introduction

Ramin Yahyapour, Christian Pérez, Erik Elmroth, Ignacio M. Llorente, Francesc Guim, and Karsten Oberle

Topic chairs

There has been a considerable discussion in the past years on the similarities and differences between Clouds and Grid Computing. This included extreme positions whether Clouds are a pure marketing hype, or whether Grids became obsoleted by failing a wide commercial adoption as a resource sharing platform. Now, we can assess that neither is true nor necessary. Grids and clouds share many similarities as they both address questions concerning access to resources in a large-scale distributed environment. Thus, there is significant overlap between the two areas in the ways that infrastructures may evolve. Grids became a common production infrastructure especially in the scientific domain including cluster and HPC resources. Clouds became a common provisioning paradigm in service infrastructures with many public/commercial providers or private clouds within data centers. It is very successful in creating a layered architecture that separates the infrastructure access from applications through virtualization. Infrastructure as a Service can be utilized to run arbitrary applications. Similarly, applications can be broken down to several software services which run on such virtual infrastructures. Grids target a similar space by combining resources from different providers in a networked infrastructure.

Both domains cannot only co-exist, but also benefit each other. The core research challenges are very similar and include, for example, topics like resource management, scheduling, SLA management, security or workflow management.

The topic 6 of Euro-Par addresses such core research topics Grid Cluster and Cloud Computing. The call for participation asked for contributions on Grid and Cloud middlewares, applications and platforms. In the following, you will find contributions which focus on different aspects.

We saw an increased interest in the aggregation and federation of Grids and Clouds which requires suitable models and protocols to interoperate between systems. This includes questions on migration of virtual machines between cloud providers as well as the consideration of virtual clusters on top of Clouds. Such approaches need suitable solutions for efficient data management.

Similarly, Quality-of-Service and Service-Level-Agreement gained attention in the scientific and industrial domain. The set of selected papers include work in the area of SLA management for Cloud environments in which a rule based approach is proposed to manage such agreements.

Efficiency of infrastructure management remains a key topic. This includes aspects of Green computing by considering carbon efficiency of nodes. The load balancing and scheduling has been subject to research for many years. However,

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 405–406, 2011.

this aspect is still very relevant due to the complexity at hand. This year's contribution include autonomic self-management aspects, as well as optimization mechanisms through load-balancing

The number of submission to this topic reflected the high interest in this area. The selection process was very competitive and we are happy to achieve a very good coverage of different perspectives. All papers were reviewed by at least three, usually four, independent reviewers. The selection process was not easy as many papers provided very good research insights and interesting approaches.

We would like to thank all the reviewers, for their time and effort, who helped us in the selection process. At the same time, we would like to thank all authors who help to maintain Euro-Par as one of the premier scientific conferences at which innovative ideas for Grid, Cluster and Cloud computing are presented.

Self-economy in Cloud Data Centers: Statistical Assignment and Migration of Virtual Machines

Carlo Mastroianni[1], Michela Meo[2], and Giuseppe Papuzzo[1]

[1] ICAR-CNR, Rende (CS), Italy
{mastroianni,papuzzo}@icar.cnr.it
[2] Politecnico di Torino, Italy
michela.meo@polito.it

Abstract. The success of Cloud computing has led to the establishment of large data centers to serve the increasing need for on-demand computational power, but data centers consume a huge amount of electrical power. The problem can be alleviated by mapping virtual machines, VMs, which run client applications, on as few servers as possible, so that some servers with low traffic can be put in low consuming sleep modes. This paper presents a new approach for the adaptive assignment of VMs to servers and their dynamic migration, with a twofold goal: reduce the energy consumption and meet the Service Level Agreements established with users. The approach, based on ant-inspired algorithms, founds on statistical processes: the mapping and migration of VMs are driven by Bernoulli trials whose success probability depends on the utilization of single servers. Experiments highlight the two main advantages with respect to the state of the art: the approach is self-organizing and mostly decentralized, since each server locally decides whether or not a new VM can be served, and the migration process is continuous and adaptive, thus avoiding the need for the simultaneous reassignment of many VMs.

1 Introduction

The need for on-demand computing, i.e., the possibility of using computational resources on a pay-as-you-go basis, was identified many years ago, but so far it is has been hindered by technological constraints. Recently, the availability of powerful data centers and high bandwidth connections have expedited the success of the Cloud computing paradigm, which is making on-demand computing a common practice for many enterprizes and scientific communities. The main advantage of this paradigm is that a company does not need to operate its own data center, with all the related costs and administration burdens, but can access to CPU power, storage facilities, software packages on the basis of current needs. For example, a Web server operated by a company can be hosted by a Cloud center, a choice that has many advantages in addition to money savings, among which higher security and availability guarantees (anti-hacker and back up procedures are managed by IT professionals), and much lower or even zero risks of under- or over-provisioning of resources. These advantages are particularly welcome by small companies, especially in their start up phase [3].

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 407–418, 2011.

One of the main issues related to the success of Cloud computing is that the ever growing number of large data centers is causing a notable increase of electrical power consumed by hardware facilities and cooling systems. This increases the cost of computation itself and affects the carbon footprint of data centers, thus aggravating, on the global scale, the problem of global warming. It has been estimated that in 2006 the energy consumed by IT infrastructures in USA was about 61 billion kWh, corresponding to 1.5% of all the produced electricity, and these figures are expected to double by 2011 [2].

A major reason for this huge amount of consumed power is the inefficiency of data centers, which are often under-utilized: it has been estimated that only 20-30% of the total server capacity is used on average [1]. Despite the adoption of techniques that try to scale the energy consumption with respect to the actual utilization of a computer (for example, Dynamic Voltage and Frequency Scaling or DVFS), an idle server still consumes approximately 65-70% of the power consumed when it is fully utilized [8]. To cope with this problem, Cloud data centers exploit the *virtualization* paradigm: user processes are not assigned directly to servers, but are first associated with Virtual Machine (VM) instances, which, in turn, are run by servers. The use of virtualization allows heterogeneous platforms to be executed on any kind of hardware facility, which facilitates the *consolidation* of VMs, that is, their clustering on as few computers as possible.

Unfortunately, the optimal mapping of VMs to servers, so as to minize energy consumption, is an NP-hard problem and requires a full knowledge of the servers load. Centralized algorithms that explore sub-optimal solutions may be computationally costly, and do not scale well with the size of the system. This paper presents a self-organizing approach that is partly inspired by the basic ant algorithms used by Deneubourg et al. [6] to model the phenomenon of larval clustering in ant colonies. In our case, the approach aims at clustering VMs in as few servers as possible, using two types of statistical procedures, for the *assignment* and the *migration* of VMs. Specifically, a new VM is assigned to one of the available servers through statistical Bernoulli trials for which the success probability depends on the current utilization of the servers. The *assignment probability function* is defined so as to favor the assignment of a VM to a highly loaded server, in order to improve VM consolidation. On the other hand, the migration procedure fosters the migration of VMs from servers in which the current utilization is either too high or too low, that is, above or below two defined thresholds. In the first case, the migration of a VM helps to prevent a possible overload of the server, which may lead to Service Level Agreement (SLA) violations. In the second case, the objective of the migration is to take VMs away from lightly loaded servers, and then power off these servers. Migration is also driven by Bernoulli trials, for which the success probability is defined by appropriate *migration probability functions*.

The use of statistical processes has two important advantages: (i) assignment and migration processes are self-organizing and mostly decentralized. The data center manager coordinates the processes, but decisions are taken locally by each server on the basis of local information; (ii) the migration process is

continuous and gradual, and the cost of migration (e.g., performance degradation) is smoothed over time, so that the quality of service perceived by users is hardly affected. Conversely, several approaches proposed in the literature (e.g., [12] and [2]) often require the simultaneous migration of many VMs. These properties, self-organization and gradual migration of VMs, favor the scalability of the approach and its capacity to adapt to the dynamic workload of client applications.

The rest of the work is organized as follows: Section 2 discusses the assignment and migration procedures; Section 3 reports the results of simulation experiments, which prove that the approach succeeds in efficiently consolidating VMs, and that power consumption is close to the theoretical minimum, while the number of SLA violations is minimized; Section 4 describes related work, and Section 5 concludes the paper and proposes some avenues for future work.

2 Assignment and Migration of Virtual Machines

This section describes the statistical procedures used for the assignment of VMs to the data center servers and for their dynamic migration. The examined scenario is pictured in Figure 1: an application request is transmitted from a client to the data center manager, which selects a VM that is appropriate for the application, on the basis of application characteristics such as the amount of required resources (CPU, memory, storage space) and the type of operating system specified by the client. Then, the VM is assigned to one of the available multi-core servers through the *assignment procedure*. The workload of the application is dynamic, that is, its demand for CPU varies with time, provided that it does not exceed the VM capacity. This is typical, for example, of Web servers, for which the CPU demand depends on the workload generated by Web users. Periodically, each server checks if its CPU utilization is between the specified upper and lower thresholds and, when this condition is violated, it activates the *migration procedure*, in order to move one VM to another server. The parameters λ and μ shown in Figure 1 are, respectively, the arrival rate of application requests and the service rate of a server core. They will be used in Section 3 for the performance analysis.

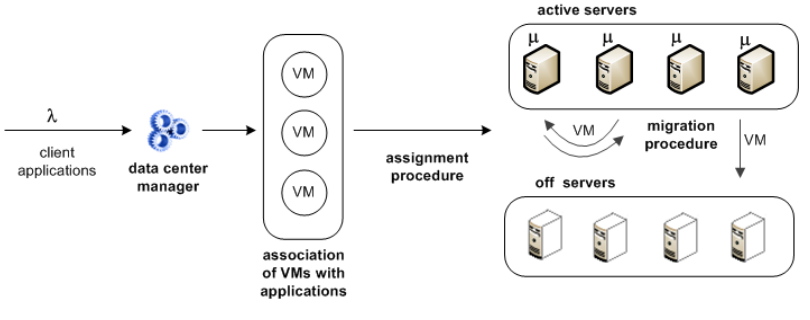


Fig. 1. Assignment and migration of VMs in a data center

2.1 Assignment Procedure

Once a client application is associated with a compatible VM, the latter must be assigned to a server for execution. The choice should take into account the following considerations: (i) it is preferable to assign the VM to a server with high CPU utilization, in order to enforce the consolidation of VMs and possibly allow idle servers to be powered off; (ii) the CPU utilization should not be too close to the server capacity: in such a case, if the VMs workload increases, the server may be unable to grant the amount of CPU required by the applications, and SLA violations may occur; (iii) the VM should be allocated on a powered off server only when strictly necessary, since switching on a server reduces consolidation and increases consumed power.

Given these objectives, the *assignment procedure* is defined as follows. The data center manager broadcasts the assignment request to servers[1]. Each active server executes a Bernoulli trial, whose success probability depends on its current CPU utilization, u (valued between 0 and 1), and on the maximum allowed utilization, T_a. The *assignment probabilistic function*, $f_{assign}(u)$, is null when $u > T_a$, otherwise it is defined as:

$$f_{assign}(u) = 1/M_p \cdot u^p \cdot (T_a - u) \qquad M_p = \frac{p^p}{(p+1)^{(p+1)}} \cdot T_a^{(p+1)} \tag{1}$$

Figure 2 shows the function graph for some values of the integer parameter p, and $T_a = 0.9$. The factor $1/M_p$ is used to normalize the maximum value to 1. The function definition ensures that the CPU utilization cannot exceed the threshold T_a (because no further VMs can be assigned when u reaches this threshold) and that VMs are preferably assigned to highly loaded servers, thus favoring consolidation. The value of u at which the function reaches its maximum - that is, the value at which assignment attempts succeed with the highest probability - is $p/(p+1) \cdot T_a$, which increases and approaches T_a as the value of p increases. Therefore, the value of p can be used to modulate the shape of the function and tune the consolidation effort.

Each server for which the Bernoulli trial succeeds, responds to the broadcast message declaring its availability to accommodate the VM. Then, the data center manager randomly selects one of these available servers, and assigns the VM to it. If all active servers are unavailable - because their utilization exceeds the threshold T_a, or because Bernoulli trials are unsuccessful - an inactive server will be switched on and will accommodate the VM. If this is not possible, because all the servers are already active, the VM will be forcedly assigned to any server that has some spare CPU fraction (such a server may be chosen after a second broadcast request), or it will be put in a waiting queue: this is a hint that the number of servers is too low to sustain the load.

[1] The broadcast strategy is the most reasonable when all the servers are located in a single high-speed network. If the servers are grouped in multiple clusters, a more efficient alternative can be to forward the request only to a subset of servers. The behavior is nearly equivalent.

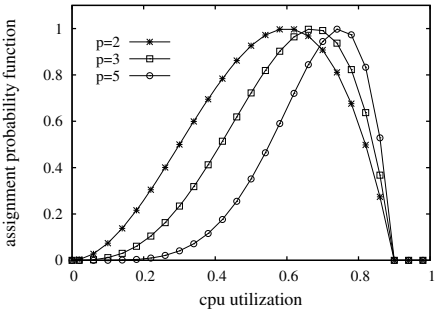


Fig. 2. Assignment probability function $f_{assign}(u)$ for different values of the parameter p. The value of the threshold T_a is set to 0.9.

One of the main advantages of the approach can now be appreciated, that is, its self-organizing and decentralized nature, since main decisions are taken locally. The manager is only required to know which servers are active and which are switched off, and to perform the random selection among the servers that are available to accommodate a new VM. The manager, though, is not requested to perform any algorithm to decide how to distribute the VMs to servers, nor to keep updated information about servers' state.

2.2 Migration Procedure

The assignment procedure allows VMs to be clustered in a reduced number of servers, as is shown in the performance evaluation section. Nevertheless, it can still happen that some servers are under-utilized and might be switched off. Indeed, even after an efficient allocation of VMs to computing resources, the VMs running in a server may terminate or may reduce their demand for CPU. Moreover, overload situations can also occur. In fact, the assignment of a VM to a server is made on the basis of its current CPU utilization, but the workload of other VMs in the same server may subsequently increase. This can cause SLA violations, thus affecting the degree of dependability of the data center and the quality of service offered to users. In both these situations, some VMs can be profitably migrated to other servers, either to switch off a server, or to alleviate its load.

Live migration of VMs is driven by the *migration procedure*. As opposed to other techniques recently proposed for VM migration (see the related work section), our approach is self-organizing and ensures a gradual and continuous migration process. At random time intervals, each server checks whether it is under-utilized or over-utilized and, when this occurs, evaluates the corresponding migration probability function, $f^l_{migrate}(u)$ or $f^h_{migrate}(u)$:

$$f^l_{migrate}(u) = (1 - u/T_l)^\alpha \tag{2}$$

$$f^h_{migrate}(u) = (1 + \frac{u-1}{1-T_h})^\beta \tag{3}$$

In either case, the server performs a Bernoulli trial and decides whether or not to request the migration of one of the local VMs. The functions, shown in Figure 3, are defined so as to trigger the migration of VMs when the CPU utilization is, respectively, below the threshold T_l or above the threshold T_h. When the utilization is in between, migrations are inhibited. The shape of the functions can be modulated by tuning the parameters α and β, which can therefore be used to foster or hinder migrations. A migration procedure completes when the VM is successfully assigned to another server, using the assignment procedure described in Section 2.1. In the case of the migration from an overloaded server, the threshold T_a of the assignment function is set to 0.9 times the CPU utilization of the current server. This ensures that the VM migrates to a less loaded server, and prevents situations in which a VM is continuously migrated from an overloaded server to another. The new value of T_a is sent to the other servers along with the migration request, and the VM is assigned to one of the available servers. If no server is available, the VM is kept by the original server.

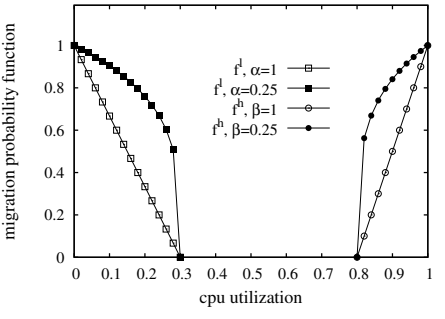


Fig. 3. Migration probability functions $f^l_{migrate}(u)$ and $f^h_{migrate}(u)$, labeled as f^l and f^h, for two values of α and β. The threshold T_l is set to 0.3, T_h is set to 0.8.

3 Performance Evaluation

The approach described in the previous section was tested with a Java simulator implemented at ICAR-CNR. The evaluated data center has N_s=100 servers, 33 of which have 4 cores, 34 have 6 cores and 33 have 8 cores. All cores have CPU frequency of 2 GHz. The Virtual Machines that host client applications have nominal CPU frequencies of 500 MHz, 1 GHz and 2 GHz. They are assigned to applications with the following probability distribution: 50% of applications are assigned to 500 MHz VMs, 25% to 1 GHz VMs, and 25% to 2 GHz VMs.

Each VM runs for a time interval generated with a Gamma distribution and average $1/\mu$ set to 100 minutes, where μ is the service rate of each server core. During its execution, the application hosted by the VM can require all the VM capacity or a fraction of it. This fraction can vary over time, as Cloud applications - in many cases Web servers - usually experience dynamic workload.

For each application, the average interval between workload changes is set to 20 minutes (with Gamma distribution), and after each interval the fraction of the VM capacity demanded by the application is extracted uniformly between 0 and 1. Requests for client applications are received by the data center manager at rate λ (see Figure 1), whose value ranges between 1.2 and 24 requests per minute. The average load of the data center, denoted as ρ, can be computed as $0.5\lambda/\mu_T$. Here, the arrival rate of requests is halved because applications ask on average for half the capacity of a VM, while $\mu_T = \mu \cdot N_s \cdot 6 \cdot 2$ is the overall service rate of the data center: indeed, the average number of cores per server is 6, and the capacity of each core (2 GHz) is twice the average frequency of a VM (1 GHz). The parameter ρ ranges between 0.05 (nearly idle data center) and 1 (data center loaded at its maximum CPU capacity), and will be used to analyze the system performance in different load conditions. Servers can be dynamically activated and powered off: an inactive server is switched on when it is asked to accommodate a new or a migrating VM; an active server is switched off (or hibernated) when all the running VMs terminate or when they are migrated to other servers.

To analyze the amount of consumed power, the model described by some recent studies (e.g., [8] and [2]) is adopted. Specifically, the power consumed by servers can be obtained with a simple relationship between CPU utilization and power consumption, assuming that an idle server consumes about 70% of the power consumed by a fully utilized server. The power consumption is expressed as $P(u) = P_{max} \cdot (0.7 + 0.3 \cdot u)$. In our tests, P_{max}, the power consumed at maximum utilization, is set to 250 W.

Figure 4 reports the average number of active servers, in steady condition, vs. the system load, when setting the threshold T_a of the assignment function (1) to 0.9, and with the following parameter setting for the migration functions (2) and (3): T_l=0.2, T_h=0.95, α=0.25 and β=0.25. The migration procedure is evaluated by each server every 10 minutes. The figure reports results obtained with different values of the parameter p of the assignment function. For comparison purposes, the results are shown together with three other curves. The first is the average number of servers activated when each VM is randomly assigned to one of the servers, regardless of their current utilization. The related curve is by far the highest, and has a typical negative exponential trend. The VM mapping problem can be formulated in terms of the *bin packing problem*, i.e., the NP-hard problem of allocating objects of heterogeneous sizes in as few bins as possible [12]. The second curve corresponds to the optimal solution of this problem, i.e., when the minimum number of servers is used to accommodate the VMs. The curve labeled as *BFD* corresponds to the performance achievable when the bin packing problem is solved with the Best Fit Decreasing algorithm, which has quadratic complexity and guarantees to use at most 11/9 MIN+1 servers, where MIN is the minimum number of servers [13].

Figure 4 shows that our approach performs better than the BFD, especially when the load is high, and that the number of active servers is very close to the optimal curve. Of course, reducing the number of active servers allows the data

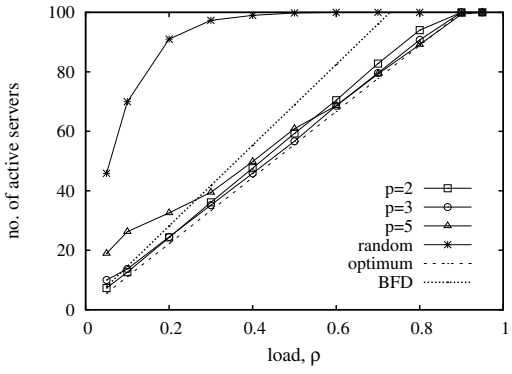


Fig. 4. Average number of active servers for different values of the parameter p of the assignment function. The meaning of the other curves is explained in the text.

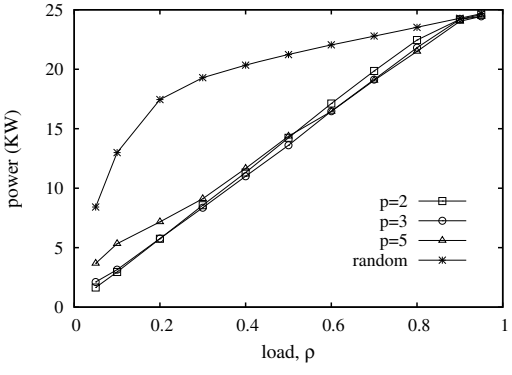


Fig. 5. Average power consumed by the data center for different values of the parameter p of the assignment function

center to save power, as appears in in Figure 5. The "green" behavior of our approach is testified by the fact that consumed power increases almost linearly with load. The two figures also show that the performance is not very sensitive to the value of p, which is a sign of robustness. Nevertheless, larger values of p can be used to improve consolidation (and reduce power consumption) in high load conditions, because they increase the probability of allocating a VM to a highly loaded server. Conversely, a low value of p is preferable when the load is low. The tuning of p can be done dynamically by the data center manager, by estimating the overall system load. In the next experiments, the parameter p is set to 3, as this value ensures a good behavior for all load conditions.

As explained before, VM migrations can be performed either because the utilization of a server is too low or too high. In the following, the migration events of the two kinds are referred, respectively, as *l_migrations* and *h_migrations*. Any migration causes a slight performance degradation of the application hosted by the VM for the time necessary to migrate, which in general is estimated in the

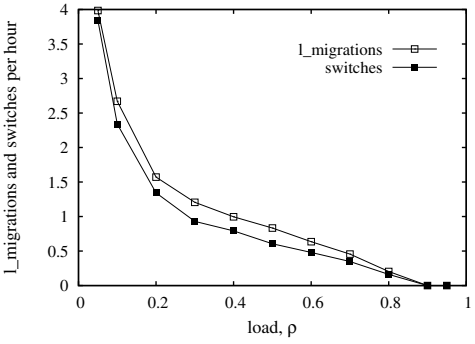


Fig. 6. Frequency of l_migrations and server switches vs. load.

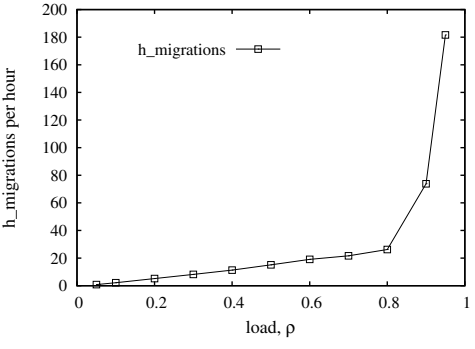


Fig. 7. Frequency of h_migrations vs. load.

order of 60 seconds [9]. Similarly, the activation of an off server needs a start up time and additional power. Therefore, it is important to limit the frequency of migrations and switches, though a certain number is essential for VM consolidation and power reduction. Figure 6 reports the frequencies of l_migrations and server activations experienced in the whole data center. Both frequencies are inversely proportional to the load. In fact, with high load, most servers are always active and highly loaded, so both events are impossible or rare. With low load, many servers are off, and the assignment procedure has more chances to assign a VM to an inactive server, which is then switched on. Since this server is initially under-utilized, it will likely attempt a migration procedure in the near future, which explains the higher migration frequency. Both frequencies are always lower than 4 events per hour in the whole data center, which is an easily sustainable burden.

Conversely, the frequency of h_migrations, reported in Figure 7, is directly proportional to the load. The trend is nearly linear, but becomes exponential when the load approaches the data center capacity: this suggests that new servers should be acquired when the load exceeds 0.8. Finally, Figure 8 reports the percentage of time in which the VMs allocated to a server demand more CPU

than what the server can provide, which may lead to SLA violations. This index, in accordance with recent studies [2], is used to measure the QoS level offered to users. Conditions for potential SLA violations are rare when ρ is lower than 0.8, then their frequency increases rapidly. The figure reports the index values obtained with and without the use of h_migrations, which clearly testifies the beneficial impact of these events.

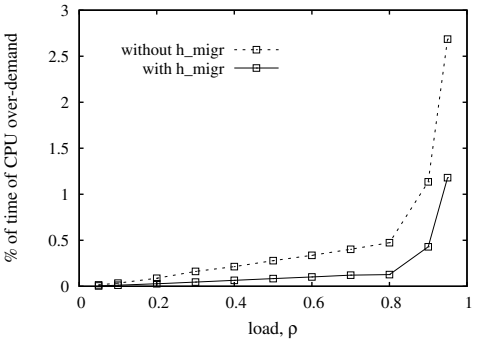


Fig. 8. Percentage of time in which the VMs require more CPU than that offered by a server, with and without the use of h_migrations

4 Related Work

As the Cloud computing paradigm rapidly emerges, a notable amount of studies focus on algorithms and procedures that aim at improving the "green" characteristics of Cloud data centers. A common aspect is the use of virtualization as a means to consolidate applications on as few servers as possible and in this way reduce power consumption. Some approaches - e.g., [4] and [10] - try to forecast the demand and aim at determining the minimum number of servers that should be switched on to satisfy the client requests, so as to reduce energy consumption and maximize data center revenue. However, even a correct setting of this number is only a part of the solution: algorithms are needed to decide how the VMs should be mapped to servers in a dynamic environment, and how live migration of VMs can be exploited to unload servers and switch them off when possible, or to avoid SLA violations.

As mentioned in Section 3, the problem of dynamically mapping VMs to servers is in some way similar to the *bin packing problem*, and the analogy is indeed exploited in recent research, for example in [12] and in [2]. Live migration of VMs between servers is adopted in [2] and by the VMWare Distributed Power Management system, using lower and upper utilization thresholds to enact migration procedures. All these approaches represent important steps ahead for the deployment of green-aware data centers, but still they share a couple of notable drawbacks: (i) the centralized manager is required to execute complex algorithms and solve a problem that is inherently NP-hard, and must always be aware of the state of all the servers, which becomes an issue in large and highly

dynamic data centers; (ii) mapping strategies may require the concurrent migration of many VMs, which can cause considerable performance degradations during the reassignment process. Conversely, the approach presented here is self-organizing, decentralized for the most part (assignment and migration decisions are taken autonomously by each server), and uses a gradual migration process.

Bio-inspired algorithms and protocols are emerging as a useful means to manage distributed systems. Assignment and migration procedures presented here are partly inspired by the *pick* and *drop* operations performed by some species of ants that cluster items in their environment [6]. The pick and drop paradigm, though very simple and easy to implement, has already proved to be surprisingly powerful: for example, it was used to cluster and order resources in P2P networks, in order to facilitate their discovery [7]. Another ant-inspired mechanism was proposed in [5]: in this study, the data center is modeled as a P2P network, and ant-like agents explore the network to collect information that can later be used to migrate VMs and reduce power consumption. Since the mapping of VMs to servers is essentially an optimization problem, evolutionary and genetic algorithms can also represent a valid solution. In [11], a genetic algorithm is used to optimize the assignment of VMs, and minimize the number of active servers. The main limitations of this kind of approach are the need for a strong centralized control and the difficulties in the setting of key parameters, such as the population size and the crossover and mutation rates.

5 Conclusion and Future Work

This paper presents an approach that aims at minimizing the number of active servers and reducing power consumption in Cloud data centers. The core of the proposal stands in the statistical and self-organizing procedures that are used to assign Virtual Machines to servers, and to migrate them when this helps either to power off under-utilized computers or to prevent possible SLA violations in highly loaded servers. Simulation experiments show that the adopted techniques succeed in the combined objective of reducing power consumption and ensuring a good level of the QoS experienced by users, but the novelty of the approach requires further research to better assess its performance and explore its potentialities. Some of the avenues are: (i) a deeper analysis of the sensitivity to parameter values; (ii) a study of scalability properties; preliminary evaluations are promising, as performance seems to improve with the data center size, which is not surprising given the statistical nature of the algorithms; (iii) adapt assignment and migration procedures to take into account not only the CPU utilization of servers, but also other aspects such as the necessity of assigning several VMs to the same server, when they need to cooperate with each other; (iv) the definition of mathematical models, which may help in giving a more formal foundation to the approach.

References

1. Barroso, L.A., Hölzle, U.: The case for energy-proportional computing. IEEE Computer 40(12), 33–37 (2007)
2. Beloglazov, A., Buyya, R.: Energy efficient allocation of virtual machines in cloud data centers. In: 10th IEEE/ACM Int. Symp. on Cluster Computing and the Grid, CCGrid 2010, pp. 577–578 (2010)
3. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Compututer Systems 25(6), 599–616 (2009)
4. Chen, Y., Das, A., Qin, W., Sivasubramaniam, A., Wang, Q., Gautam, N.: Managing server energy and operational costs in hosting centers. SIGMETRICS Perform. Eval. Rev. 33(1), 303–314 (2005)
5. Dubois, D.J., Mirandola, R., Barbagallo, D., Di Nitto, E.: A bio-inspired algorithm for energy optimization in a self-organizing data center. In: Self-Organizing Architectures. Springer, Heidelberg (2010)
6. Deneubourg, J.L., Goss, S., Franks, N., Sendova-Franks, A., Detrain, C., Chrétien, L.: The dynamics of collective sorting: robot-like ants and ant-like robots. In: First International Conference on Simulation of Adaptive Behavior on From Animals to Animats, pp. 356–363. MIT Press, Cambridge (1990)
7. Forestiero, A., Mastroianni, C., Spezzano, G.: So-grid: A self-organizing grid featuring bio-inspired algorithms. ACM Transactions on Autonomous and Adaptive Systems 3(2) (May 2008)
8. Greenberg, A., Hamilton, J., Maltz, D.A., Patel, P.: The cost of a cloud: research problems in data center networks. SIGCOMM Comput. Commun. Rev. 39(1), 68–73 (2009)
9. Hirofuchi, T., Ogawa, H., Nakada, H., Itoh, S., Sekiguchi, S.: A live storage migration mechanism over wan for relocatable virtual machine services on clouds. In: 9th IEEE/ACM Int. Symp. on Cluster Computing and the Grid, CCGrid 2009 (2009)
10. Mazzucco, M., Dyachuk, D., Deters, R.: Maximizing cloud providers' revenues via energy aware allocation policies. In: 10th IEEE/ACM Int. Symp. on Cluster Computing and the Grid, CCGrid 2010, pp. 131–138 (2010)
11. Mi, H., Wang, H., Yin, G., Zhou, Y., Shi, D., Yuan, L.: Online self-reconfiguration with performance guarantee for energy-efficient large-scale cloud computing data centers. In: 2010 IEEE Int. Conference on Services Computing, SCC 2010, Miami, Fl, USA, pp. 514–521 (July 2010)
12. Verma, A., Ahuja, P., Neogi, A.: pMapper: Power and migration cost aware application placement in virtualized systems. In: Issarny, V., Schantz, R. (eds.) Middleware 2008. LNCS, vol. 5346, pp. 243–264. Springer, Heidelberg (2008)
13. Yue, M.: A simple proof of the inequality FFD (L) $\leq$ 11/9 OPT (L) + 1, for all L for the FFD bin-packing algorithm. Acta Mathematicae Applicatae Sinica 7(4), 321–331 (1991)

An Adaptive Load Balancing Algorithm with Use of Cellular Automata for Computational Grid Systems

Laleh Rostami Hosoori and Amir Masoud Rahmani

Department of Computer Engineering
Islamic Azad University, Science and Research Branch
Tehran, Iran
l.rostami@srbiau.ac.ir, rahmani@srbiau.ac.ir

Abstract. Load balancing algorithms play a challenging, complicated, and important role in the performance of computational Grid systems. In this paper, we present a decentralized adaptive load balancing algorithm with use of cellular automata, named LBA_CA. Each computing node in the Grid system is modeled as a cell of proposed cellular automata and can be in four states. Cellular automata (abbreviated to CA) are used for designing a load balancing algorithm for computational Grids because of its distributed and dynamic manner. In addition, such natural properties of CA make LBA_CA an appropriate local load balancing algorithm for each cluster of computational Grids. Due to resource heterogeneity and communication overheads exist in computational Grid systems; we take account of several issues in LBA_CA such as processing power of computing nodes and communication latency. The main goal of our algorithm is to reduce the average response time of arrival jobs. The performance of our algorithm is evaluated in terms of several metrics including the average response time of jobs, processor utilization, percent of executed jobs, and average Off time in relation to considerable variations in transition time, service time, and number of jobs.

Keywords: Load balancing; Cellular Automata; Computational Grid systems; Distributed systems.

1 Introduction

The computational Grid is a promising platform that provides large resources for distributed algorithmic processing [1]. Computational Grid environments promise to support resource sharing and coordinated problem solving in dynamic multi-institutional Virtual Organizations [2]. End users and applications see this environment as a big virtual computing system. The systems connected together by a grid might be distributed globally, running on multiple hardware platforms, under different operating systems and owned by different organizations. However, computational Grids have different constraints and requirements than those of traditional high-performance computing systems, such as heterogeneous computing resources and considerable communication delays [3]. In distributed systems, every node has different processing speed and system resources, so in order to enhance the

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 419–430, 2011.

utilization of each node and minimize the average response time of jobs, load balancing will play a critical role [4].

In general, load balancing algorithms can be classified as centralized or decentralized. In the centralized algorithms (e.g., [5]), one computing node makes all the load-balancing decisions. In the decentralized algorithms (e.g., [6]), all computing nodes are involved in the load-balancing decisions.

Moreover, Load balancing algorithms are classified as static, dynamic, adaptive, or hybrid, based on the used information. Static load balancing algorithms (e.g., [7]) assume that all required information about computing nodes, jobs, and communication network that influence load-balancing decisions are determined in advance, on the other hand, dynamic load balancing algorithms (e.g., [8] and [9]) attempt to gather the current state information to make more informative load-balancing decisions. Adaptive algorithms are a special type of dynamic algorithms where the parameters of the algorithm and/or the scheduling policy themselves are changed based on the global state of the system [10]. Finally hybrid algorithms (e.g.,[11]) attempt to combine the merits of static and dynamic load balancing algorithms in order to minimize their weak points.

A cellular automaton is a discrete dynamical system that consists of a regular network of definite state automata (cells) that change their states depending on the states of their neighbors, according to a local update rule. All cells change their state simultaneously, using the same update rule. The process is repeated at discrete time steps. It turns out that amazingly simple update rules may produce extremely complex dynamics when applied in this fashion. A well known example is the *Game-of-life* by John Conway [12]. For this reason, CA is often used to model real-world phenomena. CA is also considered as a model of highly parallel and distributed computations in multiprocessor and distributed systems [13]. They were used to find solutions of such problems as scheduling and resource management [14].

This paper presents a load balancing algorithm with using CA. The remainder of the paper is organized as follows: In section 2, an introduction to cellular automata is addresses. Section 3 describes our proposed load balancing algorithm in detail. Section 4 discusses our simulation and results of evaluation. Finally, section 8 concludes this paper.

2 Cellular Automata

Cellular automata are a class of discrete dynamical systems, consisting of an array of nodes (cells) of n-dimension. Each cell can be in one of k different states at a given time t [15]. At each discrete time, each cell may change its state, in a way determined by the local transition rules of the particular CA. The transition rules describe precisely how a given cell should change states, depending on its current state and the states of its neighbors. The neighborhood of a given cell must be specified explicitly.

More precisely, CA consists of 4 major components. At first all cells in the CA constitute the *cellular space*, which may be of any dimension, and is of infinite extent. For example, one-dimensional CA can be visualized as having a cell at each integral point on the real number line [15]. A two-dimensional CA has cells at all points in the plane that has only integral coordinates. Minutely *state set* is a finite set whose

elements are all the possible distinct states of the cells. The state of cell i at discrete time t is denoted by $S_i(t)$. Then *neighbourhood* is defined as the neighbours of each cell. $V_i(t)$ denotes the neighborhood of cell i at time t. Finally, each cell transforms from its current state to a new state (at the next time) based on its current state and the states of its neighbors, according to the transition rules. f denotes the *transition rule* of cell i in (1):

$$S_i(t + 1) = f(S_i(t), V_i(t)) . \quad (1)$$

3 The Proposed Load Balancing Algorithm

We assume that the computational Grid environment consists of three major components including processors, communication network, and jobs. Our Grid system consists of M heterogeneous processors, P_1; P_2;...; P_M. It is assumed each processor P_i has several different attributes. Firstly *processing power (W_i)* is the ratio of the processing power of the processor P_i to the processing power of the reference processor P_{ref} (with a relative W_{ref} equal to 1) in the system. In this paper, we elect the slowest processor as P_{ref}. Minute attribute is *job queue*. It is assumed that each processor has an infinite capacity job queue to store jobs waiting for execution. Finally *neighbours (N_i)* defined as a set of processors that are directly connected to the processor P_i.

The jobs are assumed to be computationally intensive, mutually independent, and can be executed at any processor. Several different attributes are assumed for each job J_j. First attribute is *arrival time (AT_j)* of job J_j. It is assumed that the arrival rate of jobs follows Poisson distribution with mean λ. In other words, the inter-arrival time is exponentially distributed with mean$1/\lambda$. It is assumed that *service time (ST_j)* of job J_j follows an exponential distribution with mean $1/\mu$. We assume that job J_j will be missed if it isn't executed until its *deadline time (DT_j)*. In addition, deadline times are distributed exponentially with a given mean.

One of the important parts of Grid computing system is communication network that connects processors to each other. The network topology is varying, since computational Grid systems are dynamic in nature. Thus, our model assumes no specific topology for the network and generates a random topology. Due to diverse network topologies in computational Grid systems, network heterogeneity also exists.

All items that each processor takes account of during the proposed load balancing algorithm are described below:

- *Transition Time (T_n):* The period of time, at the end of that each cell i of CA transforms from its current state $S_i(T_{n-1})$ to a new state (at the next time) $S_i(T_n)$, in a way determined by transition rules f.
- *$Q_i(t)$:* The number of jobs waiting in the processor P_i's job queue at time t
- *$\lambda_i(T_n)$:* The number of arrived jobs at the processor P_i during the interval $[T_{n-1}, T_n]$
- *$\mu_i(T_n)$:* The number of served jobs at the processor P_i during the interval $[T_{n-1}, T_n]$
- *$l_i(T_n)$:* $Q_i(T_n)$ divided by $\mu_i(T_n)$
- *W_i:* The processing power of P_i in relation to the reference processor P_{ref}
- *$l_N(T_n)$:* The average of normalized load in the neighbours N_i of the processor P_i during the interval $[T_{n-1}, T_n]$ as presented in (2):

$$l_N(T_n) = \frac{\sum_{i \in \text{Neighbourhood}} W_i * l_i(T_n)}{\sum_{i \in \text{Neighbourhood}} W_i} \tag{2}$$

- *$S_i(T_n)$:* The cell state of processor P_i during the interval $[T_{n-1}, T_n]$, denoted as below in (3):

$$S_i(T_n)=(S_{ix}(T_n), S_{iy}(T_n)) \ . \tag{3}$$

- *$Sender_p(T_n)$:* The percent of the processor P_i's neighbours N_i whose $S(T_n)$ are Sender, during the interval $[T_{n-1}, T_n]$ as presented in (4):

$$\text{Sender}_p(T_n) = \frac{\sum_{i \in \text{Neighourhood}} S_{ix}(T_n) * S_{iy}(T_n)}{\sum_{i \in \text{Neighourhood}} i} \ . \tag{4}$$

- *$Receiver_p(T_n)$:* The percent of the processor P_i's neighbours N_i whose $S(T_n)$ are Receiver, during the interval $[T_{n-1}, T_n]$ as presented in (5):

$$\text{Receiver}_p(T_n) = \frac{\sum_{i \in \text{Neighourhood}} \overline{S_{ix}}(T_n) * S_{iy}(T_n)}{\sum_{i \in \text{Neighourhood}} i} \ . \tag{5}$$

- *Threshold:* The load threshold (The default value is 1)
- *CL_j^k:* The communication latency between processor P_i and the processor P_k when the job J_j transferred.
- *$\tilde{l}_k(t)$:* The estimated added load on the processor P_k at time t, as presented in (6):

$$\tilde{l}_k(t) = \frac{\lambda_k(T_{n-1})}{\mu_k(T_{n-1})} * (t - T_{n-1}) \ . \tag{6}$$

- *ST_j^i:* The duration in which the job J_j will expectedly be served on the processor P_i, as denoted in (7):

$$ST_j^i = \frac{ST_j}{W_i} \ . \tag{7}$$

- *$ET_j^i(t)$:* When the job J_j will expectedly leave the system, if it is served on the processor P_i , as described in (8):

$$ET_j^i(t) = \frac{Q_i(t)}{\mu_i(T_{n-1})} + ST_j^i \ . \tag{8}$$

- *$ET_j^k(t)$:* When the job J_j will expectedly leave the system, if it is sent form the processor P_i to P_k to be served, as formulated in (9):

$$ET_j^k(t) = \max(l_k(T_{n-1}) + \tilde{l}_k(t), CL_j^k) + ST_j^k \ . \tag{9}$$

Our goal is to minimize the *Average Response Time (ART)* as calculated in (10):

$$\mathrm{ART} = \frac{1}{\mathrm{N}} \sum_{\mathrm{j=1}}^{\mathrm{N}} \mathrm{ExecutionTime}_{\mathrm{j}} - \mathrm{ArrivalTime}_{\mathrm{j}} \ . \tag{10}$$

Where N is the number of jobs executed by the system, $ExecutionTime_j$ is the time when the job J_j is completely executed, and $ArrivalTime_j$ is the time when the job J_j arrives. As obviously clear, ART takes account of the communication latency, waiting time in queues, and service time.

Our algorithm has proposed a CA with following features. All processors of an arbitrary network are cells of *cellular space*, in other words each processor is considered as a cell. *State set* of CA includes four states. The state of cell i at time t represented by the tuple $S_i(t)$ as in (11), where each element of the tuple can be a binary number (0 or 1):

$$S_i(t) = (S_{ix}(t), S_{iy}(t)) \ . \tag{11}$$

We assumed each cell can be in four states. At first, whenever a processor becomes overloaded in comparison with its neighbours, it is assumed as a *sender* processor. It means the processor will send all incoming jobs to its neighbours, which are in receiver state. This state is represented by tuple (1,1). Minutely whenever a processor becomes balanced in comparison with its neighbours, it is denoted as a *balanced* processor. It means the processor will run jobs, which exist in its queue and will accept new jobs as well. Meanwhile, the processor will refuse jobs sent form sender processors. This state is represented by tuple (1,0). Thirdly whenever a processor becomes under-loaded in comparison with its neighbours, it is assumed as a *receiver* processor. It means the processor will run jobs, which exist in its queue and will accept all new jobs and jobs sent form sender processors. This state is represented by tuple (0,1). Whenever a processor is idle for a while, it will be denoted as an *off* processor. It means the processor will go off. This state is represented by tuple (0,0).

The *neighbourhood* $V_i(t)$ of processor P_i at time t is the set of its neighbours N_i in the communication network. Finally the *transition rule* is the same for all processors as denoted by f in (12) and described in depth in Table 1.

$$S_i(T_n) = f(S_i(T_{n-1}), V_i(T_{n-1})) \ . \tag{12}$$

Table 1. Transition rule

$S_i(T_n)$	$f(S_i(T_{n-1}), V_i(T_{n-1}))$
Sender (1,1)	If $(l_i(T_n) - l_N(T_n)) >$ Threshold
Balanced (1,0)	If $\|l_i(T_n) - l_N(T_n)\| \le$ Threshold
Receiver (0,1)	If $(l_i(T_n) - l_N(T_n)) <$ Threshold or If $S_i(T_{n-1})$ == Off and $Sender_p(T_n) \ge Receiver_p(T_n)$
Off (0,0)	If $(l_i(T_n), \lambda_i(T_n), \mu_i(T_n)) == (0, 0, 0)$ If $S_i(T_{n-1})$ == Off: $S_i(T_n)$ = Off Else: $S_i(T_n)$ will be Off with probability 0.01

Our proposed model focuses on the dynamic and distributed properties of cellular automata, in order to offer an efficient load balancing algorithm. LBA_CA gathers

up-to-date information in each transition time. Nowadays, energy conservation becomes a hot discussion all over the world. So LBA_CA put some computing nodes in the *Off* state under special circumstances. This feature might be a step toward approaching energy conservation.

LBA_CA consists of two procedures. *Transition Procedure* indicates processes that each processor performs in each transition time T_n in order to run transition rule and update its state. *Main Procedure* is invoked whenever a new job J_j arrives to processor P_i at time t. If the processor P_i is in *Sender* or *Off* state, this procedure firstly will generate a set of processors which are suitable to execute the job J_j maybe including also the processor P_i. Afterwards it will assign different probability to each processor of the set. Then the job J_j will be sent to a processor with the highest probability. Otherwise, the job J_j will be executed by the processor P_i or kept in its queue.

```
Transition Procedure:
Each processor P_i runs the following statements
simultaneously at the Transition time (T_n):
 StateInfo = (λ_i(T_n), μ_i(T_n), l_i(T_n))
 Transfer StateInfo to neighbours N_i
 TransitionInfo = (l_N(T_n), Sender_p(T_n), Receiver_p(T_n))
 Call Transition rule

Main Procedure:
If S_i(T_n) == Sender OR S_i(T_n) == Off
  Calculate ET_j^i(t)
  If Receiver_p(t) ≤ 0.1
    pb = Generate a probability in range [0.9, 1]
  Else
    pb = Generate a probability in range [0, 1]
  Add (P_i, pb) to the set P_ET
  For each P_k in N_i
    If S_k(T_n) == Receiver
      Calculate ET_j^k(t)
      If ET_j^k(t) < ET_j^i(t)
        If Receiver_p(t) ≥ 0.9
          pb = Generate a probability in range [0.9, 1]
        Else
          pb = Generate a probability in range [0, 1]
        Add (P_k, pb) to the set P_ET
  Choose a processor as P_dest in P_ET with highest
  probability
  Send the job J_j to the processor P_dest
Else
  If P_i is Idle
    Execute J_j
  Else
    Put J_j into the job queue of P_i
```

4 Simulation and Results

In the simulation, our algorithm is compared with ELISA (Estimated Load Information Scheduling Algorithm). In ELISA [16], each processor exchanges the actual queue length, the actual service rate, and the estimated arrival rate at state exchange times. The queue lengths of its neighbours are estimated at estimation times using information received at state exchange times. The main reason that ELISA has been chosen for the comparison is it considers regular time intervals in order to make load-balancing decisions. Therefore, we can compare two algorithms more precisely by varying intervals. In this paper, four performance metrics are considered, addressed in detail as follow:

1. *Average Response Time (ART)*
2. *Processor Utilization (U)*: The utilization U_i of the processor P_i calculates as mentioned in (13):

$$U_i = \frac{Busy_i}{Busy_i + Idle_i} . \tag{13}$$

Where $Busy_i$ is the amount of time that the processor P_i remains busy and $Idle_i$ is the amount of time that the processor P_i remains idle.

3. *Percent of Executed Jobs (PEJ)*: Percent of jobs that their deadlines do not miss.
4. *Average Off Time (AOT)*: Let us begin by $offTime_i$ that is the amount of time that the processor P_i remains off. Next AOT is defined as presented in (14). It should be mentioned that this metric is unique to LBA_CA.

$$AOT = \frac{1}{M}\sum_{i=1}^{M} offTime_i . \tag{14}$$

4.1 Simulation Model

It is assumed that our simulated Grid system includes 60 heterogeneous computing nodes. Processing powers of nodes are assumed to follow a uniform distribution in range [1, 20]. Communication latency between each processing nodes is chosen from a lognormal distribution with a mean of 0.2 time units and a standard deviation 0.5. It is assumed that time unites are minutes.

In most cases, the number of executed jobs is 10000 unless otherwise stated explicitly. 300 extra jobs are executed at first that are considered as “warm-up jobs”. After warm-up jobs, performance metrics are traced. Jobs arrive at the Grid system according to a Poisson process with rate 1. Service times and deadline times of jobs are assumed to follow an exponential distribution with a mean of 50 and 300 minutes as default values, respectively. Other parameters in LBA_CA are the transition time (T_n) and threshold assumed 10 minutes and 1, as the default values, respectively.

4.1.1 Effect of Number of Jobs

In order to measure the effect of this factor, we increase the number of jobs from 10000 to 50000. At first, as can be seen from Fig. 1, the average response time of

LBA_CA is significantly lower than that of ELISA although it slightly increases due to sharp rises in number of jobs. By using LBA_CA, it clearly shows that jobs leave the system sooner than when ELISA runs due to better load-balancing decisions. The average improvement factor of LBA_CA in terms of the average response time under the effect of number of jobs is 28.7% over ELISA.

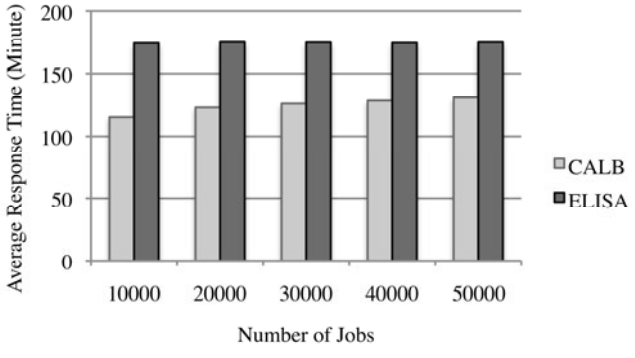


Fig. 1. Effect of number of jobs on average response time of the both algorithms

Afterwards, the amount of difference between the minimum and maximum of processor utilization in the Grid system partly states the quality of load-balancing services provided. As can be easily seen from Fig. 2, the minimum and maximum utilization of processors of LBA_CA are approximately 47.1% closer to the average utilization of the Grid system than those of ELISA. Therefore, this performance metric approves improvements in load-balancing decisions of LBA_CA as well as previous one.

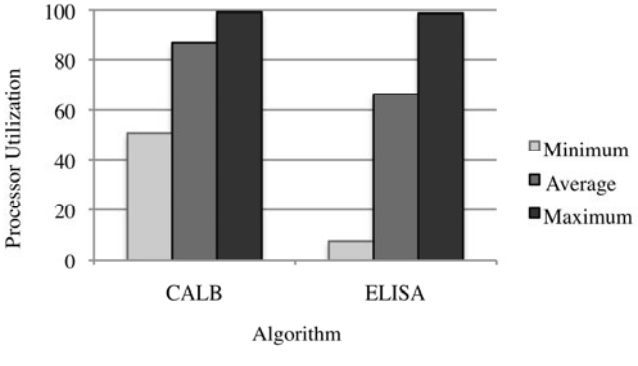


Fig. 2. Comparing processor utilization of the both algorithms

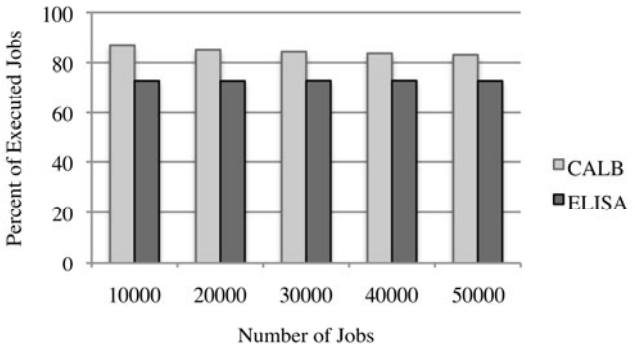


Fig. 3. Effect of number of jobs on percent of executed jobs of the both algorithms

After that Fig. 3 illustrates that the percent of executed jobs for LBA_CA gradually falls nearly from 87% to 83%. Meanwhile, the percent of executed jobs for ELISA remains fairly constant at about 73%. However, LBA_CA shows the average improvement factor 15.8% over ELISA, in terms of the percent of executed. The reason of this improvement is LBA_CA reduces the waiting time of jobs in comparison to ELISA through properly balancing the load across the processors.

Lastly, Fig. 4 wonderfully shows that the average off time has a linear growth with a slope about 0.0002. This metric indicates that some computing nodes usually exist in the Grid system do not take part in load-balancing process due to their inappropriate network situation such as link bandwidths. Therefore, it might be a good idea to turn off such computing nodes in order to conserving energy.

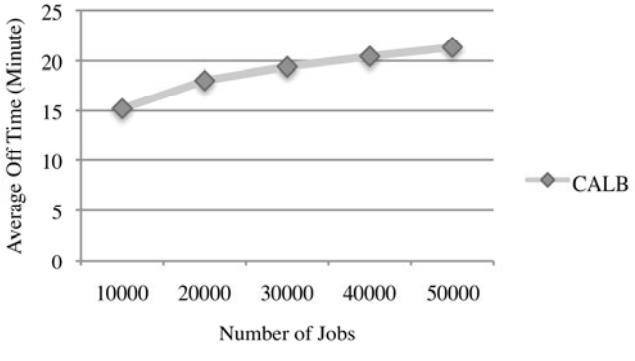


Fig. 4. Effect of number of jobs on average off time of LBA_CA

4.1.2 Effect of Service Time

In this part, it is assumed that the mean of service time varies from 12 to 150 minutes. In order to keep the percent of executed jobs acceptable, the mean of deadline times

for all jobs is fairly raised as the corresponding service time increases. Fig. 5 illustrates that changes in the service time cause dramatic increases in the average response time of the both algorithms. When the mean of service time is under or equal to 50 minutes, LBA_CA shows better results than ELISA. On the other hand, when it is above 50 minutes, ELISA indicates more acceptable results than LBA_CA. However, the average improvement factor of LBA_CA in terms of the average response time under the effect of service time is 13.8% over ELISA.

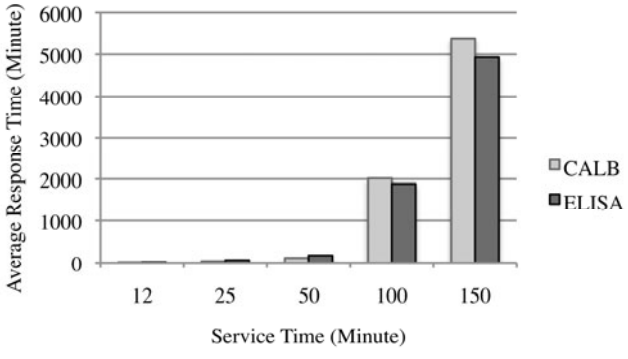


Fig. 5. Effect of service time on average response time of the both algorithms

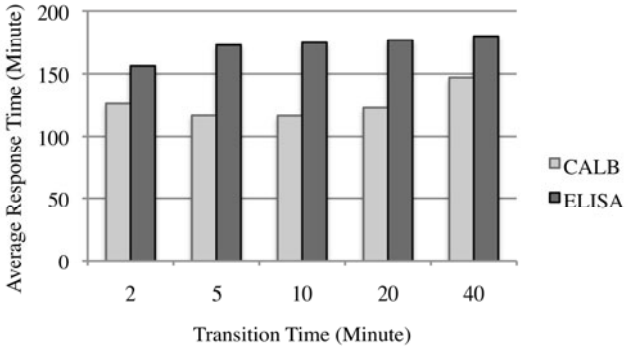


Fig. 6. Effect of transition time on average response time of the both algorithms

4.1.3 Effect of Transition Time

In order to find the most proper transition time (T_n) in terms of performance metrics for running transition rule, we experimented on LBA_CA with different transition times in the wide range of 2 to 40 minutes. It should be mentioned that the state exchange time and estimation time of ELISA are assumed equal to transition time and five times the length of it, respectively. Firstly, in Fig. 6, for transition times under or equal to 10 minutes, LBA_CA certainly shows a slight downward trend of this performance metric. But whenever the transition time rises more than 10 minutes, the average response time of LBA_CA gradually increases. On the other hand, this metric

of ELISA shows an increasing growth. However, LBA_CA indicates the better performance than ELISA due to the average improvement factor of 26.8%. Since LBA_CA reaches its minimum at the transition time of 10 minutes, it is concluded that the most proper time for the transition time is about 10 minutes.

5 Conclusion

Natural dynamic and distributed properties of CA convince us to use it in our proposed local load balancing algorithm and make LBA_CA partly practical for each cluster of computational Grid systems. As mentioned before, each computing node assumed as a cell of CA. Each cell of our proposed CA can be in four state including *Sender*, *Balanced*, *Receiver*, and *Off*. We address resource heterogeneity and communication overheads in LBA_CA through taking several parameters into account such as processing power of computing nodes and communication latency. Moreover, since energy conservation gets a high priority in each aspect of our lives; LBA_CA puts that node into Off state under special circumstances.

LBA_CA attempts to improve the average response time of jobs, although LBA_CA is evaluated in terms of some other performance metrics including processor utilization, the percent of executed jobs, and the average off time. In order to experiment LBA_CA, several items such as number of jobs, service time and transition time are varied in wide ranges of values and it is compared with ELISA. As addressed in depth, LBA_CA plainly performs far better than ELISA in all performance metrics measured by the average improvement factor between 10% and 47%.

Consequently, since LBA_CA gathers up-to-date state information in each transition time, communication overheads may partly increase, although LBA_CA takes account of communication latency for load-balancing decisions. Therefore, it might be a good idea to use estimation methods in some intervals of time instead of providing accurate information. However, it seems that CA is a proper instrument for designing load balancing algorithms and it can be applied to design more efficient load balancing algorithms in future.

References

1. Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San Francisco (1999)
2. Foster, I.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. In: Sakellariou, R., Keane, J.A., Gurd, J.R., Freeman, L. (eds.) Euro-Par 2001. LNCS, vol. 2150, pp. 1–4. Springer, Heidelberg (2001)
3. Subrata, R., Zomaya, A.Y., Landfeldt, B.: Game-Theoretic Approach for Load Balancing in Computational Grids. IEEE Transactions on Parallel and Distributed Systems 19(1), 66–76 (2008)
4. Lu, K., Zomaya, A.Y.: A Hybrid Policy for Job Scheduling and Load Balancing in Heterogeneous Computational Grids. In: 6th International Symposium on Parallel and Distributed Computing, p. 19. IEEE Computer Society, Washington, D.C (2007)
5. Shivaratri, N., Krueger, P., Singhal, M.: Load Distributing for Locally Distributed Systems. Computer 25(12), 33–44 (1992)

6. Lu, K., Subrata, R., Zomaya, A.Y.: Towards Decentralized Load Balancing in a Computational Grid Environment. In: Chung, Y.-C., Moreira, J.E. (eds.) GPC 2006. LNCS, vol. 3947, pp. 466–477. Springer, Heidelberg (2006)
7. Penmatsa, S., Chronopoulos, A.T.: Game-theoretic static load balancing for distributed systems. Journal of Parallel and Distributed Computing (2010)
8. Nasir, H.J.A., Mahamud, K.R.K., Din, A.M.: Load Balancing Using Enhanced Ant Algorithm in Grid Computing. In: 2nd International Conference on Computational Intelligence, Modelling and Simulation, pp. 160–165. IEEE Computer Society Press, Washington, D.C (2010)
9. Zheng, Q., Tham, C.K., Veeravalli, B.: Dynamic Load Balancing and Pricing in Grid Computing with Communication Delay. Journal of Grid computing 6(3), 239–253 (2008)
10. Shah, R., Veeravalli, B., Misra, M.: On the Design of Adaptive and Decentralized Load-Balancing Algorithms with Load Estimation for Computational Grid Environments. IEEE Transactions on Parallel and Distributed Systems 18(12), 1675–1686 (2007)
11. Yan, K.Q., Wang, S.S., Wang, S.C., Chang, C.P.: Towards a hybrid load balancing policy in grid computing system. Journal Expert Systems with Applications 36(10), 12054–12064 (2009)
12. Berlekamp, E.R., Conway, J.H., Guy, R.K.: Winning Ways for Your Mathematical Plays, vol. 2. Academic Press, New York (1982)
13. Gramb, T., Bornholdt, S., Grob, M., Mitchell, M., Pellizzari, T.: Computation in Cellular Automata: A Selected Review. In: Mitchell, M. (ed.) Non-Standard Computation: Molecular Computation - Cellular Automata - Evolutionary Algorithms - Quantum Computers, pp. 95–140. Wiley-VCH Verlag GmbH & Co, Weinheim (1998)
14. Swiecicka, A., Seredynski, F., Zomaya, A.Y.: Multiprocessor Scheduling and Rescheduling with Use of Cellular Automata and Artificial Immune System Support. IEEE Transactions on Parallel and Distributed Systems 17(3), 253–262 (2006)
15. Kari, J.: Theory of cellular automata: A survey. Theoretical Computer Science 334(1-3), 3–33 (2005)
16. Anand, L., Ghose, D., Mani, V.: ELISA: An Estimated Load Information Scheduling Algorithm for Distributed Computing Systems. Computers & Mathematics with Applications 37(8), 57–85 (1999)

Shrinker: Improving Live Migration of Virtual Clusters over WANs with Distributed Data Deduplication and Content-Based Addressing

Pierre Riteau[1,2], Christine Morin[2], and Thierry Priol[2]

[1] Université de Rennes 1, IRISA, Rennes, France
[2] INRIA, Centre INRIA Rennes - Bretagne Atlantique, Rennes, France
Pierre.Riteau@irisa.fr, {Christine.Morin,Thierry.Priol}@inria.fr

Abstract. Live virtual machine migration is a powerful feature of virtualization technologies. It enables efficient load balancing, reduces energy consumption through dynamic consolidation, and makes infrastructure maintenance transparent to users. While live migration is available across wide area networks with state of the art systems, it remains expensive to use because of the large amounts of data to transfer, especially when migrating virtual clusters rather than single virtual machine instances. As evidenced by previous research, virtual machines running identical or similar operating systems have significant portions of their memory and storage containing identical data. We propose Shrinker, a live virtual machine migration system leveraging this common data to improve live virtual cluster migration between data centers interconnected by wide area networks. Shrinker detects memory pages and disk blocks duplicated in a virtual cluster to avoid sending multiple times the same content over WAN links. Virtual machine data is retrieved in the destination site with distributed content-based addressing. We implemented a prototype of Shrinker in the KVM hypervisor and present a performance evaluation in a distributed environment. Experiments show that it reduces both total data transferred and total migration time.

Keywords: Virtualization, Live Migration, Wide Area Networks, Cloud Computing.

1 Introduction

The complete encapsulation of execution environments (applications combined together with their underlying OS) in virtual machines (VMs) allowed the development of live virtual machine migration [4,20]. This mechanism relocates a virtual machine from one host to another with minimal downtime, usually not noticeable by users. This offers numerous advantages for data center management, including:

Load balancing: VMs can be dynamically migrated depending on workload, offering more efficient usage of computing resources.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 431–442, 2011.

Reduced energy consumption: VMs with low workloads can be consolidated to fewer physical machines, making it possible to power off nodes to reduce energy consumption.

Transparent infrastructure maintenance: Before physical machines are shut down for maintenance, administrators can relocate VMs to other nodes without noticeable interruption of service for users.

Mainstream hypervisors usually do not support live migration between different data centers connected with wide area networks (WANs). To avoid transferring persistent state of VMs, which can be of large size (from hundreds of megabytes to dozens of gigabytes and beyond), they depend on shared storage, usually not accessible across different data centers. They also require that migrated VMs stay in the same local network, to keep network connections uninterrupted after migration to new host machines.

These restrictions prevents users from getting the benefits of live virtual machine migration over WANs. It would allow administrators to load balance workload between several data centers, and offload VMs to other sites whenever a site-wide maintenance is required. Users with access to private clouds (private computing infrastructures managed with cloud computing stacks) could seamlessly migrate VMs between private and public clouds depending on resource availability. It could also allow them to leverage variable price between competing cloud providers.

Fortunately, state of the art systems allow to use live migration over WANs by migrating storage [3,16,10,32] and network connections [3,7,32]. However, the large amounts of data to migrate make live migration over WANs expensive to use, especially when considering migrations of virtual clusters rather than single VM instances.

Previous research showed that VMs running identical or similar operating systems have significant portions of their memory and storage containing identical data [30,5]. In this paper, we propose Shrinker, a live virtual machine migration system leveraging this common data to improve live virtual cluster migration between data centers interconnected by wide area networks. Shrinker detects memory pages and disk blocks duplicated in a virtual cluster to avoid sending multiple times the same content over WAN links. Virtual machine data is retrieved in the destination site with distributed content-based addressing. We implemented a prototype of Shrinker in the KVM hypervisor [12] and present a performance evaluation in a distributed environment. Our experiments show that it reduces both total data transferred and total migration time.

This paper is organized as follows. Section 2 presents related work. Section 3 covers the architecture of Shrinker. Section 4 describes our prototype implementation in the KVM hypervisor, presents our experiments and analyzes the results. Finally, Section 5 concludes and discusses future work.

2 Background and Related Work

Live virtual machine migration [4,20] has traditionally been implemented with pre-copy [27] algorithms. Memory pages are transferred to the destination host

while the virtual machine is still executing. Live migration continues sending modified memory pages until it enters a phase where the virtual machine is paused, the remaining pages are copied, and the virtual machine is resumed on the destination host. This phase is responsible for the downtime experienced during live migration, which should be kept minimal. This downtime can range from a few milliseconds to seconds or minutes, depending on page dirtying rate and network bandwidth [1].

Numerous works have been proposed to improve live migration and optimize its performance metrics: total data transferred, total migration time, and downtime. They can be classified in two types: optimizations of the pre-copy approach, and alternatives to pre-copy. Optimizations include compression, delta page transfer, and data deduplication. Compression is an obvious method to reduce the amount of data transferred during live migration. Jin et al. [11] use adaptive compression to reduce the size of migrated data. Their system chooses different compression algorithms depending on memory page characteristics. Delta page transfer [6,32,26] optimizes the transmission of dirtied pages by sending difference between old pages and new pages, instead of sending full copies of new pages. Data deduplication [34,32] detects identical data inside the memory and disk of a single virtual machine and transfers this data only once.

An alternative to pre-copy is live migration based on post-copy [8,9], which first transfers CPU state and resumes the VM on the destination host. Memory pages are then fetched from the source host on demand, in addition to a background copying process to decrease the total migration time and quickly remove the residual dependency on the source host. Similarly, SnowFlock [13] uses demand-paging and multicast distribution of data to quickly instantiate VM clones on multiple hosts. Another alternative was proposed by Liu et al. [15], based on checkpointing/recovery and trace/replay. By recording nondeterministic events and replaying them at the destination, live migration efficiency is greatly improved. However, their approach is not adapted to migrate SMP guests.

All these systems were designed in the context of live migration of single VMs, and do not take advantage of data duplicated across multiple VMs. However, previous research [30,5,33,17] has shown that multiple VMs have significant portions of their memory containing identical data. This identical data can be caused by having the same versions of programs, shared libraries and kernels used in multiple VMs, or common file system data loaded in buffer cache. This identical data can be leveraged to decrease memory consumption of colocated VMs by sharing memory pages between multiple VMs [30,5,33,17]. The same observation has been made for VM disks [21,23,14], which is exploited to decrease storage consumption. To our knowledge, Sapuntzakis et al. [25] were the first to leverage identical data between multiple VMs to improve migration. However, their work predates live migration and supported only suspend/resume migration, where the VM is paused before being migrated. Additionally, they only took advantage of data available on the destination node, limiting the possibility of finding identical data. A similar approach was also proposed by Tolia et al. [28].

To allow live migration of VMs over WANs, two issues need to be solved: lack of shared storage and relocation to a different IP network. Storage can be migrated with algorithms similar to memory migration [3,16,10,32]. Network connections can be kept uninterrupted after migration using tunneling [29,3], VPN reconfiguration [32] or Mobile IPv6 [7]. The architecture of Shrinker supports storage migration. Additionally, since Shrinker focuses solely on optimizing data transmission during live virtual cluster migrations over WAN, it is compatible with all solutions for keeping network connections uninterrupted. Therefore, we do not discuss further this issue, as it is out of scope of this paper.

3 Architecture of Shrinker

We propose Shrinker, a system that improves live migration of virtual clusters over WANs by decreasing total data transferred and total migration time. During live migration of a virtual cluster between two data centers separated by a WAN, Shrinker detects memory pages and disk blocks duplicated among multiple virtual machines and transfers identical data only once. In the destination site, virtual machine disks and memory are reconstructed using distributed content-based addressing. In order to detect identical data efficiently, Shrinker leverages cryptographic hash functions. These functions map blocks of data, in our case memory pages or disk blocks, to hash values of fixed size, also called digests. These functions differ from ordinary hash functions because they are designed to render practically infeasible to find the original block of data from a hash value, modify a block of data while keeping the same hash value, and find two different blocks of data with the same hash value. Using hash values to identify memory pages and disk blocks by content is interesting because hash values are much smaller than the data they represent. For instance, a 4 kB memory page or disk block is mapped to a 20 bytes hash value using the SHA-1 [19] cryptographic hash function, a size reduction of more than 200 times.

We first present the architecture of Shrinker, and then discuss security considerations caused by our use of cryptographic hash functions.

3.1 Architecture Overview

In a standard live migration of a virtual cluster composed of multiple VMs running on different hypervisors, each live VM migration is independent. VM content is transferred from each source host to the corresponding destination host, with no interaction whatsoever between the source hypervisors or between the destination hypervisors. As a consequence, when migrating a virtual cluster over a WAN, data duplicated across VMs is sent multiple times over the WAN link separating the source and destination hypervisors.

To avoid this duplicated data transmission, Shrinker introduces coordination between the source hypervisors during the live migration process. This coordination is implemented by a service, running in the source site, that keeps track of which memory pages and disk blocks have been sent to the destination site.

Before sending a memory page or a disk block, a source hypervisor computes the hash value of this data and queries the service with this hash value. If no memory page or disk block with the same hash value has previously been sent to the destination site, the service informs the hypervisor that it should transfer the data. The service also registers the hash value in its database. Later, when the service receives a subsequent query for another memory page or disk block with the same hash value, it informs the querying hypervisor that the data has already been sent to the destination site. Based on this information, the hypervisor sends the hash value of the data to the destination host instead of the real content. This mechanism essentially performs data deduplication by replacing duplicated transmissions of memory pages and disk blocks by transmissions of much smaller hash values. Figure 1 illustrates a live migration between two pairs of hypervisors, with a cryptographic hash function creating 16-bit hash values.[1]

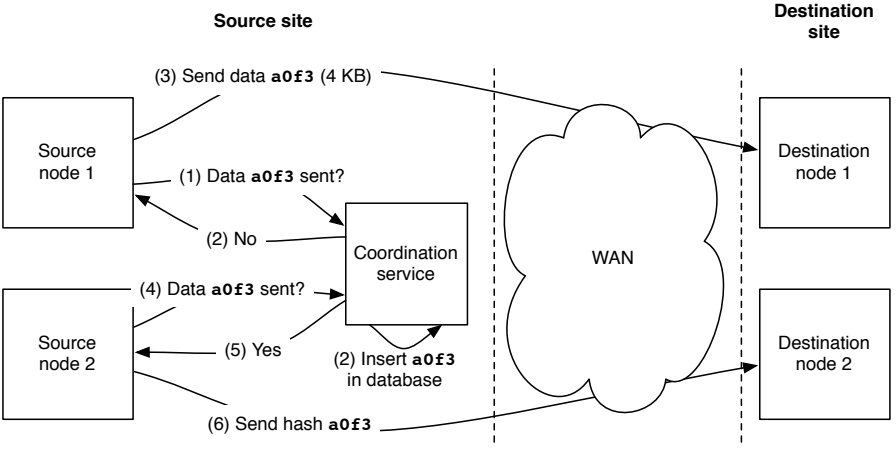


Fig. 1. Distributed data deduplication used in Shrinker to avoid sending duplicate data on a WAN link

Source node 1 has to send a memory page or disk block identified by the hash value `a0f3`. Instead of sending it directly like in a standard live migration, it first queries the coordination service (step 1). Since this is the first time that this data has to be sent to the destination site, the service informs source node 1 that it should send the data. It also updates its internal database to keep track of this hash value (step 2). After receiving the answer from the coordination service, source node 1 sends the data to destination node 1 (step 3). Afterwards, source node 2 queries the service for the same data (step 4). The service informs

[1] Note that this small hash value size is chosen to improve readability of the figure. In a real scenario, Shrinker can not use such hash function, since 65,536 different hash values would likely create collisions even for virtual machines of moderate sizes.

source node 2 that this data has already been sent to a node in the destination site (step 5), which prompts source node 2 to send the hash value `a0f3` (2 bytes) instead of the full content (4 kB) to destination node 2 (step 6).

Since destination nodes receive a mix of VM data (memory and disk content) and hash values from source nodes, they need to reconstruct the full VM content before the VM can be resumed. This is where distributed content-based addressing is used. When a full memory page or disk block is received by a destination node, its hash value and the IP of the node are registered into an indexing service, running in the destination site. When destination nodes receive hash values from source nodes, they query the indexing service to discover a node that has a copy of the content they are requesting. After contacting a node and receiving a copy of the content, they register themselves in the indexing service for this data, which increases the number of hosts capable of sending this data to another hypervisor. This process is illustrated in Fig. 2.

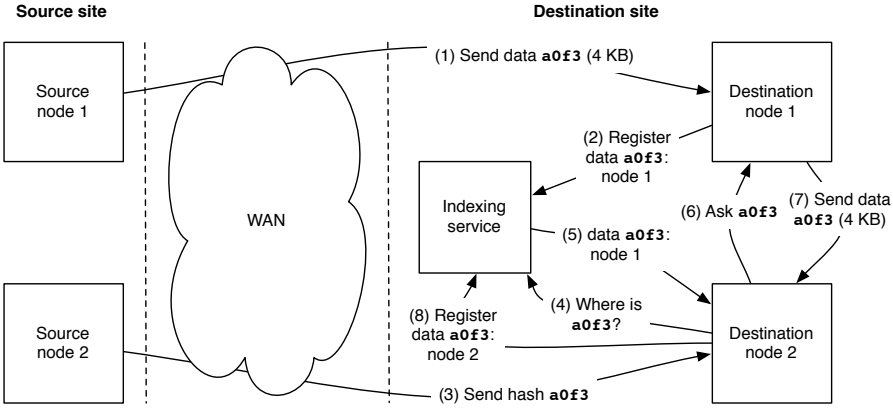


Fig. 2. Distributed content-based addressing used in Shrinker to reconstruct VM content on destination nodes

As in Fig. 1, destination node 1 receives a memory page or disk block identified by hash `a0f3` (step 1). It registers itself as having a copy of this data in the indexing service (step 2). Afterwards, as in Fig. 1, source node 2 sends hash value `a0f3` to destination node 2 (step 3). In order to get the data identified by this hash value, destination node 2 queries the indexing service to discover a host which has a copy of the corresponding content (step 4). The indexing service informs it that destination node 1 has a copy of this data (step 5). Destination node 2 asks destination node 1 for this content (step 6), receives it (step 7), and registers itself in the indexing server as having a copy (step 8).

Since the live migration is performed while the virtual machine is still executing, it is possible that data that was sent to the destination site has since been overwritten and is not accessible in any destination hypervisor. All

destination hypervisors open a communication channel with their corresponding source hypervisor. This channel can be used to directly request data to the source node.

3.2 Security Considerations

The possible number of different 4 kB memory pages and disk blocks (2^{4096}) is bigger than the number of possible hash values (2^{160} for SHA-1). As a consequence, the use of cryptographic hash functions opens the door to collisions: it is theoretically possible that memory pages and disk blocks with different content map to an identical hash value. However, the properties offered by cryptographic hash functions allow us to use these hash values with a high confidence. The probability p of one or more collisions occurring is bounded by (1), where n is the number of objects in the system and b the number of bits of hash values [22]:

$$p \leq \frac{n(n-1)}{2} \times \frac{1}{2^b} \ . \tag{1}$$

If we consider a very large virtual cluster consisting of 1 exabyte (2^{60} bytes) of 4 kB memory pages and disk blocks migrated by Shrinker using the SHA-1 hash function, the collision probability is around 10^{-20}. This is considered to be much less than other possible faults in a computing system, such as data corruption undetectable by ECC memory. However, (1) gives the probability of an accidental collision. Although the theoretical number of operations to find a collision is approximately $2^{\frac{n}{2}}$ (birthday attack), attackers can exploit weaknesses of the hash function algorithm to find collisions more easily. For example, researchers have shown attacks against SHA-1 that decrease the number of operations to find collisions from 2^{80} to 2^{69} [31]. Assuming that finding collisions is possible, an attacker capable of storing arbitrary data in VMs (for instance, by acting as a client interacting with web servers) could inject colliding data in these VMs. After migrating them with Shrinker, memory content would be corrupted because two different pages would have been replaced by the same data. If such attacks become practically feasible in the future, Shrinker can use stronger cryptographic hash functions, such as those from the SHA-2 family. Even though these hash functions are more computationally expensive, Shrinker remains interesting as long as the hash computation bandwidth is larger than the network bandwidth available to each hypervisor.

4 Implementation and Performance Evaluation

In this section, we first describe the implementation of our Shrinker prototype. Then, we present and analyze the results of our experiments performed on the Grid'5000 testbed.

4.1 Implementation

We implemented a prototype of Shrinker in the KVM hypervisor [12]. KVM is divided in two main parts. The first part is composed of two loadable kernel

modules providing the core virtualization infrastructure. The second part is a modified version of QEMU [2] running in user space to provide higher level features, including virtual device emulation and live migration. This prototype is fully implemented in the user space component of KVM version 0.14.0-rc0 and is about 2,000 lines of C code. We use Redis [24] version 2.2.0-rc4, a high performance key-value store, to implement the coordination and indexing service. Additional dependencies are OpenSSL, used to compute hash values, Hiredis, a C client library for Redis, and libev, a high-performance event loop library.

Support for data deduplication of storage migration in our prototype is not yet fully finalized. As such, in the following experiments, we use the storage migration mechanism of KVM, with copy-on-write images. The use of copy-on-write images allows KVM to send only storage data that has been modified since the boot of the VM, minimizing the amount of storage data to migrate. We expect our finalized prototype to offer even greater amounts of data deduplication for storage than for memory, since VM disks typically present large amounts of identical data, as shown in previous work (c.f. Sec. 2).

As explained in Sec. 3, before sending a memory page, source hypervisors need to contact the coordination service to know if the same page as already been sent to the destination site. Performing this query in a sequential manner would drastically reduce the live migration throughput (because of the round-trip time between source hypervisors and the coordination service). To overcome this problem, our implementation performs these queries in a pipelined manner. Queries for multiple memory pages are sent in parallel, and the decision of sending the full content or the hash is made when the answer is received. The same method is used on the destination site for hypervisors to get memory content.

The coordination service is implemented using the Redis `SETNX` command, with a memory page hash as key. If the hash is not already known by the service, it is registered and the return value notifies the source hypervisor that it was the first to register it. Otherwise, no change is performed in the service and the source hypervisor is notified that the hash was already registered.

The indexing service is implemented using the set data structure of Redis. For each registered memory page, there is one corresponding set which holds information about hypervisors having a copy of the page. When destination hypervisors register a memory page, they send a `SADD` command with the page hash as key and an IP/port pair as value. When other destination hypervisors need to get a page, they send a `SRANDMEMBER` command, which selects a random hypervisor in the set corresponding to the queried page and returns its IP/port information. After connecting on this IP and port, they query the content of the page. Finally, when a destination hypervisor doesn't hold any more copy of a page, it unregisters it with a `SREM` command.

4.2 Evaluation Methodology

All the experiments presented in this paper are run on the paradent cluster of the Grid'5000 site of Rennes. We use Carri System CS-5393B nodes supplied with 2 Intel Xeon L5420 processors (each with 4 cores at 2.5 GHz), 32 GB of

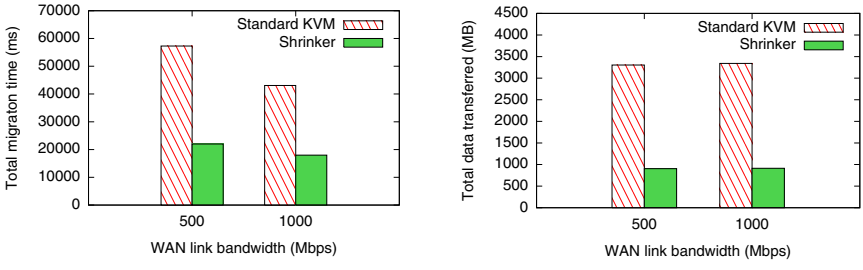
memory, and Gigabit Ethernet network interfaces. Physical nodes and VMs use the AMD64 port of Debian 5.0 (Lenny) as their operating system, with a 2.6.32 Linux kernel from the Lenny backports repository. VMs are configured to use 1 GB of memory. Two instances of Redis are running, each on a dedicated node.

To study the performance of our prototype, we configured a dedicated node to act as a router between source and destination hypervisors. Using the `netem` Linux kernel module, this router emulates wide area networks with different bandwidth rates. The emulated latency is set to a round trip time of 20 ms.

For our performance evaluation, we use a program from the NAS Parallel Benchmarks (NPB) [18], which are derived from computational fluid dynamics applications. We evaluate performance metrics of live migration during the execution of this workload. We measure total migration and total data transmitted by live migration. During our experiments, we discovered that the downtime caused by a live migration in KVM was overly important because of a miscalculation of the available bandwidth. This is why we do not report numbers for downtime, as they may be not representative of a correct behavior. We are investigating the issue and will report it to the KVM community.

4.3 Performance Results

Figure 3(a) shows the total migration time of a 16 VMs cluster executing the ep.D.16 program, for two different bandwidth rates. Figure 3(b) shows the total amount of transmitted data during the live migration of these 16 VMs, for the same two bandwidth rates.



(a) Average total migration time of 16 VMs running ep.D.16

(b) Total data transmitted over the WAN link during the live migration of 16 VMs running ep.D.16

Fig. 3. Performance evaluation of Shrinker

First, we observe that, whatever the bandwidth, the same amount of data is transmitted for both bandwidth rates. However, we can see that Shrinker sends much less data over the WAN link than the standard KVM migration. This allows Shrinker to reduce the total migration time, for instance from one minute to 20 seconds for the 500 Mbps link. Note that in the regular KVM live migration

protocol, memory pages containing only identical bytes are already compressed and sent efficiently. As such, the improvement showed by Shrinker come from real data deduplication, and not from deduplication of zero pages.

5 Conclusion

In this paper, we presented the design and prototype implementation of Shrinker, a system that improves live migration of virtual clusters over WANs by decreasing total data transferred and total migration time. Shrinker detects memory pages and disk blocks duplicated among multiple virtual machines to transfer identical data only once. Virtual machine data is retrieved in the destination site with distributed content-based addressing. We implemented a prototype of Shrinker in the KVM hypervisor. Our experiments show that it reduces both total data transferred and total migration time.

In the future, we will finalize our Shrinker prototype to perform data deduplication on storage migration and evaluate its performance. We also plan to study how Shrinker can allow destination hypervisors to fetch data from local VM image repositories found in most cloud computing infrastructures to further decrease the amount of data sent over WAN links.

Acknowledgments. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

1. Akoush, S., Sohan, R., Rice, A., Moore, A.W., Hopper, A.: Predicting the Performance of Virtual Machine Migration. In: International Symposium on Modeling, Analysis, and Simulation of Computer Systems (MASCOTS 2010), pp. 37–46 (2010)
2. Bellard, F.: QEMU, a Fast and Portable Dynamic Translator. In: Proceedings of the 2005 USENIX Annual Technical Conference (USENIX 2005), pp. 41–46 (2005)
3. Robert, B., Evangelos, K., Anja, F., Harald, S.: Live wide-area migration of virtual machines including local persistent state. In: Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE 2007), pp. 169–179 (2007)
4. Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I., Warfield, A.: Live Migration of Virtual Machines. In: Proceedings of the 2nd Symposium on Networked Systems Design & Implementation (NSDI 2005), pp. 273–286 (2005)
5. Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A.C., Varghese, G., Voelker, G.M., Vahdat, A.: Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In: 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008), pp. 309–322 (2008)
6. Hacking, S., Hudzia, B.: Improving the live migration process of large enterprise applications. In: Proceedings of the 3rd international Workshop on Virtualization Technologies in Distributed Computing (VTDC 2009), pp. 51–58 (2009)

7. Harney, E., Goasguen, S., Martin, J., Murphy, M., Westall, M.: The Efficacy of Live Virtual Machine Migrations Over the Internet. In: Proceedings of the 3rd International Workshop on Virtualization Technology in Distributed Computing (VTDC 2007), pp. 1–7 (2007)
8. Hines, M.R., Gopalan, K.: Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2009), pp. 51–60 (2009)
9. Hirofuchi, T., Nakada, H., Itoh, S., Sekiguchi, S.: Enabling Instantaneous Relocation of Virtual Machines with a Lightweight VMM Extension. In: Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid 2010), pp. 73–83 (2010)
10. Hirofuchi, T., Ogawa, H., Nakada, H., Itoh, S., Sekiguchi, S.: A Live Storage Migration Mechanism over WAN for Relocatable Virtual Machine Services over Clouds. In: Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2009), pp. 460–465 (2009)
11. Jin, H., Deng, L., Wu, S., Shi, X., Pan, X.: Live Virtual Machine Migration with Adaptive Memory Compression. In: Proceedings of the 2009 IEEE International Conference on Cluster Computing, Cluster 2009 (2009)
12. Kivity, A., Kamay, Y., Laor, D., Lublin, U., Liguori, A.: kvm: the Linux Virtual Machine Monitor. In: Proceedings of the 2007 Linux Symposium, vol. 1, pp. 225–230 (June 2007)
13. Lagar-Cavilla, H.A., Whitney, J.A., Scannell, A.M., Patchin, P., Rumble, S.M., Lara, E.d., Brudno, M., Satyanarayanan, M.: SnowFlock: rapid virtual machine cloning for cloud computing. In: Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys 2009), pp. 1–12 (2009)
14. Liguori, A., Hensbergen, E.V.: Experiences with Content Addressable Storage and Virtual Disks. In: Proceedings of the First Workshop on I/O Virtualization, WIOV 2008 (2008)
15. Liu, H., Jin, H., Liao, X., Hu, L., Yu, C.: Live migration of virtual machine based on full system trace and replay. In: Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing (HPDC 2009), Garching, Germany, pp. 101–110 (2009)
16. Luo, Y., Zhang, B., Wang, X., Wang, Z., Sun, Y., Chen, H.: Live and incremental whole-system migration of virtual machines using block-bitmap. In: 2008 IEEE International Conference on Cluster Computing (Cluster 2008), pp. 99–106 (2008)
17. Milos, G., Murray, D.G., Hand, S., Fetterman, M.: Satori: Enlightened Page Sharing. In: Proceedings of the 2009 USENIX Annual Technical Conference, USENIX 2009 (2009)
18. NASA Advanced Supercomputing Division: NAS Parallel Benchmarks, http://www.nas.nasa.gov/Software/NPB/
19. National Institute of Standards and Technology: Secure Hash Standard (April 1995)
20. Nelson, M., Lim, B.-H., Hutchins, G.: Fast Transparent Migration for Virtual Machines. In: Proceedings of the 2005 USENIX Annual Technical Conference (USENIX 2005), pp. 391–394 (2005)
21. Partho, N., Kozuch, M.A., O'Hallaron, D.R., Harkes, J., Satyanarayanan, M., Tolia, N., Toups, M.: Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines. In: Proceedings of the 2006 USENIX Annual Technical Conference (USENIX 2006), pp. 1–6 (2006)

22. Quinlan, S., Dorward, S.: Venti: A New Approach to Archival Storage. In: Proceedings of the Conference on File and Storage Technologies (FAST 2002), pp. 89–101 (2002)
23. Rhea, S., Cox, R., Pesterev, A.: Fast, inexpensive content-addressed storage in foundation. In: Proceedings of the 2008 USENIX Annual Technical Conference (USENIX 2008), pp. 143–156 (2008)
24. Sanfilippo, S.: Redis, http://redis.io
25. Sapuntzakis, C.P., Chandra, R., Pfaff, B., Chow, J., Lam, M.S., Rosenblum, M.: Optimizing the migration of virtual computers. In: Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002), pp. 377–390 (2002)
26. Svärd, P., Hudzia, B., Tordsson, J., Elmroth, E.: Evaluation of Delta Compression Techniques for Efficient Live Migration of Large Virtual Machines. In: Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2011), pp. 111–120 (2011)
27. Theimer, M.M., Lantz, K.A., Cheriton, D.R.: Preemptable remote execution facilities for the V-system. In: Proceedings of the Tenth ACM Symposium on Operating Systems Principles (SOSP 1985), pp. 2–12 (1985)
28. Tolia, N., Bressoud, T., Kozuch, M., Satyanarayanan, M.: Using Content Addressing to Transfer Virtual Machine State. Tech. rep., Intel Corporation (2002)
29. Travostino, F., Daspit, P., Gommans, L., Jog, C., de Laat, C., Mambretti, J., Monga, I., van Oudenaarde, B., Raghunath, S., Wang, P.Y.: Seamless live migration of virtual machines over the MAN/WAN. Future Gener. Comput. Syst. 22(8), 901–907 (2006)
30. Waldspurger, C.A.: Memory resource management in VMware ESX server. In: Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002), pp. 181–194 (2002)
31. Wang, X., Yin, Y.L., Yu, H.: Finding Collisions in the Full SHA-1. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17–36. Springer, Heidelberg (2005)
32. Wood, T., Ramakrishnan, K., Shenoy, P., van der Merwe, J.: CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines. In: Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2011 (2011)
33. Wood, T., Tarasuk-Levin, G., Shenoy, P., Desnoyers, P., Cecchet, E., Corner, M.: Memory Buddies: Exploiting Page Sharing for Smart Colocation in Virtualized Data Centers. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2009), pp. 31–40 (2009)
34. Zhang, X., Huo, Z., Ma, J., Meng, D.: Exploiting Data Deduplication to Accelerate Live Virtual Machine Migration. In: IEEE International Conference on Cluster Computing (Cluster 2010), pp. 88–96 (2010)

Maximum Migration Time Guarantees in Dynamic Server Consolidation for Virtualized Data Centers

Tiago Ferreto[1], César A.F. De Rose[1], and Hans-Ulrich Heiss[2]

[1] Faculty of Informatics, PUCRS, Brazil
tiago.ferreto@pucrs.br, cesar.derose@pucrs.br
[2] Technische Universitaet Berlin, Germany
heiss@cs-tu-berlin.de

Abstract. Server consolidation is a vital mechanism in modern data centers in order to minimize expenses with infrastructure. In most cases, server consolidation may require migrating virtual machines between different physical servers. Although the downtime of live-migration is negligible, the amount of time to migrate all virtual machines can be substantial, delaying the completion of the consolidation process. This paper proposes a new server consolidation algorithm, which guarantees that migrations are completed in a given maximum time. The migration time is estimated using the max-min fairness model, in order to consider the competition of migration flows for the network infrastructure. The algorithm was simulated using a real workload and shows a good consolidation ratio in comparison to other algorithms, while also guaranteeing a maximum migration time.

1 Introduction

Server consolidation is a key feature in current virtualized data centers. It focuses on minimizing the amount of resources required to handle the data center workload, and therefore, it has a direct impact on the costs of the data center infrastructure. Due to the variations in demand of the applications executed in the data center, virtual machines capacities should be periodically revisited in order to provide good performance to the applications and minimize overprovisioning. A server consolidation algorithm evaluates these capacity changes and derives a new mapping of virtual machines to the available resources. The new mapping may require migrating virtual machines among physical servers, which can be performed using live-migration techniques with negligible downtime.

Although the migration of virtual machines is imperceptible for the users, migrating several virtual machines concurrently can require a considerable amount of time, which results in a larger delay in the server consolidation process. Therefore, estimating the total migration time is essential when planning the server consolidation of virtual machines. Even so, current server consolidation algorithms disregard this matter.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 443–454, 2011.

This paper proposes a new server consolidation algorithm which minimizes the number of required physical servers to handle a set of virtual machines, while also guaranteeing that the migrations performed in the transition to the new mapping be completed in a specified maximum time. The algorithm was evaluated using real workloads from TU-Berlin and compared with common implementations of the server consolidation problem, such as heuristics and linear programming. The results obtained show that the algorithm is able to guarantee a maximum migration time in most cases and also minimizes considerably the number of migrations performed, while requiring a small amount of additional physical servers.

2 Related Work

The server consolidation problem consists in mapping a set of virtual machines with different capacities to a set of physical servers in order to minimize the number of physical servers required. This problem is analogous to the classic bin-packing problem, which is classified as an NP-hard problem [6], and aims at mapping a set of items with different capacities into a minimal set of bins. There are several approaches in the literature to solve this problem, in which the most common use heuristics and linear programming. Some of the most common heuristics for the bin-packing problem are [6,11]: first-fit, best-fit, worst-fit and almost worst-fit. Each heuristic uses a different policy to select which bin should receive an item. A common optimization is to order the items in decreasing order before starting the mapping process, resulting in the algorithms: first-fit decreasing, best-fit decreasing, worst-fit decreasing and almost worst-fit decreasing. The main goal of heuristics is to find good solutions at a reasonable computational cost, however it does not guarantee optimality. Another common approach is to use linear programming to find an optimal solution. The drawback of this approach is that it usually has higher requirements in terms of computing power and time.

Applications running in data centers usually present periods of high and low utilization. In order to minimize the amount of active resources, the capacity required in each virtual machine is periodically evaluated, and a new mapping is produced using server consolidation. Server consolidation techniques have widespread adoption in virtualized data centers. However, the process of mapping virtual machines to physical servers is not trivial. Depending on the application requirements and the resource provider goals, different strategies can be applied. Several works have been published in the last years proposing different approaches in the server consolidation process.

Andrzejak *et al.* [2] proposed static and dynamic server consolidation algorithms based on integer programming and genetic algorithm techniques. The algorithms were evaluated using a production workload containing traces from enterprise applications. The results present the benefits of using server consolidation, showing that the same workload could be allocated in a much smaller number of physical servers. The genetic algorithm resulted in solutions as good

as with integer programming, with the benefit of reaching the solution much faster. In this work, migrations are considered to happen instantaneously, i.e., there is no migration cost included in the algorithms.

Speitkamp and Bichler [4,15] described linear programming formulations for the static and dynamic server consolidation problems. They also designed extension constraints for limiting the number of virtual machines in a physical server, guaranteeing that some virtual machines are assigned to different physical servers, mapping virtual machines to a specific set of physical servers that contain some unique attribute, and limiting the total number of migrations for dynamic consolidation. In addition, they proposed an LP-relaxation based heuristic for minimizing the cost of solving the linear programming formulations.

Bobroff *et al.* [5] proposed a dynamic server consolidation algorithm, which focus on minimizing the cost of running the data center. The cost is measured using a penalty over underutilized and overloaded physical servers, and over service level agreements (SLA) violations, defined as CPU capacity guarantees. The algorithm uses historical data to forecast future demand and relies on periodic executions to minimize the number of physical servers to support the virtual machines.

Khanna *et al.* [9] proposed a dynamic management algorithm, which is triggered when a physical server becomes overloaded or underloaded. The main goals of their algorithm are to: i) guarantee that SLAs are not violated (SLAs are specified considering mainly response time and throughput); ii) minimize migration cost; iii) optimize the residual capacity of the system; and iv) minimize the number of physical servers used. Migration cost is defined as the amount of resources used by each virtual machine.

Wood *et al.* [17] developed the Sandpiper system for monitoring and detecting hotspots, and remapping/reconfiguring virtual machines whenever necessary. In order to choose which virtual machines to migrate, Sandpiper sorts them using a volume-to-size ratio (VSR), which is a metric based on CPU, network, and memory loads. Sandpiper tries to migrate the most loaded virtual machine from an overloaded physical server to one with sufficient spare capacity.

Mehta and Neogi [13] introduced the ReCon tool, which aims at recommending dynamic server consolidation in multi-cluster data centers. ReCon considers static and dynamic costs of physical servers, the costs of virtual machine migration, and the historical resource consumption data from the existing environment in order to provide an optimal dynamic plan of virtual machines to physical server mapping over time. Virtual machine migration costs are defined as directly related to amount of resources used by the VM, such as CPU and memory. Similarly, Verma *et al.* [16] developed the pMapper architecture and a set of server consolidation algorithms for heterogeneous virtualized resources. The algorithms take into account power and migration costs and the performance benefit when consolidating applications into physical servers. In the pMapper architecture, the migration cost is analyzed as the impact of the migration during the application execution. However, the migration cost model only considers the impact when migrating a single virtual machine, i.e. the migration cost does not change when several migrations occur concurrently.

Despite the several approaches investigated in server consolidation, none of them have already studied the real impact of virtual machines migration in the server consolidation process. Most of the works that deal with virtual machine migration only take into account the number of migrations, or the amount of memory transferred (related to the amount of resources used by the VM), but the impact of these transfers in the completion of the consolidation process have never been explored. It is necessary to estimate how long does it take to migrate each virtual machine considering that they compete for the network infrastructure.

3 Server Consolidation Algorithm

The server consolidation problem focus on minimizing the number of physical servers required to map a list of virtual machines, respecting the capacity of physical servers and demands of virtual machines. In the algorithm proposed here, besides minimizing the number of physical servers, it also aims at establishing a maximum migration time during the server consolidation process. The migration time directly reflects the amount of time required to complete the consolidation.

One of the main challenges is to accurately estimate migration time, taking into consideration that migrations are performed in a shared network infrastructure, and it may affect the available bandwidth for each migration. In the proposed algorithm, the evaluation of the available bandwidth for each migration is performed using the max-min fairness (MMF) model [8,14]. This model is often considered in the context of IP networks carrying elastic traffic. It presents the following properties: i) all transfers have the same priority over the available bandwidth, ii) link bandwidths are fairly shared among transfers being allocated in order of increasing demand, iii) no transfer gets a capacity larger than its demand, and iv) transfers with unsatisfied demands get an equal share of the link bandwidth.

Given a set of network links with respective bandwidths and the links used by each migration, it is possible to obtain the available bandwidth for each migration using a progressive filling algorithm which respects the MMF model properties [3]. The algorithm initializes the bandwidth available for each transfer with 0. It increases the bandwidth for all transfers equally, until one link becomes saturated. The saturated links serve as a bottleneck for all transfers using them. The bandwidths for all transfers not using these saturated links are incremented equally until one or more new links become saturated. The algorithm continues, always equally incrementing all transfer bandwidths not passing through any saturated link. When all transfers pass through at least one saturated link, the algorithm stops.

The migration time is measured as the time it takes to transfer the current memory allocation of each virtual machine, with the available bandwidth obtained using the MMF model. However, it is not possible to simply divide one by the other. Considering that some migrations finish before others, the available

bandwidth for each migration can change, and the remaining amount of memory to be transferred should take into account the new available bandwidth. Therefore, migration time is measured in incremental steps.

After each migration finishes, the amount of time passed is added to the migration time of all virtual machines, and the amount of memory transferred during this time is decreased from the total amount of memory to transfer. If there is no more memory to transfer, the migration is removed from the set of running migrations. The algorithm continues until there are no more running migrations. In the end, we have an estimation of the migration time of each migration.

The proposed algorithm is divided in two distinct phases. The first phase aims at finding a feasible mapping of virtual machines to physical servers that minimizes the maximum migration time of all virtual machines. In the second phase, the feasible mapping is iteratively modified in order to produce solutions using a smaller number of physical servers, but also respecting a maximum migration time threshold. In cases where it is not possible to guarantee migration times smaller than a specified threshold, the algorithm finds a solution that minimizes the number of virtual machines that have their migration times higher than the specified threshold.

3.1 First Phase: Minimizing Migration Time

The first phase is based on the traditional descent method for local neighborhood search. Based on an initial solution, a set of small modifications to this solution is derived. The result obtained with each modification is analyzed and the best one is chosen. If this modification optimizes the current solution, then it is applied and the process repeats, otherwise the algorithm stops.

Algorithm 1 presents the algorithm for the first phase. It starts using a copy of the current mapping as the current solution. The repetition of the current mapping results in zero migrations, however the physical servers can become overloaded, i.e., the physical server capacity can not be able to handle the changes of the virtual machines demands mapped to it. The strategy is to remove each overloaded physical server from this overload state, migrating some of its virtual machines to other physical servers, choosing every time the alternative that results in minimal migration times. The algorithm generates migration alternatives for each overloaded physical server. The physical server that performs migrations which results in minimal migration time is chosen. The migrations are included in the current solution, and the process repeats, until there are no more overloaded physical servers.

The migration alternatives for each overloaded physical server are generated as the combinations of virtual machines that remove the physical server from the overload state. For example, given an overloaded physical server with capacity 100, packing virtual machines: v_1 with demand 50, v_2 with demand 40, v_3 with demand 30, and v_4 with demand 20, the algorithm generates the following combinations of virtual machines: (v_1), (v_2), and (v_3, v_4). Each combination of virtual machines is applied in the current solution using the best-fit decreasing

heuristic and its cost is evaluated. The cost function considers the maximum migration time and also the sum of the migration times of all migrations. The goal is to find a feasible mapping, respecting physical servers capacities and virtual machines demands, which minimizes this cost function, i.e., results in the lowest maximum migration time.

The algorithm repeats until there are no more overloaded physical servers. In the end, the result generated will contain at least the same amount of physical servers as the present mapping. The second phase is used to decrease the number of physical servers, while guaranteeing that the maximum migration time stays under a given threshold.

Algorithm 1. First phase of the server consolidation algorithm

```
current_solution ← getCurrentMapping()
while there are overloaded physical servers in current_solution do
  for all p in overloaded physical servers do
    alternatives ← generateMigrationAlternatives(p)
    for all alt in alternatives do
      sol ← bestFitDecreasing(alt)
      cost ← calculateCost(sol)
    end for
  end for
  current_solution ← getAlternativeWithLowestCost()
end while
```

3.2 Second Phase: Minimizing the Number of Physical Servers

The second phase is implemented using the tabu search metaheuristic [7]. The main idea of tabu search is to maintain a memory about previous local searches, in order to avoid performing repeatedly the same moves, returning to the same solution and staying confined into a local optima. The tabu search method is based on a repetition of steps that explores the possible solutions for the problem.

The second phase is presented in Algorithm 2, and it starts by using as initial solution the mapping resulted from the first phase. The strategy is to select in each iteration one physical server to empty, reassigning its virtual machines to other physical servers. The selection of the physical server is based on a filling function, proposed by [12], which gives a measure of easiness to empty a physical server. The function gives higher priority to physical servers with low occupied capacity and more virtual machines packed on it. The physical server with the lowest filling index, according to the filling function, and that is not in the tabu list is chosen. The tabu list stores a list of previously chosen physical servers, and the goal of the tabu list is to avoid choosing repeatedly the same physical servers to empty.

After choosing the physical server using the filling function, the virtual machines mapped to it are retrieved and the physical server is set as unavailable during this iteration, in order to avoid remapping all virtual machines to it again. The list of virtual machines is used as input to a permutation function, which returns lists with these virtual machines in all possible orderings. Each alternative is evaluated, applying the worst-fit heuristic to map the virtual machines to the physical servers. The alternative that provides a solution with best cost, according to the cost function, is selected and its solution is defined as the current solution. The cost function that should be minimized combines the number of physical servers and the number of breaks of maximum migration time. This last term refers to the number of migrations with migration time higher than the specified threshold.

The physical server chosen at first is included in the tabu list and set as available again to pack virtual machines in the next iterations. The tabu list size has a fixed capacity, and when this capacity is exceeded, the oldest entry is removed. The tabu list size should be smaller than the number of physical servers. If the cost of the current solution is smaller than the cost of the best solution, then the current solution is defined as the best solution. This process repeats until the best solution does not present any enhancements in a pre-specified number of iterations. The final solution can result in a situation that guaranteeing the maximum migration time is not possible, however it will minimize the number of breaks of maximum migration time.

4 Evaluation

The evaluation of the server consolidation algorithm was performed using workloads composed of traces from servers of the Technical University of Berlin (TU-Berlin), which are normally used by researchers and students to execute computational experiments. Each workload contains samples of CPU and memory utilization per hour during a week, totalizing 168 samples per trace. The workloads present different characteristics, such as: number of traces, average CPU utilization, average memory utilization, and average variability. The variability in a trace indicates the percentage of consecutive samples that present a variation in CPU or memory values. A variability of 0% indicates that the trace keeps with same CPU and memory utilization during the whole trace duration, whereas 100% indicates that each consecutive sample presents a different CPU or memory utilization. The higher the variability, higher also is the probability of changes in the mapping of virtual machines to physical machines, and hence a higher number of migrations can be performed. Table 1 presents the workloads and its characteristics.

The physical infrastructure simulated in the experiments is a data center composed of 100 homogeneous physical servers with CPU and memory capacities equal to 100. Each server is connected to a single crossbar switch using bidirectional links forming a star network topology. Each link has a capacity of 100 per unit of time. It means that it takes 1 unit of time to transfer all memory from one physical server to another one using full link capacity.

Algorithm 2. Second phase of the server consolidation algorithm

```
current_solution ← getFirstPhaseMapping()
best_solution ← current_solution
repeat
  for all p in physical_servers not in tabu_list do
    p_index ← getFillingIndex()
  end for
  p ← getPhysicalServerWithLowestIndex()
  vms ← getVirtualMachinesFrom(p)
  set p as unavailable
  moves ← getPermutations(vms)
  for all m in moves do
    sol ← worstFit(m)
    cost ← calculateCost(sol)
  end for
  current_solution ← getAlternativeWithLowestCost()
  insertIntoTabuList(p)
  set p as available
  if calculateCost(current_solution) < calculateCost(best_solution) then
    best_solution ← current_solution
  end if
until termination condition
```

Table 1. Details of TU-Berlin workload groups

	Number of traces	Avg. CPU utilization (%)	Avg. memory utilization (%)	Avg. variability (%)
Workload 1	43	25.2	28.36	17
Workload 2	61	32.37	39.39	41
Workload 3	36	47.28	48.67	63

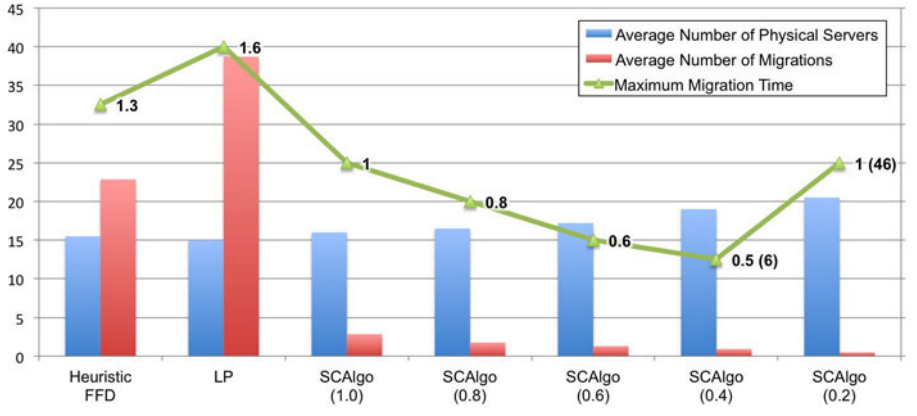
The server consolidation algorithm proposed was compared with typical implementations of the server consolidation problem using heuristics and linear programming. The heuristics implemented were: first-fit decreasing (FFD), best-fit decreasing (BFD), worst-fit decreasing (WFD) and almost worst-fit decreasing (AWFD). They were all implemented using the Python language. The linear programming (LP) solution was implemented using Zimpl [10] and solved using the SCIP [1] solver. The solver was configured with a timeout of 5 minutes, i.e., if the solver can not find the optimal result in 5 minutes, it returns the best result found so far. This approach is usually used since linear programming problems can take a long time to find an optimal solution. All experiments were performed on a Intel Core 2 Duo processor with 2.4 GHz and 4 GBytes of memory. The server consolidation algorithm proposed was implemented using the Python language. The second phase (using Tabu Search) was configured with tabu list size

equal to 5 and to terminate when the solution does not present any changes in the last 10 iterations (termination condition). The first mapping is performed using the first-fit heuristic since there is no previous mapping to be used by the algorithm. The server consolidation algorithm was executed using five different thresholds of maximum migration time. The thresholds are: 1.0, 0.8, 0.6, 0.4 and 0.2 units of time.

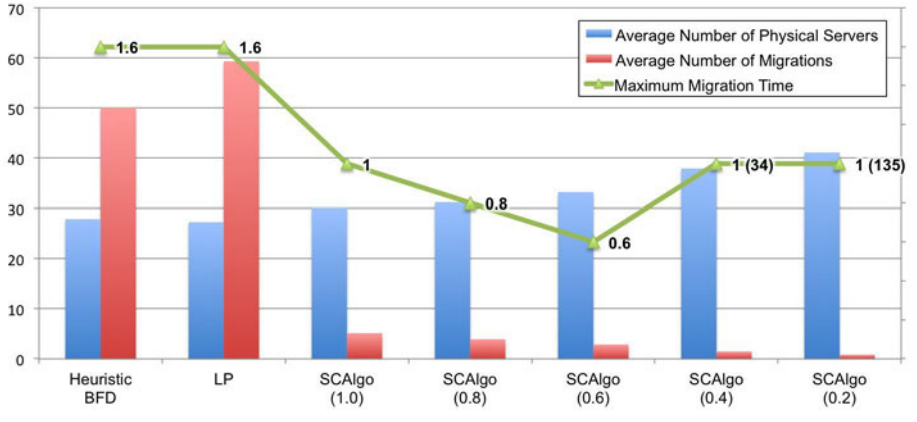
For each experiment combining algorithm and workload, the metrics measured are: the average number of physical servers required to process the workload, the average number of migrations required, and the maximum migration time. Figures 1a, 1b and 1c present the results obtained with the algorithms for each workload. The heuristics bar only presents the heuristic that presented the best solution.

LP presented the lowest average number of physical servers in all workloads, but it was closely followed by the best heuristic. However, LP also presented the highest average number of migrations, requiring migrating almost all virtual machines each consolidation step in all workloads. The average percentage of virtual machines migrated each consolidation step using LP are: 90% for workload 1, 97.2% for workload 2 and 95.8% for workload 3 using LP. These high values are due to the aggressive methods applied by linear programming in order to find an optimal solution. Despite the lowest number of physical servers, the high number of migrations is a huge obstacle for its utilization in a real environment. The heuristics required a lower number of migrations, but with a considerable increase when using workloads with higher variability. Heuristics tend to keep the mapping of virtual machines in the same physical servers when there is a low variation in the virtual machines capacities. Besides presenting a high number of migrations, LP and heuristics also presented, as expected, a high maximum migration time, considering that these algorithms only try to optimize the number of physical servers used by the workload.

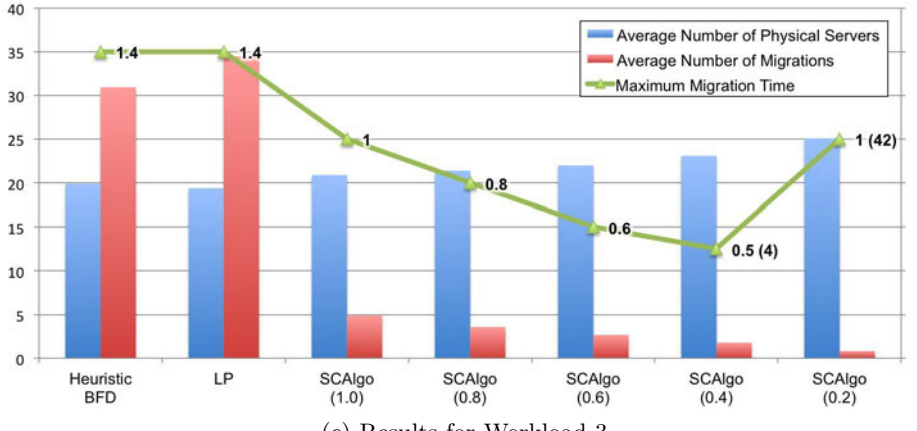
The proposed server consolidation algorithm (named as SCAlgo in the charts) was able to maintain the maximum migration time for the first three thresholds: 1.0, 0.8 and 0.6. The reduction in the number of required migrations is considerable, requiring only a small increase in the number of physical servers. In workload 1, with the time threshold of 1.0, the algorithm migrates an average of 6.5% of the virtual machines against 90% using LP, requiring only an increase of a single machine in average. Workloads 2 and 3 migrate an average of 8.3% (against 97.2% in LP) and 13.6% (against 95.8% in LP) of virtual machines, while requiring an increase of approximate 2.9 and 1.5 machines in average in comparison to the results obtained using LP. In the cases where the proposed server consolidation could not guarantee the maximum migration time (thresholds of 0.4 and 0.2), the algorithm minimized the number of migration breaks, i.e., the number of migrations that presented migration times higher than the specified threshold. In the charts, the number of migration breaks is presented in parentheses besides the maximum migration time for thresholds 0.4 and 0.2. This limitation is directly related to the characteristics of the workload, specially regarding the amount of memory utilization. Even when using



(a) Results for Workload 1



(b) Results for Workload 2



(c) Results for Workload 3

full network capacity, virtual machines with high memory demands cannot be transferred during the maximum migration time specified.

5 Conclusion and Future Work

This paper presented a new server consolidation algorithm to be used in virtualized data centers that, besides minimizing the number of physical servers used, also guarantees that all necessary migrations occur during a specified maximum migration time. The maximum migration time has direct relation to the completion of the consolidation process and, therefore, should be taken into account in the server consolidation algorithm. Several algorithms have been proposed for the server consolidation problem, but none of them have focused on ensuring a maximum migration time in order to minimize the delay in the consolidation process. The server consolidation algorithm proposed has been evaluated using real workloads and typical solutions for server consolidation using linear programming and heuristics. The results obtained indicate that the proposed algorithm can efficiently provide guarantees using common thresholds of time, with a huge decrease in the number of migrations performed and a slight increase in the number of additional physical servers required. As future work, we intend to perform more experiments using different workloads and optimize the algorithm in order to provide guarantees of maximum migration time in more complex scenarios. We also intend to analyze how resource providers can use the maximum migration time in SLAs in order to offer a more controlled environment for its users.

References

1. Achterberg, T.: SCIP - a framework to integrate constraint and mixed integer programming. Tech. Rep. 04-19, Zuse Institute Berlin (2004)
2. Andrzejak, A., Arlitt, M., Rolia, J.: Bounding the resource savings of utility computing models. Technical report hpl-2002-339, Hewlett Packard Laboratories (2002)
3. Bertsekas, D., Gallager, R.: Data Networks. Prentice-Hall, Englewood Cliffs (1992)
4. Bichler, M., Setzer, T., Speitkamp, B.: Capacity planning for virtualized servers. In: Proceedings of the 16th Annual Workshop on Information Technologies and Systems, WITS 2006 (2006)
5. Bobroff, N., Kochut, A., Beaty, K.: Dynamic placement of virtual machines for managing sla violations. In: Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management (2007)
6. Coffman Jr., E., Garey, M., Johnson, D.: Approximation Algorithms for Bin Packing - A Survey. In: Approximation algorithms for NP-hard problems, PWS Publishing Co. (1996)
7. Floudas, C.C.A., Pardalos, P.M.: Encyclopedia of Optimization. Springer-Verlag New York, Inc., Secaucus (2006)
8. Ioannis, D.: New Algorithm for the Generalized Max-Min Fairness Policy based on Linear Programming. IEICE Transactions on Communications (2005)

9. Khanna, G., Beaty, K., Kar, G., Kochut, A.: Application performance management in virtualized server environments. In: Proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium, NOMS 2006 (2006)
10. Koch, T.: Rapid Mathematical Programming. Ph.D. thesis, Technische Universitaet Berlin (2004)
11. Kou, L., Markowsky, G.: Multidimensional bin packing algorithms. IBM Journal of Research and development 21(5) (1977)
12. Lodi, A., Martello, S., Vigo, D.: TSpack: A Unified Tabu Search Code for Multi-Dimensional Bin Packing Problems. Annals of Operations Research 131(1-4) (2004)
13. Mehta, S., Neogi, A.: ReCon: A Tool to Recommend Dynamic Server Consolidation in Multi-Cluster Data Centers. In: Proceedings of the IEEE Network Operations and Management Symposium, NOMS 2008 (2008)
14. Nace, D., Nhatdoan, L., Klopfenstein, O., Bashllari, a.: Max-min fairness in multi-commodity flows. Computers & Operations Research 35(2) (2008)
15. Speitkamp, B., Bichler, M.: A Mathematical Programming Approach for Server Consolidation Problems in Virtualized Data Centers. IEEE Transactions on Services Computing (2010)
16. Verma, A., Ahuja, P., Neogi, A.: pMapper: Power and migration cost aware application placement in virtualized systems. In: Proceedings of the ACM/IFIP/USENIX 9th International Middleware Conference (2008)
17. Wood, T., Shenoy, P.J., Venkataramani, A., Yousif, M.S.: Sandpiper: Black-box and gray-box resource management for virtual machines. Computer Networks 53(17) (2009)

Enacting SLAs in Clouds Using Rules

Michael Maurer[1], Ivona Brandic[1], and Rizos Sakellariou[2]

[1] Vienna University of Technology, Distributed Systems Group,
Argentinierstraße 8, 1040 Vienna, Austria
{maurer,ivona}@infosys.tuwien.ac.at
[2] University of Manchester, School of Computer Science, U.K.
rizos@cs.man.ac.uk

Abstract. The emergence of Cloud Computing raises the question of dynamically allocating resources of physical (PM) and virtual machines (VM) in an on-demand and autonomic way. Yet, using Cloud Computing infrastructures efficiently requires fulfilling three partially contradicting goals: first, achieving low violation rates of Service Level Agreements (SLA) that define non-functional goals between the Cloud provider and the customer; second, achieving high resource utilization; and third achieving the first two issues by as few time- and energy consuming reallocation actions as possible. To achieve these goals we propose a novel approach with escalation levels to divide all possible actions into five levels. These levels range from changing the configuration of VMs over migrating them to other PMs to outsourcing applications to other Cloud providers. In this paper we focus on changing the resource configuration of VMs in terms of storage, memory, CPU power and bandwidth, and propose a knowledge management approach using rules with threat thresholds to tackle this problem. Simulation reveals major improvements as compared to recent related work considering SLA violations, resource utilization and action efficiency, as well as time performance.

1 Introduction

One of the main challenges Cloud Computing providers face is the question of dynamically allocating resources in an on-demand way. *Service Level Agreements* (SLAs) settle non-functional requirements between the Cloud Computing providers and their customers. These SLAs contain Quality of Service (QoS) goals, which are expressed as, e.g., "storage $\geq 1000GB$". Penalties that have to be paid to the customers in case these goals are violated are also part of the SLA. Thus, Cloud Computing providers face three contradicting challenges: First, they aim for providing enough resources for every application. Second, they try to efficiently use their resources and only allocate what applications currently really need. Third, they consider energy consumption of reallocation actions and strive for an efficient usage of these executed actions.

In [10] we presented Case Based Reasoning (CBR) for decision making in the MAPE-K (Monitoring, Analysis, Planning, Execution, Knowledge) cycle of an autonomic SLA enactment environment in Clouds. We evaluated it by a generic

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 455–466, 2011.

simulation engine we developed and showed the suitability of CBR for resource-efficient SLA management. However, we also determined some drawbacks of CBR as far as its learning performance and its scalability were concerned. Therefore, in this paper we design and implement a rule-based knowledge management (KM) approach, and utilize the same simulation engine enhanced by more accurate and general utility functions to evaluate it and reevaluate CBR. Using rules we attempt to improve not only SLA adherence and resource allocation efficiency as discussed in [10], but also new aspects, i.e., the efficient use of reallocation actions, as well as scalability. Additionally, we adapt a wholesome view of different adaptation levels like virtual machine (VM) configuration or VM migration, and propose a hierarchical model of so called *escalation levels* for dynamically and efficiently managing resource allocation for Cloud Computing infrastructures.

The challenge in this work is to evaluate KM techniques for autonomic SLA enactment in Cloud Computing infrastructures that fulfill the three following conflicting goals: (i) achieving low SLA violation rates; (ii) achieving high resource utilization such that the level of allocated but unused resources is as low as possible; and (iii) achieving (i) and (ii) by as few time- and energy consuming reallocation actions as possible. We will call this problem *resource allocation problem* throughout the rest of the paper.

The main contributions of this paper are:

1. partitioning the resource allocation problem for Cloud infrastructures into several subproblems by proposing *escalation levels* that structure all possible reaction possibilities into different subproblems using a hierarchical model.
2. designing, implementing and evaluating a rule-based approach to propose a solution for one of the subproblems presented in 1), i.e., for *virtual machines* in Cloud infrastructures, and comparing it to the CBR approach.

2 Related Work

Concerning related work, we have determined two different ways to compare our work with other achievements in this area. Whereas the first level compares other works dealing with SLA enactment and resource efficiency, the second one considers the area of knowledge management.

At first, considerable work on optimizing resource usage while keeping QoS goals has been conducted. One general main difference to our approach consists of the fact that related work examines either only certain subsystems of large-scale distributed systems, as [8] the performance of memory systems, or constrain themselves to one or two specific SLA parameters. Whereas Petrucci et al. [14] or Bichler et al. [4] consider one general resource constraint, Khanna et al. [2] only investigate response time and throughput, and Kalyvianaki [6] CPU usage. [5,17] examine specific use cases of web servers deployed in Cloud-like environments by investigating horizontal scaling of servers. The Sandpiper framework [18], which offers black-box and gray-box resource management for VMs, provides a quite similar approach to ours. Contrary to our project, though, Sandpiper plans reactions just after violations have occurred. Also the VCONF model by Rao

et al. [15] has similar goals as presented in Section 1, but depends on specific parameters, can only execute one action per iteration and neglects the energy consumption of executed actions. Other papers focus on different escalation levels (as described in Section 3). [19,12] focus on VM migration and [11] on turning on and off physical machines, whereas our paper focuses on VM re-configuration.

Secondly, there has been work on KM of SLAs, especially rule-based systems. Paschke et al. [13] look into a rule based approach in combination with the logical formalism ContractLog. It specifies rules to trigger after a violation has occurred, e.g., it obliges the provider to pay some penalty, but it does not deal with avoidance of SLA violations. Others inspected the use of ontologies as knowledge bases (KBs) only at a conceptual level. Koumoutsos et al. [9] view the system in four layers (i.e., business, system, network and device) and break down the SLA into relevant information for each layer, but give no implementation details. Bahati et al. [3] also use policies, i.e., rules, to achieve autonomic management. They provide a system architecture including a KB and a learning component, and divide all possible states of the system into so called regions, which they assign a certain benefit for being in this region. A bad region would be, e.g., response time > 500 (too slow), fair region response time < 100 (too fast, consuming unnecessary resources) and a good region $100 \leq$ response time ≤ 500. The actions are not structured, but are mixed together into a single rule, which makes the rules very hard to manage and to determine a salience concept behind them. However, we share the idea of defining "over-utilized", "neutral" and "under-utilized" regions. Our KM system allows to choose any arbitrary number of resource parameters that can be adjusted on a VM. Moreover, our paper provides a more wholesome approach than related work and integrates the different action levels that work has been carried out on.

3 Escalation Levels

This section presents a methodology of dividing the resource allocation problem into smaller subproblems using a hierarchical approach. It demonstrates which actions can be executed in what level to achieve SLA adherence and efficient resource allocation for Cloud infrastructures. We call these levels *escalation levels* and present them in Table 1. The idea is that every problem that occurs should be solved on the lowest escalation level. Only if this is not possible, the problem is tried to be solved on the next level, and again, if this fails, on the next one, and so on. The levels are ordered in a way such that lower levels offer faster and more local solutions than higher ones. The first escalation level ("change VM configuration") works locally on a PM and tries to change the amount of storage or memory, e.g., that is allocated to the VM from the PM resources. Then, migrating applications (escalation level 2) is more light-weight than migrating VMs and turning PMs on/off (escalation levels 3 and 4). For all three escalation levels already the whole system state has to be taken into account to find an optimal solution. The problem stemming from escalation level 3 alone can be formulated into a *Binary integer problem* (BIP), which is known to be NP-complete [7]. The proof is out of scope for this paper, but a similar approach can

Table 1. Escalation levels

1. Change VM configuration.
2. Migrate applications from one VM to another.
3. Migrate one VM from one PM to another or create new VM on appropriate PM.
4. Turn on / off PM.
5. Outsource to other Cloud provider.

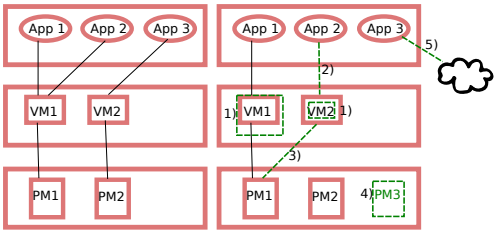
be seen in [14]. The last escalation level has least locality and greatest complexity, since the capacity of other Cloud infrastructures have to be taken into account too, and negotiations have to be started with them as well.

Also the rule-based approach benefits from this hierarchical action level model, because it provides a salience concept for contradicting rules. Without this concept it would be troublesome to determine which of the actions, e.g., "Power on additional PM with extra-storage and migrate VM to this PM", "Increase storage for VM by 10%" or "Migrate application to another VM with more storage" should be executed, if a certain threshold for allocated storage has been exceeded. The proposed rule-based approach will present a solution for escalation level 1.

Figure 1 visualizes the escalation levels from Table 1 in the context of Infrastructure as a Service (IaaS) before and after actions are executed. Figure 1(a) shows applications *App1* and *App2* deployed on *VM1* that is itself deployed on *PM1*, whereas *App3* runs on *VM2* running on *PM2*. Figure 1(b) shows example actions for all five escalation levels. The legend numbers correspond to the respective numbering of the escalation levels.

- *Escalation level 1*: At first, the autonomic manager tries to change VM configuration. Actions 1) show *VM1* being up-sized and *VM2* being down-sized.
- *Escalation level 2*: If the attempt to increase a certain resource for a VM in escalation level 1 fails, because some resource cannot be increased anymore due to the constraints of the PM hosting the VM, in level 2 the autonomic manager tries to migrate the application to another larger VM that fulfills the required specifications from level 1. So if, e.g., provided storage needs to be increased from 500 to 800GB, but only 200 GB are available on the respective VM, then the application has to be migrated to a VM that has at least the same resources as the current one plus the remaining 100GB of storage. Action 2) shows the re-deployment of *App2* to *VM2*. Due to possible confinements of some applications to certain VMs, e.g., a user deployed several applications that need to work together on one VM, this escalation might be skipped in some scenarios.
- *Escalation level 3*: If there is no appropriate VM available in level 2, in level 3 the autonomic manager tries to create a new VM on an appropriate PM or migrate the VM to a PM that has enough available resources. Action 3) shows the re-deployment of *VM2* to *PM1*.

– *Escalation level 4*: Again, if there is no appropriate PM available in level 3, the autonomic manager suggests turning on a new PM (or turning it off if the last VM was emigrated from this PM) in level 4. Action 4) shows powering on a new PM (*PM3*).
– *Escalation level 5*: Finally, the last escalation level 5 tries to outsource the application to another Cloud provider as explained, e.g., in the Reservoir project [16]. Action 5) outsources *App3* to another Cloud provider.



(a) Before action execution (b) After action execution

Fig. 1. Actions used in 5 escalation levels: before and after action execution

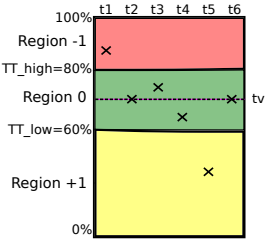


Fig. 2. Example behavior of actions at time intervals t1-t6

4 Rule-Based Approach for VM Level

This section describes the rule-based approach for escalation level 1.

4.1 Prerequisites

For resource management, we need to define how the *measured*, *provided* and *agreed* values interrelate, and what an SLA violation actually is [10]. The *measured* value (1) represents the amount of a specific resource that is currently used by the customer. The amount of *allocated* (2) resource determines to what extent a specific resource can be used by the customer, i.e., how much of the resource is allocated to the VM hosting the application. The *agreed* value (3) corresponds to the Service Level Objective (SLO) agreed in the SLA. An *SLA violation* occurs, if less is provided (2) than the customer utilizes (or wants to utilize) (1) with respect to the limits set in the SLA (3).

Dealing with SLA-bound resource management, where resource usage is paid for on a "pay-as-you-go" basis with SLOs that guarantee a minimum capacity of these resources as described above, raises the question, whether the Cloud provider should allow the consumer to use more resources than agreed. We will refer to this behavior as *over-consumption*. Since the consumer will pay for every additional resource, it should be in the Cloud provider's interest to allow over-consumption as long as this behavior does not endanger the SLAs of other consumers. Thus, Cloud providers should not allow over-consumption when the

Table 2. Resource policy modes

green	Plenty of resources left. Over-consumption allowed.
green-orange	Heavy over-consumption is forbidden. All applications that consume more than τ% (threshold to be specified) of the agreed resource SLO are restrained to $\tau/2$% over-consumption
orange	Resource is becoming scarce, but SLA demand can be fulfilled if no over-consumption takes place. Thus, over-provisioning is forbidden.
orange-red	Over-provisioning forbidden. Initiate outsourcing of some applications.
red	Over-provisioning forbidden. SLA resource requirements of all consumers cannot be fulfilled. If possible, a specific choice of applications is outsourced. If not enough, applications with higher reputation points or penalties are given priority over applications with lower reputation points / penalties. SLAs of latter ones are deliberately broken to ensure SLAs of former ones.

resulting penalties they have to pay are higher than the expected revenue from over-consumption. To tackle this problem, we introduce five policy modes for every resource that describe the interaction of the five escalation levels. As can be seen in Table 2 the policy modes are green, green-orange, orange, orange-red and red. They range from low utilization of the system with lots of free resources left (policy mode green) over a scarce resource situation (policy mode orange) to an extreme tight resource situation (policy mode red), where it is impossible to fulfill all SLAs to its full extent and decisions have to be made which SLAs to deliberately break and which applications to outsource.

4.2 Design and Implementation

In order to know whether a resource r is in danger of under-provisioning or already is under-provisioned, or whether it is over-provisioned, we calculate the current utilization $ut^r = \frac{\text{use}^r}{\text{pr}^r} \times 100$, where use^r and pr^r signify how much of a resource r was used and provided, respectively, and divide the percentage range into three regions by using the two "threat thresholds" TT^r_{low} and TT^r_{high}:

- Region −1: Danger of under-provisioning, or under-provisioning ($> TT^r_{high}$)
- Region 0: Well provisioned ($\leq TT^r_{high}$ and $\geq TT^r_{low}$)
- Region +1: Over-Provisioning ($< TT^r_{low}$)

The idea of this rule-based design is that the ideal value that we call *target value* $tv(r)$ for utilization of a resource r is exactly in the center of region 0. So, if the utilization value after some measurement leaves this region by using more (Region -1) or less resources (Region +1), then we reset the utilization to the target value, i.e., we increase or decrease allocated resources so that the utilization is again at

$$tv(r) = \frac{TT^r_{low} + TT^r_{high}}{2}\%.$$

As long as the utilization value stays in region 0, no action will be executed. E.g., for $r = \text{storage}, TT^r_{low} = 60\%$, and $TT^r_{high} = 80\%$, the target value would be $tv(r) = 70\%$. Figure 2 shows the regions and measurements (expressed as utilization of a certain resource) at time steps $t_1, t_2, \ldots, t_6$. At t_1 the utilization

of the resource is in Region −1, because it is in danger of a violation. Thus, the KB recommends to increase the resource such that at the next iteration t_2 the utilization is at the center of Region 0, which equals the target value. At time steps t_3 and t_4 utilization stays in the center region and consequently, no action is required. At t_5, the resource is under-utilized and so the KB recommends the decrease of the resource to $tv(r)$, which is attained at t_6. Additionally, if over-provisioning is allowed in the current policy mode, then the adjustment will always be executed as described regardless of what limit was agreed in the SLA. On the other hand, if over-provisioning is not allowed in the current policy mode, then the rule will allocate at most as much as agreed in the SLA (SLO^r).

The concept of a rule increasing resource r is depicted in Figure 3. The rule executes if the current utilization ut^r and the predicted utilization $ut^r_{predicted}$ of the next iteration (cf. next paragraph) both exceed TT^r_{high} (line 2). Depending on what policy level is active the rule either sets the provided resource pr^r to the target value $tv(r)$ for policy levels green and green-orange (line 3) or to at most what was agreed in the SLA (SLO^r) plus a certain percentage ϵ to account for rounding errors when calculating the target value in policy levels orange, orange-red and red (line 5). A similar rule scheme for decreasing a resource can be seen in Figure 4. The main difference is that it does not distinguish between policy modes and that it sets the provisioned resource to at least a minimum value $minPr^r$, which may be 0, that is needed to keep the application alive (line 4). The rule is executed if the current utilization ut^r and the predicted utilization $ut^r_{predicted}$ of the next iteration both lie below TT^r_{low} (line 2).

A large enough span between the thresholds TT^r_{low} and TT^r_{high} helps to prevent oscillations of repeatedly increasing and decreasing the same resource. However, to further reduce the risk of oscillations, we suggest to calculate a prediction for the next value based on the latest measurements. Thus, an action is only invoked when the current AND the predicted measurement exceed the respective TT. So, especially when only one value exceeds the TT, no action is executed.

1 **IF**
2 $ut^r > TT^r_{\text{high}}$ AND $ut^r_{predicted} > TT^r_{\text{high}}$
3 **THEN**
4 Set pr^r to $\frac{\text{use}^r}{tv(r)}$ for policy modes green, green-orange.
5 Set pr^r to $\min(\frac{\text{use}^r}{tv(r)}, SLO^r * (1+\epsilon/100))$ for policy modes orange, orange-red, red.

Fig. 3. Rule scheme for increasing a resource

1 **IF**
2 $ut^r < TT^r_{\text{low}}$ AND $ut^r_{predicted} < TT^r_{\text{low}}$
3 **THEN**
4 Set pr^r to $\max(\frac{\text{use}^r}{tv(r)}, minPr^r)$.

Fig. 4. Rule scheme for decreasing a resource

The rules have been implemented using the Java rule engine Drools [1]. The Drools engine sets up a knowledge session consisting of different rules and a working memory. Rules get activated when specific elements are inserted into the working memory such that the conditional "when" part evaluates to true. Activated rules are then triggered by the simulation engine. In our case, the simulation engine inserts measurements and SLAs of applications into the working

memory. Different policy modes will load slightly modified rules into the Drools engine and thus achieve a high adaptability of the KM system reacting to the general performance of the Cloud infrastructure. As opposed to the CBR based approach in [10], the rule-based approach is able to fire more than one action at the same iteration, which inherently increases the flexibility of the system. Without loss of generality we can assume that one application runs on one VM (several applications' SLAs can be aggregated to form one VM SLA) and we assume the more interesting case of policy modes orange, orange-red or red, where over-provisioning is not allowed.

5 Evaluation

In this section we evaluate the quality of the proposed rule-based approach measured by a utility function, as well as its time performance and scalability.

5.1 Utility-Driven Evaluation

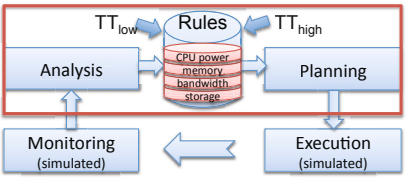


Fig. 5. Simulation engine evaluating a rule-based knowledge management system

We evaluated the rule-based approach for escalation level 1 with the simulation engine described in [10]. This simulation engine simulates measurements of SLA parameters for an arbitrary number of VMs, forwards them to the KB, asks the KB for appropriate actions, and simulates the execution of these actions; thus it traverses the complete MAPE cycle in one iteration as depicted in Figure 5. As resources for IaaS one can use all parameters that can be adapted on a VM. For the evaluation we chose to take the following parameters and SLOs: *storage* $\geq$ 1000GB, *incoming bandwidth* $\geq$ 20 Mbit/s, *outgoing bandwidth* $\geq$ 50 Mbit/s, *memory* $\geq$ 512 MB, and *CPU power* $\geq$ 100 MIPS (Million Instructions Per Second).

The simulation engine keeps track of the SLA of every VM, the violations thereof, resource utilization and costs of action execution. All evaluations for this subsection are executed with 100 iterations and 500 applications. We investigate low, middle and high values for TT^r_{low} and TT^r_{high}, where $TT^r_{\text{low}} \in \{30\%, 50\%, 70\%\}$ and $TT^r_{\text{high}} \in \{60\%, 75\%, 90\%\}$ for all resources stated above. We combine the TTs to form eight different scenarios as depicted in Table 3. The workload follows an (increasing or decreasing) trend for an a-priori unknown period of time and different for every resource. As the intensity of the trend varies for every iteration, the simulation comprises both, slow developments and rapid changes, and thus simulates workload in a quite general way.

To be able to compare the utility of the individual threshold pairs, we define a generic cost function that maps SLA violations, resource wastage and the costs of

Table 3. 8 Simulations Scenarios for TT_{low} and TT_{high}

	Scenarios							
	1	2	3	4	5	6	7	8
TT_{low}	30%	30%	30%	50%	50%	50%	70%	70%
TT_{high}	60%	75%	90%	60%	75%	90%	75%	90%

executed actions into a monetary unit, which we want to call *Cloud EUR*. First, we define a penalty function $\mathfrak{p}^r(p) : [0, 100] \rightarrow \mathbb{R}^+$ that defines the relationship between the percentage of violations p (as opposed to all possible violations) and the penalty for a violation of resource r. Second, we define a function wastage $\mathfrak{w}^r(w) : [0, 100] \rightarrow \mathbb{R}^+$ that relates the percentage of unused resources w to the energy in terms of money that these resources unnecessarily consume. Third, we define a cost function $\mathfrak{a}^r(a) : [0, 100] \rightarrow \mathbb{R}^+$ from the percentage of executed actions a (as opposed to all possible actions that could have been executed) to the energy and time costs in terms of money. The total cost c is then defined as

$$c(p, w, c) = \sum_r \mathfrak{p}^r(p) + \mathfrak{w}^r(w) + \mathfrak{a}^r(a). \tag{1}$$

We assume functions $\mathfrak{p}^r$, $\mathfrak{w}^r$ and $\mathfrak{a}^r$ for this evaluation with $\mathfrak{p}^r(p) = 100p$, $\mathfrak{w}^r(w) = 5w$, and $\mathfrak{a}^r(a) = a$ for all r. The intention behind choosing these functions is (i) to impose very strict fines in order to proclaim SLA adherence as top priority, (ii) to weigh resource wastage a little more than the cost of actions.

In Figure 6 we compare the outcome of the rule-based approach evaluating the aforementioned eight scenarios. From Figure 6(a) we see that in terms of SLA violations Scenario 1 achieves the best result, where only 0.0908% of all possible violations occur, and the worst result with Scenario 8, with a still very low violation rate of 1.2040%. In general, the higher the values are for TT_{high}, the worse is the outcome. The best result achieved with CBR was at 7.5%.

Figure 6(b) shows resource utilization. We see that the combination of high TT_{low} and high TT_{high} (Scenario 8) gives the best utilization (83.98%), whereas low values for TT_{low} and TT_{high} lead to the worst utilization (62.03% in Scenario 1). Still, compared to CBR which scored a maximum of 80.36% and a minimum of 51.81%, the rule-based approach generally achieves better results. Also, when comparing resource allocation efficiency (RAE), which is defined as

$$\text{RAE} = \frac{u}{v + 1}, \tag{2}$$

where u is the average utilization over all resources and v is the number of violations, the rule-based approach achieves a maximum of 795.9 and a minimum of 69.8 (see Figure 6 (e)), whereas CBR achieves 10.0 at most.

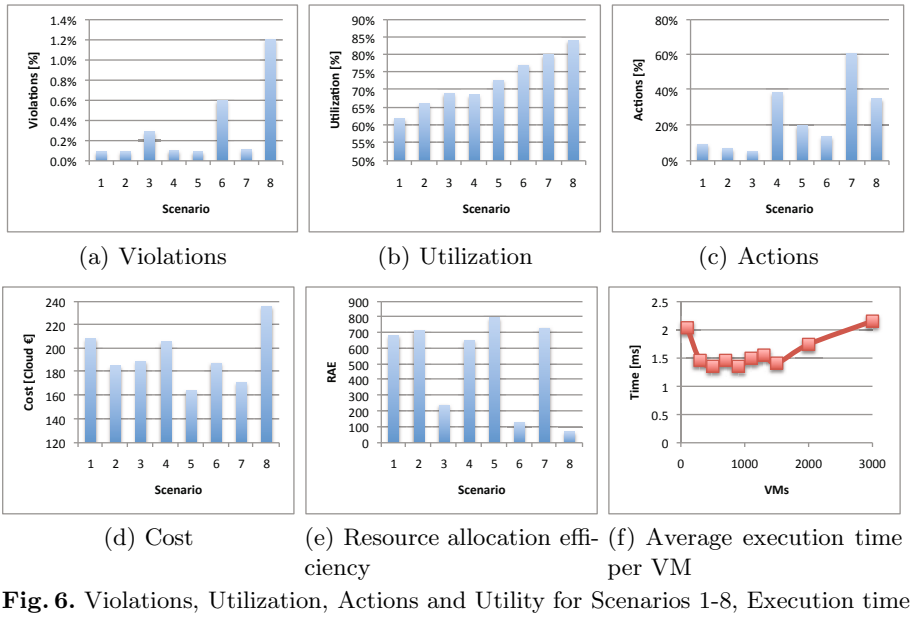
The percentage of all executed actions as compared to all possible actions that could have been executed is shown in Figure 6(c). One observes that the greater the span between TT_{low} and TT_{high} is, the less actions have to be executed. Most actions (60.75%) are executed for Scenario 7 (span of only 5% between TT values), whereas least actions (5.44%) are executed for Scenario 3 (span of 60%

between TT values). CBR almost always recommended exactly one action and hardly ever (in about 1% of the cases) recommended no action.

Figure 6(d) shows the costs for each scenario using Equation 1. The best trade-off between the three terms is achieved by Scenario 5 that has medium values for TT^r_{low} and TT^r_{high}. It has a very low violation rate of 0.0916%, a quite elaborate utilization of 72.90%, but achieves this with only 19.79% of actions. Scenario 7 achieves a better violation and utilization rate but at the cost of an action rate of 60.75%, and consequently has higher costs. The lowest cost value for CBR is 923 Cloud EUR, the highest 2985 Cloud EUR.

If the utility of the decision decreases for a certain time frame (as cost increases), the KB could determine the cost summand in Equation 1 that contributes most to this decrease. For any resource r, if the term is $\mathfrak{p}$, then decrease TT^r_{high}. If the term is $\mathfrak{w}$, then increase TT^r_{low}. Otherwise, if the term is $\mathfrak{c}$, then widen the span of TT^r_{high} and TT^r_{low}, i.e., increase TT^r_{high} and decrease TT^r_{low}. We plan to investigate this in our future research.

Summarizing, we have seen that in all 8 scenarios the proposed approach outperforms the CBR approach with respect to the SLA violation rate (up to 82 times better results) and the resource allocation efficiency (up to 80 times better results). 7 out of 8 scenarios achieved better results in terms of actions needed and were better than the worst CBR value for utilization, whereas only one scenario was better than the best CBR utilization value. However, accumulating these results into cost, all rule-based scenarios outperform CBR by a factor of at least 4 (worst rule-based scenario (236) compared to best CBR result (923)), which to a large extent is due to the huge number of violations that the rule-based approach is able to prevent and the high number of actions it can save.



(a) Violations (b) Utilization (c) Actions

(d) Cost (e) Resource allocation efficiency (f) Average execution time per VM

Fig. 6. Violations, Utilization, Actions and Utility for Scenarios 1-8, Execution time

5.2 Performance-Driven Evaluation

As far as time performance and scalability is concerned, the performance tests are very encouraging. We executed 100 iterations from 100 to 3000 VMs. We performed every test twice and calculated the average execution time as well as the average time it took for the simulation engine to handle one VM. As shown in Figure 6(f) the execution time per VM stays quite constant for up to 1500 VMs, and thus average execution time is about linear. For 3000 VMs, it took $647s/100 = 6.47s$ for one iteration to treat all VMs. The high time consumption per VM for 100 VMs in Figure 6(f) is due to the initialization of the rule knowledge base which takes over-proportionally long for just a small number of VMs and does not weigh so much for more VMs.

CBR took 240s for 50VMs and 20 iterations. Thus, CBR took $240s/20 = 12s$ for one iteration to treat all VMs, which is twice as long as the rule-based approach takes, which even has 60 times more VMs. However, CBR implements learning features, what the rule-based currently does not, and could be sped up by choosing only specific cases to be stored in the KB.

6 Conclusion and Outlook

This paper structured the set of possible actions to govern Cloud infrastructures into five escalation levels from changing the configuration of virtual machines over migrating them to other physical machines to outsourcing applications to other Cloud providers. A use case has been developed together with resource policy modes that govern the high-level behavior of Cloud infrastructures. We developed a rule-based knowledge management approach to tackle the first of the five escalation levels: dynamic adaptation of VM configuration in an energy-efficient way. We proposed a rule-based approach and showed that it had several advantages over an approach using Case Based Reasoning (CBR). We tested the rule-based approach using 8 scenarios that differed in the threat thresholds employed to mark the limits between "regular performance", over- and under-utilization. In almost all scenarios, we gained even better results with the rule-based approach than with CBR. In the future we want to ameliorate the cost functions by relating them to real-world measurements of energy consumption and with it learn and adjust the high and low threat thresholds.

Acknowledgments. The work described in this paper is supported by the Vienna Science and Technology Fund (WWTF) under grant agreement ICT08-018 Foundations of Self-Governing ICT Infrastructures (FoSII) and by COST-Action IC0804 on Energy Efficiency in Large Scale Distributed Systems.

References

1. Drools, http://www.drools.org
2. Application Performance Management in Virtualized Server Environments (2006), http://dx.doi.org/10.1109/NOMS.2006.1687567

3. Bahati, R.M., Bauer, M.A.: Adapting to run-time changes in policies driving autonomic management. In: ICAS 2008: Proceedings of the 4th Int. Conf. on Autonomic and Autonomous Systems. IEEE Computer Society, Washington, DC, USA (2008)
4. Bichler, M., Setzer, T., Speitkamp, B.: Capacity Planning for Virtualized Servers. Presented at Workshop on Information Technologies and Systems (WITS), Milwaukee, Wisconsin, USA (2006)
5. Dutreilh, X., Rivierre, N., Moreau, A., Malenfant, J., Truck, I.: From data center resource allocation to control theory and back. In: 2010 IEEE 3rd International Conference on Cloud Computing (CLOUD), 2010, pp. 410–417 (July 2010)
6. Kalyvianaki, E., Charalambous, T., Hand, S.: Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In: Proceedings of the 6th International Conference on Autonomic Computing, ICAC 2009, pp. 117–126. ACM, New York (2009), http://doi.acm.org/10.1145/1555228.1555261
7. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (eds.) Complexity of Computer Computations: Proc. of a Symp. on the Complexity of Computer Computations, pp. 85–103. Plenum Press (1972)
8. Khargharia, B., Hariri, S., Yousif, M.S.: Autonomic power and performance management for computing systems. Cluster Computing 11(2), 167–181 (2008)
9. Koumoutsos, G., Denazis, S., Thramboulidis, K.: SLA e-negotiations, enforcement and management in an autonomic environment. In: Modelling Autonomic Communications Environments, pp. 120–125 (2008)
10. Maurer, M., Brandic, I., Sakellariou, R.: Simulating autonomic SLA enactment in clouds using case based reasoning. In: Di Nitto, E., Yahyapour, R. (eds.) ServiceWave 2010. LNCS, vol. 6481, pp. 25–36. Springer, Heidelberg (2010)
11. Mazzucco, M., Dyachuk, D., Deters, R.: Maximizing cloud providers' revenues via energy aware allocation policies. In: CLOUD 2010, pp. 131–138 (2010)
12. Meng, X., Isci, C., Kephart, J., Zhang, L., Bouillet, E., Pendarakis, D.: Efficient resource provisioning in compute clouds via VM multiplexing. In: Proceeding of the 7th International Conference on Autonomic Computing, ICAC 2010, pp. 11–20. ACM, New York (2010), http://doi.acm.org/10.1145/1809049.1809052
13. Paschke, A., Bichler, M.: Knowledge representation concepts for automated SLA management. Decision Support Systems 46(1), 187–205 (2008)
14. Petrucci, V., Loques, O., Mossé, D.: A dynamic optimization model for power and performance management of virtualized clusters. In: e-Energy 2010, pp. 225–233. ACM, New York (2010)
15. Rao, J., Bu, X., Xu, C.-Z., Wang, L., Yin, G.: Vconf: a reinforcement learning approach to virtual machines auto-configuration. In: ICAC 2009, pp. 137–146. ACM, New York (2009), http://doi.acm.org/10.1145/1555228.1555263
16. Rochwerger, B., et al.: The RESERVOIR model and architecture for open federated cloud computing. IBM Journal of Research and Development 53(4) (2009), http://www.research.ibm.com/journal/rd/534/rochwerger.pdf
17. Singh, R., Sharma, U., Cecchet, E., Shenoy, P.: Autonomic mix-aware provisioning for non-stationary data center workloads. In: ICAC 2010, pp. 21–30. ACM, New York (2010), http://doi.acm.org/10.1145/1809049.1809053
18. Wood, T., Shenoy, P., Venkataramani, A., Yousif, M.: Sandpiper: Black-box and gray-box resource management for virtual machines. Computer Networks 53(17), 2923–2938 (2009)
19. Yazir, Y.O., Matthews, C., Farahbod, R., Neville, S., Guitouni, A., Ganti, S., Coady, Y.: Dynamic resource allocation in computing clouds using distributed multiple criteria decision analysis. In: 2010 IEEE 3rd International Conference on Cloud Computing (CLOUD), pp. 91–98 (2010)

DEVA: Distributed Ensembles of Virtual Appliances in the Cloud

David Villegas and Seyed Masoud Sadjadi

School of Computing and Information Sciences
Florida International University
Miami, FL, USA
{dvill013,sadjadi}@cs.fiu.edu

Abstract. Low upfront costs, rapid deployment of infrastructure and flexible management of resources has resulted in the quick adoption of cloud computing. Nowadays, different types of applications in areas such as enterprise web, virtual labs and high-performance computing are already being deployed in private and public clouds. However, one of the remaining challenges is how to allow users to specify Quality of Service (QoS) requirements for composite groups of virtual machines and enforce them effectively across the deployed resources. In this paper, we propose an Infrastructure as a Service resource manager capable of allocating *Distributed Ensembles of Virtual Appliances* (DEVAs) in the Cloud. DEVAs are groups of virtual machines and their network connectivities instantiated on heterogeneous shared resources with QoS specifications for individual entities as well as their connections. We discuss the different stages in their lifecycle: declaration, scheduling, provisioning and dynamic management, and show how this approach can be used to maintain QoS for complex deployments of virtual resources.

1 Introduction

Infrastructure as a Service (IaaS) clouds allow users to instantiate Virtual Machines (VMs) on demand in remote shared resources for a certain period of time. One of the currently faced challenges in such systems is allowing users to specify fine-grained requirements for groups of resources and ensure that the promised Quality of Service (QoS) is met for them, not only in terms of individual machines, but also in their aggregate traffic assignment. This requirement is essential to run certain parallel and distributed workloads such as scientific applications that rely on low network latencies or high bandwidth, for example. We propose an IaaS cloud manager to tackle this problem at different levels: user request definition, scheduling of virtual resources and management of physical infrastructure to secure the requested service.

In order to maximize resource utilization, providers assign VMs to shared physical infrastructure. Consequently, mechanisms need to be implemented to ensure that utilization is fairly distributed according to the requested allocation. These measures have to consider various aspects such as VM placement,

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 467–478, 2011.

creation of virtual network links between them that provide the appropriate bandwidth and latency, and dynamic monitoring and management of the composite allocations. Our proposed work in this paper is an attempt to address the above mentioned issues in the current IaaS implementations such as Amazon [1], OpenNebula [14], Eucalyptus [11] or Nimbus [6].

Our approach allows users to submit requests by specifying the requirements of their application. A cloud resource manager is in charge of brokering for the appropriate resources and acquiring them for the desired time. This process is akin to the act of planning for traditional computing equipment, where hardware —architecture, processor speed, memory, switching and routing devices, or machine interconnections— can be carefully tailored based on costs and capabilities, except with the benefits of the cloud, such as elasticity or pay-per-use. An additional advantage is that, by defining concrete Service Level Agreements (SLA), the broker can use heterogeneous resources, reducing the fragmentation between clouds with different capabilities. This fact can also be employed to enable federation among providers.

In this paper, we implement a heuristic placement algorithm based on the *assign* mapping method used in Emulab [4] for network topologies. Next, we create a cloud interface on top of OpenNebula which accepts user requests for Distributed Ensembles of Virtual Appliances (DEVAs) with annotated network connections and VM descriptions. We also define *DEVA agents* in charge of enforcing QoS, isolating traffic, monitoring resource usage and tunneling packets between remote resources to seamlessly create layer 2 networks among ensemble members. Finally, we perform experiments to validate our architecture and demonstrate how QoS can be fulfilled.

The results demonstrate that our system can be used to instantiate groups of VMs in clouds with user-defined QoS requirements to execute different types of applications. The DEVA IaaS manager enables fine-grained control over the allocated resources, allowing users to request the appropriate infrastructure and providers to apply the right policies so that resources are not allocated in excess.

2 System Overview

Figure 1 depicts the general architecture of the system and the flow of interactions among the different components. Users prepare a request description based on their application requirements and send it to the Cloud Interface component. When a request is received, it is parsed and a graph describing the virtual deployment is generated. The Mapper component tries to find the most appropriate resources based on pre-defined site policies and the fulfillment of the user specified requirements. The resulted mapping is then sent to a VM provisioner, for example, the OpenNebula [14], which is the one that we used in our prototype implementation. The provisioner transfers the required VM images to the destination nodes, instantiates them, and notifies the DEVA Agents in every hosting node. Each agent then applies the appropriate network mechanisms to perform traffic monitoring, isolation and traffic control. Agents also monitor the state of

the VMs and notify the central manager. The steps mentioned in the process are enumerated and discussed in detail next.

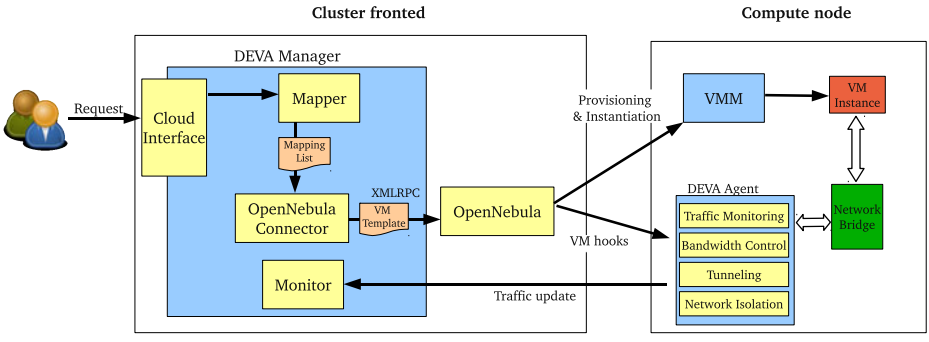


Fig. 1. General architecture

We define a *Distributed Ensemble of Virtual Appliances* (or DEVA) as a group of virtual appliances, including virtual machines, virtual network devices, and their connections, altogether with a set of QoS requirements applied either globally or to individual members of the ensemble. DEVAs can be described using XML, and sent to a resource manager capable of processing and instantiating them within a pool of physical resources. Our current implementation supports single VM requirements such as CPU power and memory, network bandwidth in megabits per second and latency, as *low* or *high*.

3 DEVA Manager

3.1 Mapping of DEVAs

After a DEVA description is sumitted to the DEVA Manager, the Mapper module decides where to place each of the individual components. During this process, there is a match-making algorithm that selects those resources that can fulfill the request: this stage considers both individual VMs (*i.e.,* available CPU and memory), and the whole ensemble (network connectivity, available bandwidth, *etc.*). We assume that physical resources may be heterogeneous, and that they may belong to different administrative domains. We implement a centralized match-making approach, where the main process is on the DEVA manager front-end node and has all the information about the available resources, even if they are located on different sites.

As it has already been discussed in the literature, the process of mapping a virtual topology to a physical one is an NP-hard problem, making comprehensive algorithms too costly. Instead, we have adopted the *assign* algorithm [12], used in emulab [4], for the mapping stage.

Our implementation of *assign* differs from the original one in various aspects based on the different targeted use. While this algorithm was originally designed

to solve the so called *network testbed mapping problem* (*i.e.*, how to find an optimal or close to optimal mapping from a virtual network topology to a phisical one), we take a more pragmatic approach, focusing on providing the required connectivity and QoS rather than an exact replica of the topology.

The algorithm considers five types of connection mapping with different scores that lead to various solutions. These types are:

- **Trivial**: Both VMs share the same physical host, thus sharing an internal connection.
- **Direct**: The hosting machines are directly connected through a cable.
- **IntraSwitch**: The hosting machines are connected to the same switch.
- **InterSwitch**: The hosting machines are connected to different switches which in turn are connected through a cable.
- **InterRouter**: The hosting machines are in different layer 2 networks, connected by one or more routers in between.

Each of the itemized connection types has a cost in terms of latency and possible bandwdith, which is accounted for in the mapping process. The algorithm gives each connection a score, promoting results with better connectivity. The case of *InterRouter* connections is special since, when a connection between two machines that share a logical layer 2 link is mapped to this kind of connection, tunneling will be necessary at the provisioning stage, which also results in additional network latency. The algorithm takes this fact in consideration, creating a policy violation if the mapped connection won't be able to fulfill the request. In particular, low latency links from the DEVA request mapped to InterRouter connection may result in a policy violation.

3.2 DEVAs across Heterogeneous Resources

The previous phase of the process is in charge of finding a good mapping between the virtual topology and the physical resources. The *assign* algorithm outputs a list of pairs of *virtual resource* to *physical resources* mappings altogether with the policies in the links. The next step takes this mapping and realizes it.

We use the *OpenNebula* virtual infrastructure manager, version 1.4, to provision and control individual machines. *OpenNebula* is an IaaS resource manager that receives requests via a command line interface or remote procedure calls and instantiates the appropriate VMs, described by a template file. The DEVA manager receives the output of the infrastructure mapper and translates it to calls to *OpenNebula*'s XMLRPC protocol in order to provision and start the VMs. The process to realize a DEVA involves four steps:

1. Translate the original DEVA request into *OpenNebula* VM templates
2. Send instantiation requests to *OpenNebula*, using the mapping results to indicate the appropriate hosting machines
3. Apply the network configuration at each machine to ensure QoS is met
4. Monitor traffic at each host and aggregate it at the DEVA manager to form a picture of the overall state

OpenNebula's VM template requires some information that is provided in the original DEVA request, such as the location of the kernel, ramdisk and filesystem images. Other data is generated by the DEVA manager, such as the VM's MAC address, and the rest of information is provided by the mapping algorithm, for example the destination host. The template is created dynamically and sent to *OpenNebula*, which is in charge of transferring the required images from a central repository to the host machine and starting the VM. Each of the instantiated VMs is identified by its uniquely generated MAC address, controlled by the central DEVA manager.

The next step consists of applying the required network configuration at each host in order to perform traffic monitoring, network isolation, bandwidth management and intelligent routing between *ensemble* members. To accomplish this, each physical machine needs to know the details about the hosted VM's network configuration. We use *OpenNebula's* hook functionality for this. Hooks are small scripts that can be configured to run at certain points of *OpenNebula's* request lifecycle, such as when a VM is started, stopped or removed. They are executed either in the head node or in the hosting machine. Since we need a process to manage network settings accordingly to the original user's request at each node, we have developed a daemon (called a *DEVA Agent*) that runs at each host that can perform these actions.

4 DEVA Agents

A *DEVA agent* runs as a background process that listens for new requests, runs some pre-defined commands on the host machines, monitors network and VM behavior, and creates VPN (Virtual Private Network) tunnels between sites. The DEVA Manager notifies the appropriate agent of a new VM member using a hook, which sends a command through the agent's specified port. There is one designated agent per site that creates VPN tunnels, called a *site gateway*. All agents in a site know the address of the site gateway and can send requests to create new tunnels. There are three supported commands: ADD, DEL and TUN.

When an agent receives an ADD request, it looks for the virtual network interface of the specified VM and queries the DEVA manager to retrieve global information about the DEVA, such as which ensemble members this VM is connected to and what is the requested network QoS. After this information is returned, the agent can perform the appropriate actions. In the case that one or more of the ensemble members share a virtual layer 2 connection with VMs assigned to machines in different domains, the agent on the host machine of the newly assigned VM issues an additional call to its corresponding site gateway asking for a tunnel to be created between the VMs using the TUN command. Finally, the DEL command is analogous to ADD, which basically removes a VM from a host.

Different DEVAs are completely isolated from one another at network layer 2. When the DEVA agent receives a request to add a new VM, it creates *ebtables* [1] rules to block all traffic except those frames originating from or directed to other ensemble members in the same logical network.

When an ADD request is received by the agent, it retrieves a list of the VM's neighbors from the DEVA manager, and then it adds two rules for each of them: one to allow outgoing traffic from the VM's unique MAC address to the neighbor's one, and the reciprocal rule to accept packets originating from the neighbor's address with the local VM as the destination. ARP requests are special since they do not have a unique target, and therefore an additional rule is added to allow this kind of packets from and to the network.

In the original DEVA request, each link in the ensemble may be annotated with a desired latency and bandwidth. The *DEVA Agent* consults the manager to retrieve the bandwidth constraints between the local VM and each of its neighbors. For each pair, the agent creates a queuing class discipline in the kernel's traffic control module using the *tc* command. Next, packets are marked in the kernel and filtered to use the appropriate class. We use the Hierarchical Tocken Bucket (HTB) for its versatility and good performance.

Site gateways are in charge of routing frames that are not targetted to the host's Local Area Network. Since DEVAs may be distributed among different networks that are not reachable at layer 2, agents must encapsulate the frames that are directed to another network and send them. In our architecture, each site must have a VPN server and allow client connections from other sites. Also, each site that needs to tunnel traffic has to have at least one site gateway. When the site gateway agent starts, it runs a VPN client for each of the remote sites and connects to them. The client is configured to create a special *tap* device, in such a way that all traffic send through a tap will be tunneled to a different site. Then agents that instantiate new VMs retrieve the neighboring VM hosts, and for each host that is located outside of the VM's domain, they send a TUN request to the local site gateway so that packets originating from the VM directed to the destination neighbor are tunneled through the appropriate channel.

5 Experimental Results

Here we perform different experiments to demonstrate the use of DEVAs as a viable cloud resource management approach. We have run several measurements to quantify the performance and scalability of our design and the prototype implementation. We evaluate the following hypothesis experimentally:

1. The overhead of the DEVA agents is small when executing High-Performance Computing applications.
2. Traffic is effectively isolated between different deployed DEVAs.
3. User requested QoS is fulfilled through the execution of applications.

[1] http://ebtables.sourceforge.net

For all experiments, we employ our Magellan cluster, which is composed of 8 nodes, each of them with a Pentium 4 CPU at 3 GHz and Hyper Threading and 1 GB of memory. The nodes are connected using a 1 Gbps ethernet link and a Gigabit switch. The head node of the cluster has two network interfaces, one is connected to the Internet and the other to the private network where the other 7 nodes are also connected. Each node runs CentOS 5.3 and the Rocks cluster administration software version 5.2 with the Xen roll, which provides Xen 3.0.3.

5.1 Overhead Measurement

In the first set of experiments, we quantify the overhead imposed by the agents at each host. The agents control network isolation and bandwidth usage. Each agent manages incoming and outgoing traffic at the virtual network interface of each VM by filtering packets based on the source and destination addresses. In our prototype implementation, since each VM has a unique MAC address, the agent can control traffic for each pair of ensemble members, which implies that there are two ebtable rules for each pair to manage incoming and outgoing traffic.

For this experiment, we execute the Weather Research and Forecast (WRF) package [8], a simulation software used for atmospheric research, using different setups. WRF uses the Message Passing Interface (MPI) middleware [2] for communication between processes, and can be executed with different number of processes. We have previously studied the behavior of this program [7], and demonstrated that its communication model is highly sensitive to network delays. In particular, link latencies of more than 1 ms result in slower executions when adding more nodes to the computation, making it impossible to perform blind scaling across Internet.

In this macro-benchmark, we perform executions of WRF for 1, 2, 4 and 8 nodes using the physical cluster running Xen's Domain 0, a set of OpenNebula instantiated VMs, and a DEVA instantiated through our manager. Each process is assigned to a different node. For all runs that involve VMs, we use a base CentOS 5.3 Xen image with 512 MB of memory and a 2.6.18 Linux kernel. We installed the required software to compile and run WRF version 2.2.

Figure 2 shows the execution time for the three considered cases, namely the physical cluster, the VMs instantiated through OpenNebula and the DEVA VMs. The slowdown between the physical execution and OpenNebula's execution is entirely produced by the Xen virtualization overhead. It can be observed that when the number of nodes grows, so does the amount of communication among them. This factor is specially significant, since I/O virtualization is known to have a huge performance hit. The use of more sophisticated network drivers and newer hardware would certainly alleviate the slowdown, although this is outside the scope of the current paper. The overhead produced by the filtering and monitoring by the agents is minimal in this case, averaging to less than 1 percent.

[2] http://www.mpi-forum.org

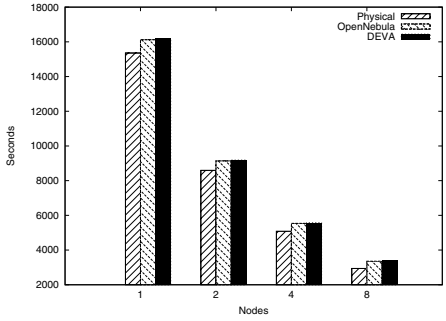


Fig. 2. WRF Execution time

The next experiment performs a micro-benchmark of the same parameters as the previous one. We investigate the performance impact of adding the necessary filtering rules to provide isolation between VMs in different networks. For this experiment, we instantiate two VMs connected to a virtual switch, but we add rules as if there were a greater number of VMs in the DEVA. Since rules are directly dependent on the number of connections a VM has to other VMs, this allows us to measure the slowdown produced by those rules in relation to the size of the broadcast group.

Figure 3 shows bandwidth, round trip time and number of generated rules in relation to the VM connections. We use *netperf* to measure the total available bandwidth between two physical machines, two VMs instantiated with 0, 10 and 100 connections instantiated through the DEVA manager. Note that the number of rules depends on how many ensemble members are in the same broadcast group (*i.e.*, share the same virtual link) and not the size of the DEVA.

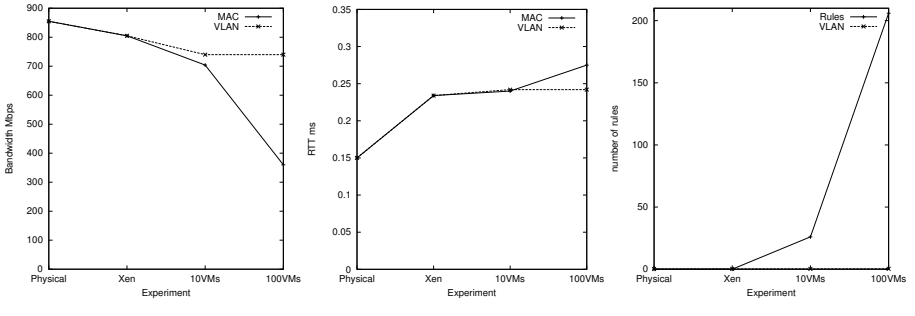


Fig. 3. Overhead introduced by filtering rules

From the results, we can see that the impact of adding more VMs is increasingly high, specially in terms of total bandwidth. Although 10 connections still has a tolerable overhead of 6.2% over the zero-connection case, adding more VMs results in a great overhead due to the number of filtering rules that need to be added.

The approach of assigning unique MAC addresses to control network isolation is therefore not scalable for large groups of ensemble members. To address this problem, we employ VLANs to identify which virtual network each packet belongs to. VLANs are defined by adding 4 additional bytes to each datagram at the source interface. To implement this method, we modified the DEVA manager to assign a unique id to each broadcast group. The downside of this solution is that VLANs need to be supported by the physical switch, while the individual MAC filtering is network agnostic.

5.2 Isolation and QoS Conservation

Isolation is also demonstrated in the provided QoS. Traffic from one DEVA should not impact bandwidth and latency allocated by the manager to another one. The only exception to this is when best effort links are requested, in which case no guarantee is made by the system.

In the next experiment, we test the effects of multiple DEVAs running in our Magellan cluster. First we execute two applications that share the same physical resources, and we constrain the allocated bandwidth to ensure each of them has the requested QoS. We compare it to the same case when allocating the VMs individually through the IaaS manager, *OpenNebula*.

For this experiment, we choose two applications with different behaviors: first, we create HTTP traffic between two VMs, one of them with the Apache Web server version 2.2.3, and the other with the *apache benchmark* tool. Next, we simulate network traffic by transferring files between another pair of VMs. The two clients share one host, and the two servers are placed in another host.

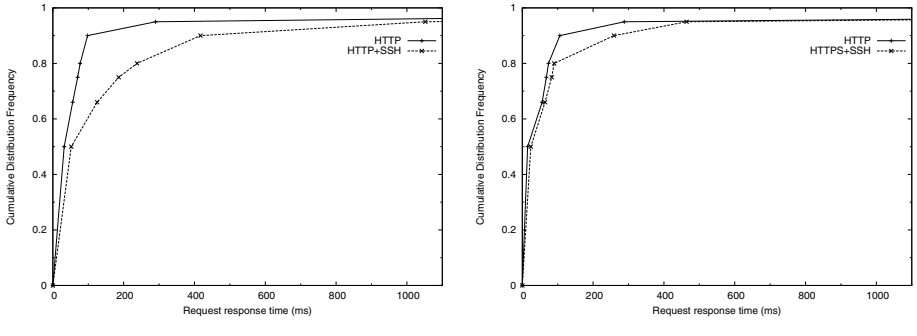


Fig. 4. HTTP traffic between two VMs with and without network contention. Right figure uses DEVA traffic control mechanisms to fulfill requested QoS.

Figure 4 shows the cumulative distribution function of the Web server response time. The left figure shows unmanaged network traffic: while half of the requests take similar time, the waiting time for the other half increases up to four times with the additional network load. In the right figure, two DEVAs are requested with different requirements: the pair of VMs with the HTTP traffic has a 600 Mbps virtual link, while the other two VMs have a virtual link of 40Mbps. It

can be seen that the response time when additional traffic is generated remains similar to the case in which only the HTTP requests take place. As the figure shows, 80% of requests are completed in the same time, while the top 20% experience delays up to three times.

5.3 Use of Heterogeneous DEVAs and Resources

Finally, we run an experiment to demonstrate how VMs in a DEVA can be placed across different administrative domains while QoS is enforced. In this case, we provision a DEVA with three VMs: two of them are connected among them with a 160 Mbps virtual link, and the third is connected to the first two with a 16 Mbps link. Next, we reduce the available nodes in the Magellan cluster to two and add a physical machine from another cluster, Mind, located in a different campus at FIU. The *assign* algorithm maps two of the VMs to Magellan and the third one to Mind, and creates a VPN tunnel among them to provide a virtual network. We calculate link bandwidth among VMs by using *netperf* and the round trip time by averaging 50 pings between the machines. Figure 5 shows how the virtual ensemble maintains the requested QoS.

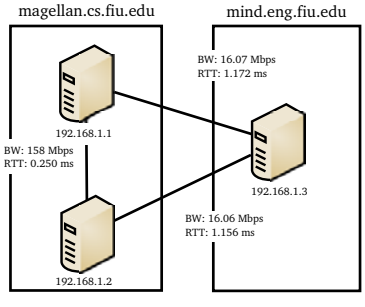


Fig. 5. Intersite deployment of a DEVA

6 Related Work

Amazon EC2 [1] is perhaps the most prominent example of IaaS public cloud. Users can request any number of virtual machines to be instantiated in the shared infrastructure. VM capabilities are defined by the requested instance type, and price is set accordingly to the time and characteristics of the used VMs. One of the main shortcomings of EC2 is the lack of QoS assurances for network traffic: while processor, memory and disk capabilities are well defined, users can't make assumptions about the network.

Eucalyptus [11], OpenNebula [14] and Nimbus [6] are IaaS cloud implementations that have some similarities with EC2. However, these solutions do not support composite groups of VMs with a defined network QoS in the requests. Another difference is in the deployment of VMs accross different domains. *OpenNebula* supports interoperation by implementing different protocols such as the

Open Cloud Computing Interface (OCCI) or by extending the local resources into public clouds. In our case, we the DEVA manager decides when to create a tunnel to connect VMs in different sites based on the requested QoS.

Another type of solutions focus on the network virtualization aspect, rather than in the resource management and providing an interface for users to manage composite groups of virtual resources. VIOLIN [13], VNET [15], VINE [16] or IPOP [3] are examples of such systems.

Also, DEVAs have similitudes with virtual clusters such as [9] or [10]. Differently than in our work, these solutions focus on instantiating the required virtual resources and providing the appropriate software and network configuration.

Finally, our work has points in common with network testbeds, where many of the problems of provisioning execution environments to replicate network topologies have to be solved. Emulab [4] allows users to create network experiments over shared resources by requesting a configuration of virtual hosts and connections. The main difference from our work and Emulab is that the latter is principally targeted for repeatable network experiments, while our system is designed to host virtual environments to run possibly long lasting applications. Also, since our primary goal is not to replicate the user's network characteristics, we can make some optimizations in the requested topologies. In GENI, [2] describes a similar approach in which ORCA [5] is extended to support additional networking infrastructure to create multi-site VM deployments via VLAN tags. Our work is more focused in the placement aspect and QoS fulfillment.

7 Conclusions and Future Work

We have described an approach to instantiate groups of Virtual Machines in the cloud while fulfilling their composite QoS requirements. Our experiments indicate that this implementation is viable and can be used to execute different workloads with specific network and processing requirements. As future work, we plan to further investigate the dynamic behavior of DEVAs, and how to respond to varying traffic and resource utilization. The DEVA manager can perform actions to further control QoS of the sytem by migrating VMs among resources, or adjusting the allocations according to the global state. Finally, we want to explore different placement policies among sites to accomplish site-specific and global goals such as lower power utilization or higher throughput.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. OISE-0730065 and HRD-083309.

References

1. Amazon elastic compute cloud, http://aws.amazon.com/ec2
2. Baldine, I., Xin, Y., Mandal, A., Renci, C.H., Chase, U.-C.J., Marupadi, V., Yumerefendi, A., Irwin, D.: Networked cloud orchestration: A geni perspective. In: 2010 IEEE GLOBECOM Workshops (GC Wkshps), pp. 573–578 (2010)

3. Ganguly, A., Agrawal, A., Boykin, P.O., Figueiredo, R.: Ip over p2p: Enabling self-configuring virtual ip networks for grid computing. In: In Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS 2006), pp. 1–10 (2006)
4. Hibler, M., Ricci, R., Stoller, L., Duerig, J., Guruprasad, S., Stack, T., Webb, K., Lepreau, J.: Large-scale virtualization in the emulab network testbed. In: USENIX 2008 Annual Technical Conference on Annual Technical Conference, pp. 113–128. USENIX Association, Berkeley (2008)
5. Irwin, D., Chase, J., Grit, L., Yumerefendi, A., Becker, D., Yocum, K.G.: Sharing networked resources with brokered leases. In: Proceedings of the Annual Conference on USENIX 2006 Annual Technical Conference, p. 18. USENIX Association, Berkeley (2006)
6. Keahey, K., Foster, I., Freeman, T., Zhang, X., Galron, D.: Virtual workspaces in the grid. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 421–431. Springer, Heidelberg (2005)
7. Martinez, J.C., Wang, L., Zhao, M., Sadjadi, S.M.: Experimental study of large-scale computing on virtualized resources. In: Proceedings of the 3rd International Workshop on Virtualization Technologies in Distributed Computing (VTDC 2009) of the IEEE/ACM 6th International Conference on Autonomic Computing and Communications (ICAC 2009), Barcelona, Spain, pp. 35–41 (June 2009)
8. Michalakes, J., Dudhia, J., Gill, D., Henderson, T., Klemp, J., Skamarock, W., Wang, W.: Reseach and Forecast Model: Software Architecture and Performance. In: 11th ECMWF Workshop on the Use of High Performance Computing In Meteorology, Reading, UK, pp. 156–168 (October 2004)
9. Murphy, M.A., Kagey, B., Fenn, M., Goasguen, S.: Dynamic provisioning of virtual organization clusters. In: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID 2009, pp. 364–371. IEEE Computer Society, Washington, DC, USA (2009)
10. Nishimura, H., Maruyama, N., Matsuoka, S.: Virtual clusters on the fly - fast, scalable, and flexible installation. In: IEEE International Symposium on Cluster Computing and the Grid, pp. 549–556 (2007)
11. Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D.: The eucalyptus open-source cloud-computing system. In: IEEE International Symposium on Cluster Computing and the Grid, pp. 124–131 (2009)
12. Ricci, R., Alfeld, C., Lepreau, J.: A solver for the network testbed mapping problem. SIGCOMM Comput. Commun. Rev. 33, 65–81 (2003)
13. Ruth, P., Jiang, X., Xu, D., Goasguen, S.: Virtual distributed environments in a shared infrastructure. Computer 38, 63–69 (2005)
14. Sotomayor, B., Montero, R.S., Llorente, I.M., Foster, I.: Virtual infrastructure management in private and hybrid clouds. IEEE Internet Computing 13, 14–22 (2009)
15. Sundararaj, A.I., Dinda, P.A.: Towards virtual networks for virtual machine grid computing. In: Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3, p. 14. USENIX Association, Berkeley (2004)
16. Tsugawa, M., Fortes, J.A.B.: A virtual network (vine) architecture for grid computing. In: International Parallel and Distributed Processing Symposium, p. 123 (2006)

Benchmarking Grid Information Systems

Laurence Field[1] and Rizos Sakellariou[2]

[1] CERN, Geneva, Switzerland
Laurence.Field@cern.ch
[2] The University of Manchester, Manchester, UK
rizos@cs.man.ac.uk

Abstract. Grid information systems play a central role in today's production Grid infrastructures, enabling the discovery of a range of information about the Grid services that exist in an infrastructure. As the number of services within these infrastructures continues to grow, it must be understood whether the current implementations are able to scale to meet the future requirements. Existing approaches for evaluating Grid information systems mainly focus on performance metrics and do not consider the quality of the information itself. This paper proposes a comprehensive benchmarking methodology for the evaluation of Grid information systems which includes a metric to assess the quality of the information returned. Using this methodology, two commonly used Grid information system implementations, Metadata Directory Service (MDS) and the Berkeley Database Information Index (BDII), are evaluated using data obtained from the Enabling Grids for E-SciencE (EGEE) production Grid.

1 Introduction

Grid information systems enable users, applications and services to discover which Grid services exist in a Grid infrastructure along with further information about their structure and state [2,7]. Information describing each Grid service originates from the Grid service itself and hence the Grid service, in terms of Grid computing, is the primary information source. Grid information systems provide an interface that can be used to resolve queries over these information sources, which can be numerous and widely distributed geographically. From the perspective of an information consumer, the information system can be represented by an abstract interface to which queries can be sent and a query response is received. How the Grid information system resolves that query, taking into consideration all the information sources, is an implementation detail for the specific system. What matters from the users perspective is the overall quality of the service provided by the interface.

As Grid infrastructures grow, more Grid services and hence information sources will exist in the infrastructure, which will increase the total volume of information. In addition, a larger infrastructure also experiences greater utilization and the amount of queries made to the information system will also increase. Ensuring that Grid information systems will meet the future requirements in

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 479–490, 2011.

terms of information volume and query load, is one of the main challenges facing today's production Grid infrastructures.

In recent years, a number of Grid information systems have been developed. Despite their differences [4], there is a great deal of commonality between their core functionality and deployment scenarios. Grid infrastructure managers need to compare these different Grid information systems implementations in order to choose which solution best meets the needs of their infrastructure. This is a decision that should not be taken lightly. Large scale distributed infrastructures can exist for many years, during which time it can be too difficult to migrate between different implementations. A benchmarking methodology for evaluating and comparing these different implementations would provide valuable insight for Grid infrastructure managers. Such a methodology should evaluate the Grid information system in order to understand its scalability limitations. The results of this evaluation could then be compared with the estimated future requirements from growth projections of that Grid infrastructure.

Existing literature evaluating Grid information systems [3,9,11,12,13,14,15] focuses mainly on performance metrics such as response time (that is, how quickly a user query is answered) or throughput (that is, how many concurrent queries can be handled over a period of time). However, these metrics alone cannot be used to assess the overall performance of Grid information systems as they neglect some important aspects such as the nature of the queries (different queries have different requirements) as well as the quality of the answers delivered by the Grid information system. In fact, a recent study [5], whose results largely motivated the present paper, demonstrated that how up-to-date the information delivered by a Grid information system is may vary significantly across different types of Grid information; this affects the quality of the answers obtained and must be considered in any evaluation study.

Taking into account the above, this paper presents a comprehensive benchmarking methodology to evaluate Grid information systems, which is based on the assessment of three different components: (i) a set of commonly executed queries; (ii) query response time; and (iii) quality of information returned. To the best of our knowledge, this is the first time that such a comprehensive benchmarking methodology for Grid information systems, which incorporates the use of a *quality metric*, is proposed. The proposed benchmarking methodology is then used to evaluate two different Grid information system implementations; the Metacomputing Directory Service (MDS) [2] from the Globus project and the Berkeley Database Information Index (BDII) [6], which is a simplified implementation of Metadata Directory Service (MDS). The evaluation takes place on the Enabling Grids for E-science (EGEE) [8] infrastructure, the largest multidisciplinary Grid infrastructure in the world.

The paper is outlined as follows. Section 2 gives a critical evaluation of previous, related work. Section 3 describes in detail the components of the benchmarking methodology and how the metrics included can be measured. The evaluation

of the two Grid information systems using the benchmarking methodology presented is given in Section 4. Finally, some concluding remarks can be found in Section 5.

2 Related Work

This section reviews work that has been done to evaluate Grid information systems. As already mentioned, and will be observed next, most of this work does not assess the quality of the information obtained.

In a frequently cited study [13,14,15], the functional components of three systems were grouped into similar types and the performance of each component type was evaluated against those of their counterparts. The focus was to evaluate the effect of a large number of users and information sources for each implementation. In addition to the average query response time, a throughput metric (average number of queries per second) was also used. Both of these metrics were measured for different numbers of concurrent queries and information sources. However, in this study little attention was given to the data itself. A single query (return all information for a specific information source) was used and it was assumed that the response was always *correct* (i.e., up-to-date, or current, or fresh). In addition, the information volume used was small (10Kb) compared with today's infrastructures.

A study investigating the scalability and analyzing the performance of an information system [3] addressed some of these deficiencies by using real data from a large-scale Grid infrastructure. It also investigated how the performance is affected by the size of the query response, however, again only the average query response time was used as a metric and again it was assumed that all responses were correct.

A Grid information benchmark has previously been proposed [9], based on a set of queries and scenarios, which were used to compare the access language and platform capabilities for three different database platforms serving Grid information. Although in total 13 different queries were used, the specific case for using this set was not made. The databases were populated with synthetic but realistic information about Grid services, which conformed to the GLUE information model [1]. Short synthetic workloads (scenarios) were used to measure the query response time of concurrent query requests to the repository. The major difference with our work is that we include a quality metric and actual results from a production Grid.

Of particular interest is the study in [12], which, in addition to query response time, proposed two further metrics, network overhead and information freshness, for evaluating Grid information systems. Network overhead is defined as the number of bytes that a crawler downloads to update the information in the system. The crawler in this context is a specific implementation detail of this system and although important, it is not a concern for the user from a quality of service perspective. Information freshness, on the other hand is of great concern to the user. Information of the Grid information system is considered fresh (or

correct), at a given point in time, if it is synchronized with its real-world equivalent value at the information source (that is, both values are the same). This concept of freshness is useful as a quality metric for the information returned, and it is missing from other studies. However, the proposed calculation of freshness may not be feasible in a real system as it requires a continuous comparison of all values.

All of the above studies demonstrate the need to evaluate Grid information systems. They show that the query response time is a key metric and should be a core part of any proposed benchmarking method. The studies also show that the query response time can be affected by four main factors; the total information volume, the type of query (both due to the complexity and size of the response), the number of concurrent queries (query loading) and the implementation (hardware, software and internal architecture).

Extending the current state-of-art further, our benchmarking methodology takes into account the above, also including a metric to measure the quality of the query response.

3 Methodology

The benchmarking methodology will be based on the GLUE 1.3 information model [1] as this information model is used by the majority of today's production Grid infrastructures [10]. The information model used is a key part of any benchmarking study as it defines a number of constants of relevance to the benchmarking method. These include the object types, and hence expected frequency of change for those object types, the composition of the queries and the size of the query response. If an alternative information model is used, these constants will have to be measured for that information model. Due to the different constants being used, benchmarking results cannot be compared for different information models.

The volume of information and the use case for the benchmarking methodology will be taken from a large-scale production Grid infrastructure, the infrastructure from the Enabling Grids for E-science (EGEE) project. This is a fair choice to use as a reference for the benchmarking methodology as EGEE operates the largest multi-disciplinary Grid infrastructure in the world. In addition, it also makes use of the GLUE 1.3 information model. As of November 2010, the EGEE Grid information system contained information representing 4102 Grid services deployed between 375 sites. This information represents an information size of 102MB in the LDIF format, which corresponds to an average of 272KB per site or 25KB per service.

Set of queries: In order to consider queries commonly made to the EGEE Grid information system, we used the information provided in [3] to choose a top-ten of queries. The type of each query and its frequency as a percentage of the total number of queries made to the system are shown in Table 1.

We further analyzed the characteristics of each query in terms of the query response size and the object type that they use. As different objects experience

Table 1. The top ten queries made to the EGEE infrastructure as a percentage of the total number of queries

	Query	%
Q1	Find the Closest Computing Service to a Storage Service	19.6
Q2	Find the VO's Storage Area for a Storage Service	17.7
Q3	Find all Storage Services	16.3
Q4	Find a Storage Service	15.5
Q5	Find the Closest Storage Service to a Computing Service	7.8
Q6	Find all Services for a VO	6.8
Q7	Find all Computing Services for a VO	2.1
Q8	Find all Storage Areas for a VO	2.1
Q9	Find all Sub Clusters	1.5
Q10	Find the VO's Computing Share for a Computing Service	1.4

a different frequency of change [5], the object type being used by each query may indicate how up-to-date the results of this query are expected to be. These characteristics are shown in Table 2.

A few observations are interesting to note. All queries have a similar level of complexity: they either return all the objects of a particular type or filter by only one predicate. As such, the two key differences between the ten queries are the size of the query response and the object type that is queried.

The size of the query response allows a classification of the queries into three groups: (i) queries that return information about one service (denoted by 'small' in the table); (ii) queries that return all information for an object type (denoted by 'large' in the table); and (iii) queries that filter information for one object type (denoted by 'varying' in the table). For queries of the latter group the actual size of the response varies depending on the filter being used. To ensure that the filter used for a particular query provides a good representation of the query response size for that query, all possible values for that filter were evaluated and

Table 2. Characteristics of each of the top ten queries

Query	Size(bytes)	Size(class.)	Object	Frequency
Q1	2858	small	CESEBind	Low
Q2	5436	small	SA	High
Q3	315225	large	SE	High
Q4	642	small	SE	High
Q5	1875	small	CESEBind	Low
Q6	49801	varying	Service	High
Q7	16607	varying	CE	High
Q8	12348	varying	SA	High
Q9	8959015	large	SubCluster	High
Q10	6009	varying	VOView	High

the response size in each case was calculated. The median size was then chosen (and is shown in the table) to avoid the average being skewed by the few large query responses. Regarding the object type used by each query, we classify these objects according to the classification in [5] into high-frequency objects (those that experience over 1% changes per day) and low-frequency objects (those that experience less than 1% changes per day).

Query response time: Each query is executed 50 times against the deployed instance of the Grid information system under evaluation. Each time, the query response time (QRT) is measured from the start of the client connection to when all the data for the query response has been received by the client. The average query response time can be calculated using Equation 1, where $n = 50$.

$$QRT_{average} = \frac{\sum QRT}{n} \tag{1}$$

Quality of information: For each query, the (α, β)-currency will be used to assess the quality of information returned. This metric, first investigated in the context of the Grid in [5], can be used to calculate a probability, α, that a selected object value stored by the Grid information system is current with respect to a grace period β. The probability α can be calculated by using Equation 2, where β is the age of the information and λ is a constant for each object type.

$$\alpha = e^{-\lambda\beta} \tag{2}$$

The method used to measure β, the age of the information, depends on the specific Grid information system implementation. If a timestamp is available for when the information was created, this can be compared to the time when the query was executed. Alternatively, the time between when the information was created and when the query was executed will need to be measured. The latter will be the method that will be used later in our benchmarking comparison of MDS and BDII. We assume that the information is fresh and the query is executed as soon as the cache has been updated. Hence, we use the time it takes to update the cache as the time between when the information was created and when the query was executed; this provides a value for β.

The constant λ can be calculated using Equation 3, where f is the frequency of change for a specific object type and N is the number of objects of that type,

$$\lambda = \frac{f}{N}. \tag{3}$$

Further information regarding the frequencies of change for the GLUE 1.3 object types (as well as the number of objects for each type) can be found in [5].

4 Benchmarking MDS and BDII

This section benchmarks two Grid information system implementations, MDS and BDII, using the benchmarking methodology described in the previous section.

4.1 Background

The MDS query interface hides the details [2,7] of the MDS architecture and implementation from the client. This interface is provided in MDS by the Grid Index Information Service (GIIS) using the *Grid Resource Information Protocol* (GRIP). The GIIS maintains a dynamic registry for information source locations populated from registration notifications received via the *Grid Resource Registration Protocol* (GRRP). The GIIS parses each incoming GRIP request and then forwards this request to one or more information sources found in the registry depending on the type of information requested. To improve performance [15], each response may be cached for a configurable period of time by specifying the cache time-to-live (TTL) per information source as part of the registration. In the MDS implementation, the standard Lightweight Directory Access Protocol (LDAP) has been adopted for both the GRRP and GRIP, where it was used to define the data model, query language and wire protocol, and the GIIS is implemented as a special purpose back-end for an OpenLDAP server.

The Berkeley Database Information Index (BDII) is a simplified implementation of the MDS GIIS. The main simplification is the absence of the GRRP protocol with the dynamic registry functionality being replaced by a static registry where new information source locations are added manually by a system administrator. The other simplification is that the BDII maintains its information cache using a parallel process rather than caching query results. This serves to decouple the cache update process from incoming queries; therefore, the information sources are isolated from the queries and this results in a predictable behavior of the system.

In both implementations, the information is cached and queries are evaluated using this cache. The implementations diverge in the method employed to refresh that cache. Further discussion on the impact of the cache update method can be found in Section 4.5.

4.2 Experiment Setup

The MDS and BDII software was installed and configured on a physical machine. The machine used had a 1.80GHz single processor Intel Pentium and 1Gb of memory. The MDS version used was 2.4, which is based on OpenLDAP 2.0 and the BDII version used was 5.1.9, which is based on OpenLDAP 2.3. A second machine was used to run client queries. This machine had four 2.66GHz Intel Xeon CPUs and 8Gb of memory. The two machines were connected through a 100 Mbps LAN.

The caches were populated with data representing the information from the EGEE production infrastructure and the cache update processes were disabled.

4.3 Query Response Time

Each query was executed in a single thread that iterated 50 times over the same query. The average response time for each query type made to the MDS GIIS and the BDII can be seen in Figure 1.

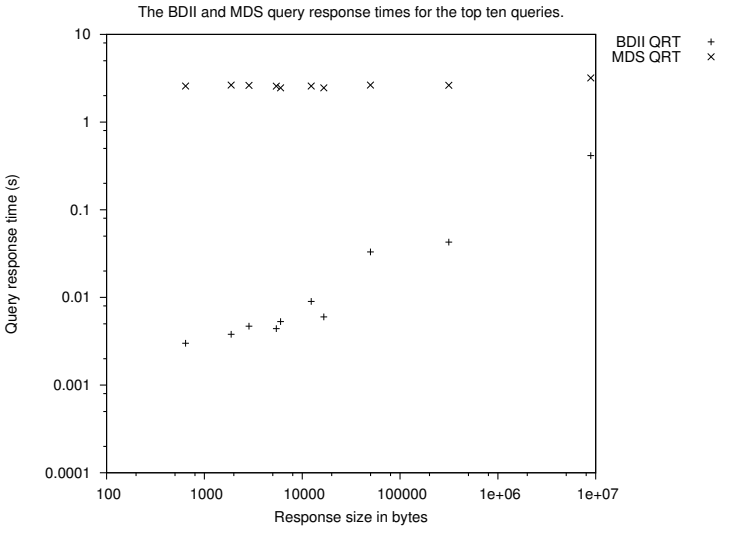


Fig. 1. Average query response time for the top ten queries made to MDS and BDII

These results show that for MDS GIIS the query response time is almost constant, approximately 2.6s, for all query result sizes, with the exception of the 8.9Mb result size, which causes the query response time to increase to 3.2s. This suggests that the connection overhead and query execution time are the main factors that affect the query response time for the MDS GIIS. The query result size, and hence transfer overhead, only has an impact for large query response sizes. By contrast, the BDII implementation has much lower query response times, which seem to be affected by the query response size. This suggests that the connection overhead and query execution time are much lower for the BDII implementation and that the transfer overhead is the main factor that affects the query response time.

4.4 Quality of Information

As already mentioned in Section 3, for the value of β we can use the time it takes to update the cache. However, we note that the BDII and MDS GIIS differ in the way the cache is updated. In the MDS GIIS, the cache is updated synchronously with the query, whereas with the BDII this update is done asynchronously. This means that for the MDS GIIS, when the cache is stale, the next query will trigger the update process and subsequent queries will block until this process has finished. In the BDII a parallel process updates the cache while the cache is still responding to queries. In either case, the time to update the cache is the minimal value for β in Equation 2, with the maximum value being defined by the TTL of the cache in the MDS GIIS or the delay between update cycles in the BDII. The best-case scenario, which will be measured here, is where these values are zero.

Table 3. The result of calculating (α, β)-currency for the MDS GIIS and BDII

Query	Object	λ	α_{MDS}	α_{BDII}
Q1	CESEBind	1.14×10^{-08}	1.000	1.000
Q2	SA	5.11×10^{-04}	0.279	0.902
Q3	SE	4.85×10^{-04}	0.298	0.907
Q4	SE	4.85×10^{-04}	0.298	0.907
Q5	CESEBind	1.14×10^{-08}	1.000	1.000
Q6	Service	4.86×10^{-04}	0.298	0.907
Q7	CE	1.91×10^{-03}	0.008	0.681
Q8	SA	5.11×10^{-04}	0.279	0.902
Q9	SubCluster	2.28×10^{-06}	0.994	1.000
Q10	VOView	1.48×10^{-03}	0.025	0.743

To measure the query response time for the MDS GIIS the cache TTL was set to zero so that every query made to the cache would trigger the update process. A simple query that returned zero results was used to trigger the update. The query response time for 10 successive queries was measured and the average query response time was calculated. The average query response time when the cache is valid was also calculated. The difference between the query response time when the cache is valid and invalid gives the cache update time. The cache update time for the MDS GIIS was 2496s with σ=13.6s.

For the BDII, the update process is instrumented and the update time is recorded in the log file of the update process. Thus, the average time for 10 updates was calculated. The cache update time for the BDII was 201s with σ=14.9s.

Using the cache update times for β (that is, β_{MDS}=2496 and β_{BDII}=201) and the frequencies of change from [5] to calculate λ, the value of α was calculated using Equation 2 for each query. The results are shown in Table 3. The values α_{MDS} and α_{BDII} correspond to the probability that the information is current for the MDS GIIS and BDII, respectively.

4.5 Discussion

Comparing the query response times of the BDII and MDS GIIS implementations for a single query thread, the result depends on the query used. For Q4, the query with the smallest response size (642 bytes), the query response times are 0.003s and 2.58s for the BDII and the MDS GIIS, respectively. For Q9, the query with the largest response size (8.5Mbytes), the query response times are 0.41s and 3.19s, respectively. The BDII implementation delivers much better performance than the MDS GIIS for queries with a small response size, however, this advantage is reduced for larger response sizes.

As stated previously, looking at the query response time is only one aspect of the overall quality of service. The quality of the information returned also needs to be considered. In the case of querying a low frequency object type, for example Q1 (CESEBind), it is almost certain that all information is current.

However, for the case of querying a high frequency object type, Q7 (CE), the probability that the information is current is 0.008 for the MDS GIIS, while, for the BDII, the probability that the information is current is 0.681. We note that these figures represent the best-case scenario as they assume the original information is current. The value for α will degrade further during the period after the caches have been updated. The length of this period can be defined by a configuration parameter in both instances. Knowledge of (α, β)-currency can be used to set an optimal value for this parameter, however, the limit is reached when the value is set to zero.

Comparing the query response times will show which implementation gives a faster response, however, it does not indicate whether the query response time is acceptable. In both implementations, the query response time may be acceptable for a particular deployment scenario. Including the (α, β)-currency metric gives additional insight by providing a measurement of the quality of the information returned. What values are acceptable will be dependent on the particular deployment scenario, however, we can make some general statements about the result. Any value less than 1 would result in the client having to include a probabilistic approach when using this information. Any value less than 0.5 means that the information returned is more likely to be incorrect than it is likely to be correct. With this interpretation, the MDS GIIS is returning more incorrect information when querying high frequency object types than correct information and although the BDII is returning more correct information than incorrect information when querying high frequency object types, a probabilistic approach to using that information would be required.

We also note that when the cache in the MDS GIIS is stale, the next query will trigger the cache to be updated and subsequent queries will be blocked until this process has finished. Unlike the MDS GIIS, the BDII can be queried while the cache is being refreshed. To see how the query load affects the cache update time and hence β, the query Q3 was used to produce a query load using a single thread. This query load caused the cache update time to increase to 1100s with σ=145 while the average query response time for the query increased from 0.0427s with σ=0.00114 to 0.0472s with σ=0.0339. Although for low frequency object types, it is still almost certain that the information in the cache is current (which implies better quality), the value of α for high frequency object types was reduced from 0.907 to 0.58 (for Q3). Also, the values for Q7 and Q10 were reduced from 0.681 and 0.743 to 0.12 and 0.2, respectively.

As a final remark, we note that a potentially important aspect for Grid information system benchmarking, which has not been a central issue in this paper, but in other similar studies, is how the query response time varies with the number of parallel queries. In some preliminary results, the query response time for varying numbers of concurrent queries executed against the MDS GIIS and BDII was measured. The results showed that for the MDS GIIS doubling the number of query threads essentially doubles the query response time. However, with only 4 query threads, a few queries time out, that is they exceed the 20s threshold set

in the test framework as a protection against queries hanging indefinitely. With only 6 query threads, up to 35% of the queries time out and with 8 query threads up to 52% of the queries time out. For the BDII up to 100 parallel query threads were used, as provisional tests showed that BDII could handle much more than 8 query threads. Even with 100 query threads, many of the queries return a result in less than 1s. The exception is for queries that return large query responses, where timeouts were observed with 75 and 100 query threads. Although briefly mentioned here, the issue of concurrent query execution is highly relevant for today's production Grid infrastructures; however, lack of space does not allow us to cover it in-depth and has been covered in other related studies.

5 Conclusion

This paper investigated the benchmarking of Grid information systems and proposed a comprehensive benchmarking methodology, suggesting that the addition of (α, β)-currency as a metric for describing the quality of a Grid Information System would give additional insight. Two Grid information system implementations that have had production exposure, MDS and the BDII, were then benchmarked using this methodology. The results showed that in terms of response time, the BDII implementation gives better performance than the MDS GIIS for queries with a small response size, 0.003s and 2.58s for the BDII and the MDS GIIS respectively for Q4 (642 bytes). However, this advantage is reduced for larger response sizes, giving 0.41s and 3.19s for the BDII and the MDS GIIS, respectively, for Q9 (8.5Mbytes).

While this comparison showed that the query response time for the BDII was better (faster) than for the MDS GIIS, a statement could not be made on whether or not this result was acceptable for deployment in a production Grid infrastructure. Including the (α, β)-currency metric gave additional insight by providing a quality measurement for the information returned. For low frequency object types, it is almost certain that the information in the cache is current. However, for the high frequency object type in Q7 (CE), the probability that the information is current is 0.008 and 0.681 for the MDS GIIS and the BDII, respectively. This suggests that for Q7, the MDS GIIS would mainly be returning incorrect values and that a probabilistic approach would be required for the information returned from the BDII.

In conclusion, this paper has presented a comprehensive methodology for benchmarking Grid information systems. This methodology demonstrated that including the concept of (α, β)-currency gives additional insight over using the query response time alone. Our experimental results have also indicated that the existing Grid information system used in the EGEE production infrastructure is not particularly suited for high frequency object types. Future work can consider further improvements of the query set or more exhaustive experimental results.

References

1. Andreozzi, S., Burke, S., Field, L., Litmaath, M.: GLUE schema version 1.3, http://glueschema.forge.cnaf.infn.it/Spec/V13
2. Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C.: Grid information services for distributed resource sharing. In: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing, San Francisco, CA, USA, pp. 181–194 (2001)
3. Ehm, F., Field, L., Schulz, M.W.: Scalability and performance analysis of the EGEE information system. Journal of Physics: Conference Series 119(6), 062029 (2008)
4. Field, L., Andreozzi, S., Konya, B.: Grid information system interoperability: The need for a common information model. In: Proceedings of the 4th IEEE International Conference on eScience, Indianapolis, IN, USA, pp. 501–507 (2008)
5. Field, L., Sakellariou, R.: How dynamic is the grid? Towards a quality metric for grid information systems. In: Proceedings of the 11th ACM/IEEE International Conference on Grid Computing, Brussels, Belgium, pp. 113–120 (2010)
6. Field, L., Schulz, M.W.: Grid deployment experiences: The path to a production quality LDAP based grid information system. In: Proceedings of the Conference for Computing in High-Energy and Nuclear Physics, pp. 723–726 (2004)
7. Fitzgerald, S., Foster, I., Kesselman, C., von Laszewski, G., Smith, W., Tuecke, S.: A directory service for configuring high-performance distributed computations. In: Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing, Portland, OR, USA, pp. 365–375 (1997)
8. Gagliardi, F., Jones, B., Grey, F., Heikkurinen, M.: Building an infrastructure for scientific grid computing: status and goals of the EGEE project. Philosophical Transactions. Series A, Mathematical, Physical, and Engineering Sciences 363(1833), 1729–1742 (2005)
9. Plale, B., Jacobs, C., Jenson, S., Liu, Y., Moad, C., Parab, R., Vaidya, P.: Understanding grid resource information management through a synthetic database benchmark/workload. In: Proceedings of the 4th IEEE International Symposium on Cluster Computing and the Grid (CCGrid), Chicago, IL, USA, pp. 277–284 (2004)
10. Riedel, M.: Interoperation of world-wide production e-Science infrastructures. Concurrency and Computation: Practice and Experience 21(8), 961–990 (2009)
11. Smith, W., Waheed, A., Meyers, D., Yan, J.: An evaluation of alternative designs for a grid information service. In: Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing, Pittsburgh, PA, USA, pp. 185–192 (2000)
12. Zanikolas, S., Sakellariou, R.: An importance-aware architecture for large-scale grid information services. Parallel Processing Letters 18(3), 347–370 (2008)
13. Zhang, X., Freschl, J., Schopf, J.: A performance study of monitoring and information services for distributed systems. In: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing, Seattle, WA, USA, pp. 270–281 (2003)
14. Zhang, X., Freschl, J.L., Schopf, J.M.: Scalability analysis of three monitoring and information systems: MDS2, R-GMA, and Hawkeye. Journal of Parallel and Distributed Computing (JPDC) 67(8), 883–902 (2007)
15. Zhang, X., Schopf, J.: Performance analysis of the globus toolkit monitoring and discovery service, MDS2. In: IEEE International Conference on Performance, Computing, and Communications, Phoenix, AZ, USA, pp. 843–849 (2004)

Green Cloud Framework for Improving Carbon Efficiency of Clouds

Saurabh Kumar Garg[1], Chee Shin Yeo[2], and Rajkumar Buyya[1]

[1] Cloud Computing and Distributed Systems Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
{sgarg,raj}@csse.unimelb.edu.au
[2] Distributed Computing Group
Computing Science Department
Institute of High Performance Computing, Singapore
yeocs@ihpc.a-star.edu.sg

Abstract. The energy efficiency of ICT has become a major issue with the growing demand of Cloud Computing. More and more companies are investing in building large datacenters to host Cloud services. These datacenters not only consume huge amount of energy but are also very complex in the infrastructure itself. Many studies have been proposed to make these datacenter energy efficient using technologies such as virtualization and consolidation. Still, these solutions are mostly cost driven and thus, do not directly address the critical impact on the environmental sustainability in terms of CO_2 emissions. Hence, in this work, we propose a user-oriented Cloud architectural framework, i.e. Carbon Aware Green Cloud Architecture, which addresses this environmental problem from the overall usage of Cloud Computing resources. We also present a case study on IaaS providers. Finally, we present future research directions to enable the wholesome carbon efficiency of Cloud Computing.

Keywords: Cloud Computing, Green IT, Resource Management.

1 Introduction

Cloud Computing provides a highly scalable and cost-effective computing infrastructure for running IT applications such as High Performance Computing (HPC), Web and enterprise applications which require ever-increasing computational resources. The emergence of Cloud Computing has rapidly changed the paradigm of ownership-based computing approach to subscription-oriented computing by providing access to scalable infrastructure and services on-demand. The Cloud users can store, access, and share any amount of information online. Similarly, small and medium enterprises/organizations do not have to worry about purchasing, configuring, administering, and maintaining their own computing infrastructure. They can instead focus on improving their core competencies by exploiting a number of Cloud Computing benefits such as low cost,

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 491–502, 2011.

datacenter efficiencies, on-demand computing resources, faster and cheaper software development capabilities.

However, Clouds are essentially datacenters hosting application services offered on a subscription basis. They require high energy usage to maintain their operations. Today, a typical datacenter with 1000 racks needs 10 Megawatt of power to operate [19]. High energy usage results in high energy cost. Thus, for a datacenter, the energy cost is a significant component of its operating and upfront costs. In addition, in April 2007, Gartner estimated that the Information and Communication Technologies (ICT) industry generates about 2% of the total global CO_2 emissions, which is equal to the aviation industry [8]. According to a report published by the European Union [1], a decrease in emission volume of 15–30% is required before the year 2020 to keep the global temperature increase below 2°C. Thus, the rapidly growing energy consumption and CO_2 emission of Cloud infrastructure has become a key environmental concern [20] [4].

Hence, energy efficient solutions are required to ensure the environmental sustainability of this new computing paradigm. Up to now, as datacenters are the major elements of Cloud Computing resources, most solutions primarily focus on minimizing the energy consumption of datacenters which indirectly minimizes the CO_2 emission [2]. However, although such solutions can decrease the energy consumption to a great degree, they do not ensure the minimization of CO_2 emissions as a whole. For example, consider a Cloud datacenter which uses cheap energy generated by coal. The usage of such a datacenter will only increase CO_2 emissions.

Therefore, we propose a user-oriented Carbon Aware Green Cloud Architecture for reducing the carbon footprint of Cloud Computing in a wholesome manner without sacrificing the Quality of Service (QoS) (such as performance, responsiveness and availability) offered by multiple Cloud providers. Our architecture is designed such that it provides incentives to both users and providers to utilize and deliver the most "Green" services respectively. Our evaluation results in the context of IaaS Clouds show that a large amount of CO_2 savings can be gained using our proposed architecture. The contributions of this paper are:

- a novel Carbon Aware Green Cloud Architecture that aims to reduce CO_2 emissions without impacting the service performance; and
- a Carbon Efficient Green Policy (CEGP) for carbon-based scheduling that can reduce the carbon footprint of Cloud Computing by 25% compared to a basic Cloud resource management system.

2 Related Work

Most works improve the energy efficiency of Clouds by addressing the issue within a particular datacenter and not from the usage of Clouds as a whole. They focus on scheduling and resource management within a single datacenter to reduce the amount of active resources executing the workload [2]. The consolidation of Virtual Machines (VMs), VM migration, scheduling, demand

projection, heat management, temperature aware allocation, and load balancing are used as basic techniques for minimizing energy consumption. Virtualization plays an important role in these techniques due to its several benefits such as consolidation, live migration and performance isolation.

Some works also propose frameworks to enable the energy efficiency of Clouds from user and provider perspectives. From the provider perspective, GreenCloud architecture [16] aims to reduce virtualized datacenter energy consumption by supporting optimized VM migration and VM placement. Similar work is presented by Lefevre et al. [14] who propose Green Open Cloud (GOC). GOC is designed for next generation Cloud datacenter that supports facilities like advance reservation. GOC aggregates the workload by negotiating with users so that idle servers can be switch-off longer.

Although these works maximize the energy efficiency of Cloud datacenters, they do not consider CO_2 emission which measures the environmental sustainibility of Cloud Computing. Even if a Cloud provider has used most energy efficient solutions for building his datacenter, it is still not assured that Cloud Computing will be carbon efficient. Greenpeace [10] indicates that current datacenters are really not environmentally friendly as Cloud providers are more concerned about reducing energy cost rather than CO_2 emission. For instance, Google Datacenter in Lenoir, NC, USA, uses 50.5% of dirty energy generated by coal. Thus, our previous work [7] proposes policies to simultaneously maximize the Cloud provider's profit and minimize the CO_2 emission of its non-virtualized datacenters. Le et al. [13] consider a similar multi-datacenter scenario, but with a different perspective of leveraging green energy by capping the brown energy. In contrast, here we propose an architectural framework which focuses on reducing the carbon footprint of Cloud Computing as a whole. Specifically, we consider all the elements of Cloud computing including Software, Platform, and Infrastructure as a Service. We also present a carbon aware policy for IaaS providers.

3 Carbon Aware Green Cloud Architecture

We propose Carbon Aware Green Cloud Architecture (Figure 1), which considers the goals of both users and providers while curbing the CO_2 emission of Clouds. Its elements include:

1. **Third Party:** Green Offer Directory and Carbon Emission Directory listing available green Cloud services and their energy efficiency respectively;
2. **User:** Green Broker accepting Cloud service requests (i.e. software, platform, or infrastructure) and selecting the most green Cloud provider; and
3. **Provider:** Green Middleware enabling the most carbon efficient operation of Clouds. The components of this middleware vary depending on the Cloud offerings (i.e. SaaS, PaaS, or IaaS).

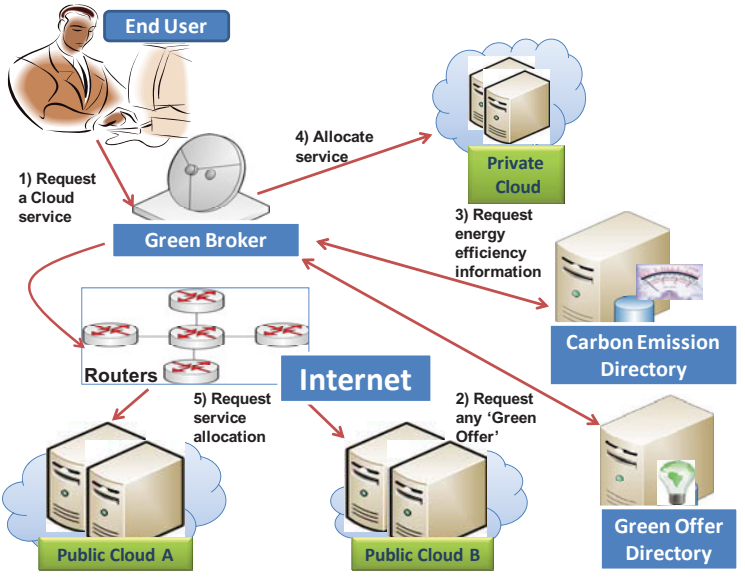


Fig. 1. Carbon Aware Green Cloud Architecture

3.1 Third Party: Green Offer Directory and Carbon Emission Directory

We propose two new elements, i.e. Green Offer Directory and Carbon Emission Directory, which are essential to enforce the green usage of Cloud Computing. Governments have already introduced energy ratings for datacenters and various laws to cap the energy usage of these datacenters [12] [22]. There is also increasing awareness on the impact of greenhouse gases on climate change [10]. Therefore, users will likely prefer using Cloud services of providers which ensure the minimum carbon footprint. Cloud providers can also use these directories as an advertising tool to attract more users. For instance, Google has released the energy efficiency of its datacenters [17]. Hence, the introduction of such directories is practical in the current context of Cloud Computing.

Cloud providers register their services in the form of 'Green Offers' to a Green Offer Directory which is accessed by Green Broker. These offers consist of the type of service provided, pricing, and time when it can be accessed for the least CO_2 emission. The Carbon Emission Directory maintains data related to the energy efficiency of Cloud services, which include the Power Usage Effectiveness (PUE) and cooling efficiency of Cloud datacenters which are providing the service, network cost, and CO_2 emission rate of electricity. Hence, Green Broker can get the current status of energy parameters for using various Cloud services from Carbon Emission Directory.

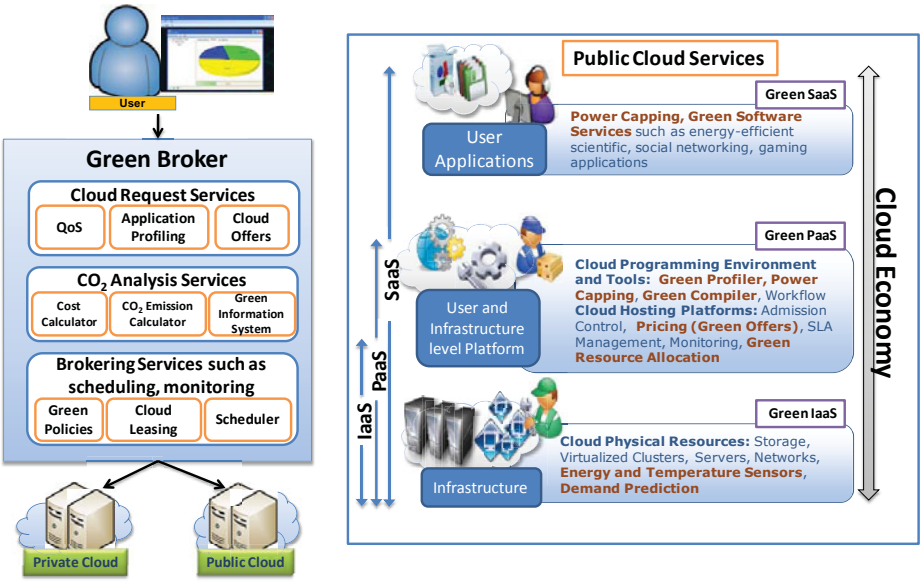


Fig. 2. (a) Green Broker and (b) Green Middleware components for each Cloud service (SaaS, PaaS, and IaaS)

3.2 User: Green Broker

Green Broker (Figure 2) has similar responsibility as a typical Cloud broker, i.e. to lease Cloud services on behalf of users and schedule their applications. Its first layer comprises Cloud request services that analyze the requests and their QoS requirements. Its second layer calculates the cost and carbon footprint of leasing particular Cloud services based on information about various Cloud offerings and current CO_2 emission factors obtained from Green Offer Directory and Carbon Emission Directory respectively. With these calculations, Green Policies make the decisions of leasing Cloud services. If no exact match is found for a request, alternate 'Green Offers' are suggested to users by Cloud Request Services.

The carbon footprint of a user request depends on the type of Cloud service it requires, i.e. SaaS, PaaS and IaaS, and is computed as the sum of CO_2 emission due to data transfer and service execution at datacenter. SaaS and PaaS requests use CO_2 emission per second (CO2PS) to reflect long term usage, while IaaS request uses CO_2 emission as data transfer is mostly once.

– **SaaS and PaaS Request (CO_2 emission per second):**

$$CO2PS_{SaaS/PaaS} = (r_{dT}^{CO2} E_{dT} \times a_{dT}) + (r^{CO2} \times \frac{1}{DCiE} \times E_{serv}) \quad (1)$$

where r_{dT}^{CO2} is the CO_2 emission rate per joule of energy spent from the user's machine to the datacenter, E_{dT} is the per-bit energy consumption

of data transfer, a_dT is the data bits transferred per second, r^{CO2} is the CO_2 emission rate where the datacenter is located, $DCiE$ is the power efficiency of the datacenter defined as the fraction of total power dissipated that is used for IT resources, and E_{serv} is the energy spent per second by the server for executing the user's request. The total power dissipated by a Cloud provider is used not only for computers, but also for other purposes, including power conditioning, HVAC (Heating, Ventilating, and Air Conditioning), lighting, and wiring [9]. Therefore, DCiE is the most appropriate parameter for selecting Cloud providers.

– **IaaS Request (CO_2 emission):**

$$CO2_{IaaS} = (r_{dT}^{CO2} E_{dT} \times IOdata) + (r^{CO2} \times \frac{1}{DCiE} \times E_{serv} \times Vtime) \quad (2)$$

where $IOdata$ is the data transferred to run application on VM leased from Clouds and $Vtime$ is the time for which VM is active.

3.3 Provider: Green Middleware

To support carbon aware Cloud Computing, a Cloud provider must implement "Green" conscious middleware at various layers depending on the type of Cloud service offered (SaaS, PaaS, or IaaS) (Figure 2) as follows:

– **SaaS Level:** SaaS providers mainly offer software installed in their own datacenters or resources leased from IaaS providers. Therefore, they require Power Capping component to limit the usage of software services by each user. This is especially important for social networking and game applications where users become completely unaware of their actions on environmental sustainability. SaaS providers can also offer Green Software Services deployed on carbon efficient datacenters with less replications.
– **PaaS Level:** PaaS providers in general offer platform services for application development and their deployment. Thus, to ensure energy efficient development of applications, relevant components such as Green Compiler to compile applications with the minimum carbon footprint and carbon measuring tools for users to monitor the carbon footprint of their applications. For example, JouleSort [19] is a Green Profiler providing energy efficiency benchmarks to measure the energy required to perform an external sort.
– **IaaS level:** IaaS providers play the most crucial role in the success of Green Cloud Architecture since IaaS not only offers independent infrastructure services, but also support other services (SaaS and PaaS) offered by Clouds. They use the latest technologies for IT and cooling systems to have the most energy efficient infrastructure. By using virtualization and consolidation, the energy consumption is further reduced by switching off unutilized servers. Energy and Temperature Sensors are installed to calculate the current energy efficiency of each IaaS provider and their datacenters. This information is advertised regularly by Cloud providers in the Carbon Emission Directory. Various green scheduling and resource provisioning policies will ensure

minimum energy usage. In addition, IaaS providers can design attractive 'Green Offers' and pricing schemes providing incentives for users to use their services during off-peak or maximum energy efficiency hours.

4 Case Study: IaaS Cloud

To illustrate the effectiveness of our proposed architecture in reducing the energy and CO_2 emissions across the entire Cloud infrastructure in a unified manner, we present a simple scenario focussed on IaaS. It considers multiple IaaS providers offering computational resources to run HPC jobs. A user request consists of an application, its estimated length in time, the deadline to complete execution, and the number of resources required. Requests are submitted to Green Broker which interprets and analyzes the service requirements before deciding where to execute them.

Cloud datacenters have different CO_2 emission rates and energy costs based on their locations. Each datacenter updates this data to Carbon Emission Directory for facilitating carbon efficient scheduling. For this study, we consider three CO_2 emission related parameters: CO_2 emission rate (kg/kWh) (r_i^{CO2}), average DCiE ($Ieff_i$), and VM power efficiency ($VMeff_i$). The VM power efficiency is the amount of power dissipated by fully active VM running at maximum utilization level [3]. In Green Offer Directory, IaaS providers specify the maximum number of VMs that can be initiated at a particular time for achieving the highest energy efficiency due to the variation in datacenter efficiency with time and load [18] and power capping technologies used within the datacenter [15].

5 Carbon Efficient Green Policy (CEGP)

We develop Carbon Efficient Green Policy (CEGP) for Green Broker to periodically select the Cloud provider with the minimum carbon footprint and initiate VMs to run the jobs (Algorithm 1). Based on user requests at each scheduling interval, Green Broker obtains information from Carbon Emission Directory about the current CO_2 emission related parameters of providers as described in Section 4 (Line 2). The QoS requirements of a job j is defined in a tuple (d_j, n_j, e_j, f_j^m), where d_j is the deadline to complete job j, n_j is the number of CPUs required for job execution, and e_j is the job execution time when operating at the CPU frequency f_j^m (Line 3).

CEGP then sorts the incoming jobs based on Earliest Deadline First (EDF) (Line 4), before sorting the Cloud datacenters based on their carbon footprint (Line 5). CEGP schedule jobs to IaaS Clouds in a greedy manner to reduce the overall CO_2 emission. For IaaS providers, CEGP uses three main factors to calculate the CO_2 emission: CO_2 emission rate, DCiE, and CPU power efficiency. The carbon footprint of an IaaS Cloud i is given by: $r_i^{CO2} \times \frac{1}{Ieff_i} \times \frac{1}{VMeff_i}$ where $VMeff_i$ can be calculated by Cloud providers based on the proportion of resources on a server utilized by the VM using tools such as PowerMeter [3]. If a VM consumes the power equivalent to a processor running at f_i frequency

```
1  while current_time < next_schedule_time do
2      RecvCloudPublish(P);
       //P contains information of Cloud datacenters
3      RecvJobQoS(Q);
       //Q contains information of Cloud users
4  Sort jobs in ascending order of deadline;
5  Sort datacenters in ascending order of r_i^CO2 × 1/Ieff_i × 1/VMeff_i;
6  foreach job j ∈ RecvJobQoS do
7      foreach datacenter i ∈ RecvCloudPublish do
8          if isInitiatedVM(i) then
9              if MaxIniVMlimitReached(i) then
10                 Try to schedule the job j on already initiated VMs;
11                 if job j is missing deadline then
12                     continue;
13                 break;
14             else
15                 InitiateVM(i) and schedule job j;
16                 break;
```

Algorithm 1. Carbon Efficient Green Policy (CEGP)

level, then we can use the following power model [5] [23] to calculate its power efficiency: $\beta_i + \alpha_i(f_i)^3$, where β_i is the static power dissipated by the CPU and α_i is the proportionality constant. Therefore, the approximate energy efficiency of VM is: $VMeff_i = |\frac{f_i}{\beta_i+\alpha_i(f_i)^3}|$. If job j executes at CPU frequency f, then its CO_2 emission will be the minimum when it is allocated to the datacenter with the minimum CO_2 emission rate r_i^{CO2}, maximum DCiE value $Ieff_i$, and maximum CPU power efficiency $VMeff_i$. CEGP then assigns jobs to VMs initiated on each Cloud datacenter according to this ordering (Line 6–16).

6 Performance Evaluation and Results

We use the Lawrence Livermore National Laboratory (LLNL) Thunder trace from Feitelson's Parallel Workload Archive (PWA) [6] with the highest resource utilization of 87.6% to ideally model a heavy HPC workload scenario. The trace contains the submit time, requested number of CPUs, and actual runtime of jobs. We use a methodology proposed by Irwin et al. [11] to synthetically assign deadlines through two classes, namely Low Urgency (LU) and High Urgency (HU). We set LU jobs to have a deadline mean of 12, which is 3 times longer than HU jobs with a deadline mean of 4. The arrival sequence of jobs from the HU and LU classes is randomly distributed.

Provider Configuration: We model 8 different IaaS providers with different configurations as listed in Table 1. Power parameters (i.e. CPU power factors and frequency level) of the CPUs at different datacenters are derived from Wang and Lu's work [23]. Green Broker uses CEGP to schedule jobs periodically at a scheduling interval of 50 seconds, which is to ensure that Green Broker can receive at least one job in every scheduling interval. The DCiE value of Cloud

Table 1. Characteristics of Cloud datacenters

Location of Cloud Datacenter	CO_2 Emission Rate (kg/kWh) [a]	CPU Power Factors		CPU Frequency Level
		β	α	f_i
New York, USA	0.389	65	7.5	1.8
Pennsylvania, USA	0.574	75	5	1.8
California, USA	0.275	60	60	2.4
Ohio, USA	0.817	75	5.2	2.4
North Carolina, USA	0.563	90	4.5	3.0
Texas, USA	0.664	105	6.5	3.0
France	0.083	90	4.0	3.2
Australia	0.924	105	4.4	3.2

[a] CO_2 emission rates are drived from a US Department of Energy (DOE) document [21] (Appendix F-Electricity Emission Factors 2007).

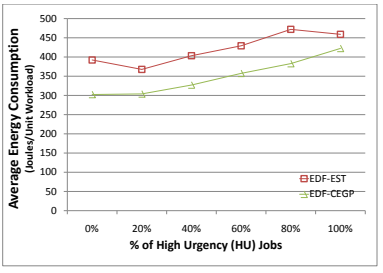
datacenters is randomly generated using a uniform distribution between [0.33, 0.80] as indicated in the study conducted by Greenberg et al. [9].

Experimental Scenarios: We compare the carbon efficiency of CEGP with a performance-based scheduling algorithm (Earliest Start Time (EST)) using two metrics: average energy consumption and CO_2 emissions. EST schedules jobs to the datacenter where jobs can start as earliest as possible with the least waiting time. The average energy consumption shows the amount of energy saved by our green framework using CEGP compared to an existing approach using EST which just focus on performance, whereas the average CO_2 emission shows its corresponding environmental impact. We examine two experimental scenarios: 1) comparison of CEGP with EST and 2) effect of relationship between CO_2 emission rate and datacenter power efficiency $DCiE$. The first scenario demonstrates how our proposed architecture can achieve higher carbon efficiency. The second scenario reveals how the relationship between CO_2 emission rate and DCiE can affect the achievement of carbon efficiency. Hence, we consider two types of relationship between CO_2 emission rate and DCiE: 1) datacenter with the highest CO_2 emission rate has the highest DCiE (HH) and 2) datacenter with the highest CO_2 emission rate has the lowest DCiE (HL). We generate 8 DCiE values using uniform distribution between [0.33, 0.80] and assign them to the 8 datacenters to achieve HH and HL configurations accordingly.

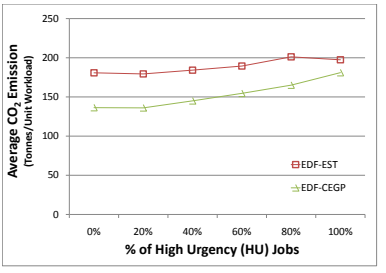
6.1 Comparison of CEGP with Performance-Based Algorithm (EST)

We compare CEGP with EST for datacenters with HH configuration. The effect of job urgency on energy consumption and CO_2 emission is prominent. As the percentage of HU jobs with more urgent (shorter) deadlines increases, the energy consumption (Figure 3(a)) and CO_2 emission (Figure 3(b)) also increase due to more urgent jobs running on datacenters with lower DCiE value and at the highest CPU frequency to avoid deadline violations.

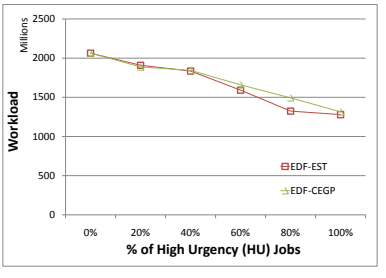
It is clear that our proposed architecture using CEGP (EDF-CEGP) can reduce up to 23% of the energy consumption (Figure 3(a)) and 25% of the CO_2 emission (Figure 3(b)) compared to an existing approach using EST (EDF-EST) across all datacenters. CEGP is also able to complete very similar amount



(a) Energy Consumption



(b) CO_2 Emission



(c) Workload Executed

Fig. 3. Comparison of CEGP with performance-based algorithm (EST)

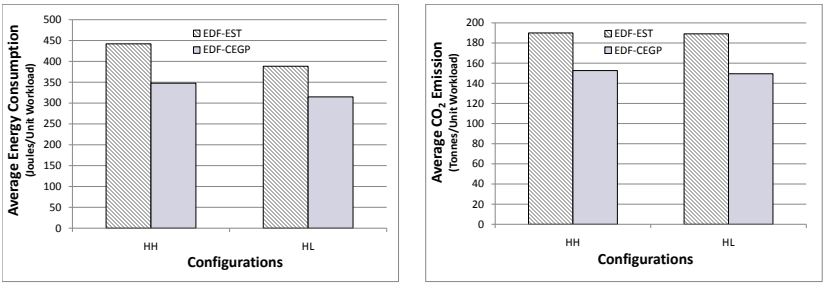
of workload[1] as EST (Figure 3(c)), but with much less energy consumption and CO_2 emission. This highlights the importance of considering the DCiE and CO_2 emission related factors in achieving the carbon efficient usage of Cloud Computing. In particular, CEGP can reduce energy consumption (Figure 3(a)) and CO_2 emission (Figure 3(b)) even more when there are more LU jobs with less urgent (longer) deadline.

6.2 Effect of Relationship between CO_2 Emission Rate and Datacenter Power Efficiency *DCiE*

This experiment analyzes the impact of different configurations (HH and HL) of datacenters with respect to CO_2 emission rate and datacenter power efficiency *DCiE* based on 40% of high urgency jobs.

In both HH and HL configurations, CEGP reduces CO_2 emission and energy consumption between 23% to 25% (Figure 4(a) and 4(b)). Therefore, we infer that for other configurations, we will also achieve similar carbon efficiency in Cloud Computing by using CEGP. Moreover, in Figure 4(a), there is a decrease in energy consumption of all the Cloud datacenters from HH to HL configuration by using EST, while there is almost no corresponding decrease by using CEGP.

[1] workload=$\sum$(job execution time $\times$ number of required processors).



(a) Energy Consumption (b) CO_2 Emission

Fig. 4. Effect of relationship between CO_2 emission rate and DCiE

This shows that how important is the consideration of global factors such as DCiE and CO_2 emission rate in order to improve the carbon footprint of Cloud Computing.

7 Conclusion

In this paper, we present a Carbon Aware Green Cloud Architecture to improve the carbon footprint of Cloud Computing taking into account its global view. Our architecture is designed such that it provides incentives to both users and providers to utilize and deliver the most "Green" services respectively. Therefore, it embeds components such as Green broker from user side to ensure the execution of their applications with the minimum carbon footprint. Similarly, from provider side, we propose features for next generation Cloud providers who will publish the carbon footprint of their services in public directories and provide 'Green Offers' to minimize their overall energy consumption. We also propose a Carbon Efficient Green Policy (CEGP) for Green broker which schedules user application workload with urgent deadline on Cloud datacenters with more energy efficiency and low carbon footprint.

Further, the simulation-based evaluation of our architecture is done in multiple IaaS Cloud provider scenario. We compare two scheduling approaches to prove how our proposed architecture helps in improving carbon and energy footprint of Cloud Computing. Performance evaluation results show how our proposed architecture using a Green Policy CEGP can save up to 23% energy while improving the carbon footprint by about 25%. Therefore, these promising results show that by using our architectural framework carbon footprint and energy consumption of Cloud Computing can be improved.

In the future, we will investigate different 'Green Policies' for Green broker and also how Cloud providers can design various 'Green Offers' based on their internal power efficiency techniques such as VM consolidation and migration. We will also conduct experiments for our architecture using real Clouds.

References

1. Baer, P.: Exploring the 2020 global emissions mitigation gap (December 2008), http://www.ippr.org/uploadedFiles/globalclimatenetwork/Exploring_the_Mitigation_Gap[1].pdf
2. Beloglazov, A., Buyya, R., Lee, Y., Zomaya, A.: A Taxonomy and Survey of Energy-Efficient Data Centers and Cloud Computing Systems. In: Zelkowitz, M. (ed.) Advances in Computers. Elsevier, San Francisco (2011)
3. Bohra, A.E.H., Chaudhary, V.: Vmeter: Power modelling for virtualized clouds. In: Proc. of 24th IEEE IPDPS Workshops, Atlanta, USA (2010)
4. Cameron, K.W.: Trading in Green IT. Computer 43(3), 83–85 (2010)
5. Chen, Y., et al.: Managing server energy and operational costs in hosting centers. ACM SIGMETRICS Performance Evaluation Review 33(1), 303–314 (2005)
6. Feitelson, D.: Parallel workloads archive (2011), http://www.cs.huji.ac.il/labs/parallel/workload
7. Garg, S., Yeo, C., Anandasivam, A., Buyya, R.: Environment-conscious scheduling of HPC applications on distributed cloud-oriented data centers. Journal of Parallel and Distributed Computing 71(6), 732–749 (2011)
8. Gartner: Gartner Estimates ICT Industry Accounts for 2 Percent of Global CO2 Emissions (April 2007), http://www.gartner.com/it/page.jsp?id=503867
9. Greenberg, S., et al.: Best practices for data centers: Results from benchmarking 22 data centers. In: ACEEE Summer Study on Energy Efficiency in Buildings (2006)
10. Greenpeace International: Make IT green: Cloud computing and its contribution to climate change (2010)
11. Irwin, D., Grit, L., Chase, J.: Balancing risk and reward in a market-based task service. In: Proc. of 13th IEEE HPDC, Honolulu, USA (2004)
12. Kurp, P.: Green computing. Commun. ACM 51, 11–13 (2008)
13. Le, K., et al.: Managing the cost, energy consumption, and carbon footprint of internet services. ACM SIGMETRICS Perf. Eval. Review 38(1), 357–358 (2010)
14. Lefèvre, L., Orgerie, A.C.: Designing and evaluating an energy efficient Cloud. The Journal of Supercomputing 51(3), 352–373 (2010)
15. Lefurgy, C., Wang, X., Ware, M.: Power capping: a prelude to power shifting. Cluster Computing 11(2), 183–195 (2008)
16. Liu, L., et al.: GreenCloud: a new architecture for green data center. In: Proc. of 6th International Conference on Autonomic Computing, Barcelona, Spain (2009)
17. Miller, R.: Google: Raise Your Data Center Temperature (October 2008), http://www.datacenterknowledge.com/archives/2008/10/14/google-raise-your-data-center-temperature
18. Patel, C., et al.: Energy Aware Grid: Global Workload Placement based on Energy Efficiency. Technical Report HPL-2002-329, HP Labs, Palo Alto (2002)
19. Rivoire, S., Shah, M.A., Ranganathan, P., Kozyrakis, C.: Joulesort: a balanced energy-efficiency benchmark. In: Proc. of ACM SIGMOD, Beijing, China (2007)
20. Tomlinson, B., Silberman, M.S., White, J.: Can More Efficient IT Be Worse for the Environment? Computer 44, 87–89 (2011)
21. U.S. DOE: Voluntary Reporting of Greenhouse Gases: Appendix F. Electricity Emission Factors, http://www.eia.doe.gov/oiaf/1605/pdf/Appendix
22. U.S. EPA: Report to Congress on Server and Data Center Energy Efficiency, Public Law 109-431 (August 2007)
23. Wang, L., Lu, Y.: Efficient Power Management of Heterogeneous Soft Real-Time Clusters. In: Proc. of 29th IEEE RTSS, Barcelona, Spain (2008)

Optimizing Multi-deployment on Clouds by Means of Self-adaptive Prefetching

Bogdan Nicolae[1], Franck Cappello[1,2], and Gabriel Antoniu[3]

[1] INRIA Saclay, France
bogdan.nicolae@inria.fr
[2] University of Illinois at Urbana Champaign, USA
cappello@illinois.edu
[3] INRIA Rennes Bretagne Atlantique, France
gabriel.antoniu@inria.fr

Abstract. With Infrastructure-as-a-Service (IaaS) cloud economics getting increasingly complex and dynamic, resource costs can vary greatly over short periods of time. Therefore, a critical issue is the ability to deploy, boot and terminate VMs very quickly, which enables cloud users to exploit elasticity to find the optimal trade-off between the computational needs (number of resources, usage time) and budget constraints. This paper proposes an adaptive prefetching mechanism aiming to reduce the time required to simultaneously boot a large number of VM instances on clouds from the same initial VM image (multi-deployment). Our proposal does not require any foreknowledge of the exact access pattern. It dynamically adapts to it at run time, enabling the slower instances to learn from the experience of the faster ones. Since all booting instances typically access only a small part of the virtual image along almost the same pattern, the required data can be pre-fetched in the background. Large scale experiments under concurrency on hundreds of nodes show that introducing such a prefetching mechanism can achieve a speed-up of up to 35% when compared to simple on-demand fetching.

1 Introduction

The Infrastructure-as-a-Service (IaaS) cloud computing model [1,2] is gaining increasing popularity both in industry [3] and academia [4,5]. According to this model, users do not buy and maintain their own hardware, but rather rent such resources as virtual machines, paying only for what was consumed by their virtual environments.

One of the common issues in the operation of an IaaS cloud is the need to deploy and fully boot a large number of VMs on many nodes of a data-center at the same time, starting from the same initial VM image (or from a small initial set of VM images) that is customized by the user. This pattern occurs for example when deploying a virtual cluster or a set of environments that support a distributed application: we refer to it as the *multi-deployment pattern*.

Multi-deployments however can incur a significant overhead. Current techniques [6] broadcast the images to the nodes before starting the VM instances, a process that can take tens of minutes to hours, not counting the time to boot the operating system itself. Such a high overhead can reduce the attractiveness of IaaS offers. Reducing this

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 503–513, 2011.

overhead is even more relevant with the recent introduction of spot instances [7] in the Amazon Elastic Compute Cloud (EC2) [3], where users can bid for idle cloud resources at lower than regular prices, however with the risk of their virtual machines being terminated at any moment without notice. In such dynamic contexts, deployment times in the order of tens of minutes are not acceptable.

As VM instances typically access only a small fraction of the VM image throughout their run-time, one attractive alternative to broadcasting is to fetch only the necessary parts on-demand. Such a "lazy" transfer scheme is gaining increasing popularity [8], however it comes at the price of making the boot process longer, as the necessary parts of the image not available locally need to be fetched remotely from the cloud repository.

In this paper we investigate how to improve on-demand transfer schemes for the multi-deployment pattern. Our proposal relies on the fact that the hypervisors will generate highly similar access patterns to the image during the boot process. Under these circumstances, we exploit small delays between the times when the VM instances access the same chunk (due to jitter in execution time) in order to prefetch the chunk for the slower instances based on the experience of the faster ones. Our approach does not require any foreknowledge of the access pattern and dynamically adapts to it as the instances progress in time. A multi-deployment can thus benefit from our approach even when it is launched for the first time, with subsequent runs fully benefiting from complete access pattern characterization.

We summarize our contributions as follows:

- We introduce an approach that optimizes the multi-deployment pattern by means of adaptive prefetching and show how to integrate this approach in IaaS architectures. (Sections 2.1 and 2.2)
- We propose an implementation of these design principles by enriching the metadata structures of BlobSeer [9,10], a distributed storage service designed to sustain a high throughput even under concurrency (Section 2.3).
- We experimentally evaluate the benefits of our approach on the Grid'5000 [11] testbed by performing multi-deployments on hundreds of nodes (Section 3).

2 Our Approach

In this section we present the design principles behind our proposal, show how to integrate them in cloud architectures and propose a practical implementation.

2.1 Design Principles

Stripe VM images in a distributed repository. In most cloud deployments [3,4,5], the disks locally attached to the compute nodes are not exploited to their full potential: they typically serve to cache VM images and provide temporary storage for the running VM instances. Most of the time, this access pattern utilizes only a small fraction of the total disk size. Therefore, we propose to aggregate the storage space of the local disks in a common pool that is used as a distributed VM image repository. This specialized service stores the images in a striped fashion: VM images are split into small equally-sized chunks that are evenly distributed among the local disks of the compute nodes.

When the hypervisor running on a compute node needs to read a region of the VM image that has not been locally cached yet, the corresponding chunks are fetched in parallel from the remote disks storing them. Under concurrency, this scheme effectively enables an even distribution of the read workload, which ultimately improves overall throughput.

Record the access pattern and use it to provide prefetching hints to subsequent remote reads. According to our observations, a multi-deployment generates a read access pattern to the VM image that exhibits two properties: (1) only a small part of the VM image is actually accessed during the boot phase (boot-sector, kernel, configuration files, libraries and daemons, etc.) and (2) read accesses follow a similar pattern on all VM instances, albeit at slightly different moments in time.

For example, Figure 1 shows the read access pattern for a multi-deployment of 100 instances booting a Debian Sid Linux distribution from a 2 GB large virtual raw image striped in chunks of 256 KB. The read access pattern is represented in terms of what chunks are accessed (disk offset) as time progresses. A line corresponds to each chunk and indicates the minimum, average and maximum time since the beginning of the multi-deployment when the chunk was accessed by the instances. It can be noticed that a large part of the disk remains untouched, with significant jitter between the times when the same chunk is accessed.

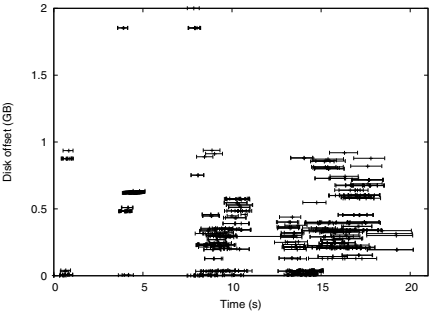


Fig. 1. Accesses to the VM image during a multi-deployment of 100 VM instances

Based on these observations, we propose to monitor two attributes: the total number of accesses to a chunk and the average access time since the beginning of the multi-deployment. These two attributes help establish the order in which chunks are accessed during the boot phase, with an increasing number of accesses leading to a higher precision. Both attributes are updated in real time for each chunk individually. Using this information, the slower instances to access a chunk can "learn from the experience" of the faster ones: they can query the metadata in order to predict what chunks will probably follow and prefetch them in the background. As shown on Figure 1, gaps between periods of I/O activity and I/O inactivity are in the order of seconds, large enough enable prefetching of considerable amounts of data.

To minimize the query overhead, we propose to piggyback information about potential chunk candidates for prefetching on top of every remote read operation to the repository. We refer to this extra information as *prefetching hints* from now on. Since remote read operations need to consult the metadata that indicates where the chunks are stored anyway, the extra overhead in order to build prefetching hints is small. However, too many prefetching hints are not needed and only generate unnecessary overhead. Thus, we limit the number of results (and thereby the number of "false positives") by introducing an *access count threshold* that needs to be reached before a chunk is considered as a viable candidate.

An example for an access threshold of 2 is depicted in Figure 3(a), where 4 instances that are part of the same multi-deployment access the same initial VM image, which is striped into four chunks: A, B, C and D. Initially, all four instances need to fetch chunk A, which does not generate any prefetching hints, as it is the only chunk involved in the requests. Next, the first instance fetches chunk B, followed by instances 2 and 3, both of which fetch chunk C. Finally instance 4 fetches chunk D. Since B is accessed only once, no prefetching hints are generated for instances 2 and 3, while chunk C becomes a prefetching hint for instance 4.

Note that a growing number of chunks that need to be stored in the repository (as a result of adding new VM images) can lead to a high overhead of building prefetching hints, which can even offset the benefits of prefetching. This in turn leads to the need to implement a scalable distributed metadata management scheme (see Section 2.3).

Prefetch chunks in the background using the hints. The prefetching hints returned with each remote read operation can be combined in order to build a prefetching strategy in the background that operates during the periods of I/O inactivity. Note that this scheme is self-adaptive: it can learn on-the-fly about "unknown" VM images during the first multi-deployment, with no need for pre-staging. After the first run, the whole access pattern can be completely characterized in terms of prefetching hints immediately after the first read occurred, which leaves room to employ optimal prefetching strategies for subsequent multi-deployments.

2.2 Architecture

A simplified IaaS cloud architecture that integrates our approach is depicted in Figure 2. The typical elements of a IaaS architecture are illustrated with a light background, while the elements that are part of our proposal are highlighted by a darker background.

A *distributed storage service* is deployed on all compute nodes. It aggregates the space available on the local disks in a common shared pool that forms the virtual machine image repository. This storage service is responsible to transparently stripe the virtual machine images into chunks. The *cloud client* has direct access to the repository and is allowed to upload and download VM images from it. Furthermore, the cloud client also interacts with the *cloud middleware* through a control API that enables

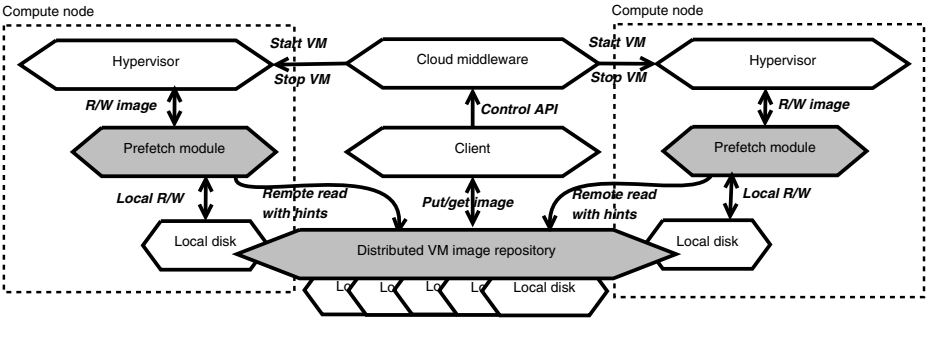


Fig. 2. Cloud architecture that integrates our approach (dark background)

launching and terminating multi-deployments. It is the responsibility of the cloud middleware to initiate the multi-deployment by concurrently launching the hypervisors on the compute nodes.

The *hypervisor* in turn runs the VM instances and issues corresponding reads and writes to the underlying virtual machine images. The reads and writes are intercepted by a *prefetching module*, responsible to implement the design principles proposed in Section 2.1. More specifically, writes are redirected to the local disk (using either mirroring [12] or copy-on-write [13]). Reads are either served locally, if the involved chunks are already available on the local disk, or transferred first from the repository to the local disk otherwise. Each read brings new prefetching hints that are used to transfer chunks in the background from the repository to the local disk.

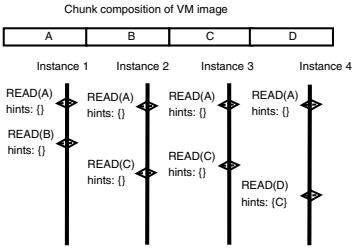
2.3 Implementation

We have chosen to implement the distributed VM image repository on top of *BlobSeer* [9,10]. This choice was motivated by several factors. First, BlobSeer enables *scalable aggregation of storage space* from the participating nodes with low overhead in order to store *BLOBs* (Binary Large OBjects). BlobSeer handles striping and chunk distribution of BLOBs transparently, which can be directly leveraged in our context: each VM image is stored as a BLOB, effectively eliminating the need to perform explicit chunk management.

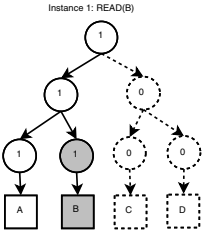
Second, BlobSeer uses a distributed metadata management scheme based on *distributed segment trees* [10] that can be easily adapted to efficiently build prefetching hints. More precisely, a distributed segment tree is a binary tree where each tree node covers a region of the BLOB, with the leaves covering individual chunks. The tree root covers the whole BLOB, while the other non-leaf nodes cover the combined range of their left and right children. Reads of regions in the BLOB imply descending in the tree from the root towards the leaves, which ultimately hold information about the chunks that need to be fetched.

In order to minimize the overhead of building prefetching hints, we add additional metadata to each tree node such that it records the total number of accesses to that node. Since a leaf can be reached only by walking down into the tree, the number of accesses to inner nodes is higher than the number of accesses to leaves. Thus, if the access count threshold is not reached, the whole sub-tree can be skipped, greatly limiting the number of chunks that need to be inspected in order to build the prefetching hints.

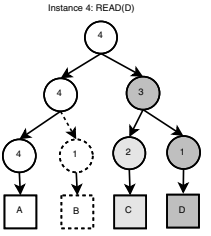
Furthermore, we designed a metadata caching scheme that avoids unnecessary remote accesses to metadata: each tree node that that has reached the threshold since it was visited the last time, is added to the cache and retrieved from there for any subsequent visits. Cached tree nodes might not always reflect an up-to-date number of accesses, however this does not affect correctness as the number of accesses can only grow higher than the threshold. Obviously, the tree nodes that are on the path towards the required chunks (i.e. those chunks that make up the actual read request) need to be visited even if they haven't reached the threshold yet, so they are added to the cache too.



(a) Evolution of remote fetches in time and the associated hints



(b) Local view of the segment tree for Instance 1 after reading chunk B



(c) Local view of the segment tree for Instance 4 after reading chunk D

Fig. 3. Adaptive prefetching by example: multi-deployment of 4 instances with a prefetch threshold of 2

An example of how this works is presented in Figures 3(b) and 3(c). Each tree node is labeled with the number of accesses that is reflected in the local cache. Figure 3(b) depicts the contents of the cache for Instance 1 at the moment when it reads chunk B. White nodes were previously added in the cache when Instance 1 accessed chunk A (access count 1 because it was the first to do so). Dark grey nodes are on the path towards chunk B and are therefore added to the local cache during the execution of the read access. Since the access count of the right child of the root is below the threshold, the whole right subtree is skipped (dotted pattern). Similarly, Figure 3(c) depicts the segment tree at the moment when Instance 4 reads chunk D. Again, white nodes on the path towards chunk A are already in the cache. Dark grey nodes are on the path towards chunk D and are about to be added in the cache. Since the access count of the leaf corresponding to chunk C (light grey) has reached the threshold, it is added to the cache as well and C becomes a prefetching hint, while the leaf of chunk B is skipped (dotted pattern).

Using this scheme, each read from the BLOB potentially returns a series of prefetching hints that are used to prefetch chunks in the background. This is done in a separate thread during the periods of I/O inactivity of the hypervisor. If a read is issued that does not find the required chunks locally, the prefetching is stopped and the required chunks are fetched first, after which the prefetching is resumed. We employ a prefetching strategy that gives priority to the most frequently accessed chunk.

3 Experimental Evaluation

This section presents a series of experiments that evaluate how well our approach performs under the multi-deployment pattern, when a single initial VM image is used to concurrently instantiate a large number of VM instances.

3.1 Experimental Setup

The experiments presented in this work have been performed on Grid'5000 [11], an experimental testbed for distributed computing that federates 9 different sites in France.

We have used the clusters located in Nancy. All nodes of Nancy, numbering 120 in total, are outfitted with x86_64 CPUs offering hardware support for virtualization, local disk storage of 250 GB (access speed ≃55 MB/s) and at least 8 GB of RAM. The nodes are interconnected with Gigabit Ethernet (measured: 117.5 MB/s for TCP sockets with MTU = 1500 B with a latency of ≃0.1 ms). The hypervisor running on all compute nodes is KVM 0.12.5, while the operating system is a recent Debian Sid Linux distribution. For all experiments, a 2 GB raw disk image file based on the same Debian Sid distribution was used.

3.2 Performance of Multi-deployment

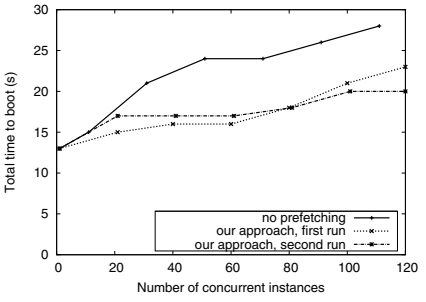
We perform series of experiments that consists in concurrently deploying an increasing number of VMs, one VM on each compute node. For this purpose, we deploy BlobSeer on all of the 120 compute nodes and store the initial 2 GB large image in a striped fashion into it. The chunk size was fixed at 256 KB, large enough to cancel the latency penalty for reading many small chunks, yet small enough to limit the competition for the same chunk. All chunks are distributed using a standard round-robin allocation strategy. Once the VM image was successfully stored, the multi-deployment is launched by synchronizing KVM to start on all the compute nodes simultaneously.

A total of three series of experiments is performed. In the first series, the original implementation with no prefetching is evaluated. In the second series of experiments, we evaluate our approach when a multi-deployment is launched for the first time such that no previous information about the access pattern is available, which essentially forces the system to self-adapt according to the prefetching hints. We have fixed the access count threshold to be 10% of the total number of instances in the multi-deployment. Finally, the third series of experiments evaluates our approach when a multi-deployment was already launched before, such that its access pattern has been recorded. This scenario corresponds to the ideal case when complete information about the access pattern is available from the beginning.

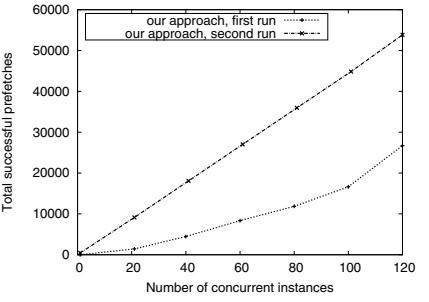
Performance results are depicted in Figure 4. As can be observed, a larger multi-deployment leads to a steady increase in the total time required to boot all VM instances (Figure 4(a)), for all three scenarios. This is both the result of increased read contention to the VM image, as well as increasing jitter in execution time. However, prefetching chunks in the background clearly pays off: for 120 instances, our self-adaptation technique lowers the total time to boot by 17% for the first run and almost 35% for subsequent runs, once the access pattern has been learned.

Figure 4(b) shows the number of successful prefetches of our approach as the number of instances in the multi-deployment grows. For the second run, almost all of the ≃450 chunks are successfully prefetched by each instance, for a total of ≃54000 prefetches. As expected, for the first run it can be clearly observed that a higher number of concurrent instances benefits the learning process more, as there are more opportunities to exploit jitter in execution time. For 120 instances, the total number of successful prefetches is about half compared to the second and subsequent runs.

Figures 5(a) and 5(b) show the remote read access pattern for a multi-deployment of 100 instances: for our approach during the first run and the second run respectively. Each line represents the minimum, average and maximum time from the beginning

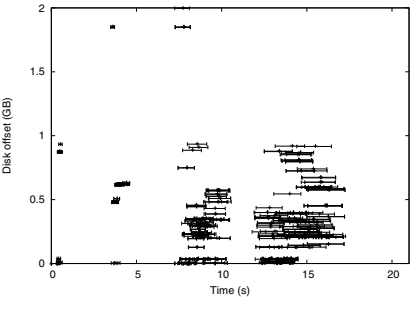


(a) Total time to boot all VM instances of a multi-deployment

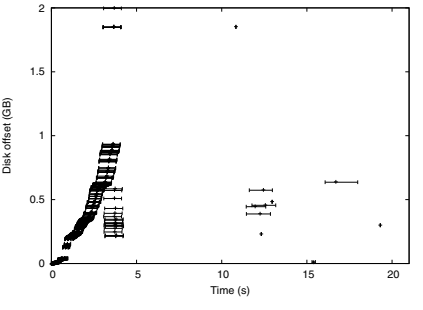


(b) Total number of remote accesses that were avoided for reads issued by the hypervisor as the result of successful prefetches

Fig. 4. Performance of self-adaptive prefetching when increasing the number of VM instances in the multi-deployment



(a) Remote accesses during the learning phase of the first-time run



(b) Remote accesses for the second and subsequent runs

Fig. 5. Remote accesses to the VM image during a multi-deployment of 100 VM instances using our approach

of the deployment when the same chunk (identified by its offset in the image) was accessed by the VM instances. The first run of our approach generates a similar pattern with the case when no prefetches are performed (represented in Figure 1). While jitter is still observable, thanks to our prefetching hints the chunks are accessed earlier, with average access times slightly shifted towards the minimum access times.

Once the access pattern has been learned, the second run of our approach (Figure 5(b)) is able to prefetch the chunks much faster, in less than 25% of the total execution time. This prefetching rush slightly increases both the remote read contention and jitter in the beginning of the execution but with the benefit of reducing both parameters during the rest of the execution. Thus, jitter accumulates to a lesser extent for a small number of concurrent instances and could be a possible explanation of why the first run is actually slightly faster than the second run for smaller multi-deployments.

4 Related Work

Many hypervisors provide native copy-on-write support by defining custom virtual machine image file formats (such as [13]). They rely on a separate read-only template as the backing image file, while storing local modifications in the derived copy-on-write file. Much like our approach, a parallel file system [14,15,16] can be relied upon to stripe and distribute the read-only image template among multiple storage elements. However, unlike our approach, a parallel file system is not specifically optimized for multi-deployments and thus does not perform prefetching that is aware of the global trend in the access pattern.

Several storage systems such as Amazon S3 [17] (backed by Dynamo [18]) have been specifically designed as highly available key-value repositories for cloud infrastructures. They are leveraged by Amazon to provide elastic block level storage volumes (EBS [8]) that support striping and lazy, on-demand fetching of chunks. Amazon enables the usage of EBS volumes to store VM images, however we are not aware of any particular optimizations for the multi-deployment pattern.

Finally, dynamic analysis of access patterns was proposed in [19] for the purpose of building adaptive prefetching strategies. The proposal uses heuristic functions to predict the next most probable disk access using the recent reference history. The algorithms involved however are designed for centralized approaches. They typically utilize only a small recent window of the reference history in order to avoid computational overhead associated with prefetching. Thanks to our distributed metadata management scheme, we can maintain a full access history that represents the global trend of the multi-deployment, which can be leveraged to perform an optimal prefetching after the first run.

5 Conclusions

This paper proposed a self-adaptive prefetching mechanism for "lazy" transfer schemes that avoid full broadcast of VM images during multi-deployments on IaaS clouds. We rely on the fact that all VM instances generate a highly similar access pattern, which is slightly shifted in time due to runtime jitter. Our proposal exploits this jitter to enable VM instances to learn from experience of the other concurrently running VM instances in order to speed-up reads not already cached on the local disk by prefetching the necessary parts of the VM image from the repository.

Our scheme is highly adaptive and does not require any past traces of the deployment, bringing a speed-up of up to 17% for the first run when compared to simple, on-demand fetching only. Once the access pattern has been learned, subsequent multi-deployments of the same VM image benefit from the full access history and can perform an optimal prefetching that further increases the speed-up to up to 35% compared to the case when no prefetching is performed.

Thanks to these encouraging results, we plan to further investigate the potential benefits of exploiting the similarity of access pattern to improve multi-deployments. In particular, we see a good potential to reduce the prefetching overhead by means of replication and plan to investigate this issue more closely. Furthermore, an interesting direction to explore is the use of push approaches (rather then pull) using broadcast algorithms once the access pattern has been learned.

Acknowledgments. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see http://www.grid5000.fr/).

References

1. Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. Commun. ACM 53, 50–58 (2010)
2. Buyya, R., Yeo, C.S., Venugopal, S.: Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In: HPCC 2008: Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications, pp. 5–13. IEEE Computer Society, Washington, DC, USA (2008)
3. Amazon Elastic Compute Cloud (EC2), http://aws.amazon.com/ec2/
4. Nimbus, http://www.nimbusproject.org/
5. Opennebula, http://www.opennebula.org/
6. Wartel, R., Cass, T., Moreira, B., Roche, E., Manuel Guijarro, S.G., Schwickerath, U.: Image distribution mechanisms in large scale cloud providers. In: CloudCom 2010: Proc. 2nd IEEE International Conference on Cloud Computing Technology and Science, Indianapolis, USA (2010) (in press)
7. Andrzejak, A., Kondo, D., Yi, S.: Decision model for cloud computing under sla constraints. In: Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2010, pp. 257–266. IEEE Computer Society, Washington, DC, USA (2010)
8. Amazon elastic block storage (ebs), http://aws.amazon.com/ebs/
9. Nicolae, B.: BlobSeer: Towards efficient data storage management for large-scale, distributed systems. PhD thesis, University of Rennes 1 (November 2010)
10. Nicolae, B., Antoniu, G., Bougé, L., Moise, D., Carpen-Amarie, A.: Blobseer: Next-generation data management for large scale infrastructures. J. Parallel Distrib. Comput. 71, 169–184 (2011)
11. Bolze, R., Cappello, F., Caron, E., Daydé, M., Desprez, F., Jeannot, E., Jégou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E.G., Touche, I.: Grid'5000: A large scale and highly reconfigurable experimental grid testbed. Int. J. High Perform. Comput. Appl. 20, 481–494 (2006)
12. Nicolae, B., Bresnahan, J., Keahey, K., Antoniu, G.: Going Back and Forth: Efficient Multi-Deployment and Multi-Snapshotting on Clouds. In: HPDC 2011: The 20th International ACM Symposium on High-Performance Parallel and Distributed Computing, San José, CA United States (2011)
13. Gagné, M.: Cooking with linux: still searching for the ultimate linux distro? Linux J. 2007(161), 9 (2007)
14. Carns, P.H., Ligon, W.B., Ross, R.B., Thakur, R.: Pvfs: A parallel file system for Linux clusters. In: Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, GA, pp. 317–327. USENIX Association (2000)
15. Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D.E., Maltzahn, C.: Ceph: a scalable, high-performance distributed file system. In: Proceedings of the 7th symposium on Operating systems design and implementation, OSDI 2006, pp. 307–320. USENIX Association, Berkeley (2006)

16. Schmuck, F., Haskin, R.: Gpfs: A shared-disk file system for large computing clusters. In: Proceedings of the 1st USENIX Conference on File and Storage Technologies, FAST 2002, USENIX Association, Berkeley (2002)
17. Amazon Simple Storage Service (S3), http://aws.amazon.com/s3/
18. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: SOSP 2007: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, pp. 205–220. ACM, New York (2007)
19. Zhu, Q., Gelenbe, E., Qiao, Y.: Adaptive prefetching algorithm in disk controllers. Perform. Eval. 65, 382–395 (2008)

Introduction

Amitabha Bagchi, Olivier Beaumont, Pascal Felber, and Alberto Montresor

Topic chairs

Peer-to-peer (P2P) systems enable computers to share information and other resources with their networked peers in large-scale distributed computing environments. The resulting overlay networks are inherently decentralized, self-organizing, and self-coordinating. Well-designed P2P systems should be adaptive to peer arrivals and departures, resilient to failures, tolerant to network performance variations, and scalable to huge numbers of peers (tens of thousands to millions). As P2P research becomes more mature, new challenges emerge to support complex and heterogeneous decentralized environments for sharing and managing data, resources, and knowledge with highly dynamic and unpredictable usage patterns. This topic provides a forum for researchers to present new contributions to P2P systems, technologies, middleware, and applications that address key research issues and challenges.

This year, three papers have been accepted for publication in the P2P track. The general trend among submitted papers was on the study of the properties of P2P networks and their extensions to new services, rather than on the design of new overlays. Each paper was evaluated by four referees.

The paper *Asynchronous Peer-to-Peer Data Mining with Stochastic Gradient Descent* by Róbert Ormándi, István Hegedűs and Mark Jelasity from the University of Szeged, Hungary, proposes a method based on stochastic gradient search to support fully decentralized data mining, with no assumptions on the reliability or synchrony of communication. The idea of applying stochastic gradient descent for SVMs to P2P platforms is particularly original and, in keeping with current trends in computing, opens out the possibility of using a P2P system as a decentralized and dynamic database, thereby creating interesting perspectives for future applications.

The other two papers accepted speak directly to the notion of dynamism that underlies P2P networks by addressing the problem of churn. The first of these, *Evaluation of P2P Systems Under Different Churn Models: Why Should We Bother?* by Marc Sánchez-Artigas and Enrique Fernández-Casado from the Universitat Rovira i Virgili, Spain, helps place the evaluation of P2P systems on a more rigorous basis by investigating the relationships between four different models suggested in the literature for churn in P2P systems. The authors study statistical properties of these models and highlight their similarities and differences. The purpose of this work is to determine if there are significant variations between the models, and hence if they provide different insights when used to study P2P systems.

The paper *ChurnDetect: Gossip-based Churn Estimator for Large-Scale Dynamic Networks* by Andrei Pruteanu, Venkat Iyer and Stefan Dulman from Delft University of Technology, the Netherlands, presents an algorithm to detect in a

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 514–515, 2011.

distributed way the rate of nodes joining/leaving in multi-hop large scale networks, even in presence of nodes behind firewalls. The algorithm relies on gossip-based communications along with a periodic reset mechanism, and mixes ideas coming from the P2P and ad-hoc communities.

We would like to take the opportunity of thanking the authors who submitted a contribution, as well as the Euro-Par Organizing Committee, and the external referees with their highly useful comments, whose efforts have made this conference and this topic possible.

Combining Mobile and Cloud Storage for Providing Ubiquitous Data Access*

João Soares and Nuno Preguiça

CITI/DI-FCT-Univ. Nova de Lisboa
Quinta da Torre, Portugal

Abstract. Users increasingly own, and use, multiple computing devices. To be able to access their personal data, at any time and in any device, users usually need to create replicas in each device. Managing these multiple replicas becomes an important issue.

In this paper we present the FEW Phone File System, a data management system that combines mobile and cloud storage for providing *ubiquitous* data access. To this end, our system takes advantage of the characteristics of mobile phones for storing a replica of a user's personal data, thus allowing these devices to be used as *personal and portable file servers*. As users tend to always carry their mobile phone with them at all times, these replicas are the basis for providing high data availability, and keeping replicas *automatically* synchronized.

Our system also uses other replicas located in web servers and cloud storage systems, to reduce the volume of data stored, and transferred to/from mobile phones, by maintaining only the information needed to obtain them.

1 Introduction

Users increasingly own, and use, multiple computing devices, from desktop computers, to laptops, tablets, consoles and mobile phones. In such an environment, data availability is an important issue, as users want to access their personal data everywhere, independently of their current machine or location.

To address this problem, users tend to rely on at least one of the following available solutions: *i*) on-line storage services (e.g. Dropbox [1], Google Docs [3]), and/or *ii*) portable storage devices (e.g. USB flash drives). While both solutions aim at providing "ubiquitous" storage space, they also force users to deal with additional problems. On-line storage services force users to trust third parties for storing their personal data. Both solutions force users to maintain synchronized replicas of the stored data, for minimizing losses due to possible device failure or for guaranteeing access in case of network disconnection. In many solutions (e.g. Google Docs, USB flash drives), synchronization is done manually, and only strict discipline avoids replicas from diverging.

* This work was partially supported by CITI and FCT/MCTES project # POSC/EIA/59064/2004, with Feder funding. João Soares was partially supported by CITI and FCT/MCTES research grant # SFRH/ BD/ 62306/ 2009.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 516–527, 2011.

In this paper, we present the FEW Phone File System (FEW), a data management system for providing high data availability to mobile users. While sharing some of the goals and solutions with existing systems [19,9,12,2,16,15], FEW provides a unique solution for offering users *ubiquitous* access to their personal data, independently of their location, device and/or network connectivity. To this end, FEW builds on the characteristics of current mobile phones, taking advantage of their *i*) storage capacity; *ii*) wireless communication capabilities (Wi-Fi and/or Bluetooth); *iii*) mobility of such devices.

In FEW, the mobile phone of a user acts as his *personal and portable file server*, storing and maintaining replicas of his personal files. This allows our system to provide ubiquitous access to these files, since users tend to carry their mobile phones with them at all times. The Wi-Fi and Bluetooth communication capabilities guarantee access to these replicas from any computer, in a location independent manner, without the hurdles of needing extra cables. The system includes an optimistic replication solution, allowing for replicas to be created and accessed when needed, including a *novel scalable update tracking solution*, and a *reconciliation algorithm based on commutative operations*.

As a large number of users' files are currently stored, or are obtained from remote web sources, FEW includes a data source verification mechanism to record the alternative sources for each file. This allows the integration of online storage systems as alternative data sources for each file, achieving both the reduction of communications, as well as data that needs to be stored in the mobile phone, while still providing high data availability. Additionally, FEW includes a data transcoding mechanism for minimizing storage consumption due to multimedia data that needs to be stored in the mobile phone - these two mechanisms allow, for example, the mobile phone to store only thumbnails of photo collections, while keeping information on how to obtain full fidelity photos from on-line systems, like Flickr, Facebook, etc.

The remainder of this paper is organized as follows: Section 2 presents the design of our systems; Section 3 details the advanced mechanisms to address limitations of using mobile phones; Section 4 presents an evaluation of the system; Section 5 discusses related work; and Section 6 concludes the paper with some final remarks.

2 System Design

The FEW Phone File System (FEW) is a data management system designed to provide high data availability to users, allowing them permanent access to their personal data, across multiple computers, independently of location, ownership, and network connectivity. In this section we present the system general design and synchronization algorithms.

2.1 Architecture

FEW is based, primarily, on a client-server architecture, where mobile phones act as *personal, and portable, file servers*, storing replicas of the personal data

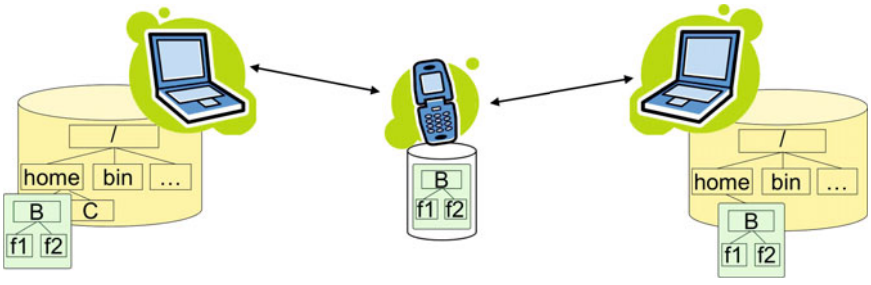


Fig. 1. FEW Phone File System architecture

of some user. A typical FEW configuration includes a set of computing devices, called nodes, accessed by a common user, and, at least, one mobile phone, as depicted in Figure 1. In the remainder of this paper, a mobile phone acting as a file server will be addressed as a *mobile server*.

Each node is responsible for managing its replicas of the user's files, which are organized into collections named *containers*. Containers are collections of files stored under the same common path name, and can be seen as subtrees of a file system, similar to volumes in Coda [18]. FEW allows containers to be created, in any client node, from any existing directory in the file system, or when replicating an existing container on a new node.

Applications access these files as any other files, i.e., using the file system interface. This allows FEW to provide data availability, without forcing applications to be modified. To this end, clients intercept and handle all file system calls executed on the files managed by the system. In the current prototype, we rely on Fuse [6] (and Fuse-J java-bindings) for implementing the client side. Typically, mobile phones only include the server component, while computer nodes run both the client and the server component of the system. This allows computers to synchronize files with mobile servers, using a client-server model, and also with other active client nodes, in a peer-to-peer fashion, as described later.

FEW fully replicates a container's complete namespace in every replica, thus allowing users to have a coherent view of its contents in every node. File contents, on the other hand, do not need to be fully replicated. As explained later, the system includes several mechanisms that allow the system to use partial or full replication policies. If necessary, file contents can be fetched on-demand when applications access files not locally replicated.

For improved performance, we use optimistic replication techniques [5] to allow replicas of the files to be created in any client computer that accesses them[1]. This way, files can be modified in any computer without requiring immediate server communication, thus addressing performance and energy limitations of mobile phones. Data consistency is maintained by a periodic synchronization

[1] Users can decide in which computer these replicas can be created, and in which replicas can be long-lived, thus addressing privacy concerns.

process, used for propagating updates from clients to servers, and vice-versa. This process includes a reconciliation mechanism for resolving possible conflicts due to concurrent updates.

2.2 Synchronization Process

To preserve consistency, allowing users to always access the latest version of their files, FEW relies on a periodic synchronization process for propagating updates between replicas of a container. Replicas are synchronized using an epidemic approach, by establishing pair-wise communication sessions. Although any node can synchronize with any other node, mobile servers tend to function as mediators for propagating updates between replicas.

This approach has several benefits. First, it allows our system to provide eventual consistency, i.e., over time, an update executed in a node will be propagated to all other nodes, even to nodes that have no network connectivity (besides short range connectivity to the mobile phone). Second, since users are expected to always carry their mobile phones with them at all times, maintaining the most recent version of a container on a mobile server allows FEW to always provide users access to the most recent version of their data.

A synchronization session is automatically initiated whenever a node detects a nearby mobile server. During this session, local updates, i.e., updates performed on a computer node, are propagated to the mobile server, while missed updates are propagated from the server to that node. This process is re-executed periodically while a mobile server is nearby. As a result, we expect a mobile server to always store the most recent version of its containers.

The synchronization process also allows nodes to gain knowledge of other existing replicas. Whenever a container is replicated on a node, the node's address is stored on the server, thus creating additional sources for obtaining the data. This information is exchanged during the synchronization process.

This allows the basic synchronization process to be extended using node-to-node synchronization. This way replicas are kept synchronized, even if the mobile server cannot be used to efficiently propagate updates among them. In this case, a node uses this information to form an overlay network for exchanging updates with existing replicas. Two situations for using such an approach have been identified. First, when the user forgets his mobile phone, it is obvious that synchronization must be performed using peer-to-peer communication. Second, when files are too large it might be too costly to store them in the mobile phone.

Synchronization Algorithm. FEW uses a two stage synchronization algorithm. First stage synchronizes the name space, i.e., it propagates unknown directory updates, while the second stage synchronizes data, i.e., files contents.

Object identifiers. Internally, all objects handled by FEW are addressed by unique identifiers (UID), assigned to each object (file or directory) when created, or during the container creation process, remaining the same during the

lifetime of that object, independently of name changes. Other information (i.e. meta-data) is also maintained for each object in a container. This information includes common file system attributes, such as file names, access permissions, etc., and system-specific attributes, such as information for tracking dependencies among versions, a digest of its contents, a list of alternative file sources, etc. This information is essential for the synchronization process.

Name Space Synchronization. For synchronizing the name space of a container, FEW maintains a log with name space updates, and an associated version vector [10]. Each log entry includes all the necessary information for reproducing the operation in other nodes, i.e., a time stamp of the operation, the type of operation (create, delete, rename), the UID of the updated object, its version, and the UID of the parent directory, while the version vector allows for recording the number of updates performed on each replica of the container. During synchronization, each node exchanges version vectors, thus allowing missed updates to be determined. The respective log entries are then exchanged, and reproduced.

Concurrent updates are deterministically resolved using the principles of commutative replicated data types [14]. Concurrent operations have been designed in a way that a deterministic result is obtained independent of their execution order, thus guaranteeing that all replicas converge to the same state.

Data Synchronization. For tracking data dependencies, FEW uses a novel approach. Updated files are initially signaled using an *'updated'* flag associated with the file and all parent directories. During synchronization, each file marked as updated is synchronized with the peer and only at this moment the version vectors are modified. Unlike the traditional approach, if only one replica has been modified and only one peer has an entry in the version vector, it is that entry that is updated to record the new version, independently of the replica in which the file has been changed. This approach allows the reduction of the number of entries in version vectors. In our scenario, where we expect a node to synchronize with a single (or a small number of) mobile servers, it is possible to keep the number of entries in version vectors equal to the number of mobile servers, as the updates performed in a node will always be reflected in the server's entry.

Additionally, in FEW, version vectors of directories summarize the updates in the subtree. Thus, during synchronization it is only necessary to propagate the version vector of the root to be able to determine which files need to be synchronized. This allows the volume of exchanged data, during synchronization, to be proportional to the number of updated files, rather than the total number of files in a container. Concurrent updates to the same file are automatically resolved by deterministically selecting a predominant file version. Additional details of the synchronization process can be found in [8,7].

3 Advanced Mechanism for Using Mobile Phones

Relying on mobile phones as file servers, imposes the following limitations that need to be addressed: *i*) limited communication bandwidth, leading to higher

data access times, compared to local hard drives; *ii*) insufficient storage capacity; *iii*) limited energy resources; and *iv*) low reliability of mobile phones [17]. Next we briefly present some of the solutions used to address these problems.

3.1 Minimizing Communications

As in any distributed file system, accessing all files directly on the file server proves impractical [11,18]. This results from limited bandwidth and high latency limitations, which increase access times, when compared to local hard drives. Also, since in our case the file server is a mobile phone, server communications are highly energy consuming operations, thus minimizing them in essential.

As explained before, FEW addresses this problem by creating temporary or long-lived replicas in the clients nodes. The system relies on periodic synchronization to keep replicas up-to-date, exchanging, during these interactions, information that is proportional to the number of updated replicas in the system (as in Cimbiosys [15]). This approach helps minimizing communications with the mobile phone, for both file access and synchronization, providing faster access times to the users files, improving the user experience, while also minimizing power consumption of due to communications.

3.2 Improving Storage Usage

Although mobile phones have increasingly larger amounts of storage capacity, they offer reduced storage space when compared to the current capacity of hard disks, and the amount of personal data users tend to store. This way, we expect to be impossible storing replicas of all files kept by an user. FEW addresses this limitation in different ways.

Container Set. Since FEW is designed to manage a set of containers, we believe users would assign, to different containers, different types of data. For instances, one container would have personal documents, like word documents, text files, spread sheets, etc., while others would contain video files, audio files, etc.. This way users can select which containers should always be available, i.e., replicated on the mobile server, and which are less important. This can be manually adjusted, thus allowing users to always keep relevant data "close by".

Integrating Remote Storage Systems. Users tend to store significant number of resources obtained from remote sites, or that have additional remote copies. Among this data, we can include, not only, files downloaded from Web sites, for example, *'pdf'* files, but also data uploaded by the user to Internet workspaces, or other remote storage systems, such as cloud storage services. Since these files tend to be preserved for long periods of time, and, in some cases, remain unchanged during this time, there is no reason why these remote sites cannot be used as alternative data sources for obtaining data.

With this in mind, FEW includes a Data Source Verification (DSV) mechanism that allows it to automatically keep track of additional sources for the files

managed by the system. DSV is a modular, plugin based component, allowing different services to be used as alternative data sources, simply by choosing the corresponding plugin, including Web servers, CVS servers, and Cloud storage systems. For instance, the current HTTP generic plugin acts as a proxy, monitoring HTTP connections. For each HTTP operation, it computes the digest for the obtained result, storing it, as well as the associated URL, for a short period of time.

When a file is created or updated in a container, the system checks with the DSV if the new contents were remotely obtained (by comparing its digest). If those contents were obtained remotely, the source URL is added, as meta-data, to the corresponding file. This information is used during the synchronization process, allowing our system to retrieve remotely stored files directly from these sources. This way, FEW prevents those contents from being stored on the mobile server, storing only their meta-data. FEW can still provide high data availability for these files, by leveraging on the alternative data sources associated with those contents for retrieving them. Doing so allows our system to reduce storage requirements.

Some DSV plug-ins require more complex solutions the one used in the HTTP plugin. For example, a CVS plugin requires additional information to work correctly, since remote files have an associated version. Thus, the CVS plugin checks, not only if a file's pathname is under control of a CVS server, but also if the version of the remote file is identical to the one stored locally. If so, the CVS server can be used as an alternative source for retrieving the file. This information is added as meta-data, thus allowing our system to retrieve these contents during synchronization.

Currently we are developing a plugin for integrating FEW with Cloud storage services, such as Dropbox [1]. As in the CVS plugin, this plugin requires additional information from the user, such as login credentials. With this information, the plugin can use the Dropbox API to create an authenticated session, using it for browsing the contents remotely stored.

Maintaining Consistency of Remote Replicas. Our system also allows the automatic update of remote replicas whenever a local replica is updated. To this end, the DSV plugin includes methods to allow these updates to be performed. For example, the HTTP plugin relies on using HTTP POST and PUT operations.

During synchronization, if remote replicas cannot be updated, either because updates are not supported by some plugin or were unsuccessful, the mobile phone is used to store the most recent version of the file. Our experience says, that most resources downloaded from Web pages are kept unchanged by users. Only personal data is regularly updated, and this information is normally stored on services that allow remote update operations to be performed.

Multimedia data. Another mechanism designed to reduce storage requirements relates to multimedia files. With the proliferation of digital cameras, and other multimedia devices, users tend to store significant amounts of multimedia contents. Since these files tend to consume considerable amounts of storage space,

it is unreasonable to store them on a mobile phone. Although some of these files can be addressed using the previous mechanism, others can not.

For these files, FEW relies on data transcoding techniques [13]. These techniques allow for multimedia contents to be adapted accordingly to the available resources on mobile servers, storing only lower fidelity versions of multimedia files. FEW automatically performs data transcoding during synchronization, immediately before transferring multimedia data to the mobile server, allowing the level of fidelity to be specified for each device.

The Data Transcoding (DTC) module includes a set of plug-ins for transcoding different types of data. In our current prototype, we include support only for transcoding a very limited number of multimedia formats, using existing applications. We also include support for transcoding generic files to an empty file. This provides support for implementing partial replication of a container. Additionally, users can configure the transcoding parameters for each supported file type. allowing these parameters to be changed according to the available storage space. For example, users can define a "more aggressive" level of transcoding when the available storage space is low.

Combining this mechanism with the previously described DSV mechanism, allows lower-fidelity versions to be used only when they are satisfactory, or when no connectivity to an alternative source is available, since the full fidelity versions can be downloaded from the alternative sources, whenever needed.

4 Evaluation

In this section we present an evaluation of the FEW Phone File System, focusing on the advanced modules: DSV and DTC. We evaluate both the importance of the modules and their performance.

4.1 Importance of DSV and DTC

For evaluating the importance of DSV and DTC modules, we have studied the percentage of user files obtained from remote web sources, which can be handled by DSV; and the percentage and relative volume of multimedia files, which can be addressed by DTC.

As we could not run our system for a long enough period and with a large enough number of users to obtain relevant statistics, we have obtained statistics analyzing the personal data of 5 users. For determining which files had been obtained from a remote source, we have used the application-specific attributes with the URL added by the Safari web browser in Mac OS X. Obviously, this is a lower-bound estimation of the files obtained remotely, as some of these users reported that they also use other browsers that do not add this attribute. Additionally, files obtained using other programs are also not computed.

Table 1 shows the type and the *relative volume* of data stored by users (office data includes, not only word documents, spreadsheets and presentations, but also digital documents such as "pdf" files). As we expected, users tend to store

Table 1. Statistics on personal files

Data Type	Relative Volume	Data Source	
		Locally Created	Web Transferred
Multimedia	56%	98%	2%
Archives	17%	68%	32%
Office	6%	53%	47%
Sources	0%	95%	5%
Other	21%	79%	21%

considerable amounts of multimedia data. In average we found that more than 50% of the files stored by current users are multimedia data. Since these files tend to be large, the use of the Data Transcoding module is essential for allowing FEW to achieve its goals.

Table 1 also presents the data obtained from the Internet using Safari. From these results, it is possible to observe that approximately 14% of the personal data has been obtained from remote sites. The largest amount of downloaded files are office and archive files, which results from the fact that users tend to store large numbers of digital documents, such as "pdf" files, and that archives are largely used to enclose other resources, such as source files. The low percentage for multimedia files can be justified by the fact that most of these files are: (1) copies of data users own(e.g. CDs) or have created (e.g. photos from cameras); (2) obtained using applications other than browsers - e.g. iTunes and peer-to-peer applications. For source files, we know that some users have a large percentage of their files in version control systems, but we could not quantify the percentage. For some multimedia files - e.g. photos, it is common for users to store them in remote sites. Thus, the files that could be handled by DSV is expected to be much higher.

These results show that keeping track of alternative sources can reduce the need for storing the contents of files, thus showing that the DSV module can be a good solution for minimizing the data that needs to be stored on a mobile server. Additionally, as multimedia files seem to be the category that has less additional sources, DTC seems a good complement to DSV.

4.2 Performance Impact of the DSV and DTC

In this section we present performance results obtained when synchronizing nodes using the DSV and the DTC modules. The results were obtained using our Java/Android prototype. The mobile server runs on an HTC Magic mobile phone running Android 1.5, while the client nodes are desktop computers with an Intel Core 2 Duo T8300 @ 2.4 GHz process, running Linux Ubuntu 9.10. Devices communicate using a Wi-fi network.

To determine the impact of the DSV module, we measured the time for synchronizing a new node with a mobile server, and those obtained performing the same operation using a third node as an additional source for transferring data. The results, presented in Table 2, show performance gains when using the DSV

Table 2. Synchronization times with DSV

Num. files	Total Size	Sync. time	
		w/ DSV	wo/ DSV
83	450 KB	5.984s	6.502s
97	2 MB	7.806s	9.171s
357	5 MB	25.467s	35.689s

Table 3. Synchronization times with DTC

Num. files	Total Size	Sync. time	
		w/ DTC	wo/ DTC
1	9 MB	5.211s	8.775s
2	8 MB	9.216s	18.556s

modules, even for small data volumes. The actual impact on performance is directly related with the bandwidth and latency of the connection with the server. Besides the performance improvement, this module allows for a considerable reduction of the volume of data transferred from the mobile phone to the node, also reducing power consumption of the mobile device.

For evaluating the impact of the DTC module, we measured the time for synchronizing a container with one and two 14 mega-pixels (4672x3104 pixels) digital photos from a node to a mobile server. Table 3 compares the values when propagating the full fidelity photos and when using the DTC module (the two photos were transcoded to a resolution of 1024x680 pixels, and color depth from 32 to 24 bits per pixel, for a size reduction from 9 MBytes to 63 kBytes). Results show a significant performance improvement and, above all, present a significant reduction in the volume of data that is transferred to and from the server. Combining these two modules allows users to access their full-fidelity data, while reducing the volume of data transferred and stored on the mobile phone, thus improving performance while reducing power consumption.

5 Related Work

Some distributed file systems (e.g. Ficus [4], Coda [18]) include support for mobile computing environments. However, the complexity associated with setting up a new server and using it in a network with private networks and firewalls lead most users to prefer using portable storage devices to transport their data.

Other solutions for addressing similar problems have been proposed recently in the literature. PersonalRAID [19] allows a portable storage devices to be used for propagating updates among several personal replicas. However, this approach makes it impossible for a user to access all his data in a new computer. Footloose [9] introduces the concept of physical eventual consistency, allowing portable devices to be used to automatically propagate updates amongst replicas. FEW extends the approach of Footloose by allowing clients to obtain data contents from other replicas (even outside the system), thus minimizing the requirements of the mobile devices. Also, we allow multimedia data to be transported efficiently.

EnsemBlue [12] supports the integration of data created in consumer devices into a common namespace, transcoding data based on application needs, using what the authors describe as *persistent queries*. FEW uses a similar approach

during the synchronization process for adapting multimedia contents based on the specifications of the user.

Unmanaged Internet Architecture [2] allows users to provide personal names to their devices and data, providing users access this data, from any device, using these names. Perspective [16] and Cimbiosys [15] provide replication solutions, allowing users to keep data replicas in multiple devices, based on the semantic description of that data. Contrarily to our system, these have no mechanism to efficiently store data in mobile devices, other than partial replication.

New cloud storage services (e.g. Dropbox) offer functionalities similar to *traditional* distributed file systems. While these minimize the complexity of setting up file servers, their use is not fully transparent (since data needs to be stored under specific directories), also requiring users to trust on third-party organizations. Additionally, these services are usually only used to store a small subset of users' files. Our system can integrate these services for improving data availability.

6 Final Remarks

FEW is a data management system designed to allow users access their personal data in any machine, independently of location and network connectivity. To this end, FEW has been designed to take advantage the storage capacity and wireless communication capabilities of current mobile phones, for maintaining replicas of the users data (the most up-to-date version). The optimistic replication approach allows for long lived replicas to be created, and accessed, whenever and wherever needed. For guaranteeing consistency, we have proposed novel techniques for tracking dependencies among replicas that can improve the scalability of commonly used version vectors.

FEW addresses the limitations of mobile devices, in particular of the storage capacity of mobile phones. A Data Transcoding module allows FEW to deal with the volume of multimedia data stored by current users, reducing storage requirements by storing lower fidelity versions of these files. A Data Source Verification module allows the system to record alternative sources for the files stored in a container. This approach explores the common case where the files stored were obtained from the Web, or are stored in some remote server. To our knowledge, FEW is the first system to combine mobile and cloud storage to provide high availability while reducing the volume of data that needs to be stored and still offering an single view of a container in all devices the users uses. This mechanism help improving the performance of the system, while reducing power consumption by reducing communications with the mobile phone.

By combining both modules, FEW allows clients to always access full-fidelity contents. This is a new feature when compared with previous solutions that use data transcoding. The obtained evaluation results suggest that the combination of DTC and DSV are important for achieving the goals of the system, since considerable percentages of files, stored by the users, are multimedia files and/or have additional Web sources.

References

1. Dropbox: Dropbox (2011), http://www.dropbox.com/
2. Ford, B., Strauss, J., Lesniewski-Laas, C., Rhea, S., Kaashoek, F., Morris, R.: Persistent personal names for globally connected mobile devices. In: Proc. of the 7th Symp. on Operating Systems Design and Implementation, pp. 233–248 (2006)
3. Google: Google docs (2009), http://docs.google.com/, http://docs.google.com/
4. Guy, R.G., Heidemann, J.S., Mak, W., Popek, G.J., Rothmeier, D.: Implementation of the Ficus Replicated File System. In: USENIX Conf. Proc., pp. 63–71 (1990)
5. Hac, A., Jin, X., Soo, J.H.: Algorithms for file replication in a distributed system. J. Syst. Softw. 14(3), 173–181 (1991)
6. Henk, C., Szeredi, M., Pavlinusic, D., Dawe, R., Delafond, S.: Filesystem in Userspace (FUSE) (December 2008), http://fuse.sourceforge.net/
7. Soares, J.: FEW Phone File System. Master's thesis, Faculdade de Ciências e Tecnologia (April 2009)
8. Soares, J., Preguiça, N.: Proving Ubiquitous Access to the User's Data Combining Mobile and Cloud Storage. Tech. Rep. 04/2011, CITI / DI-FCT-Univ. Nova de Lisboa (May 2011)
9. Paluska, J., Saff, D., Yeh, T., Chen, K.: Footloose: a case for physical eventual consistency and selective conflict resolution. In: Proc. Fifth IEEE Workshop on Mobile Computing Systems and Applications, pp. 170–179 (October 2003)
10. Parker, D.S., Popek, G.J., Rudisin, G., Stoughton, A., Walker, B.J., Walton, E., Chow, J.M., Edwards, D., Kiser, S., Kline, C.: Detection of Mutual Inconsistency in Distributed Systems. IEEE Trans. Softw. Eng. 9(3), 240–247 (1983)
11. Pawlowski, B., Juszczak, C., Staubach, P., Smith, C., Lebel, D., Hitz, D.: NFS version 3 design and implementation. In: Proc. of the Summer USENIX Conf., pp. 137–152 (1994)
12. Peek, D., Flinn, J.: Ensemblue: integrating distributed storage and consumer electronics. In: Proc. of the 7th Symp. on Operating Systems Design and Implementation, pp. 219–232 (2006)
13. Phan, T., Zorpas, G., Bagrodia, R.: Middleware support for reconciling client updates and data transcoding. In: Proc. Int. Conf. on Mobile Systems, Applications, and Services, MobiSys (2004)
14. Preguiça, N., Marques, J.M., Shapiro, M., Letia, M.: A commutative replicated data type for cooperative editing. In: Proc. of the 2009 IEEE Int. Conf. on Distributed Computing Systems, pp. 395–403 (2009)
15. Ramasubramanian, V., Rodeheffer, T.L., Terry, D.B., Walraed-Sullivan, M., Wobber, T., Marshall, C.C., Vahdat, A.: Cimbiosys: a platform for content-based partial replication. In: NSDI 2009: Proc. of the 6th USENIX Symp. on Networked systems design and implementation, pp. 261–276 (2009)
16. Salmon, B., Schlosser, S.W., Cranor, L.F., Ganger, G.R.: Perspective: semantic data management for the home. In: FAST 2009: Proccedings of the 7th Conf. on File and Storage Technologies, pp. 167–182 (2009)
17. Satyanarayanan, M.: Fundamental challenges in mobile computing. In: Proc. of the ACM Symp. on Principles of Distributed Computing, pp. 1–7 (1996)
18. Satyanarayanan, M.: The evolution of coda. ACM Trans. Comput. Syst. 20, 85–124 (2002), http://doi.acm.org/10.1145/507052.507053
19. Sobti, S., Garg, N., Zhang, C., Yu, X., Arvind Krishnamurthy, R., Wang, O.Y.: PersonalRAID: Mobile Storage for Distributed and Disconnected Computers. In: Proc. First Conf. on File and Storage Technologies, pp. 159–174 (2002)

Asynchronous Peer-to-Peer Data Mining with Stochastic Gradient Descent*

Róbert Ormándi[1], István Hegedűs[1], and Márk Jelasity[2]

[1] University of Szeged, Hungary
{ormandi,ihegedus}@inf.u-szeged.hu
[2] University of Szeged and Hungarian Academy of Sciences, Hungary
jelasity@inf.u-szeged.hu

Abstract. Fully distributed data mining algorithms build global models over large amounts of data distributed over a large number of peers in a network, without moving the data itself. In the area of peer-to-peer (P2P) networks, such algorithms have various applications in P2P social networking, and also in trackerless BitTorrent communities. The difficulty of the problem involves realizing good quality models with an affordable communication complexity, while assuming as little as possible about the communication model. Here we describe a conceptually simple, yet powerful generic approach for designing efficient, fully distributed, asynchronous, local algorithms for learning models of fully distributed data. The key idea is that many models perform a random walk over the network while being gradually adjusted to fit the data they encounter, using a stochastic gradient descent search. We demonstrate our approach by implementing the support vector machine (SVM) method and by experimentally evaluating its performance in various failure scenarios over different benchmark datasets. Our algorithm scheme can implement a wide range of machine learning methods in an extremely robust manner.

1 Introduction

Data aggregation has long been considered an important aspect of a peer-to-peer (P2P) system. In the past decade, an extensive literature has accumulated on the subject. Research has mainly focused on very simple statistics over fully distributed databases, such as the average of a distributed set of numbers [18,15], separable functions [22], or network size [20]. General SQL queries have also been implemented in this fashion [25]. The main attraction of the known fully distributed (mostly gossip-based) algorithms for data aggregation is their impressive simplicity and efficiency, combined with robustness to benign failure.

Simple statistics or queries are very useful, but often more is needed. For example, for a P2P platform that offers rich functionality to its users including

* M. Jelasity was supported by the Bolyai Scholarship of the Hungarian Academy of Sciences. This work was partially supported by the Future and Emerging Technologies programme FP7-COSI-ICT of the European Commission through project QLectives (grant no.: 231200).

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 528–540, 2011.

spam filtering, personalized search, and recommendation [24,1,3], or for P2P approaches for detecting distributed attack vectors [5], complex predictive models have to be built based on fully distributed, and often sensitive, data. At the same time, it would be highly desirable to build these models without sacrificing any of the nice properties of the aggregation algorithms mentioned above.

In sum, we need to find fully distributed, efficient, and lightweight data mining algorithms that make no or minimal assumptions about the synchrony and reliability of communication, work on fully distributed datasets without collecting the data to a central location, and make the learned models available to all participating nodes. Our contribution is that we propose a method based on stochastic gradient search that meets these requirements. In stochastic gradient search, the model of the data is gradually evolved as it is exposed to random records from the training dataset. A wide range of models—including artificial neural networks, and support vectors—can be evolved in this fashion. Stochastic gradient methods can naturally be implemented in a gossip fashion, where models perform a random walk over the network, while converging to an optimal model. Furthermore, we can even improve the performance of sequential stochastic gradient methods, exploiting the fact that there are many interacting models making random walks at the same time.

2 System and Data Model

We assume that the system consists of a potentially very large number of nodes, typically personal computing devices such as PCs or mobile devices. Every node has a network address. Every node can send messages to every other node, provided the address of the target node is available. We assume that messages can have arbitrary delays, and messages can be lost as well. In addition, nodes can join and leave at any time without warning, thus leaving nodes and crashed nodes are treated identically. Leaving nodes can join again, and while offline, they may retain their state information.

The only middleware service our algorithm relies on is the peer sampling service [16]. Through this service, each node can request uniform random samples of the nodes in the network that are likely to be online at the time of the request. The API of the service consists of a local function GETRANDOMPEER(), which returns a random node address. Many implementations of the peer sampling service are known. In this paper we apply the NEWSCAST protocol, a gossip based implementation [16]. The overhead of NEWSCAST consists of sending one message of a constant size to a random node periodically. This protocol has been extended to deal with uneven request rates at different nodes, as well as uneven distributions of message drop probabilities [30].

As for the data distribution model, we assume that each node stores exactly one data record. These records are of the same type (contain the local values of the same features) at each node. This extreme distribution model allows us to support applications that require extreme privacy where, for example, the profile of a user never leaves the computer of the user.

3 Background

The basic problem of *supervised binary classification* can be defined as follows. Let us assume that we are given a labeled database in the form of pairs of feature vectors and their correct classification, i.e. $(x_1, y_1), \ldots, (x_n, y_n)$, where $x_i \in \mathbb{R}^d$, and $y_i \in \{-1, 1\}$. The constant d is the *dimension* of the problem (the number of features). We are looking for a *model* $f : \mathbb{R}^d \to \{-1, 1\}$ that correctly classifies the available feature vectors, and that can also *generalize* well; that is, which can classify unseen examples too. For testing purposes, the available data is often partitioned into a *training set* and a *test set*, the latter being used only for testing candidate models.

Supervised learning can be thought of as an optimization problem, where we want to maximize prediction performance, which can be measured via, for example, the number of feature vectors that are classified correctly over the training set. The search space of this problem consists of the set of possible models (the *hypothesis space*) and each method also defines a specific search algorithm (often called the *training algorithm*) that eventually selects one model from this space.

Stochastic gradient search is one such generic search algorithm. Without going into too much detail, the basic idea is that we iterate over the training examples in a random order repeatedly, and for each training example, we calculate the gradient of the error function (which describes classification error), and modify the model along this gradient to reduce the error on this particular example. At the same time, the step size along the gradient is gradually reduced. In many instantiations of the method, it can be proven that the converged model minimizes the *sum* of the errors over the examples [8].

Let us now turn to support vector machines (SVM), the learning algorithm we apply in this paper [6]. In its simplest form, the SVM approach works with the space of linear models to solve the binary classification problem. Assuming a d dimensional problem, we want to find a $d-1$ dimensional separating hyperplane that maximizes the *margin* that separates examples of the two class. The margin is defined by the hyperplane as the sum of the minimal perpendicular distances from both classes.

Equation (1) states the formal SVM optimization problem, where $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$ are the parameters of model, namely the norm of the separating hyperplane and the bias parameters, respectively. Furthermore, ξ_i is the slack variable of the ith sample, which can be interpreted as the amount of misclassification error of the ith sample, and C is a trade-off parameter between generalization and error minimization.

$$\begin{aligned} \min_{w,b,\xi_i} \quad & \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\xi_i \\ \text{s.t.} \quad & y_i(w^T x_i + b) \geq 1 - \xi_i \quad \text{and} \quad \xi_i \geq 0 \quad (\forall i: \ 1 \leq i \leq n) \end{aligned} \tag{1}$$

The Pegasos algorithm is an SVM training algorithm, based on a stochastic gradient descent approach [27]. It directly optimizes a form of the above defined,

so-called primal optimization task. We will use the Pegasos algorithm as a basis for our distributed method. In this primal form, the desired model w is explicitly represented, and is evaluated directly over the training examples. Since in the context of SVM learning this is an unusual approach, let us take a closer look at why we decided to work in the primal formulation. The standard SVM algorithms solve the dual problem instead of the primal form [6]. The dual form is

$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i y_i \alpha_j y_j x_i^T x_j \\ \text{s.t.} \quad & \sum_{i=1}^{n} \alpha_i y_i = 0 \quad \text{and} \quad 0 \leq \alpha_i \leq C \quad (\forall i : \ 1 \leq i \leq n), \end{aligned} \tag{2}$$

where the variables α_i are the Lagrangian variables. The Lagrangian variables can be interpreted as the weights of the training samples, which specify how important the corresponding sample is from the point of view of the model.

The primal and dual formalizations are equivalent, both in terms of theoretical time complexity and the optimal solution. Solving the dual problem has some advantages; most importantly, one can take full advantage of the kernel-based extensions (which we have not discussed here) that introduce nonlinearity into the approach. However, methods that deal with the dual form require frequent access to the entire database to update α_i, which is unfeasible in our system model. Besides, the number of variables α_i equals the number of training samples, which could be orders of magnitude larger than the dimension of the primal problem, d. Finally, there are indications that applying the primal form can achieve a better generalization on some databases [4].

4 Related Work

Here we do not consider parallel data mining algorithms. This field has a large literature, but the rather different underlying system model means it is of little relevance to us here. Apart from the related work mentioned in the Introduction, we focus here on fully distributed data mining algorithms.

We divide fully distributed data mining algorithms into two main groups. In the first group we can find approaches that do not build models or that build minimal models like the unsupervised learners [28] or the collaborative filtering based recommender algorithms [23,2,12,31]. These types of approaches mainly use other well-studied P2P services like aggregation [18,15] with perhaps some kind of overlay support like T-Man [14]. We stress that these algorithms do not implement optimization or generative probability modeling like most of the state-of-the-art machine learning algorithms do.

In the second group there are algorithms that do build models, but require services such as round-based synchronization, and other reliability assumptions (e.g. [7]). As for SVM algorithms, we are aware of only one comparable P2P SVM implementation called Gadget SVM [13]. This algorithm applies the Push-Sum algorithm [18], but it requires round synchronization as well.

Algorithm 1. P2P Stochastic Gradient Descent Algorithm

```
1: initModel()
2: loop
3:     wait(Δ)
4:     p ← selectPeer()
5:     send currentModel to p
6: procedure onReceiveModel(m)
7:     m ← updateModel(m)
8:     currentModel ← m
9:     modelQueue.add(m)
```

Algorithm 2. P2Pegasos

```
1: procedure updateModel(m)
2:     η ← 1/(λ · m.t)
3:     if y ⟨m.w, x⟩ < 1 then
4:         m.w ← (1 − ηλ)m.w + ηyx
5:     else
6:         m.w ← (1 − ηλ)m.w
7:     m.t ← m.t + 1
8:     return m
9: procedure initModel
10:    m.t ← 0
11:    m.w ← (0, ..., 0)^T
12:    send model(m) to self
```

Hence, to the best of our knowledge there is no other learning approach designed to work in our fully asynchronous system model, and which is capable of producing a large array of state-of-the-art models.

5 The Algorithm

The skeleton of the algorithm we propose is shown in Algorithm 1. This algorithm is run by every node in the network. When joining the network, each node generates a model via initModel(). After the initialization each node starts to periodically send its current model to a random neighbor that is selected using the peer sampling service (see Section 2). When receiving the model, the node updates it using a stochastic gradient descent step based on the training sample it stores, and subsequently it stores the model. The model queue can be used for voting, as we will explain later.

Recall that we assumed that each node stores exactly one training sample. This is a worst case scenario; if more samples are available locally, then we can use them all to update the model without any network communication, thus speeding up convergence.

In this skeleton, we do not specify what kind of models are used and what algorithms operate on them. For example, a model is a $d-1$ dimensional hyperplane in the case of SVM, as described earlier, which can be characterized by a d dimensional real vector. In other learning paradigms other model types are possible. To instantiate the framework, we need to implement initModel() and updateModel(). This can be done based on any learning algorithm that utilizes the stochastic gradient descent approach. In this paper we will focus on the Pegasos algorithm [27], which implements the SVM method. The two procedures are shown in Algorithm 2.

Algorithm 3. P2Pegasos prediction procedures

```
1: procedure PREDICT(x)
2:     w ← currentModel
3:     return sign(⟨w, x⟩)
4: procedure VOTEDPREDICT(x)
5:     pRatio ← 0
6:     for m ∈ modelQueue do
7:         if sign(⟨m.w, x⟩) ≥ 0 then
8:             pRatio ← pRatio +1
9:     return sign(pRatio/modelQueue.size()−0.5)
```

We assume that the model m has two fields: $m.t \in \mathbb{N}$, which holds the number of times the model was updated, and $m.w \in \mathbb{R}^d$ that holds the linear model. The parameter $\lambda \in \mathbb{R}$ is the learning rate. In our experiments we used the setting $\lambda = 10^{-4}$. Vector $x \in \mathbb{R}^d$ is the local feature vector at the node, and $y \in \{-1, 1\}$ is its correct classification. The operator $< \cdot, \cdot >$ calculates the inner product. Line 4 gets executed if the local example x is misclassified by the model $m.w$.

The effect of the algorithm will be that the models will perform a random walk in the network while being updated using the update rule of the Pegasos algorithm. In this sense, each model corresponds to an independent run of the sequential Pegasos, hence the theoretical results of the Pegasos algorithm are applicable. Accordingly, we know that all these models will converge to an optimal solution of the SVM primal optimization problem [27]. For the same reason, the algorithm does *not need any synchronization or coordination*. Although we do not give a formal discussion of asynchrony, it is clear that as long as each node can contact at least one new uniform random peer in a bounded time after each successful contact, the protocol will converge to the optimal solution.

An important aspect of our protocol is that every node has at least one model available locally, and thus all the nodes can perform a prediction. Moreover, since there are N models in the network (where N is the network size), we can apply additional techniques to achieve a higher predictive performance than that of an output model of a simple sequential implementation. Here we implement a simple voting mechanism, where nodes will use more than one model to make predictions. Algorithm 3 shows the procedures used for prediction in the original case, and in the case of voting. Here the vector x is the unseen example to be classified. In the case of linear models, the classification is simply the sign of the inner product with the model, which essentially describes on which side of the hyperplane the given point lies. We note, that MODELQUEUE is assumed to be of a bounded size. When storing a new model in it, an old one will be removed if the queue is full. In our experiments we used a queue implementation, where the queue holds the 10 latest added models.

6 Experimental Results

We selected data sets of different types including small and large sets containing a small or large number of features. Our selection includes the commonly used Fiser's Iris data set [9]. The original data set contains three classes. Since the

Table 1. The main properties of the data sets, and the prediction error of the baseline sequential algorithms

	Iris1	Iris2	Irirs3	Reuters	SpamBase	Malicious10
Training set size	90	90	90	2000	4140	2155622
Test set size	10	10	10	600	461	240508
Number of features	4	4	4	9947	57	10
Classlabel ratio	50/50	50/50	50/50	1300/1300	1813/2788	792145/1603985
Pegasos 20000 iter.	0	0	0	0.025	0.111	0.080 (0.081)
Pegasos 1000 iter.	0	0	0.4	0.057	0.137	0.095 (0.060)
SVMLight	0	0	0.1	0.027	0.074	0.056 (–)

SVM method is designed for the binary (two-class) classification problem, we transformed this database into three two-class data sets by simply removing each of the classes once, leaving classes 1 and 2 (Iris1), classes 1 and 3 (Iris2), and classes 2 and 3 (Iris3) in the data set. In addition, we included the Reuters [11], the Spambase, and the Malicious URLs [19] data sets as well. All the data sets were obtained from the UCI database repository [10]. Table 1 shows the main properties of these data sets, as well as the prediction performance of the baseline algorithms. SVMLight [17] is an efficient SVM implementation. Note that the Pegasos algorithm can be shown to converge to the same value as SVMlight [27].

The original Malicious URLs data set has about 3,000,000 features, hence we first reduced the number of features so that we could carry out simulations. The message size in our algorithm depends on the number of features, therefore in a real application this step might also be useful in such extreme cases. We used a simple and well-known method, namely we calculated the correlation coefficient of each feature with the class label, and kept the ten features with the maximal absolute values. If necessary, this calculation can also be carried out in a gossip-based fashion [15], but we performed it offline. The effect of this dramatic reduction on the prediction performance is shown in Table 1, where the results of Pegasos on the full feature set are shown in parentheses (SVMlight could not be run due to the large size of the database).

6.1 Scenarios

The experiments were carried out in the event based engine of the PeerSim simulator [21]. The peer sampling service was provided by the NewsCast protocol. The network size is the same as the database size; each node has exactly one sample. Each node starts running the protocol at the same time. The protocol does not require a synchronized startup, but we need it here to analyze convergence in a clearly defined setting.

In our experimental scenarios we modeled message drop, message delay, and churn. The drop probability of each message was 0.5. This can be considered an extremely large drop rate. Message delay was modeled as a uniform random delay from the interval $[\Delta, 10\Delta]$, where Δ is the gossip period, as shown in

Algorithm 1. This is also an extreme delay, which is orders of magnitudes higher than what can be expected in a realistic scenario.

We also modeled realistic churn based on probabilistic models in [29]. Accordingly, we approximated the online session length with a lognormal distribution, and we approximated the parameters of the distribution using a maximum likelihood estimate based on a trace from a private BitTorrent community called FileList.org, obtained from Delft University of Technology [26]. We set the offline session lengths so that at any moment in time 90% of the peers were online. In addition, we assumed that when a peer came back online, it retained its state that it had at the time of leaving the network. We now list the scenarios we experimented with: *No failure*: there is no message drop, no delay and no churn; *Drop only*: we simulate message drop as described, but no other types of failure; *Delay only*: we simulate message delay only; *Churn only*: we simulate node churn only; *All failures*: we apply message drop, delay and churn at the same time.

6.2 Metrics

The evaluation metric we focus on is prediction error. To measure prediction error, we need to split the datasets into training sets and test sets. The ratios of this splitting are shown in Table 1. At a given point in time, we select 100 peers at random (or all the peers, if there are fewer than 100) and we calculate the average misclassification ratio of these 100 peers over the test set using the current models of the peers. The misclassification ratio of a model is simply the number of the misclassified test examples divided by the number of all test examples, which is the so called 0-1 error.

Moreover, we calculated the similarities between the models circulating in the network using the cosine similarity measure. This was done only for the Iris databases, where we calculated the similarity between all pairs of models, and calculated the average. This metric is useful for studying the speed at which the actual models converge. Note that under uniform sampling it is known that all models converge to an optimal model.

6.3 Results

Figure 1 shows the results over the Iris datasets for algorithm variants that do not apply voting for prediction. The plots show results as a function of cycles. One cycle is defined as a time interval of one gossip period Δ. Although the size of each data set is the same, the dynamics of the convergence are rather different. The reason is that the learning complexity of a database depends primarily on the inner structure of the patterns of the data, and not on the size of data set. In trivially learnable patterns a few examples are enough to construct a good model, while under complex patterns a large number of samples as well as many iterations might be required. Since Pegasos also has a similar convergence behavior, we can be sure that this is not an artifact of parallelization.

Let us now turn to the analysis of the individual effects of the different failures we modeled, comparing them to two baseline algorithms. The first baseline algorithm is SVMLight, a sequential efficient SVM solver [17] that optimizes the

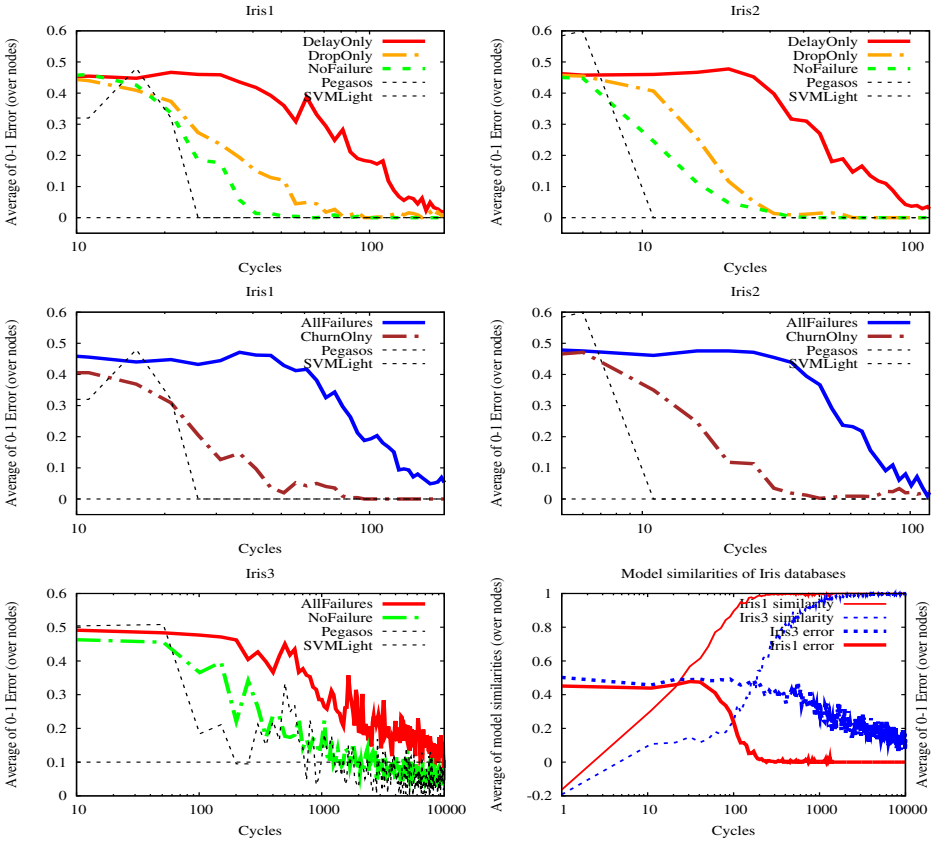


Fig. 1. Experimental results over the Iris databases

dual SVM problem given in (2). It is independent of the cycles, hence its performance is shown as a horizontal line. The second baseline algorithm is Pegasos. We ran Pegasos 100 times, and show the average error at each cycle. Note that for Pegasos each cycle means visiting another random teaching example.

Clearly, the best performance is observed under no failure. This performance is very close to that of Pegasos, and converges to SVMlight (like Pegasos does). The second best performance is observed with churn only. Adding churn simply introduces an extra source of delay since models do not get forgotten as mentioned in 6.1. The situation would be different in an adaptive scenario, which we do not consider here. In the scenario with message drop only, the performance is still very close to the ideal case. Considering the extremely large drop rates, this result is notable. This extreme tolerance to message drop comes from the fact that the algorithm is fully asynchronous, and a 25% drop rate on average causes only at most a proportional slowdown of the convergence. Among the individual failure types, extreme message delay is the most significant factor. On average, each message takes as much as 5 cycles to reach its destination. The resulting

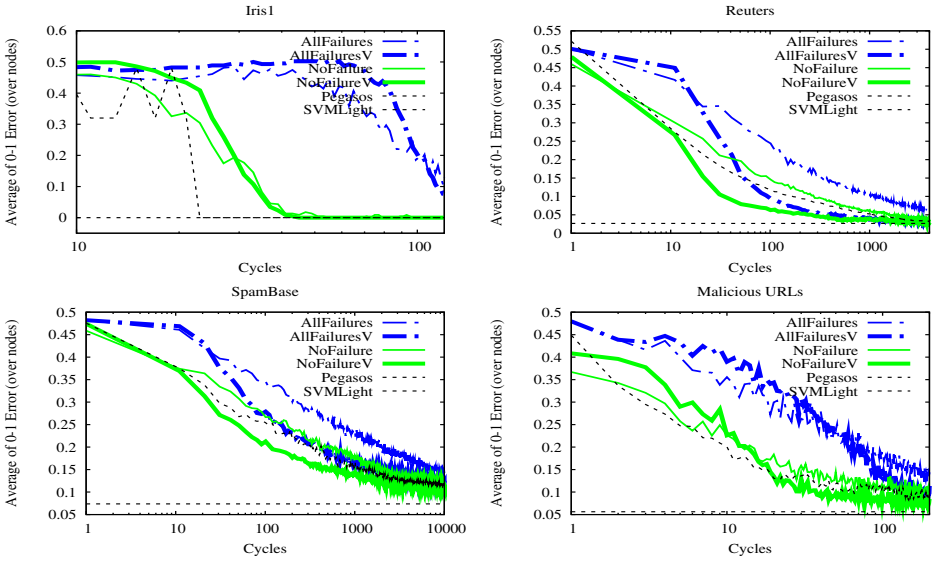


Fig. 2. Experimental results over the large databases, and the Iris1 database. Labels marked with a 'V' are variants that use voting.

slowdown is less then a factor of 5, since some messages do get through faster, which speeds up the convergence of the prediction error.

In Figure 1 we also present the convergence of the averaged cosine similarities over the nodes together with their prediction performance under no failures, without voting. We can see that in the case of each data set the models converge, so the observed learning performance is due to good models as opposed to random influences.

Although, as mentioned above, in our case convergence speed depends mainly on data patterns, and not on the database size, to demonstrate scalability we performed large scale simulations as well with our large data sets. The results can be seen in Figure 2. Here we plotted just the two scenarios with no failures and with all the failures. The figure also shows results for the variants that use voting. A general observation regarding the distinction between the P2Pegasos variants with and without voting is that voting results in a better performance in all scenarios, after a small number of cycles. In the first few cycles, the version without voting outperforms voting because there is insufficient time for the queues to be filled with models that are mature enough. On some of the databases the improvement due to voting can be rather dramatic. We note that where the test data sets were larger (see Table 1) we obtained smoother convergence curves.

7 Conclusions

In this paper we have proposed a generic framework for fully distributed data mining, which implements stochastic gradient search. Nodes in the network

gossip models that are continuously updated at each node along their random walk. We experimented with an instantiation of this framework using the Pegasos algorithm, that is a stochastic gradient descent implementation of the SVM method. Our main conclusion is that the approach is able to produce SVM models in a very hostile environment, with extreme message drop rates and delays, with very limited assumptions about the communication network. The only service that is needed is uniform peer sampling. The quality of the models are very similar to that of the sequential Pegasos algorithm. Furthermore, we can also outperform Pegasos with the help of a voting technique that makes use of the fact that there are many independent models in the network passing through each node. The models are available at each node, so all the nodes can perform predictions as well. At the same time, nodes never reveal their data, so this approach is a natural candidate for privacy preserving solutions.

References

1. Bai, X., Bertier, M., Guerraoui, R., Kermarrec, A.-M., Leroy, V.: Gossiping personalized queries. In: Proc. 13th Intl. Conf. on Extending Database Technology (EBDT 2010) (2010)
2. Bakker, A., Ogston, E., van Steen, M.: Collaborative filtering using random neighbours in peer-to-peer networks. In: Proc. 1st ACM Intl. Workshop on Complex Networks Meet Information & Knowledge Management (CNIKM 2009), pp. 67–75. ACM, New York (2009)
3. Buchegger, S., Schiöberg, D., Vu, L.-H., Datta, A.: PeerSoN: P2P social networking: early experiences and insights. In: Proc. Second ACM EuroSys Workshop on Social Network Systems (SNS 2009), pp. 46–52. ACM, New York (2009)
4. Chapelle, O.: Training a support vector machine in the primal. Neural Computation 19, 1155–1178 (2007)
5. Cheetancheri, S.G., Agosta, J.M., Dash, D.H., Levitt, K.N., Rowe, J., Schooler, E.M.: A distributed host-based worm detection system. In: Proc. 2006 SIGCOMM Workshop on Large-Scale Attack Defense (LSAD 2006), pp. 107–113. ACM, New York (2006)
6. Cristianini, N., Shawe-Taylor, J.: An introduction to Support Vector Machines and other kernel-based learning methods. Cambridge University Press, Cambridge (2000)
7. Datta, S., Giannella, C., Kargupta, H.: Approximate distributed k-means clustering over a peer-to-peer network. IEEE Trans. on Knowl. and Data Eng. 21, 1372–1388 (2009)
8. Duda, R.O., Hart, P.E., Stork, D.G.: Pattern Classification, 2nd edn. Wiley Interscience, Hoboken (2000)
9. Fisher, R.A.: The use of multiple measurements in taxonomic problems. Annals of Eugenics 7(7), 179–188 (1936)
10. Frank, A., Asuncion, A.: UCI machine learning repository (2010)
11. Guyon, I., Hur, A.B., Gunn, S., Dror, G.: Result analysis of the nips 2003 feature selection challenge. In: Advances in Neural Information Processing Systems 17, pp. 545–552. MIT Press, Cambridge (2004)

12. Han, P., Xie, B., Yang, F., Wang, J., Shen, R.: A novel distributed collaborative filtering algorithm and its implementation on p2p overlay network. In: Dai, H., Srikant, R., Zhang, C. (eds.) PAKDD 2004. LNCS (LNAI), vol. 3056, pp. 106–115. Springer, Heidelberg (2004)
13. Hensel, C., Dutta, H.: GADGET SVM: a gossip-based sub-gradient svm solver. In: Intl. Conf. on Machine Learning (ICML), Numerical Mathematics in Machine Learning Workshop (2009)
14. Jelasity, M., Babaoglu, O.: T-man: Gossip-based overlay topology management. In: Brueckner, S.A., Di Marzo Serugendo, G., Hales, D., Zambonelli, F. (eds.) ESOA 2005. LNCS (LNAI), vol. 3910, pp. 1–15. Springer, Heidelberg (2006)
15. Jelasity, M., Montresor, A., Babaoglu, O.: Gossip-based aggregation in large dynamic networks. ACM Transactions on Computer Systems 23(3), 219–252 (2005)
16. Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A.M., van Steen, M.: Gossip-based peer sampling. ACM Transactions on Computer Systems 25(3), 8 (2007)
17. Joachims, T.: Making large-scale SVM learning practical. In: Schölkopf, B., Burges, C., Smola, A. (eds.) Advances in Kernel Methods - Support Vector Learning, ch. 11, pp. 169–184. MIT Press, Cambridge (1999)
18. Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: Proc. 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2003), pp. 482–491. IEEE Computer Society, Los Alamitos (2003)
19. Ma, J., Saul, L.K., Savage, S., Voelker, G.M.: Identifying suspicious urls: an application of large-scale online learning. In: Proc. 26th Annual Intl. Conf. on Machine Learning, ICML 2009, pp. 681–688. ACM, New York (2009)
20. Massoulié, L., Merrer, E.L., Kermarrec, A.M., Ganesh, A.: Peer counting and sampling in overlay networks: random walk methods. In: Proc. 25th Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 123–132. ACM, New York (2006)
21. Montresor, A., Jelasity, M.: Peersim: A scalable P2P simulator. In: Proc. 9th IEEE Intl. Conf. on Peer-to-Peer Computing (P2P 2009), pp. 99–100. IEEE, Los Alamitos (2009), extended abstract
22. Mosk-Aoyama, D., Shah, D.: Fast distributed algorithms for computing separable functions. IEEE Transactions on Information Theory 54(7), 2997–3007 (2008)
23. Ormándi, R., Hegedűs, I., Jelasity, M.: Overlay management for fully distributed user-based collaborative filtering. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6271, pp. 446–457. Springer, Heidelberg (2010)
24. Pouwelse, J.A., Garbacki, P., Wang, J., Bakker, A., Yang, J., Iosup, A., Epema, D.H.J., Reinders, M., van Steen, M.R., Sips, H.J.: TRIBLER: a social-based peer-to-peer system. Concurrency and Computation: Practice and Experience 20(2), 127–138 (2008)
25. van Renesse, R., Birman, K.P., Vogels, W.: Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Transactions on Computer Systems 21(2), 164–206 (2003)
26. Roozenburg, J.: Secure Decentralized Swarm Discovery in Tribler. Master's thesis, Parallel and Distributed Systems Group, Delft University of Technology (2006)
27. Shalev-Shwartz, S., Singer, Y., Srebro, N., Cotter, A.: Pegasos: primal estimated sub-gradient solver for SVM. Mathematical Programming B (2010)

28. Siersdorfer, S., Sizov, S.: Automatic document organization in a p2p environment. In: Lalmas, M., MacFarlane, A., Rüger, S.M., Tombros, A., Tsikrika, T., Yavlinsky, A. (eds.) ECIR 2006. LNCS, vol. 3936, pp. 265–276. Springer, Heidelberg (2006)
29. Stutzbach, D., Rejaie, R.: Understanding churn in peer-to-peer networks. In: Proc. 6th ACM Conf. on Internet measurement (IMC 2006), pp. 189–202. ACM Press, New York (2006)
30. Tölgyesi, N., Jelasity, M.: Adaptive peer sampling with newscast. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 523–534. Springer, Heidelberg (2009)
31. Tveit, A.: Peer-to-peer based recommendations for mobile commerce. In: Proc. 1st Intl. workshop on Mobile commerce (WMC 2001), pp. 26–29. ACM Press, New York (2001)

Evaluation of P2P Systems under Different Churn Models: Why We Should Bother*

Marc Sànchez-Artigas and Enrique Férnandez-Casado

Department of Computer Engineering and Mathematics
Universitat Rovira i Virgili, Catalonia, Spain
{marc.sanchez,enriqueeduardo.fernandez}@urv.cat

Abstract. Research on peer-to-peer (P2P) systems has been hampered by the fact that few systems are actually in use, and the space of possible applications is still under scrutiny. As a consequence, new ideas have been mostly evaluated using synthetic data, traces from a few existing systems and simulators, with a poor characterization of churn. This void has lead to the formulation of a variety of models, with implications that have not yet been made altogether clear to the community. In this work, we discuss the question whether it pays off to evaluate P2P applications using more than one churn model. Although an affirmative response could appear to be obvious at first glance, we show that depending on the aspects under consideration, models can yield equivalent results, saving implementation time but leading to spurious generalizations if proper care is not taken.

1 Introduction

Simulations are the most popular tool for studying peer-to-peer (P2P) systems. The cost of implementation is less than that of large-scale experimentation and, if carefully crafted, a simulation can be more realistic than any tractable analytical model. However, when we talk about *churn*, the continuous process of user arrival and departure, it is easy to come to the question of how to model the dynamics of a system appropriately. Since there is not a clear picture yet, there is no means to protect researchers against bias in the evaluation of new ideas and applications. Despite best intentions, researchers very often assume that the results obtained from a model apply to other configurations. For this reason, it is vital to ascertain whether there exists any apparent danger in the generalization of results due to the technical peculiarities of existing churn models.

These peculiarities include the arrival process of users into the system, their ontime and offtime durations, and the proportion of temporary and permanent departures. The exponential and Pareto distributions have been widely employed to model the behavior of users [8] [12] [19], with marked differences in the results depending on the distribution. We consider both distributions in our analysis.

In this paper, we try to answer the "why bother" question, i.e., *whether it pays off to use more than one churn model to evaluate a new idea*, simply because a

* This research work has been partially funded by the Spanish Ministry of Science and Innovation through project DELFIN, TIN2010-20140-C03-03.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 541–553, 2011.

general extrapolation of results to reality might be problematic. To inform this debate, we use four state-of-the-art models and compare them according to some metrics to quantify the magnitude of their difference. Armed with these metrics, and equating all models in terms of asymptotic availability, we see that although models can seem equivalent in appearance, the peculiarities of each one can lead to disparate conclusions in practice. Proof of that is their close similarity in the Poissonity of user arrivals, which, in conjunction with the same mean availability, can give a strong sensation of indistinguishability across all models. In this case, by following a reasoning similar to that in [6], a typical false generalization might be that *lifetime-based churn models have an equivalent uniform failure model in the steady state*. On the contrary, we argue that, although a single model makes for rapid prototyping, a deeper inspection is necessary to clarify to which extent the results from one model are enough to draw firm conclusions. To whet reader's appetite, we provide here a first interesting finding of this piece of research:

Based on our analysis, *we show that for general situations, where arrivals are well-modeled by a Poisson process, any model is as good as any other, and only when results relatively depend on availability patterns, it pays off to account for their differences*. This result provides a rule of thumb to determine when a claim on churn is exaggerated, or when it can be considered adequate.

The rest of this work is organized as follows. In Section 2, we overview related work. Section 3 presents the churn models that are subject of our comparison as well as their concrete instances. Section 4 compares the models in three different relevant aspects. Section 5 concludes this work.

2 Related Work

The phenomenon of churn widely exists in P2P systems, large-scale distributed systems, etc. Quite a few measurements have been conducted to understand the characteristics of churn in recent years. Particularly enriching are the works [2, 14] and [13]. In these works, the authors found similar phenomena of system churn. They observed that the distribution of user lifetimes in real P2P systems is often heavy-tailed, which means that while most users spend several minutes per day, a handful of other users keep their computers logged in for weeks. We investigated the properties of each model using a Pareto distribution to reflect reality.

Analytical models capturing the many faces of churn are available too, such as [19, 8, 7]. Each of these works proposed a new model to analyze one aspect of P2P systems, such as connectivity in this case, each from a particular viewpoint, thereby making it hard for someone not familiar in the area to come to the right conclusions. The same happened in other areas, though it is particularly visible in the storage literature (see [18] for references therein).

Surprisingly, despite the broad literature on churn, no comparative analysis has yet been conducted to ascertain whether the peculiarities of each model are problematic to predict the performance of a system in the real world. If this was the case, the potential of a new solution may be either over- or underestimated, leading to misleading conclusions. To the best of our knowledge, this is the first

work to elucidate some of the existing differences between the models, even when the mean online and offline durations are the same in all models.

3 Models and Distributions

In this section, we describe the models we chose to support our thesis as well as the specific distributions chosen for the ON/OFF durations.

3.1 Churn Models

Despite numerous studies of the effects of churn, there are no general simulators that incorporate the most popular churn models. To fill this gap, we implemented a tool called `Affluenza`[4], available at `http://ast-deim.urv.cat/trac/affluenza/`, with several representative churn models. For this analysis, we chose four models for their characteristics and referred to each one by the name of the first author. These models were: Leonard [8], Yao [19], Duminuco [3] and Wang [17].

With the exception of Leonard, in the rest of models, each peer can be viewed as alternating between online/offline states. Formally, this can be represented as an ON/OFF right-continuous process $\{Z_i(t)\}$ on time interval $[0, \infty)$, indicating the state of user i. That is, $Z_i(t) = 1$ if user i is ON at time t, and 0 otherwise. What varies across these models is either the common assumption of stationarity in user arrivals or the distinction between temporary or permanent churn.

Yao: This model describes a stationary alternating renewal process, where users behave independently of each other. This means that for $i \neq j$, processes $\{Z_i(t)\}$ and $\{Z_j(t)\}$ are independent, and therefore users do not synchronize their arrival or departures, exhibiting uncorrelated lifetime characteristics. For each process $\{Z_i(t)\}$, this model assumes that ON durations L_i have some distribution $F_i(x)$ with mean $l_i = \mathbb{E}\{L_i\} < \infty$ and that OFF durations D_i have some distribution $G_i(x)$ with mean $d_i = \mathbb{E}\{D_i\} < \infty$, respectively. Consequently, this model takes into account the heterogeneous behavior of users as a main characteristic.

Duminuco: In this model each user can be described by a stationary alternating process with independent and identically distributed online and offline durations. However, ON/OFF durations are drawn from two general distributions $F(x)$ and $G(x)$, which are not *unique* to each user. Moreover, users may permanently leave the network or become temporarily offline according to an abandon ratio P. To prevent the network size from depleting to zero, new users arrive according to a Poisson process with rate $R = \mu P n$, where n is the target network size and μ is the disconnection rate which includes permanent and temporary disconnections.

Wang: This model can be viewed as an extension of Yao model to accommodate non-stationary dynamics like diurnal arrival/departure patterns. As Duminuco, the duration of ON periods is drawn from a general distribution $F(x)$, but OFF states are now split into two sub-states: REST and WAIT. The REST state can

be visualized as the delay between a departure and midnight. The WAIT state represents the delay from midnight until the user rejoins the system again within a given day, which follows its own distribution $F_A(x)$. Unlike prior models, Wang model allows OFF periods to be dependent on the time of day and the duration of the previous ON period.

Leonard: The main particularity of this model is the absence of OFF durations. Upon joining, each user is given a random lifetime drawn from some distribution $F(x)$ which reflects the amount of time the user stays in the network. Once the lifetime expires, the user departs from the system permanently. To maintain the network size stable, each failed node is immediately replaced by a fresh one with another random lifetime.

3.2 ON/OFF Distributions

In line with measurement results that demonstrate heavy-tailed user lifetimes in real P2P networks [2] [14], we use a shifted Pareto distribution to allow arbitrarily small lifetimes and offtimes. Heavy-tailed distributions exhibit tails that follow a power-law with small exponent in contrast to traditional distributions (Gaussian, exponential) whose tails decline faster. Any random variable whose distribution is heavy-tailed exhibits high variability, and may have infinite variance and mean. In practical terms, this means that lifetimes following a heavy-tailed distribution can give rise to extremely large values with non-negligible probability. As a result, when observed collectively, a group of users can appear to be correlated over time even when their ON/OFF processes are independent of each other.

The cumulative distribution function of a shifted Pareto is given by $F(x) = \Pr\{X \geq x\} = 1 - \left(1 + \frac{x}{\beta}\right)^{-\alpha}$, $x > 0$, $\alpha > 1$, where scale parameter $\beta > 0$ can change the mean of the distribution without affecting its range $(0, \infty)$. Observe that the mean of this distribution $\mathbb{E}\{X\} = \frac{\beta}{\alpha - 1}$ is finite only if $\beta > 1$, which we assume holds throughout the paper. Further, we use the exponential distribution as a baseline for some of the comparisons. Unless noted otherwise, configurations are as follows (notation $Par(\alpha, \beta)$ refers to $F(x) = 1 - (1 + x/\beta)^{-\alpha}$):

- *Yao*: We generate n pairs of means l_i and d_i that are randomly drawn from two Pareto distributions with $\alpha = 3$ to model heterogeneity. For mean ON periods, we use $\beta = 1$ to obtain $\mathbb{E}\{l_i\} = 1/2$ hour. For mean OFF durations, we use $\beta = 2$ to get $\mathbb{E}\{d_i\} = 1$ hour. We consider three cases:
  1. heavy-tailed system $\mathcal{H}_Y$ with $F_i(x) \sim Par(3, 2l_i)$ and $G_i(x) \sim Par(3,\ 2d_i)$;
  2. very heavy-tailed system $\mathcal{VH}_Y$ with $F_i(x) \sim Par(1.5, \frac{l_i}{2})$ and $G_i(x) \sim Par(1.5, \frac{d_i}{2})$; and
  3. exponential $\mathcal{E}_Y$ with $F_i(x) \sim exp(\frac{1}{l_i})$ and $G_i(x) \sim Par(3, 2d_i)$.
- *Duminuco*: We fix abandon probability to $P = 0.05$ to let each user stay in the system for sufficiently long time. Again, we consider three cases:
  1. heavy-tailed system $\mathcal{H}_D$ with $F(x) \sim Par(3, 1)$ and $G(x) \sim Par(3, 1)$;

2. very heavy-tailed system $\mathcal{VH}_D$ with $F(x) \sim Par(1.5, 0.25)$ and $G(x) \sim Par(1.5, 0.25)$; and
3. exponential $\mathcal{E}_D$ with $F(x) \sim exp(2)$ and $G(x) \sim Par(3, 1)$.

– *Leonard*: We investigate three cases: 1) heavy-tailed system $\mathcal{H}_L$ with $F(x) \sim Par(3, 1)$; 2) very heavy-tailed system $\mathcal{VH}_L$ with $F(x) \sim Par(1.5, 0.25)$ and 3) exponential $\mathcal{E}$ with $F(x) \sim exp(2)$.

To make the comparison fair, it is important to note here that the asymptotic availability of all Yao and Duminuco instances is 0.5. From Smith's theorem, it is easy to see that the asymptotic availability of each user i in the Yao model: $a_i = \lim_{t\to\infty} \Pr\{Z_i(t) = 1\} = \frac{l_i}{l_i+d_i}$. For sufficiently large n, the availability of Yao model becomes $a = \frac{1}{n}\sum_{i=1}^{n} a_i = 0.5$, which is equal to Duminuco availability $a = \frac{\mathbb{E}[L]}{\mathbb{E}[L]+\mathbb{E}[D]}$, where $\mathbb{E}[L] = \int_0^\infty (1 - F(x))dx$ and $\mathbb{E}[D] = \int_0^\infty (1 - G(x))dx$.

For the Wang model, we use only one instance as the purpose of this model is to be a baseline to study the arrival non-stationarity of the *a priori* uncorrelated Yao and Duminuco models. The exact configuration is given in 4.2.

4 Comparative Analysis

Our focus is on the aggregate behavior of a random groups of n users. The reason is that the particularities of churn models are more apparent for small groups of users. Observe that while replica sets, routing tables and streaming trees rarely exceed a few tens of members, entries and connections, respectively, prior works have paid too much attention to large n behavior, overlooking subtle side effects.

To capture these effects, we compare the models against each other according to three aspects: 1) the degree of Poissonity in user arrivals, 2) availability interdependence, and 3) system reliability. It is important to note that not all models can be compared against each other in all aspects. For instance, it does not make sense to evaluate the Poissinity of the Wang model since the arrivals in this model are not stationary by definition. Similarly, the absence of offline durations in the Leonard model precludes any evaluation of its behavior based on user availability. Despite this caveat, the number of divergences is high and worth to be discussed.

4.1 Poissonity in Arrivals

In the absence of a "universal" churn model, researchers have traditionally made simplifying assumptions about churn behavior in their definition and evaluation. One common modeling assumption is that user arrivals follow a Poisson process (e.g., see [7] [10] [9] [13]). Poisson processes are characterized by interarrival times that are distributed exponentially and are independent of each other. Also, the number of Poisson arrivals in non-overlapping time intervals are independent. When such a process is aggregated to large time-scales, the law of large numbers applies and the aggregated process converges to the mean quickly. Visually, the aggregated process appears "smooth" and non-bursty.

In this section, we examine the deviation from Poissonity in user arrivals for each model. This question is particularly intriguing for ON/OFF models due to

the effect of superposition. While the Palm-Khintchine Theorem states that the superposition of n renewal processes converges to a Poisson process as $n \to \infty$, it requires that each point process associated with the renewal process becomes sparser[1] as $n \to \infty$ and the various processes be independent. However, these conditions are difficult to find in real systems simply because the inter-occurrence times in the superposition process are statistically dependent, which is especially visible for small n. In this sense, it is interesting to study the Poissonity of user arrivals as a function of n, particularly at small values, since in many situations the response of a user essentially depends on the aggregate behavior of a reduced number of other users. For instance, in P2P storage systems, peers increase the availability of their data by replicating it on other users in the network. In these systems, the number of replicas to maintain high availability is significantly much lower (by orders of magnitude) than the total network size. To wit, if mean host availability is 0.5, it can be easily shown than the number of replicas to guarantee a target availability of 0.999 is only 10 [1]. This result substantiates the need to characterize arrivals at the group level.

Test for Poissonity in Arrivals. An arrival process is said to be Poisson having rate λ, $\lambda > 0$, if the interarrival times X_1, X_2, ... have a common exponential distribution function: $\Pr\{X_n \leq t\} = 1 - e^{\lambda t}, t \geq 0$. The average interarrival time is given by λ^{-1}. All the arrivals are independent of each other and the number of arrivals occurring in a given interval depends only on the length of that interval.

To evaluate whether a process is Poisson or not, we need to test whether the process is exponentially distributed and is consistent with independent arrivals. A linear trend in the Complementary Cumulative Distribution Function (CCDF) with y-axis on log scale indicates an exponential distribution. To check whether arrivals are uncorrelated, we compute the autocorrelation function (ACF), $r(k)$, at different lags. ACF of a time series X_n at lag k corresponds to its normalized auto-covariance:

$$r(k) = \frac{Cov\{X_n, X_{n+k}\}}{\sigma^2} = \frac{\sum_{n=1}^{N-k}(X_n - \mu)(X_{n+k} - \mu)}{\sum_{n=1}^{N}(X_n - \mu)^2}, \tag{1}$$

where μ and σ are the sample mean and standard deviation, respectively. If the arrivals are completely uncorrelated, the sample ACF is approximately normally distributed with mean 0 and variance $1/N$, where N is the number of samples. The 95% confidence limits for $r(k)$ can then be approximated to $0 \pm \frac{2}{\sqrt{N}}$[2]. Thus, for example, if a time series has length $N = 100$, the approximate 95% confidence limits are $\pm\frac{2}{\sqrt{100}} = \pm 0.20$.

[1] Sparsity can be formulated as follows: Given $\epsilon > 0$, for each $t > 0$ and n sufficiently large: $F_{jn}(t) \leq \epsilon$, $j = 1, \ldots, n$, where $F_{jn}(t)$ is the inter-occurrence time distribution of the j-th process. Loosely speaking, this asserts that as n increases, the processes being combined have renewals very infrequently.

[2] Observe that about 95% of values drawn from a normal distribution are within two standard deviations.

Table 1. Coefficient of determination R^2 for a 240-hour portion

Model	Scenario	n	R^2	Model	Scenario	n	R^2	Model	Scenario	n	R^2
Yao	$\mathcal{VH}_Y$	1	0.2771	*Leonard*	$\mathcal{VH}_L$	1	0.5584	*Duminuco*	$\mathcal{VH}_D$	1	0.2411
	$\mathcal{VH}_Y$	2	0.6440		$\mathcal{VH}_L$	2	0.8206		$\mathcal{VH}_D$	2	0.8766
	$\mathcal{VH}_Y$	5	0.8804		$\mathcal{VH}_L$	5	0.8903		$\mathcal{VH}_D$	5	0.9455
	$\mathcal{VH}_Y$	50	0.9934		$\mathcal{VH}_L$	50	0.9943		$\mathcal{VH}_D$	50	0.9899
	$\mathcal{VH}_Y$	100	0.9970		$\mathcal{VH}_L$	100	0.9987		$\mathcal{VH}_D$	100	0.9984
	$\mathcal{H}_Y$	1	0.8351		$\mathcal{H}_L$	1	0.7935		$\mathcal{H}_D$	1	0.5130
	$\mathcal{H}_Y$	2	0.8744		$\mathcal{H}_L$	2	0.9628		$\mathcal{H}_D$	2	0.8531
	$\mathcal{H}_Y$	5	0.9330		$\mathcal{H}_L$	5	0.9896		$\mathcal{H}_D$	5	0.9324
	$\mathcal{H}_Y$	50	0.9862		$\mathcal{H}_L$	50	0.9984		$\mathcal{H}_D$	50	0.9987
	$\mathcal{H}_Y$	100	0.9922		$\mathcal{H}_L$	100	0.9994		$\mathcal{H}_D$	100	0.9977
	$\mathcal{E}_Y$	1	0.9874		$\mathcal{E}_L$	1	0.9927		$\mathcal{E}_D$	1	0.3166
	$\mathcal{E}_Y$	2	0.9910		$\mathcal{E}_L$	2	0.9961		$\mathcal{E}_D$	2	0.5678
	$\mathcal{E}_Y$	5	0.9957		$\mathcal{E}_L$	5	0.9966		$\mathcal{E}_D$	5	0.6303
	$\mathcal{E}_Y$	50	0.9991		$\mathcal{E}_L$	50	0.9980		$\mathcal{E}_D$	50	0.6926
	$\mathcal{E}_Y$	100	0.9963		$\mathcal{E}_L$	100	0.9990		$\mathcal{E}_D$	100	0.7157

Another statistical measure is the Index of Dispersion for Intervals (IDI). Let S_k denote the sum of k consecutive inter-arrival times $S_k = X_1 + X_2 + \cdots + X_k$. The IDI or k-interval squared coefficient of variation is defined as:

$$c_k^2 = \frac{kVar\left\{S_k\right\}}{\mathbb{E}\left\{S_k\right\}^2} = \frac{kCov\left\{X_1, X_1\right\} + 2\sum_{j=1}^{k-1}(k-j)Cov\left\{X_1, X_{1+j}\right\}}{k\mathbb{E}\left\{X_1\right\}^2}. \quad (2)$$

The IDI of an ideal Poisson process equals to 1 for all k. If the arrival process has larger variance than Poisson at some time scale, then this index increases as a function of k. c_k^2 only depends on the length of the series, not on any specific part of the trace [16]. In addition, for a renewal process, $c_k^2 = c_1^2$ for all k. Observe that when $c_k^2 = c_1^2$ for all k, then $Cov\left\{X_i, X_j\right\} = 0$ for all $i, j (i \neq j)$. As a result, looking for fluctuations in the IDI sequence $\left\{c_k^2; k \geq 1\right\}$ is a good manner to test for deviations from the renewal property.

Results. We first verify if interarrival times follow an exponential distribution. From the CCDF plot of interarrival times with the y-axis in logarithmic scale, we can conjecture that the interarrival times are exponentially distributed if the plot shows approximately linear behavior. In Fig.1a, we show the CCDF for 240-hour portion of the $\mathcal{VH}_Y$ as a function of n. As shown in the figure, the shape of the tail, while never strictly linear, shows a stronger linear trend as n increases, verifying the Palm-Khintchine Theorem. For the rest of configurations, we found a similar behavior, though we have not included the plots due to lack of space. To give insight into the their potential Poissonity, Table 1 reports the coefficient of determination, R^2, for all configurations after performing linear regression on $\log_{10}\left(1 - \Pr\left\{X < x\right\}\right)$, where $\Pr\left\{X < x\right\}$ is the interarrival time distribution. If R^2 is found to be more than 99%, we can conjecture that arrivals are Poisson, which is, in general, true for $n = 100$.

However, to strictly claim Poissonity, it is necessary to verify that interarrival times are independent of each other, which can be done by inspecting the ACF. For lack of space, we only plot the autocorrelation coefficients for the very heavy-tailed instances of Yao, Duminuco and Leonard for $n = 50$. In the plot, horizontal lines correspond to 95% confidence bounds. As shown in Fig. 1b, autocorrelation coefficients for most lags lie within 95% confidence interval, which demonstrates that interarrival times are almost independent of each other.

To quantify deviation from Poisson at different time scales, we use the Index of Dispersion for Intervals (IDI). As before, we only have considered the instances $\mathcal{VH}_Y$, $\mathcal{VH}_L$ and $\mathcal{VH}_D$ for $n = 50$ due to space constraints, although we got similar results for other configurations. The results are shown in Fig. 1c. An ideal Poisson process has $c_k^2 \equiv 1$ for all k. If an arrival process has higher variance at a given timescale, c_k^2 increases with increasing k. Fig. 1c illustrates such behavior. The values of c_k^2 for large timescales rapidly diverge as k increases, which indicates a deviation from Poisson behavior. Altogether the above results show that:

- For sufficiently large n (n typically ~ 100), *arrival processes follow a Poisson process.* Given the same mean availability, such similarity in arrival processes verifies our rule of thumb that for basic analysis any model is good enough.
- For small n and heavy-tailed durations, arrival processes are not Poisson, so it critical to account for the existing differences to avoid overestimations.

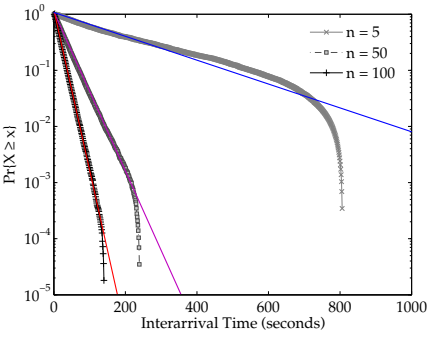
To conclude, we provide a pictorial proof of the burstiness on the the number of arrivals per time unit at two different timescales in Fig. 1d. Timescales plotted are 30 seconds and 5 minutes and the instance chosen is $\mathcal{VH}_Y$ for $n = 50$. We can see that the arrival process exhibits burstiness at the two timescales, confirming the deviation from Poisson and indicating the presence of *self-similarity.*

4.2 Availability Inter-dependence: A First Difference

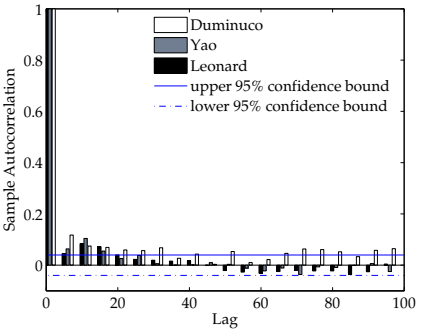
As shown in the earlier experiment, for sufficiently small n, the aggregate arrival process might be *non-stationary*[3] in certain models, in particular, in the models where the possibilities of a user to contribute too much to the arrivals is high. In this sense, it is prudent to check if *non-stationarity* induces availability patterns between users that vary with time-of-day. A low availability inter-dependence is desirable for many P2P applications, such as storage systems, where correlated churn exhibits a degree of burstiness and impacts file availability and durability greatly [18] [5]. In this case, we will see that the differences are significant. Hence, *it will be risky to extrapolate results from one model to another.*

We characterize the dependence between every pair of users using the cosine similarity between their uptime histories. We represent the uptime history of a host as a binary string of size W, where bit b_t is 1 if the host was online a time t, and -1 otherwise. We assume that samples are taken at discrete points $t_j = \Delta j$,

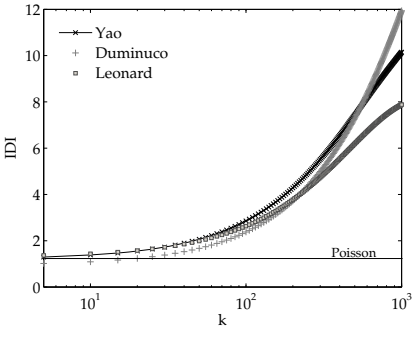
[3] Let $\lambda(t) = \sum_{i=1}^{n} \lambda_i(t)$ be the aggregate arrival rate of a random set of exactly n users, where $\lambda_i(t)$ is the arrival rate of user i in $[0, t]$. We consider that the arrival process is *non-stationary* when $\lambda(t)$ is not simply a constant λ.



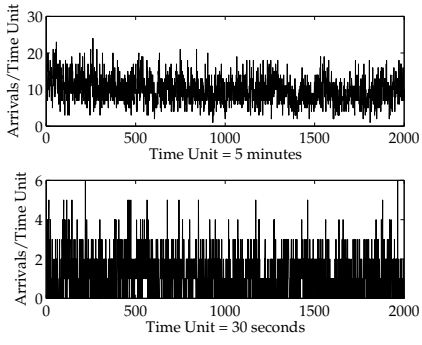
(a) CCDF plot with y-axes in log scale.



(b) ACF plot for $n = 50$.



(c) IDI plot for $n = 50$.



(d) Arrival process at two timescales.

Fig. 1. Poissonity in arrivals for very heavy-tailed instances: $\mathcal{VH}_Y$, $\mathcal{VH}_L$, $\mathcal{VH}_D$

$j = 1, 2, \ldots, W$, in the interval $[\Delta, \tau]$ for any $\Delta > 0$, where $\tau = \Delta W$. To capture time-of-day effects, we fix $\tau = 24$ hours with $\Delta = 1$ minute.

Once we have the uptime story for each host, we can construct random groups of size n, where a group contains a uptime string for each member host. Let $\boldsymbol{A}_i$ be the uptime history of user i in a group G. For each group, we then compute the average pairwise cosine similarity of all users in each group G, using the next equation:

$$\textsc{Sim}(G) = \frac{\sum_{i,j \in G, i \neq j} cosSim(\boldsymbol{A}_i, \boldsymbol{A}_j)}{\sum_{i=1}^{|G|-1} i}, \tag{3}$$

where $cosSim(\boldsymbol{A}_i, \boldsymbol{A}_j) = \frac{\sum_{k=1}^{W} \boldsymbol{A}_i[k]\boldsymbol{A}_j[k]}{\sqrt{\sum_{k=1}^{W} \boldsymbol{A}_i[k]^2}\sqrt{\sum_{k=1}^{W} \boldsymbol{A}_j[k]^2}}$ is nothing but the cosine similarity between vectors $\boldsymbol{A}_i$ and $\boldsymbol{A}_j$. Note that this measure can vary between -1 and 1. If the value of the measure is close to -1, this means that users i and j do not overlap in time. If the value of the measure is near zero, then it can be concluded that there is no correlation between users i and j, i.e., the availability of i does not tell anything about the availability of j over time. Otherwise, if the value of the cosine measure is near 1, users i and j are perfectly correlated. For P2P systems, a value close to 1 is highly undesirable as it means that users tend

to abandon the network irregularly and in batches, which can trigger unnecessary repairs in P2P storage systems [18], or induce overlay partitions [11], for instance.

To characterize availability inter-dependence, we conducted two experiments, one with $n = 5$ and the other with $n = 50$. We used the very-heavy tailed systems $\mathcal{VH}_Y$ and $\mathcal{VH}_D$ together with the following instance of the Wang model. For ON periods, we chose $F(x) \sim Par(1.5, 0.25)$ with arrival distribution $F_A(x)$ following a truncated Weibull distribution with shape parameter $k = 0.53$.

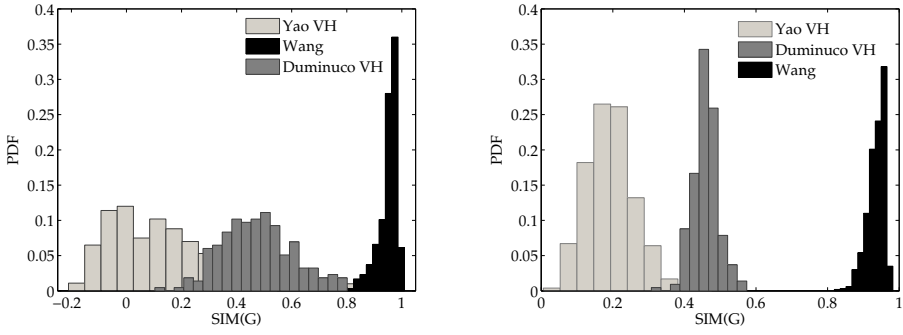
In Fig. 2, we plot the probability density function (PDF) of mean pairwise cosine similarity for a collection of 1000 random groups. As shown in the figure, PDF of cosine similarity differs notably from one model to another. As expected, cosine similarity is peaked around 0.95 for the Wang model as arrivals are highly correlated. However, what is surprising is the high level of correlation exhibited by the Duminuco configuration. This high correlation is explained by the action of the Poisson process that compensates permanent departures. On expectation, $R = \mu P n$ new users join the system every hour to compensate the users departing permanently from the system. This means that within time period $(12, 24]$, it is expected that $12\,R$ new users join the system, which equals to 6 for $n = 5$. Since these users have the first half of their availability histories set to -1 as they were initially OFF, their correlation is close to 1, which makes the average pairwise cosine similarity to be near 0.5. Even Yao configuration exhibits a certain positive correlation due to heavy tails despite that user processes are independent.

4.3 Reliability: A Second Difference

As defined by Siewiorek and Swarz in [15], the reliability of a system as a function of time, $R(t)$, is the conditional probability that the system has survived the time interval $[0, t]$, given that the system was operational at time $t = 0$. For storage systems, this might mean that the system does not lose data before the mission time t. In terms of overlay connectivity, this might mean that the system has no isolated nodes, and hence partitions, provided that the overlay was connected at time $t = 0$. Whatever the context, reliability gives us sense of the odds of failure-free operation over time t, irrespective of whether a failure represents the loss of a data unit or the isolation of a host from the network. As in the previous two sections, we construct random groups of size n and investigate the statistical behavior of $R(t)$ as a function of n. Our goal is to determine if there are any obvious difference between the models in the probability to find all the hosts of a group offline within time t, given that all were initially online at time 0.

Although the reliability function $R(t)$ can be obtained using Markov models[4] in some cases, it is generally difficult to develop closed-form expressions for the superposition of ON/OFF processes with heavy-tailed ON and OFF periods. To better understand this, let $W(t)$ be the superposition of n heavy-tailed ON/OFF processes. Clearly, $W(t) = \sum_{i=1}^{n} Z_i(t)$ and therefore, $W(t)$ returns the number of online hosts at time t. Denote by T the time when the last online host of the group

[4] One can construct a Markov chain where the states of the chain represent the number of departures and where the transitions correspond to joins and departures, what is often referred to as "*birth and death processes*" in the literature.



(a) $n = 5$. (b) $n = 50$.

Fig. 2. PDF of mean pairwise cosine similarity for 1000 random groups

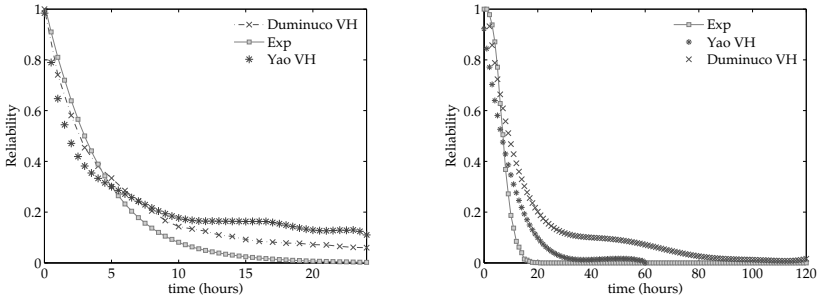
logged off. Then, it is easy to see that T can be formalized as the *first hitting time* of process $W(t)$ on level 0, i.e., $T = \inf\{t > 0 : W(t) = 0 | W(0) = n\}$. Since $R(t) = 1 - \Pr\{T < t\}$, the calculation of $R(t)$ reduces to compute the distribution of T. However, the main problem is that the exact distribution of T is difficult to develop in closed-form since it depends on *transient* properties of $W(t)$ [8].

Instead, our recommendation is to use Monte Carlo simulation to calculate $R(t)$ for the distinct churn models. We accomplish this as follows. We implement the function $I(T < t)$, which equals to 1 if the last online host departed before the mission time t, and 0 otherwise. In other words, if the group had all members offline at least once during $[0, t]$ or not. Since many trials are required to produce statistically meaningful results, the standard method of approximating reliability is to run N iterations (typically chosen experimentally) of the simulator and then make the following calculation: $R(t) = 1 - \sum_{i=1}^{N} \frac{I(T<t)}{N}$.

Since $I(T < t)$ evaluates to 1 when there is a group failure in iteration i, and 0 otherwise, $R(t)$ directly calculates the probability of no group failure in $[0, t]$.

For this experiment, we conducted two simulations, one with $n = 5$ and the other with $n = 20$. In this case, we restricted our attention to the configurations $\mathcal{VH}_Y$ and $\mathcal{VH}_D$ and a variant of Yao where $F_i(x) \sim exp(2)$ and $G_i(x) \sim exp(2)$ for all users. We called this variant Exp and we used it as a baseline to quantify the effects of heavy tails on $R(t)$. To obtain meaningful results, we set $N = 1000$.

In Fig. 3, we depict $R(t)$ as a function of the mission time t. Non-surprisingly, the results show that heavy tails increase reliability in all cases, although their effect is not homogeneous and depends on the group size. For $n = 20$, Duminuco performs better whereas for $n = 5$ is more unreliable than Yao. This phenomenon can be explained by the arrival of fresh users into the system with Poisson arrival rate $R = \mu P n$. For $n = 20$, two new users join the system every hour on average, which combined with heavy-tailed lifetimes prolong significantly the time to find all users simultaneously offline. For $n = 5$, however, the extremely large offtimes cannot be compensated by the arrival of a new user every 2 hours and reliability declines faster. This result reinforces the idea that *the peculiarities of each model*



(a) $n = 5$. (b) $n = 20$.

Fig. 3. $R(t)$ as a function of the mission time t in hours

emerge when one takes a closer look at the ON-OFF pattern of users, rather than considering only typical macroscopic aggregates such as the mean availability and the exact shape of the lifetime distributions.

5 Conclusions

In this paper, we have shown, with the aid of some measures, that the differences between existing models for churn analysis and evaluation are subtle and require of technical expertise to interpret results in an appropriate manner. For a general evaluation, we have seen than any model is as good as any other, but for a reliable extrapolation it is vital to understand if a particular singularity can overestimate simulation results. To wit, we have found that user availabilities are surprisingly more dependent on each other if a small fraction of the departures are permanent, an insight that we could have not reached without a comparison between models. For this reason, it is critical to determine to which extent the singularities of each model can affect the results, and classify them to help researchers not familiar in the area to correctly evaluate their proposals. Here, we have taken the first step.

References

1. Bhagwan, R., Tati, K., Cheng, Y.-C., Savage, S., Voelker, G.M.: Total recall: System support for automated availability management. In: NSDI 2004, pp. 337–350 (2004)
2. Bustamante, F.E., Qiao, Y.: Friendships that last: peer lifespan and its role in p2p protocols. In: Web Content Caching and Distribution, pp. 233–246 (2004)
3. Duminuco, A., Biersack, E., En-Najjary, T.: Proactive replication in distributed storage systems using machine availability estimation. In: CoNext, pp. 27–41 (2007)
4. Fernández-Casado, E., Sànchez-Artigas, M., García-López, P.: Affluenza: Towards universal churn generation. In: IEEE P2P 2010, pp. 1–2 (2010)
5. Haeberlen, A., Mislove, A., Druschel, P.: Glacier: highly durable, decentralized storage despite massive correlated failures. In: NSDI 2005, pp. 143–158 (2005)

6. Kong, J.S., Roychowdhury, V.P.: Price of structured routing and its mitigation in p2p systems under churn. In: IEEE P2P 2007, pp. 97–104 (2007)
7. Krishnamurthy, S., et al.: An analytical study of a structured overlay in the presence of dynamic membership. IEEE/ACM TON 16(4), 814–825 (2008)
8. Leonard, D., Rai, V., Loguinov, D.: On lifetime-based node failure and stochastic resilience of decentralized peer-to-peer networks. In: ACM SIGMETRICS, pp. 26–37 (2005)
9. Li, J., Stribling, J., Gil, T.M., Morris, R., Kaashoek, M.F.: Comparing the performance of distributed hash tables under churn. In: Voelker, G.M., Shenker, S. (eds.) IPTPS 2004. LNCS, vol. 3279, pp. 87–99. Springer, Heidelberg (2005)
10. Li, J., et. al.: A performance vs. cost framework for evaluating dht design tradeoffs under churn. In: INFOCOM'05. pp. 225–236 (2005)
11. Mahajan, R., Castro, M., Rowstron, A.: Controlling the cost of reliability in peer-to-peer overlays. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735, Springer, Heidelberg (2003)
12. Pandurangan, G., Raghavan, P., Upfal, E.: Building low-diameter p2p networks. In: FOCS, pp. 492–499 (2001)
13. Rhea, S., Geels, D., Roscoe, T., Kubiatowicz, J.: Handling churn in a DHT. In: USENIX 2004, pp. 127–140 (2004)
14. Saroiu, S., Gummadi, P.K., Gribble, S.D.: A measurement study of peer-to-peer file sharing systems. In: SPIE/ACM MCN 2002, pp. 156–170 (2002)
15. Siewiorek, D.P., Swarz, R.S.: Reliable computer systems (3rd ed.): design and evaluation. A.K. Peters, Ltd., Wellesley (1998)
16. Sriram, K., Whitt, W.: Characterizing superposition arrival processes in packet multiplexers for voice and data. IEEE JSAC 4(6), 833–846 (1986)
17. Wang, X., et al.: Robust lifetime measurement in large-scale p2p systems with non-stationary arrivals. In: IEEE P2P 2009, pp. 101–110 (2009)
18. Wu, D., Tian, Y., Ng, K.W., Datta, A.: Stochastic analysis of the interplay between object maintenance and churn. Comput. Commun. 31, 220–239 (2008)
19. Yao, Z., Leonard, D., Wang, X., Loguinov, D.: Modeling heterogeneous user churn and local resilience of unstructured p2p networks. In: ICNP, pp. 32–41 (2006)

Introduction

Dariusz Kowalski, Pierre Sens,
Antonio Fernandez Anta, and Guillaume Pierre

Topic chairs

Parallel computing is increasingly exposed to the development and challenges of distributed systems, such as asynchrony, long latencies, failures, network partitions, mobility, heterogeneity, malicious and selfish behavior, disconnected operations, the lack of load balancing, and many others. Furthermore, distributed systems are becoming larger, more diverse and more dynamic, for example, in terms of highly dynamic number of participants and topology changes. The Euro-Par topic dedicated to distributed systems and algorithms provides a forum for research and practice, of interest to both academia and industry, to present and discuss novel approaches in distributed computing and to explore relations between parallel processing and distributed systems.

We encouraged submission of papers across the whole area of distributed systems and algorithms, with emphasis on several classical and recent popular subareas, see the topic web page `europar2011.bordeaux.inria.fr/topic08.php` for details.

This year three papers were accepted. The paper "Productive Cluster Programming with OmpSs", by J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R.M. Badia, E. Ayguade, and J. Labarta, proposes a novel implementation of the system for cluster programming, supporting asynchrony, heterogeneity and data movement. Another paper — "On the Use of Cluster-Based Partial Message Logging to Improve Fault Tolerance for MPI HPC Applications" by T. Ropars, A. Guermouche, B. Ucar, E. Meneses, L.V. Kale, and F. Cappello — studies the communication patterns of message passing HPC applications in relation to partial message logging. The third paper "Object Placement for Cooperative Caches with Bandwidth Constraints", by U.C. Devi, M. Chetlur, and S. Kalyanaraman, develops an efficient solution to the caching problem motivated by high-volume streaming through bounded-constrained links.

We would like to take the opportunity of thanking all the authors who submitted their work to the topic, as well as all people involved in the organization and reviewing process within Euro-Par 2011. In particular, we would like to acknowledge the work of the external referees, who offered enormous help and expertise in reviewing and assessing papers from many different sub-areas covered by the topic.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, p. 554, 2011.

Productive Cluster Programming with OmpSs

Javier Bueno[1,2], Luis Martinell[1], Alejandro Duran[1], Montse Farreras[1,2], Xavier Martorell[1,2], Rosa M. Badia[1,3], Eduard Ayguade[1,2], and Jesús Labarta[1,2]

[1] Barcelona Supercomputing Center (BSC-CNS)
[2] Universitat Politècnica de Catalunya (UPC)
[3] Artificial Intelligence Research Institute (IIIA) - Spanish National Research Council (CSIC)

Abstract. Clusters of SMPs are ubiquitous. They have been traditionally programmed by using MPI. But, the productivity of MPI programmers is low because of the complexity of expressing parallelism and communication, and the difficulty of debugging. To try to ease the burden on the programmer new programming models have tried to give the illusion of a global shared-address space (e.g., UPC, Co-array Fortran). Unfortunately, these models do not support, increasingly common, irregular forms of parallelism that require asynchronous task parallelism. Other models, such as X10 or Chapel, provide this asynchronous parallelism but the programmer is required to rewrite entirely his application.

We present the implementation of OmpSs for clusters, a variant of OpenMP extended to support asynchrony, heterogeneity and data movement for task parallelism. As OpenMP, it is based on decorating an existing serial version with compiler directives that are translated into calls to a runtime system that manages the parallelism extraction and data coherence and movement. Thus, the same program written in OmpSs can run in a regular SMP machine, in clusters of SMPs, or even can be used for debugging with the serial version. The runtime uses the information provided by the programmer to distribute the work across the cluster while optimizes communications using affinity scheduling and caching of data.

We have evaluated our proposal with a set of kernels and the OmpSs versions obtain a performance comparable, or even superior, to the one obtained by the same version of MPI.

1 Introduction

Parallel and distributed programming has always been a difficult endeavour. But the rise in the number of systems and their complexity have put a stress in the way parallel applications are programmed. In recent years, there has been a significant effort to improve programming models to yield more productive models which still are able to provide good performance.

Applications for clusters have traditionally been programmed with MPI. But, while MPI allows to achieve very good performance it comes at the cost of programming at a very low-level which is error-prone and difficult to debug. Even more, as clusters become even larger obtaining high-performance requires using asynchronous communication and overlapping of computation which exacerbates the previous problems.

Shared-memory models, like OpenMP, offer a more productive and easy to debug environment but all efforts to devise implementations that could scale on clusters have

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 555–566, 2011.

failed so far except for some applications. Other models built specifically for clusters, like UPC or Co-array Fortran, try give programmers the illusion of a global address-space where data can be explicitly placed on each node and communications happen implicitly. But these models do not support well emerging task parallelism where work is more dynamic and synchronization and communication follow irregular patterns.

Asynchronous Partitioned Global Address Spaces (APGAS) languages, like X10 or Chapel, were created to address these needs but require programmers to rewrite their applications completely. Furthermore, to implement communication overlapping, data prefetch or locality scheduling, the compiler needs to implement complex and costly analysis of the application.

We designed OmpSs, which combines ideas from OpenMP[11] and StarSs[12], to try to tackle these problems. It enhances OpenMP with support for irregular and asynchronous parallelism and heterogeneous architectures. It incorporates the idea of disjoint address spaces that allows the compiler/runtime to automatically move data as necessary and perform different kinds of optimizations.

Our previous work has shown successful implementations of these ideas for multicore [12], the Cell B.E.[13] and GPUs[10]. In this work, we present the implementation of the model and evaluation with a set of applications for clusters of multicores. We show that using OmpSs the same application prepared to run in an SMP can be run in a cluster. The runtime takes care of moving the data around the different nodes as needed and of performing different optimizations to improve the overall performance.

Our results with different applications show that, compared with MPI, the speed-up obtained with OmpSs can be on par or even higher thanks to the asynchronous parallelism that can be expressed with it.

The paper is structured as follows: section 2 describes OmpSs, the programming model used to develop the presented work, section 3 presents the main contribution of this work, the design of Nanos++ for clusters, our implementation of OmpSs, section 4 shows the evaluation of Nanos++, section 5 describes related work to this project and section 6 concludes and discusses the future directions of this work.

2 OmpSs: From Multicores to Clusters

2.1 Overview

Our proposal is to have a single programming model, OmpSs, covering the different homogeneous and heterogeneous architectures in use today and opened to future ones. OmpSs is based on the OpenMP programming model with some modifications to its execution and memory model. This changes and additions come from ideas from the Star SuperScalar (StarSs) programming model.

StarSs is a programming model focused on exploiting asynchronous parallelism expressed using annotations on a sequential code like OpenMP. StarSs also targets different architectures. StarSs is actually the conjuction of a collection of programming models that targeted different architectures: CellSs, SMPSs, ClusterSs and GridSs.

Execution model. The OmpSs execution model is a thread-pool model instead of the traditional OpenMP fork-join model. The master thread starts the execution and all

other threads cooperate executing the work it creates (whether it is from worksharing or task constructs). Therefore, there is no need for a **`parallel`** region. Nesting of constructs allows other threads to become work generators as well.

Memory model. OmpSs assumes a non-homogeneous disjoint memory address space. As such shared data may reside in memory locations that are not directly accessible from some of the computational resources. Therefore, all parallel code can only safely access private data and for shared data it must specify how it is going to be used (see below). This assumption is true even for SMP machines as the implementation may reallocate shared data taking into account memory effects (e.g., NUMA).

Function tasks. OmpSs allows to annotate function declarations or definitions, *a la Cilk*[3], with a **`task`** directive. In this case, any call to the function creates a new task that will execute the function body. The data environment of the task will be captured from the function arguments.

Dependence synchronization. OmpSs integrates the StarSs dependence support[9]. It allows to annotate both task and worksharings constructs with three additional clauses:

input It specifies that the construct depends on some input data, and therefore, it is not eligible for execution until any previous construct with an **`output`** clause over the same data is completed.

output It specifies that the construct will generate some output data, and therefore, it is not elegible for execution until any previous construct with an **`input`** or **`output`** clause over the same data is completed.

inout It specifies a combination of **`input`** and **`output`** over the same data.

The target construct. To support heterogeneity and data motion between address spaces a new construct is introduced: the **`target`** construct[1]. The **`target`** construct can be applied to either **`task`**, worksharing constructs or functions. Its syntax is:

```
1 #pragma omp target [clauses]
2   task construct | worksharing construct | function definition | function header
```

Where the possible clauses are:

device. It allows to specify on which devices should be targeting the construct (e.g., cell, gpu, smp, ...). If no **`device`** clause is specified then the target devices are decided by the implementation (by default SMP).

copy_in. It specifies that a set of shared data may be needed to be transferred to the device before the associated code is going to be executed.

copy_out. It specifies that a set of shared data may be needed to be transferred from the device after the associated code is executed.

copy_inout. This clause is a combination of **`copy_in`** and **`copy_out`**.

copy_deps. It specifies that if the attached construct has any dependence clauses then they will also have copy semantics (i.e., **`input`** will also be considered **`copy_in`**, **`output`** **`copy_out`** and **`inout`** **`copy_inout`**).

implements. It specifies that the code is an alternate implementation for the target devices of the function name specified in the clause. This alternate can be used instead of the original if the implementation considers it appropriately.

The different **`copy`** clauses are advisory and not mandatory. This allows the implementation to take advantage of devices with access to the shared memory or implement different caching and prefetch techniques. To make sure that data that could have moved to a device is valid again in the host, SMP code must also use the **`copy`** clauses or appear after an OpenMP **`flush`** (either explicit or implicit).

2.2 Example

Fig. 1 shows some of these features applied to a SparseLU code. The master thread traverses the *kk* loop and because a **`task`** construct prepends the functions *lu0*, *fwd*, *bdiv* and *bmod*, it will spawn tasks for each of the calls. When the tasks are created, the runtime will use the dependence information to build a dynamic task graph. The `[BS][BS] pointer` notation specifies that a block of size $BSxBS$ is being pointed by the pointer. Task with no predecessors will be elegible for execution and they will release new tasks as they finish. Note that because of the sparseness of the computation it is difficult to have a pre-computed task dependence graph.

```
1  #pragma omp target copy_deps
2  #pragma omp task input([BS][BS] diag) inout([BS][BS] col)
3  void fwd(float *diag, float *col);
4  #pragma omp target copy_deps
5  #pragma omp task input([BS][BS] row, [BS][BS] col) inout([BS][BS] inner)
6  void bmod(float *row, float *col, float *inner);
7  #pragma omp target copy_deps
8  #pragma omp task input([BS][BS] diag) inout([BS][BS] row)
9  void bdiv(float *diag, float *row);
10 #pragma omp target copy_deps
11 #pragma omp task inout([BS][BS] diag)
12 void lu0(float *diag);
13
14 for (kk=0; kk<NB; kk++) {
15   lu0(A[kk][kk]);
16   /* fwd phase */
17   for (jj=kk+1; jj<NB; jj++) {
18     if (A[kk][jj] != NULL)
19       fwd(A[kk][kk], A[kk][jj]);
20   /* bdiv phase */
21   for (ii=kk+1; ii<NB; ii++)
22     if (A[ii][kk] != NULL)
23       bdiv (A[kk][kk], A[ii][kk]);
24   /* bmod phase */
25   for (ii=kk+1; ii<NB; ii++)
26     for (jj=kk+1; jj<NB; jj++)
27       if (A[kk][jj] != NULL) {
28         if (A[ii][jj]==NULL) A[ii][jj]=allocate_clean_block();
29         bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
30       }
31 }
32 #pragma omp taskwait
```

Fig. 1. SparseLU example with OmpSs

The **`target`** directive prepending each task specifies that the execution of the tasks requires that the specified data (in this case the same as the data dependences) is ready in the location where the task is executed. In case of a pure SMP run this information will be ignored, but for our cluster implementation it means that the runtime will need to move the data as necessary. After, the **`taskwait`** all tasks will be completed and the data will be available to the master thread (so it could print the result for example).

3 Implementation

Nanos++ overview. The OmpSs infrastructure is composed by the Mercurium compiler and the Nanos++ runtime library. The compiler gets as an input the program annotated with the OmpSs directives and generates a transformed version that invokes services of the Nanos++ runtime.

Nanos++ is an extensible runtime library that supports OmpSs. Its responsibility is to execute *task parallel* applications as specified by the compiler. Nanos++ offers mechanisms to schedule the execution of the tasks. The runtime schedules these tasks on the available resources making sure all constraints specified by the user (order, coherence, ...) are maintained. Nanos++ comes with a few scheduling policies (e.g. fifo, lifo, ...) but allows new ones by means of plug-in extensions.

Most of the runtime is independent from the actual target architectures supported (and various of these architectures can be active at the same time). Nanos++ currently supports several "conceptual" architectures: *smp*, *smp-numa*, *gpu*[10], *tasksim* (a simulated architecture)[14] and cluster.

In the following sections we describe the Nanos++ *cluster* architecture, the general ordering and coherence mechanisms that ensure the correctnes of the execution and the data-affinity scheduler that we have implemented to improve performance.

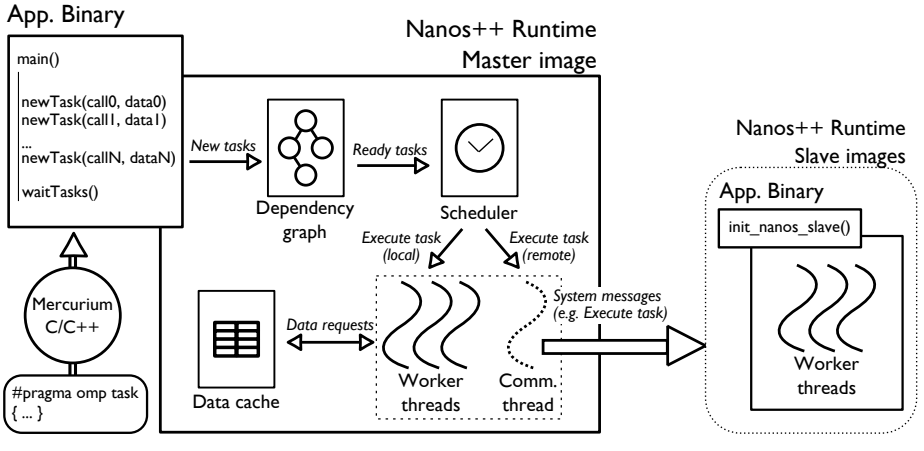


Fig. 2. Nanos++ cluster overview

Nanos++ cluster architecture. Fig. 2 shows the general design of Nanos++ for clusters. The main difference with respect to other supported architectures is that, when running in a cluster, there will be more than one image of the runtime running at the same time (i.e., one on each cluster node). When the execution starts, the first image will become the *master* image and the rest of the images will become the *slave* images. This structure creates an identical address space on each node, which gives the view of a single distributed address space. This eases the implementation of the proposed programming model(s) but it limits the total amount of memory used by the program (i.e., it can use only as much memory as is available in the master node).

All low level communications for control information and data transfers are implemented using *active messages*. We used GASNet [4] for this functionality since it offers a network-independent API with native support for various network technologies.

Initially there is only one task that is executed by the master. As this task starts creating new tasks, they will be scheduled to the local threads of the master node and to a communication thread that represents the remote nodes. When a task is scheduled to the communication thread it will be executed by a remote node. There is only one communication thread[1] that will be pooling the task pool for each node of the cluster in a round-robin fashion. This thread also keeps track of the execution of tasks on remote nodes, when a node has no task on execution, it will send a new one to it.

The remote execution of tasks is a straightforward process. First, the general coherence mechanisms of the runtime are invoked to ensure that all data that will be needed by a task is available in the remote node (and it is up-to-date). If not, data is gathered from its current location. If the data is available in the master node it is sent directly to the remote node. If it is only available in a remote node, a message is sent to that node so it sends the data to the new owner. This first step can be done concurrently with the execution of other remote tasks to overlap communication and computation but our current implementation does not apply this optimization yet.

After this, the master sends a control message with the task information to start the execution of the remote task. The slave images are constantly waiting for upcoming requests and they will start the execution of the task as soon as the request arrives. When the task finishes, another active message is sent back to the master to notify the completion of the task.

Tasks executed in a remote node can create new tasks that use the data transferred or created by their parent task. This allows scalable data decomposition to be coded in the application. These local tasks will be executed by any thread that becomes available in the node (and before going to fetch more work from the master node). Currently, we do not implement stealing between the local queues of the slave nodes.

Nanos++ coherence support. To run tasks in architectures with separated address spaces, is necessary a mechanism that copies data from the host (or where the task was created) to where the task will execute and to control the coherence between the different address spaces. With this mechanism Nanos++ is able to schedule tasks to run everywhere in the system.

A centralized directory keeps track of the physical location of the data and one software *cache* per cluster node stores data for tasks executed in that node. This *cache*

[1] Our design allows to have more than one if necessary.

manages the transfers from main memory to the remote nodes memory, avoiding unnecessary data movement and implementing different coherence policies (by default a writeback policy is used).

From the point of view of the runtime, copy operations can be of two types: synchronous and asynchronous. Having synchronous copies means that the cache will wait until all data is available when preparing a task for execution. Asynchronous copies allow the runtime to place all copy operations of a to-be-scheduled task and start doing other things while data is being transferred for that task. Although asynchronous operations are not implemented for clusters (as mentioned before), it is relatively easy to incorporate to the Nanos++ cluster architecture and would enable the runtime to overlap data transfers and computation.

It is important to notice that the coherence mechanisms assume program correctness. Applications where tasks write to the same data simultaneously without specifying dependencies result in an undefined behavior.

Nanos++ task scheduler. Minimizing the number of transfers through the network is critical in clusters to avoid network saturation and reduce the impact of latency in performance. A locality-aware scheduling policy has been implemented to favor scheduling of tasks in the remote nodes where most of their data reside. Directory entries store a map with the location of data in the system. When a new task is submitted, the scheduler computes, for each node, an affinity score based on the location and size of the data needed by the task. This score is used to place the task in the queue of the node with the highest affinity. A global queue is used when there is no node with the highest affinity.

The runtime looks for work for work on each node's queue first, if it is empty, the global queue is used, if this is empty too then another node's queue may be used to fetch a ready task, this aims to prevent application imbalance.

4 Evaluation

4.1 Methodology

In order to evaluate our runtime environment we measured the scalability of several applications on a cluster of SMPs. We implemented two versions of each application: one using OmpSs and one using MPI. With this, we compared the performance of the OmpSs version while running with Nanos++ with the performance obtained by the MPI version. We consider this a good measure of how good can OmpSs can be compared to a well known standard like MPI.

Environment. The benchmarks were run in the MareNostrum cluster of PowerPC 970MP @ 2.3GHz processors. Each node has 2 CPUs with 2 cores each and 4 Gb of physical memory and runs the SLES 10 operating system. The interconnection network of the cluster is based on Myrinet hardware along with the Myrinet Express driver. All benchmarks were compiled using the Mercurium C++ compiler version 1.3.5.7 using the GCC compiler as the backend compiler, -O3 optimization level was always used.

OmpSs was run using the MPI conduit for GASNet. Since there is no specific conduit available for the Myrinet Express driver, the MPI conduit allowed us to indirectly use Myrinet Express through the MPICH library that was installed on the system.

Applications

Matrix Multiply. It performs a dense matrix multiplication of two square matrices. Each matrix is divided in blocks; in the MPI version this is used to tile the execution of the algorithm, in the OmpSs version tiling is also applied, however the algorithm is structured slightly different. In the OmpSs version there are two different types of tasks; the mission of the first type tasks is to create the other type of tasks, which are the ones that perform the computation, and also distribute the data implicitly during the process. Using this schema the second tasks benefit from better data locality, since the parent task already requested their needed data. The MPI version and the OmpSs version used a simple kernel in order to perform the matrix multiplication. The matrices had 32x32 blocks of 400x400 doubles on both versions.

NAS EP. The EP benchmark generates pairs of Gaussian random deviates according to a specific scheme. The main loop keeps all its data private until the end of the execution, where a reduction is done. The implementation for OmpSs divides the main loop of the benchmark in tasks, and implements the reduction manually. We implemented the OmpSs version porting it from the C implementation of the 2.3 NAS Parallel Benchmarks. The MPI version comes from the original NPB v2.3 for Fortran. We used the class C problem size.

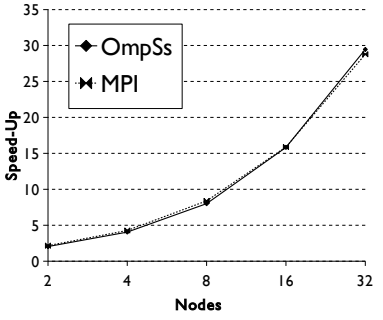
STREAM. STREAM is a benchmark that measures memory bandwith for simple kernels, intended for use with large data sets. It performs 4 simple operations on a three one dimensional arrays. It does not share any data between nodes so, as EP, we expected to be able to run STREAM without any problems with OmpSs. We used 500 Mb arrays in order to use the maximum memory that the GASNet configuration used could handle.

Sparse LU. The Sparse LU computes a LU decomposition on a sparse matrix and can have empty blocks. Due to the sparseness, the total number of tasks generated is less than in a regular LU. Task parallelism with dependencies can benefit from this situation as it can overlap multiple iterations at the same time whereas MPI needs barriers across different iterations. The matrix size used was the same as for the Matrix Multiply, 32x32 blocks of 400x400 doubles.

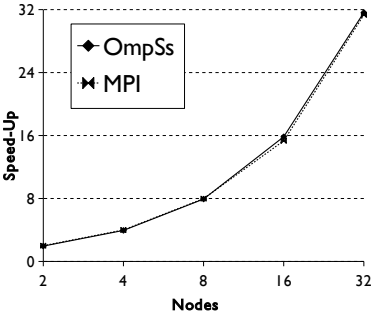
Experiments. We run the selected applications with different configurations of numbers of nodes to obtain the speed-up of each application for both OmpSs and MPI. We selected the biggest data set possible, since we did not wanted that the performance obtained was limited due to using a small problem size. As a baseline, for the speed-up we use the execution time of the serial version.

4.2 Results

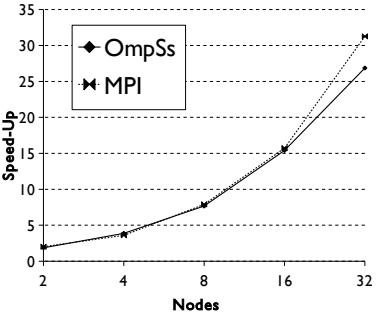
Matrix Multiply. The benchmark achieves a good performance on OmpSs, almost identical to the MPI version. This is somewhat expected since Matrix Multiply has a lot of data parallelism that can be exploited efficiently using either MPI or task parallelism. Figure 3(a) shows the results obtained for both OmpSs and MPI, perfect scalability is not achieved since communication and computation are not overlapped, but the scalability of the OmpSs code is on par with the MPI code.



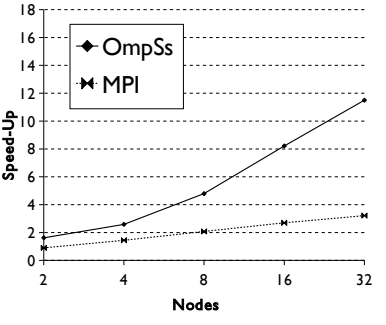
(a) Matrix Multiply (32x32 blocks of 400x400 doubles)



(b) NAS EP (Class C)



(c) STREAM (500Mb arrays)



(d) Sparse LU (32x32 blocks of 400x400 doubles)

Fig. 3. Results comparing OmpSs and MPI

STREAM. The results obtained by the STREAM benchmark are almost on par with MPI. As seen in figure 3(c) We achive almost the same scalability, only losing a little bit of performance due to the centralized initialization of the tasks. MPI outperforms OmpSs because of the SPMD model, since, besides the first synchronization done when initializing the MPI runtime, there is nothing to setup. On the other hand the creation of tasks in OmpSs takes some time, and it is proportional to the number of tasks, in addition, the distribution of these tasks also takes time proportional to the number of nodes of the execution.

Sparse LU. Sparse LU performs better than MPI since it exploits the advantages of having fine-grained tasks with dependences. This is specially important in sparse matrices since data parallelism is lower than in non-sparse ones. The MPI code can also be optimized in order to be conscious of this sparseness, however the changes we did to the benchmark only optimized the amount of data transferred between nodes, keeping the same application structure. On the other hand, the changes to the OmpSs code were minimal, and the benefits measured were greater than with the MPI version. Figure 3(d) shows this effect, where the OmpSs application achieves higher scalability than MPI on any number of nodes.

NAS EP. The EP benchmark has almost no data sharing among tasks, so it fits well on the set of applications that can achieve a good performance on distributed environments. With OmpSs there is no exception and the results showed a perfect lineal speed-up, on par with the original MPI implementation. Figure 3(b) shows the results we obtained executing the class C of the benchmark.

5 Related Work

Probably the best well known examples of parallel programming models are OpenMP and MPI. However, each of them has its own disadvantages and there has always been a good number of projects trying to address them or proposing new features in order to make them more suitable for the HPC systems we can find nowadays.

OpenMP was designed to provide a high productivity environment to produce parallel programs. Originally focused on dealing with loop-based parallelism, it was recently updated to the version 3.0, which includes new ways of expressing parallelism in the form of *tasks* [11].

OpenMP has influenced many projects due to its ease of use and simplicity. Cilk[3] is an example of a programming model which also provides task-based parallelism that can be expressed with simple keywords in a sequential code. Distributed Shared Memory (DSM) systems have also tried to offer this simple vision of a distributed environment by adding an extra software and/or hardware layer to the memory hierarchy that virtualizes the address space of the applications, allowing OpenMP, or other applications conceived for shared memory, to run on distributed memory architectures. The big disadvantage of these systems is that the memory access time increases dramatically, difficulting the task of achieving a reasonable performance. Techniques like data pre-send and pre-fetch along with relaxed memory consistency [8] have tried to overcome this, however, only a limited number of applications have benefited from such techniques.

Basumallik et al.[2] presented another approach that aimed to translate OpenMP to MPI, focusing on parallel loops. While it has achieved a good performance when running several OpenMP benchmarks, it does not offer asynchronous parallelism like we provide with OmpSs.

MPI has been a de facto standard in parallel programming for distributed environments. It offers explicit communication calls to transfer data among a set of processes running on different nodes of a cluster. The main disadvantage is that this approach can be complex to apply to some applications, and it requires a lot of effort from the programmer.

Partitioned Global Address Space (PGAS) programming models have tried to simplify all this burden, and offer a more friendly environment to develop distributed applications. They try to accomplish this by providing a global address space, which is distributed among the memory of each node of the execution. With this, they offer the programmer a simplified vision of the distributed environment, easing the development and porting of sequential applications to the PGAS. UPC [7] is one of these models, it takes also ideas from OpenMP in the form of compiler annotations but also has explicit communication calls.

Chapel [5] and X10 [6] implement an Asynchronous PGAS (APGAS), which offer asynchronous parallelism and mechanisms to synchronize it. In both environments is the responsibility of the programmer to deal with the data distribution and coherence.

An alternative way to provide asynchronous parallelism on clusters is the one explored by Marjanovic et al.[15], a hybrid programming model that composes SMPSs, a programming model that inspired OmpSs, with MPI. The main idea is to encapsulate the communications in tasks so they are executed when the data is ready. This technique achieves an asynchronous dataflow execution of both communication and computation.

6 Conclusions and Future Work

We have presented an implementation of OmpSs for clusters, a programming model that aims to be a high productivity environment without loss of performance when compared to other solutions. Coming from StarSs and OpenMP, OmpSs parallelization comes in the form of compiler directives that can be used to annotate sequential code. With this annotated code, the Mercurium C++ compiler can generate parallel code to run on top of the Nanos++ runtime. Applications built this way can be run on several architectures including GPUs and clusters of SMPs. In this work, we have evaluated the performance of different applications when running on a cluster of SMPs, and we have compared the performance obtained against the same applications developed with MPI. The performance achieved by OmpSs is on par with the performance obtained by MPI and even in some cases it can outperform MPI thanks to the asynchronous parallelism implemented in the form of task-based parallelism with data dependencies.

Since Nanos++ is a young project there is still much work to be done. We plan to implement techniques that allow us to overlap computation and communication, in the form of pre-sending or pre-fetching data before a tasks starts the execution, this will be done in collaboration with a more conscious scheduling. Also, one of our goals is to being able to scale further than the number of nodes that we have presented on this work, and also to offer a better handling of multiple threads on slave images. To achieve this, we will have to implement techniques to improve data distribution and allow local memory allocation on the remote nodes. This will allow us to fully use the physical memory of the cluster. Another direction we would like to explore is to include other devices into the cluster architecture, making Nanos++ capable of managing the execution of a single application on a cluster composed by SMPs and GPU devices.

Acknowledgments. We thankfully acknowledge the support of the European Commission through the HiPEAC-2 Network of Excellence (FP7/ICT 217068) and the ENCORE project (FP7-248647), the support of the Spanish Ministry of Education (TIN2007-60625, and CSD2007-00050), the Generalitat de Catalunya (2009-SGR-980) and the TEXT project (IST-2007-261580).

References

1. Ayguade, E., Badia, R., Cabrera, D., Duran, A., Gonzalez, M., Igual, F., Jimenez, D., Labarta, J., Martorell, X., Mayo, R., Perez, J.M., Quintana-Orti, E.S.: A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures. In: IWOMP: Evolving OpenMP in an Age of Extreme Parallelism, Dresden, Germany, pp. 154–167 (June 2009)
2. Basumallik, A., Eigenmann, R.: Towards automatic translation of openmp to mpi. In: Proceedings of the 19th Annual International Conference on Supercomputing, ICS 2005, pp. 189–198. ACM, New York (2005)
3. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. SIGPLAN Not. 30(8), 207–216 (1995)
4. Bonachea, D.: GASNet Specification, v1.8. Technical report, U.C. Berkeley (2006)
5. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the chapel language. Int. J. High Perform. Comput. Appl. 21, 291–312 (2007)
6. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, New York, NY, USA (2005)
7. UPC Consortium. UPC Language Specifications v1.2 (May 2005)
8. Costa, J.J., Cortes, T., Martorell, X., Ayguade, E., Labarta, J.: Running OpenMP applications efficiently on an everything-shared SDSM. J. Parallel Distrib. Comput. (May 2006)
9. Duran, A., Pérez, J.M., Ayguadé, E., Badia, R.M., Labarta, J.: Extending the OpenMP Tasking Model to Allow Dependent Tasks. In: OpenMP in a New Era of Parallelism, pp. 111–122. Springer, Heidelberg (2008)
10. Ferrer, R., Planas, J., Bellens, P., Duran, A., Gonzalez, M., Martorell, X., Badia., R., Ayguade, E., Labarta, J.: Optimizing the Exploitation of Multicore Processors and GPUs with OpenMP and OpenCL. In: Proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC 2010) (October 2010)
11. OpenMP ARB. OpenMP Application Program Interface, v. 3.0 (May 2008)
12. Josep, M., Perez, R.M.: Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. In: IEEE Int. Conference on Cluster Computing, pp. 142–151 (September 2008)
13. Perez, J.M., Bellens, P., Badia, R.M., Labarta, J.: CellSs: Making it easier to program the Cell Broadband Engine processor. IBM Journal of Research and Development 51(5), 593–604 (2007)
14. Rico, A., Duran, A., Cabarcas, F., Ramirez, A., Etsion, Y., Valero, M.: Trace-driven Simulation of Multithreaded Applications. In: Proceedings of the 2011 ISPASS (to appear, 2011)
15. Ayguadé, E., Marjanovic, V., Labarta, J., Valero, M.: Effective communication and computation overlap with hybrid mpi/smpss. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2010, pp. 337–338. ACM, New York (2010)

On the Use of Cluster-Based Partial Message Logging to Improve Fault Tolerance for MPI HPC Applications

Thomas Ropars[1], Amina Guermouche[1,2], Bora Uçar[3], Esteban Meneses[4], Laxmikant V. Kalé[4], and Franck Cappello[1,4]

[1] INRIA Saclay-Île de France, France
thomas.ropars@inria.fr
[2] Université Paris-Sud
guermou@lri.fr
[3] CNRS and ENS Lyon, France
bora.ucar@ens-lyon.fr
[4] University of Illinois at Urbana-Champaign, USA
{emenese2,kale,cappello}@illinois.edu

Abstract. Fault tolerance is becoming a major concern in HPC systems. The two traditional approaches for message passing applications, coordinated checkpointing and message logging, have severe scalability issues. Coordinated checkpointing protocols make all processes roll back after a failure. Message logging protocols log a huge amount of data and can induce an overhead on communication performance. Hierarchical rollback-recovery protocols based on the combination of coordinated checkpointing and message logging are an alternative. These partial message logging protocols are based on process clustering: only messages between clusters are logged to limit the consequence of a failure to one cluster. These protocols would work efficiently only if one can find clusters of processes in the applications such that the ratio of logged messages is very low. We study the communication patterns of message passing HPC applications to show that partial message logging is suitable in most cases. We propose a partitioning algorithm to find suitable clusters of processes given the communication pattern of an application. Finally, we evaluate the efficiency of partial message logging using two state of the art protocols on a set of representative applications.

1 Introduction

The generation of HPC systems envisioned for 2018-2020 will reach Exascale from 100s of millions of cores. At such scale, failures cannot be considered as rare anymore and fault tolerance mechanisms are needed to ensure the successful termination of the applications. In this paper, we focus on message passing (MPI) HPC applications. For such applications, fault tolerance is usually provided through rollback-recovery techniques [9]: the state of the application processes is saved periodically in a checkpoint on a reliable storage to avoid

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 567–578, 2011.

restarting from the beginning in the event of a failure. In most cases, rollback-recovery protocols are used flat: the same protocol is executed for all processes.

Flat rollback-recovery protocols have several drawbacks. Coordinated checkpointing requires to restart all processes in the event of a failure leading to a massive waste of resources and energy. Message logging protocols need to log all messages contents and delivery order, which leads to high storage resource occupation, high communication overhead and high energy consumption.

One way to cope with these limitations is to use hierarchical rollback-recovery protocols [11, 12, 15, 16, 21]. Clusters of processes are defined during the execution and different protocols are used inside and between clusters, giving a hierarchical aspect to the protocol. Typically, a partial message logging protocol logs messages between clusters to confine the effects of a process failure to one cluster and a coordinated checkpointing protocol is used within clusters [21, 15]. The efficiency of these protocols depends on two conflicting requirements: i) the size of the clusters should be small to limit the rollbacks in the event of a failure; ii) the volume of inter-cluster messages should be low to limit the impact of logging on failure free performance.

This paper provides three main contributions: i) it shows that suitable process clustering can be found in most applications; ii) it proposes a bisection-based partitioning algorithm to find such clusters in an application based on its execution communication pattern; iii) it shows that partial message logging limits the amount of computing resources wasted for failure management compared to flat rollback-recovery protocols.

The paper is organized as follows. Section 2 details the context of this work, and presents the related work on MPI applications communications analysis. Section 3 analyzes a set of execution communication patterns in MPICH2. Section 4 presents our bisection-based partitioning algorithm, designed to address the partitioning problem for a partial message logging protocol. Using this algorithm, we evaluate the performance of two *state-of-the-art* partial message logging protocols [11, 15] on a set of representative MPI HPC applications. Results are presented in Section 5. Finally, conclusions are detailed in Section 6.

2 Context

In this section, we first present existing hierarchical rollback-recovery protocols. Then we present the applications studied in this paper. Finally, we detail the related work on analyzing characteristics of MPI HPC applications.

Hierarchical Rollback-Recovery Protocols. Rollback-recovery techniques are based on saving information during the execution of an application to avoid restarting it from the beginning in the event of a failure. One of the main concerns is the amount of resources wasted with respect to computing power or energy. Several factors are contributing to this waste of resources: i) the overhead on performance during failure free execution; ii) the amount of storage resources used to save data; iii) the amount of computation rolled back after a failure.

Rollback-recovery protocols are usually divided into two categories: checkpointing-based and logging-based protocols [9]. In checkpointing protocols, processes checkpoints can be either coordinated at checkpoint time, taken independently or induced by the communications. For all these protocols, a single failure implies the rollback of all processes in most cases, which is a big waste of resources. On the other hand, message logging protocols log the content as well as the delivery order (determinant) of the messages exchanged during the execution of the application to be able to restart only the failed processes after a failure. However, logging all messages during a failure free execution can be very wasteful regarding communications and storage resources.

Hierarchical rollback-recovery protocols divide the processes of the applications into clusters and apply different protocols for the communications inside a cluster and for the communications among clusters. Our work focuses on partial message logging protocols [11, 12, 15, 21] which apply a checkpointing protocol inside the clusters and a message logging protocol among the clusters. They are attractive at large scale because only one cluster has to rollback in the event of a single failure. These protocols can work efficiently if the volume of inter-cluster messages is very low in which case the cost of message logging is small.

In this paper, we use two of these protocols for evaluations. Meneses et al. [15] use a coordinated checkpointing protocol inside clusters. Intra-cluster messages determinants are logged to be able to replay these messages in the same order after the failure and reach a consistent state. Considering a single failure, determinants can be logged in memory. In this study [15] is comparable to [12] and [21]. Guermouche et al. [11] propose an uncoordinated checkpointing protocol without domino effect, relying on the send-determinism of MPI HPC applications. Using this protocol, an ordered set of p clusters can be defined. Only messages going from one cluster to a *higher* cluster are logged, limiting the number of clusters to roll back after a failure to $(p+1)/2$ on average. Thanks to send-determinism, this protocol does not require any determinant to be logged and can tolerate multiple concurrent failures.

Applications Studied. We study a representative set of MPI HPC applications to see if partial message logging could be used. Thirteen dwarfs have been defined and seven of them represent seven main classes of computation and communication patterns corresponding to numerical methods for high-end simulation in the physical sciences [2]: dense linear algebra, sparse linear algebra, spectral methods, N-body methods, structured grids, unstructured grids, and MapReduce. This paper does not consider MapReduce applications because rollback-recovery is not adapted in this case. Our set of applications includes five of the NAS Parallel Benchmarks (NPB) [3] containing BT, LU, CG, FT, and MG; three of NERSC-6 Benchmarks [1] (GTC, MAESTRO, and PARATEC); one of the Sequoia Benchmarks, `http://asc.llnl.gov/sequoia/benchmarks/`, (LAMMPS); and an Nbody kernel. Table 1 summarizes the dwarfs covered.

HPC Applications Communications Characteristics. The communication patterns of most of the applications considered in this paper have already

Table 1. Dwarfs covered by the studied applications

Dense linear algebra	Sparse linear algebra	Spectral methods	N-body simulations	Structured grids	Unstructured grids
BT, LU, PARATEC	CG, MAESTRO	FT, PARATEC	GTC, LAMMPS, Nbody	MG, GTC, MAESTRO, PARATEC	MAESTRO

been published [1, 18]. Previous studies highlighted some properties. First, most MPI applications make use of collective communications, but in general with a very small payload size that remains invariant with respect to the problem size [20]. Second, the communication graphs of many MPI HPC applications have a low degree of connectivity [13], which might indicate that processes can be partitioned into clusters.

3 Communication Patterns

In this section, we first study the communication patterns of some collective operations in MPICH2. Then, we present communication patterns we collected by running some applications. To get the communication patterns of MPI applications execution, we modified the code of MPICH2[1] to collect data on communications. The applications run on Rennes Grid'5000 cluster over TCP.

We focus on collective communications because they could generate patterns that are hard to cluster (they involve all processes in the application). Figure 1 presents the set of communication patterns used for MPICH2 collective communications, for a power-of-two number of processes (64 processes) and short messages (4 bytes). Details on the implementation of collective communications in MPICH2 can be found in [19]. The recursive doubling algorithm, Fig. 1(a), is used to implement *MPI_Allgather* and *MPI_Allreduce* operations. The recursive halving algorithm, used in *MPI_Reduce_scatter*, has the same communication pattern. A binomial tree, Fig. 1(b), is used in *MPI_Bcast*, *MPI_Reduce*, *MPI_Gather* and *MPI_Scatter*. These two patterns are easily clusterizable using for instance clusters of size 16. The last pattern, corresponding to a store-and-forward algorithm, is the one used for *MPI_Alltoall*. This pattern is more difficult to cluster, but the use of *MPI_Alltoall* should be limited in applications targeting very large scale. For large messages, many of the collective operations do involve communications between all application processes. Clustering is also difficult to apply to such patterns. However, as mentioned in Section 2, the payload for collective communications is usually small: on the set of applications we tested, we only found this pattern in NPB FT.

Although some of the collective communication patterns define natural clusters (see Fig. 1(a)), some others do not (see Fig. 1(c)). Furthermore, the point-to-point communications can render the patterns more sophisticated, so much so that the size or the number of clusters cannot be known a priori. Figure 2 presents the communication pattern of MAESTRO and GTC executed on 512

[1] http://svn.mcs.anl.gov/repos/mpi/mpich2/trunk:r7592

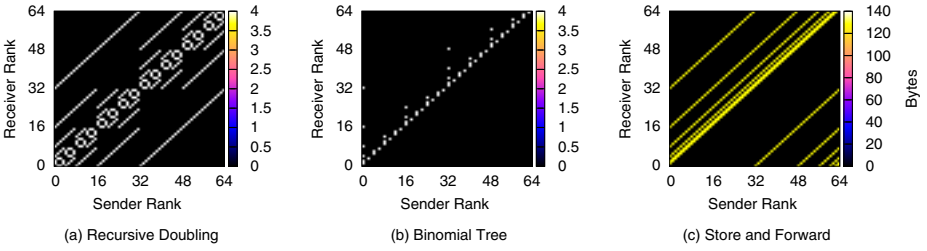


Fig. 1. Communication patterns inherent in collective communications of MPICH2

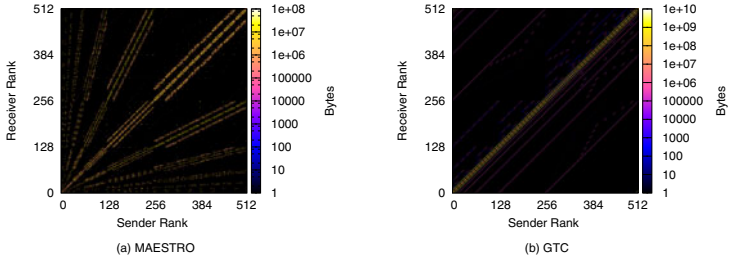


Fig. 2. Communications patterns of two applications

processes. As exemplified by these two applications, an automated means of clustering is strictly required to identify clusters of processes in an application.

4 Partitioning for Partial Message Logging Protocols

Here, we propose a bisection-based partitioning algorithm to automate clustering for partial message logging protocols. The objectives of the clustering are to reduce the inter-cluster communications, to increase the number of clusters, and to limit the maximum size of a cluster. We first illustrate the limits of existing tools and then present our new method.

4.1 Two Possible Approaches

A simplified version of the partitioning problem in which the objectives are to minimize the size of the logged messages and the maximum size of a part corresponds to the NP-complete graph partitioning problem (see the problem ND14 in [10]). This can be easily seen by considering a graph whose vertices represent the processes and whose edges represent the communication between the corresponding two processes. Using heuristics for the graph partitioning problem would require knowing the maximum part size. This can be done but requires an insight into the application and the target machine architecture.

A common variant of the above graph partitioning problem specifies the number of parts (in other words, specifies the average size of a part) and requires

parts to have similar sizes (thusly reducing the maximum size of a partition). The problem remains NP-complete [4]. Tools such as MeTiS [14] and Scotch [17] can be used to solve this problem. A possible but not an economical way to use those tools in our problem is to partition the processes for different number of parts (say 2, 4, 8, ...) and try to select the best partition encountered.

4.2 Bisection-Based Partitioning

Bisection based algorithms are used recursively in graph and hypergraph partitioning tools, including PaToH [6], MeTiS and Scotch, to partition the input graph or hypergraph into a given number of parts. Simply put, for a given number K of parts, this approach divides the original graph or hypergraph into two almost equally sized parts and then recursively partitions each of the two parts into $K/2$ parts until the desired number of parts is obtained.

We adapt the bisection based approach and propose a few add-ons to address our partitioning problem. The proposed algorithm is seen in Algorithm 1. The algorithm accepts a set of P processes and a matrix M of size $P \times P$ representing the communications between the processes where $M(u, v)$ is the volume of messages sent from the process u to the process v. The algorithm returns the number of parts $K^\star$ and the partition $\Pi^\star = \langle P_1, P_2, \ldots, P_K^\star \rangle$. In the algorithm, the operation $\Pi \leftarrow \Pi \ominus P_k \oplus P_{k_1} \oplus P_{k_2}$ removes the part P_k from the partition Π and adds P_{k_1} and P_{k_2} as new parts, where $P_k = P_{k_1} \cup P_{k_2}$. We use $M(U, V)$ to represent the messages from processes in the set U to those in the set V.

Algorithm 1. The proposed partitioning algorithm

Input: The set of P processes $= \{p_1, p_2, \ldots, p_P\}$; the $P \times P$ matrix M representing the communications between the processes

Output: $K^\star$: the number of process parts; $\Pi^\star = \langle P_1, P_2, \ldots, P_K^\star \rangle$: a partition of processes

1: $K^\star \leftarrow K \leftarrow 1$; $B^\star \leftarrow B \leftarrow 0$
2: $\Pi^\star \leftarrow \Pi \leftarrow \langle P_1 = \{p_1, p_2, \ldots, p_P\} \rangle$ ▶ a single part
3: **while** there is a part to consider **do**
4: Let P_k be the largest part
5: **if** ShouldPartition(P_k) **then**
6: $\langle P_{k_1}, P_{k_2} \rangle \leftarrow$Bisect($P_k, M(P_k, P_k)$)
7: **if** AcceptBisection(P_{k_1}, P_{k_2}) **then**
8: $K \leftarrow K + 1$
9: $\Pi \leftarrow \Pi \ominus P_k \oplus P_{k_1} \oplus P_{k_2}$ ▶ replace P_k
10: $B \leftarrow B + \sum M(P_{k_1}, P_{k_2})$ ▶ Update volume of inter-parts messages
11: **if** BestSoFar(Π) **then**
12: $\Pi^\star \leftarrow \Pi$; $B^\star \leftarrow B$; $K^\star \leftarrow K$ ▶ save for output
13: **else**
14: mark P_k in order not to consider it again at lines 3 and 4.
15: **return** $\Pi^\star$

The algorithm uses bisection to partition the given number of processes into an unknown number of parts. Initially, all the processes are in a single part.

At every step, the largest part is partitioned, if it should be, into two (by the subroutine Bisect), and then if the bisection is acceptable (determined by the subroutine AcceptBisection), that largest part is replaced by the two parts resulting from the bisection. If this new partition is the best one seen so far (tested in subroutine BestSoFar), it is saved as a possible output. Then, the algorithm proceeds to another step to pick the largest part. We now add a few details. Let P_k be the largest part, then if P_k should be partitioned (tested in the subroutine ShouldPartition), it is bisected into two P_{k_1} and P_{k_2} and tested for acceptance; if not accepted then the bisection is discarded and P_k remains as is throughout the algorithm. Notice that when a bisection of P_k is accepted, then P_k is partitioned for good; if any BestSoFar test after this bisection returns true, then P_k will not be in the output $\Pi^\star$.

The computational core of the algorithm is the Bisect routine. This routine accepts a set of processes and the communication between them and tries to partition the given set of processes into two almost equally sized parts by using existing tools straightforwardly. One can use MeTiS or Scotch quite effectively if the communications are bidirectional ($M(u,v) \neq 0 \implies M(v,u) \neq 0$), in which case $\frac{M+M^T}{2}$ can be used. Alternatively one can use PaToH as each communication (bidirectional or not) can be uniquely represented as an edge.

The routines ShouldPartition, AcceptBisection, and BestSoFar form the essence of the algorithm. The routine BestSoFar requires a cost function to evaluate a partition. The cost function should be defined based on the metric the user wants to optimize, e.g., the performance overhead, and the characteristics of the targeted partial message logging protocol. It is function of the part sizes and of the volume of inter-parts messages. We define a cost function for both of the mentioned rollback-recovery protocols later in Section 5.1.

ShouldPartition is used to stop partitioning very small parts. It returns false if the size of the part in question is smaller than a threshold. Although we mostly used 1 as threshold in our experiments, using a larger threshold will make the algorithm faster. One could select this threshold based on a minimal part size considering the properties of the target machine architecture.

The routine AcceptBisection returns true in two cases. The first one is simply that for Π and $\Pi' = \Pi \ominus P_k \oplus P_{k_1} \oplus P_{k_2}$, we have $\text{cost}(\Pi') \leq \text{cost}(\Pi)$ according to the cost function. If this does not hold, we may still get better partitions with further bisections. To this end, we have adapted the graph strength formula [7]. For a partition $\Pi = \langle P_1, P_2, \ldots, P_K \rangle$, we use $\text{strength}(\Pi) = \frac{B}{K-1}$ as its strength, B being the volume of inter-parts data. We accept the bisection if $\text{strength}(\Pi') \leq \text{strength}(\Pi)$. If the bisection reduces the strength, then it can be beneficial, as it increases the size of the logged messages only a little with respect to the number of parts.

5 Evaluation

This section presents our experimental results. For the evaluations, we consider the two partial message logging protocols described in Section 2. We start by

defining a cost function for each of them, evaluating the amount of wasted computing resources for failure management. Then we present the results obtained by running our implementation of the proposed partitioning algorithm using these cost functions, on the set of applications executions data we collected. We first show that these results overcome coordinated checkpointing and message logging protocols. Then we validate our partitioning algorithm by comparing our results to the results obtained by the existing graph partitioning tools.

5.1 Defining a Cost Function

We define a cost function to evaluate the amount of computing resources wasted by failure management, for the two partial message logging protocols. We would like to stress that the cost functions we define in this section do not give a very precise evaluation of the protocols, because a lot of parameters would have to be taken into account. We use some parameters, that we consider realistic, to provide an insight of the protocols cost. The main goal in this section, is to show that we manage to find clustering configurations with good trade-off between the clusters size and the amount of data logged.

We model the cost of a partial message logging protocol for a given clustering configuration as a function of the total volume of logged messages and the size of the clusters. We use a formula of the form

$$\text{cost}(\Pi) = \alpha \times L + \beta \times R \tag{1}$$

where L is the ratio of logged data and R is the ratio of processes to restart after a failure. The multipliers α and β are the cost associated with message logging and with restarting processes after a failure, respectively.

To evaluate the overhead of message logging, we use results from [11], where message logging impact on communications (latency and bandwidth) on a high performance network results in a 23% performance drop on average. Therefore, $\alpha = 23\%$ can be considered as a maximum theoretical overhead induced by message logging for a communication-bounded application. For the sake of simplicity, we do not consider in this study the cost of logging intra-cluster messages reception order in Meneses et al. protocol [15].

To evaluate the amount of computing resources wasted during recovery after a failure, we consider the global performance of the system. While a subset of the processes are recovering from a failure, the other processes usually have to wait for them to progress. We assume that the resources of the hanging processes could be temporarily allocated for other computations, until all application processes are ready to resume normal execution. So β includes only the amount of resources wasted by rolling back processes after a failure.

To compute β, we consider an execution scenario with the following parameters [5]: a Mean Time Between Failure ($MTBF$) of 1 day; 30 minutes to checkpoint the whole application (C); 30 minutes to restart the whole application (Rs). The optimum checkpoint interval (I) can be computed using Daly's formula [8]: $I = \sqrt{(2 \times C \times (MTBF + Rs))} = 297min$. This formula was originally used in coordinated checkpointing protocols, but we think we can safely apply

it to our case. Assuming that failures are evenly distributed a failure occurs at time $\frac{I}{2}$ on average. The total time lost per MTBF period can be approximated by $\frac{I}{2} + Rs = 179min$, that is $12,4\%$ of the period, giving $\beta = 12.4\%$.

The two other parameters, L and R, are protocol-dependent. In the protocol proposed by Meneses et al. [15], all inter-cluster messages are logged and only the cluster where the failure occurs roll back. Considering a partition $\Pi = \langle P_1, P_2, \ldots, P_K \rangle$ which entails a volume of B inter-cluster messages over a total volume of messages D,

$$\text{cost}(\Pi) = 23\% \times \frac{B}{D} + 12.4\% \times \frac{\sum_k |P_k|^2}{P^2} \tag{2}$$

where $\frac{\sum_k |P_k|^2}{P^2}$ is the average number of processes restarting after a failure if the failures are evenly distributed among the P application processes. As described in Section 2, the protocol proposed by Guermouche et al. [11] only logs half of the inter-cluster messages but requires to roll back $\frac{K+1}{2}$ clusters on average after a failure:

$$\text{cost}(\Pi) = 23\% \times \frac{B}{2 \times D} + 12.4\% \times \frac{K+1}{2} \times \frac{\sum_k |P_k|^2}{P^2} \quad . \tag{3}$$

5.2 Results

We first evaluate the cost of partial message logging. Table 2 presents the results obtained by running our partitioning algorithm with PaToH and cost function (2). The results show that for all applications except FT and MAESTRO, our tool was able to find a clustering configuration where less than 15% of the processes have to roll back on average after a failure, while logging less than 20% of the data exchanged during the execution.

To get an insight on the quality of the costs obtained, they might be compared to the cost of a coordinated checkpointing and a message logging protocol. With a coordinated checkpointing ($L = 0$, $R = 1$), the amount of wasted resources would be 12.4%. With a message logging protocol ($L = 1$, $R = \frac{1}{P}$), this amount would be 23%. The results show that for all applications except FT, using the partial message logging protocol minimizes the cost. For applications based on dense linear algebra (BT, LU, SP, and PARATEC) or N-body methods (GTC, LAMMPS, and Nbody), the cost obtained is always below 5%.

Table 3 presents the results obtained by running our partitioning algorithm with PaToH and cost function (3). We evaluate it only with the applications having a symmetric communication pattern, because in this protocol, messages logging is not bidirectional and our partitioning tool does not take this in account yet. In all the tests except MAESTRO, we managed to find clusters such that the ratio of rolled back processes is around 55% while logging less than 5.3% of the messages. The cost function of the two partial message logging protocols cannot be compared because the first one is only valid for a single failure case while the second one can handle multiple concurrent failures.

Table 2. Partitioning for the protocol of Meneses et al

	Size	Nb Clusters	Min/Max cluster size	Processes to roll back	Log/Total Amount of data (in GB)	Cost
NPB BT	1024	8	123/133	12.5%	201/1635 (12.3%)	4.37
NPB CG	1024	32	32/32	3.1%	910/5606 (16.2%)	4.12
NPB FT	1024	2	502/522	50%	432/864 (50%)	17.7
NPB LU	1024	16	64/64	6.25%	67/700 (9.7%)	3.0
NPB MG	1024	8	128/128	12.5%	20/107 (18.5%)	5.8
NPB SP	1024	8	123/133	12.5%	366/2989 (12.2%)	4.4
GTC	512	16	32/32	6.25%	240/3654 (6.6%)	2.3
MAESTRO	1024	4	252/259	25%	55/309 (17.7%)	7.17
PARATEC	1024	13	64/128	8.5%	2262/23914 (9.4%)	3.23
LAMMPS	1024	8	127/129	12.5%	0.3/4 (7.6%)	3.3
Nbody	1024	30	31/61	3.5%	80/2733 (2.9%)	1.1

Table 3. Partitioning for the protocol of Guermouche et al

	Size	Nb Clusters	Min/Max cluster size	Processes to roll back	Log/Total Amount of data (in GB)	Cost
NPB LU	1024	16	64/64	53.1%	34/700 (4.8%)	7.7
MAESTRO	1024	4	250/262	62.5%	27/309 (8.9%)	9.78
PARATEC	1024	16	61/68	53.1%	1285/23914 (5.3%)	7.8
LAMMPS	1024	8	127/129	56.2%	0.15/4 (3.8%)	7.9

Table 4. Evaluating three tools on PARATEC with the cost function (2)

	2	4	8	13	16	32	64
PaToH	7.07	4.52	3.42	3.47	3.36	3.95	5.04
run time	0.60	0.57	0.60	0.59	0.60	0.61	0.60
MeTiS	7.47	5.36	5.38	5.47	5.92	3.82	4.96
Scotch	7.09	4.56	3.44	3.37	3.22	3.67	4.94

To validate our bisection-based partitioning algorithm, we used PaToH, MeTiS and Scotch as outlined in Section 4.1. Table 4 presents the result of running the three tools on PARATEC with the cost function (2). We ran the experiment with $K = 2, 4, 8, 16, 32, 64$, and also with $K = 13$ which is the result provided by our tool (see table 2). First, it has to be noticed that the number of clusters found by our tool is close to the number of clusters that minimizes the cost function with PaToH and Scotch. Second, only Scotch manages to slightly improve the cost compared to our tool. However, if we use our tool with Scotch instead of PaToH, we obtain exactly the same cost. This is mostly due to the fact that Scotch obtains well balanced partitions for any given K, and that the β in (2) is relatively high. In cases where β is smaller, the partitioner has a higher degree of freedom. Whereas the proposed method automatically exploits this leeway, it is hard to specify the imbalance parameter for the three existing tools we have used (we have not reported these experiments, but this was observed for β around 5). We conclude from these results that the proposed algorithm manages to find good clusters without taking a number of clusters as input. The row "run time" below PaToH contains the running time of PaToH with the given K compared to that of the proposed

algorithm (which finds $K = 13$). As is seen, two runs of PaToH take more time than a single run of the proposed algorithm (despite the overheads associated with repeated calls to the library, including converting the data structure).

6 Conclusion

Partial message logging protocols, combining a checkpointing and a message logging protocol, are an attractive rollback-recovery solution at very large scale because they can provide failure containment by logging only a subset of the application messages during the execution. To work efficiently, such protocols require to form clusters of processes in the application, such that inter-cluster communications are minimized. In this paper, we showed that such clustering can be done in many MPI HPC applications. To do so, we analyzed the communication patterns we gathered from the execution of a representative set of HPC MPI applications. To find clusters, we proposed a bisection-based partitioning algorithm that makes use of a cost function evaluating the efficiency of a partial message logging protocol for a given clustering configuration. Contrary to existing graph partitioning tools, this algorithm does not require the number of clusters as input. We defined a cost function for two *state-of-the-art* partial message logging protocols and ran tests on our set of execution data. With both protocols, results show that we were able to get a good trade-off between the size of the clusters and the amount of logged messages. Furthermore, with Meneses et al. protocol, percentage of processes to rollback and of message to log is in many cases around 10%, which is an order of magnitude improvement compared to flat rollback-recovery protocols. This result encourages us to continue our work on partial message logging protocols.

Acknowledgments. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr). This work was supported by INRIA-Illinois Joint Laboratory for Petascale Computing and the ANR RESCUE project.

References

[1] Antypas, K., Shalf, J., Wasserman, H.: NERSC-6 Workload Analysis and Benchmark Selection Process. Technical Report LBNL-1014E, Lawrence Berkeley National Laboratory, Berkeley (2008)

[2] Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, University of California, Berkeley (2006)

[3] Bailey, D., Harris, T., Saphir, W., van der Wilngaart, R., Woo, A., Yarrow, M.: The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center (1995)

[4] Bui, T.N., Jones, C.: Finding good approximate vertex and edge partitions is NP-hard. Information Processing Letters 42, 153–159 (1992)
[5] Cappello, F.: Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. International Journal of High Performance Computing Applications 23, 212–226 (2009)
[6] Çatalyürek, Ü.V., Aykanat, C.: PaToH: A multilevel hypergraph partitioning tool, version 3.0. Technical Report BU-CE-9915, Bilkent Univ.(1999)
[7] Cunningham, W.H.: Optimal attack and reinforcement of a network. J. ACM 32, 549–561 (1985)
[8] Daly, J.: A model for predicting the optimum checkpoint interval for restart dumps. In: Proceedings of the 2003 International Conference on Computational Science, ICCS 2003, pp. 3–12. Springer, Heidelberg (2003)
[9] Elnozahy, E.N(M.), Alvisi, L., Wang, Y.-M., Johnson, D.B.: A Survey of Rollback-Recovery Protocols in Message-Passing Systems. ACM Computing Surveys 34(3), 375–408 (2002)
[10] Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1979)
[11] Guermouche, A., Ropars, T., Brunet, E., Snir, M., Cappello, F.: Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic Message Passing Applications. In: 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011), Anchorage, USA (2011)
[12] Ho, J.C.Y., Wang, C.-L., Lau, F.C.M.: Scalable Group-Based Checkpoint/Restart for Large-Scale Message-Passing Systems. In: 22nd IEEE International Parallel and Distributed Processing Symposium, Miami, USA (2008)
[13] Kamil, S., Shalf, J., Oliker, L., Skinner, D.: Understanding ultra-scale application communication requirements. In: Proceedings of the 2005 IEEE International Symposium on Workload Characterization, pp. 178–187 (2005)
[14] Karypis, G., Kumar, V.: MeTiS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0. Univ. Minnesota, Minneapolis (1998)
[15] Meneses, E., Mendes, C.L., Kale, L.V.: Team-based Message Logging: Preliminary Results. In: 3rd Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids (CCGRID 2010) (May 2010)
[16] Monnet, S., Morin, C., Badrinath, R.: Hybrid Checkpointing for Parallel Applications in Cluster Federations. In: Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2004), pp. 773–782. IEEE Computer Society, Washington, DC, USA (2004)
[17] Pellegrini, F.: SCOTCH 5.1 User's Guide. LaBRI (2008)
[18] Riesen, R.: Communication Patterns. In: Workshop on Communication Architecture for Clusters CAC 2006, Rhodes Island, Greece, IEEE, Los Alamitos (2006)
[19] Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of Collective Communication Operations in MPICH. International Journal of High Performance Computing Applications 19(1), 49–66 (2005)
[20] Vetter, J.S., Mueller, F.: Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures. Journal of Parallel and Distributed Computing 63, 853–865 (2003)
[21] Yang, J.-M., Li, K.F., Li, W.-W., Zhang, D.-F.: Trading Off Logging Overhead and Coordinating Overhead to Achieve Efficient Rollback Recovery. Concurrency and Computation: Practice and Experience 21, 819–853 (2009)

Object Placement for Cooperative Caches with Bandwidth Constraints

UmaMaheswari C. Devi, Malolan Chetlur, and Shivkumar Kalyanaraman

IBM Research – India, Bangalore

Abstract. The projected growth in video traffic delivered to mobile devices is expected to stress the backhaul and core of a broadband wireless network. Caches deployed at the edge elements, such as base stations, are one of alleviating this stress. Limits on the sizes of the base station caches and restrictions on frequent upgrades to the hardware necessitate that techniques that can increase the hit rates with the growing traffic, given the constraints, be explored. In this paper, we consider using cooperative caching schemes for the purpose. The edge elements are connected via bandwidth-constrained links, and hence, the assumption made in most prior work that the cooperating nodes are located on a high-speed network do not apply here. We show that the problem of placing objects to maximize hit rate in such a bandwidth-constrained caching system is NP-hard in the strong sense. We develop an efficient placement algorithm when the caches have identical characteristics and show that its performance is within a constant factor of the optimal under practical conditions. We also discuss how to extend the algorithm for the non-identical case. Our simulation experiments show that in practice, the performance of our algorithm is very close to the optimal and a few tens of cooperating nodes are sufficient to significantly increase the hit rate even with a 1% base cache size.

1 Introduction

Data and Video-on-Demand (VoD) traffic delivered over mobile networks are projected to grow tremendously in the next few years [5]. Current wireless infrastructures are not provisioned to handle this growth. The projected growth is hence expected to significantly increase the stress on not just the wireless channel, but also the wired backhaul and core of a cellular network. Wireless network operators are therefore seeking optimizations that can ease this pressure and help defer infrastructure upgrades.

One simple and effective method to reduce the backhaul traffic is to cache frequently requested content at the edge elements, such as base stations (BS) and central controllers (CC). The limit on the size of a cache that can be placed at a BS is lower than that of traditional Internet caches by an order of magnitude or more. While the smaller caches might be capable of providing good hit rates to traditional web traffic, the same may not hold for VoD and other types of multimedia traffic. This is due to the facts that video objects are much larger in size and the number of video clips is increasing by the day. The latter growth is spurred by the growth in user-generated content and IP and mobile TV, which produce numerous shows per day. In such a scenario, adequate hit rates may be obtained for the growing traffic by increasing the effective cache size by enabling cooperation and sharing of objects among the caching nodes.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 579–593, 2011.

Cooperative caching has been studied previously for traditional wired networks. However, as discussed below, most of the prior work assumes that the caching nodes are placed on a high-speed network, and hence, that bandwidth available for inter-cache communication is not a constraint.[1] Also, most of the work focuses on minimizing the average object access latency. In contrast, the bandwidth available for inter-BS or inter-CC communication in the wireless edge is limited, and our focus is on reducing the network traffic in the backhaul and core by reducing the byte miss ratio in the edge. Hence, we explore placing objects in a set of cooperating caches such that the hit rate (not access latency) of the caches is optimized subject to not violating the inter-cache communication bandwidth (ICCB) constraints.

Cellular BSs and CCs are organized in a two-level hierarchy with a few 100s of BSs connected to a CC. We assume that any pair of BSs may communicate either directly or via their parent CC. In either case, we assume that the bandwidth available for transferring objects among BSs is limited. In such a setup, we first consider the problem of placing video objects at the BSs (assuming that there is no cache at the CC) and specifying how those objects should be shared for the hit rate is maximized. We show that this problem is NP-hard in the strong sense even when all the caches have identical sizes and see identical object access patterns, an assumption made in several studies. We then develop an efficient object placement algorithm for the case of identical caches and show that it has a constant-factor approximation ratio under conditions expected to hold in practice. Thirdly, we discuss how to extend the algorithm when second-level caches are available at the CCs and relax the assumption that caches are identical. Finally, we evaluate our scheme through simulations.

Our placement algorithm is centralized and the idea is to periodically determine the placement map at a common node such as the parent CC using object popularity information gathered from the BSs. The maps would then be distributed to the BSs, which would subsequently cache the newly specified objects the first time each is requested. To reduce object churn, knowledge of objects already cached at the various nodes may be used while laying out a new map.

Related Work: Cooperative caching was (among other works) first explored as part of the Harvest project [4]. A notable follow up was the Summary Cache [6], which introduced efficient directory services. The above efforts were on the development of techniques and protocols for directing URL requests that miss to sibling or parent nodes and did not deal with bandwidth constraints.

The above work was followed by a good amount of research on object placement algorithms for a cooperating cluster of web caches. Significant works in this area include [11], which provides a 13.93-approximation algorithm for placing objects to minimize the average access cost while imposing no constraints on the available bandwidth, and [13], which extends the algorithm in [11] by including bandwidth constraints. [13] however assumes that the cache size at each node is very large and an object will be stored locally after the initial miss. Hence, unlike our paper, the bandwidth constraints apply only for the initial object placement and for periodic object updates, and not for serving requests to objects from peer nodes on a continuous basis. Also, since video objects are

[1] Works that impose a bandwidth constraint differ in the objective explored or in the exact nature of the bandwidth constraint imposed.

immutable, the need for object updates is obviated, as is the initial object placement as discussed earlier.

Replicating objects in a content-distribution network (CDN) to minimize the average number of ASs traversed [10], heuristics for minimizing the end-user retrieval cost subject to meeting end-user QoS under the assumption of sufficient combined storage for all the objects [14], and placing objects in a CDN layered on a P2P overlay [12] have also been studied. P2P techniques are redundant for BS caches as BSs are reliable entities.

The work that comes closest to ours is [2], which considers placing video objects in a 2-level hierarchical network with costs on bandwidth usage on the links connecting the nodes such that the total bandwidth cost is minimized and proposes distributed algorithms with constant-factor approximation ratios. Bandwidth constraints cannot be converted to bandwidth costs, and hence, the solution of [2] does not extend to the problem in this paper, although there is similarity in the structure of our solution and theirs.

Placing *all* of a large set of video objects in the video hub offices of a large-scale distributed VoD system while minimizing the cost of total byte transfer subject to link bandwidth limits is considered in [1]. The problem differs from ours since we also need to determine *which* of the objects need to be placed. Another problem with a similar flavor, that of distributing chunks of videos among end clients with local connectivity to reduce the stress on the access networks is considered in [9].

The rest of the paper is organized as follows. Our system model is described in Sec. 2. Sec. 3 develops a placement algorithm, derives an approximation ratio for it, and discusses extensions to the algorithm. Simulation studies are described in Sec. 4. Sec. 5 concludes.

2 System Model and Problem Formulation

We consider a cooperative caching proxy system of N nodes, $1, \ldots, N$, where node j is provided with a cache of size C_j. Each request to each of a set of M video objects, where the size of object i is S_i bytes, pass through one of the N nodes. If a requested object is cached at the node that receives the request, it is served from the node's local cache. The nodes are assumed to be provided with dedicated bandwidths, referred to as *inter-cache communication bandwidth*, (ICCB), both in the upload and download directions, that may be used for letting an object cached at a node to be borrowed by a peer node. Hence, a request that misses at a node's local cache can be served from either a peer node or the origin server. The upload and download bandwidth limits at node j are denoted B_j^u and B_j^d, respectively. The average demand in requests per second for object i at node j is denoted R_{ij}; its bandwidth is hence $R_{ij} \cdot S_i$ bytes/sec. The total bandwidth of objects borrowed by/from a node cannot exceed its ICCB limits. We are concerned with placing a subset of M objects at the N nodes and designating how objects are shared such that the total bytes served per second from the caches (byte hit rate) is maximized. Note that since nodes have dedicated ICCB, an object served by borrowing from a peer cache is considered a hit. We refer to the set of all caches at the N nodes as the *combined* or *collective* cache.

Let x_{ij} be a 0-1 integer variable denoting whether object i is placed at cache j. Similarly, let x_{ijk} denote whether object i placed in cache j is borrowed by cache k. The problem of placing objects at the caches and determining how objects are shared to maximize the byte hit rate, denoted O_Place_Gen, can then be formulated as shown in the inset to the right.

O_Place_Gen

$$\text{Maximize} \sum_{i=1}^{M}\sum_{j=1}^{N}(x_{ij}\cdot R_{ij}\cdot S_i + \sum_{k=1}^{N} x_{ikj}\cdot R_{ij}\cdot S_i)$$

$$\text{subject to} \sum_{i=1}^{M} S_i \cdot x_{ij} \le C_j, \quad j = 1,\ldots,N \tag{1}$$

$$x_{ik} + \sum_{j=1}^{N} x_{ijk} \le 1,\ i = 1,\ldots,M,\ k = 1,\cdots,N \tag{2}$$

$$x_{ijk} \le x_{ij}, \quad i = 1,\ldots,M,\ j,k = 1,\ldots,N \tag{3}$$

$$\sum_{i=1}^{M}\sum_{j=1}^{N} x_{ijk}\cdot R_{ik}\cdot S_i \le B_k^u, \quad k = 1,\ldots,M \tag{4}$$

$$\sum_{i=1}^{M}\sum_{k=1}^{N} x_{ijk}\cdot R_{ik}\cdot S_i \le B_j^d, \quad j = 1,\ldots,M \tag{5}$$

$$x_{ij} \in \{0,1\}, x_{ijk} \in \{0,1\}, \quad i = 1,\ldots,M, \quad j,k = 1,\ldots,N \tag{6}$$

Object i served from node j, either using a copy locally cached at it or borrowed from another node k, would lead to $R_{ij}\cdot S_i$ fewer bytes per second requested from the hosting servers and transported over the core and backhaul networks. The objective function is therefore as indicated. The constraints in (1) account for the limits on the cache sizes. Constraint (2) prevents a node from both caching a node locally as well as borrowing from one or more caches, while (3) ensures that node k borrows an object i from node j only if i is cached at j. (4) and (5) ensure that the limits on uplink and downlink bandwidths available for inter-cache transport are not violated at any node.

In O_Place_Gen, the objective function and all the constraints are linear in the decision variables, so it is an integer linear program. Solving it with generic integer program methods can therefore require exponential time. It turns out that even a simpler special case of the problem with uniform object and cache sizes, denoted S and C, respectively, identical uplink and downlink bandwidth limits, denoted B, and identical popularity distributions at all nodes, denoted R_i for object i (the bandwidth for object i would be $R_i \cdot S$ bytes per sec at all nodes), is actually NP-hard in the strong sense, so an exact solution to it or the general problem cannot be obtained in polynomial time using alternative methods either, unless P=NP. The special case, denoted O_Place_Spl, is obtained from O_Place_Gen by replacing R_{ij}'s and R_{ik}'s with R_i, S_i's with S, C_j with C, and B_k^u and B_j^d with B. A complete problem statement is omitted due to space constraints.

3 Hardness Result and Approximation Algorithm

The special case of the object placement problem O_Place_Spl is NP-hard in the strong sense as we show in the longer version of this paper. The reduction is from the 3-PARTITION (3-PART) problem, which is NP-complete in the strong sense. Hence, a pseudo-polynomial-time algorithm or an FPTAS are also not possible for O_Place_Spl, apart from a polynomial-time algorithm.

3.1 Hardness Proof

3-PART is a number problem [7, pp. 224 and 94] defined as follows.

Definition 1 (3-PART)**:** Given set E of $3m$ elements, $e_1, e_2, \ldots, e_{3m}$, a bound $K \in \mathbb{Z}_+$, and a size $s(e_i) = s_i \in \mathbb{Z}_+$ for each $e_i \in E$ such that $K/4 < s_i < K/2$ and $\sum_{i=1}^{3m} s_i = mK$. The problem is to determine whether E can be partitioned into m disjoint sets $E_1, E_2, \ldots, E_m$ such that $\sum_{e \in E_i} s(e) = K$, for $1 \leq i \leq m$.

Theorem 1. O_Place_Spl *is NP-hard in the strong sense.*

Proof: To prove the theorem, we show that the decision version of O_Place_Spl is NP-complete in the strong sense. It is easy to see that the decision version is in NP. We provide a pseudo-polynomial reduction [7] from 3-PART to it.

Consider an arbitrary instance of 3-PART and construct an instance of O_Place_Spl, denoted objpl-3-part, from it, as follows. Let $N = m$, $M = 3m$, $C = 3 \cdot S$, $S = 1$, and $B = (m-1) \cdot K$. Let $R_i = s_i/S$ for $1 \leq i \leq N$. Let $B_i = R_i \cdot S = s_i$ denote the bandwidth of object i. We now show that a solution to 3-PART exists if and only if there is a solution to objpl-3-part with objective value exactly equal to $N \cdot m \cdot K = m^2K$.

$\Leftarrow$ Assume that there is a solution to objpl-3-part with objective value exactly m^2K.

Because $C = 3 \cdot S$, each node can store at most three objects locally in its cache. We first show that this number is exactly three. If some node stores fewer than three objects, then the total number of objects stored in the combined cache is less than $3m$. Thus, there exists at least one object that is not served by the combined cache, and hence, the objective value of the solution to objpl-3-part cannot equal or exceed m^2K, which contradicts our assumption. Thus, each node stores exactly three objects in its cache. Next, we show that for each node, the total bandwidth of the three objects stored in its cache is exactly K. For this, first note that for the objective value to equal m^2K, each of the m nodes should serve all the $3m$ objects from the combined cache. The total bandwidth of all the objects is mK. The total bandwidth of the objects that a node borrows from other caches cannot exceed $(m-1) \cdot K$ (since the downlink bandwidth at each node (B) is limited to $(m-1) \cdot K$). Hence, each node should serve the remaining $mK - (m-1)K = K$ bytes from its local cache. Thus, the total bandwidth of the three objects that each node caches is exactly K. Therefore, since $B_i = s_i$, a solution to 3-PART can be obtained from a solution to objpl-3-part by assigning element e_i to set E_j if object i is assigned to node j (that is $x_{ij} = 1$).

$\Rightarrow$ A solution to objpl-3-part with objective value exactly m^2K that satisfies cache capacity constraints and bandwidth constraints can be obtained by simply setting $x_{ij} = 1$ if e_i is assigned to set E_j, and $x_{ijk} = 1$ for all $k \neq j$, if $x_{ij} = 1$.

The reduction can be performed in polynomial time. All numbers in objpl-3-part are polynomially bounded by the numbers in 3-PART. Thus, the decision version of O_Place_Spl is NP-complete in the strong sense. O_Place_Spl is hence NP-hard in the strong sense. ■

3.2 Efficient Placement Algorithm

In this section, we focus on designing an efficient centralized algorithm for solving O_Place_Spl. We assume that at least a few of the top K most popular objects have

bandwidth at most $B/(N-1)$. ($K = C/S$, the number of objects that fit in a cache.) Otherwise, the scope for cooperation would be very limited and one may consider enabling cooperation among fewer caches (*i.e.*, with smaller N).

Identifying Heuristics. Before presenting an algorithm for O_Place_Spl, we present some rules of thumb that have been used in its design. In what follows, we will refer to an object that is cached at all N nodes as *fully replicated*. An object that is cached at two or more nodes, but not all nodes, is said to be *partially replicated*, and one that is cached at a single node as *unreplicated*. An unreplicated object that is shared by all nodes is said to be *totally shared*, while an unreplicated or a partially-replicated object that is shared by some but not all nodes or borrowed partly by all nodes is referred to as *partially shared*. An object cached at a node is said to be partly borrowed by a peer node if part of the requests to the object at the peer node that are evenly distributed is served from the caching node.

Rule 1. *Cache objects with the largest bandwidths* (*in fully-replicated, partially-replicated, or unreplicated manner*).

This rule is quite obvious and is used by most caching systems.

Rule 2. *Since ICCB is constrained, replicate, either fully or paritially, objects of larger bandwidths.*

If ICCB is abundant, then bytes served from the collective cache can be maximized by having unique copies of as many objects as possible in the constituent caches and serving those objects from the combined cache at every node. In such a case, most objects (if not all) are unreplicated and the rule does not apply.

If ICCB is limited, then it can be shown that fully replicating one or more objects will serve to increase the combined hit rate. To see that replicating higher bandwidth objects is beneficial, suppose a higher bandwidth object, H, is unreplicated or partially replicated, while a lower bandwidth object, L, is fully replicated. Then, a node $\mathcal{N}$ that does not cache H has two options: it either fetches H from the hosting server or borrows it from a peer. It is easy to see that simply replacing L by H would in the former case increase the number of bytes served locally from $\mathcal{N}$ (while not decreasing the number of bytes borrowed from the other caches and served). In the latter case, the replacement would lead to H being served locally. The downlink bandwidth that consequently gets freed up at $\mathcal{N}$ can be used to borrow at least L, and potentially, a few more objects. So, the total bytes that $\mathcal{N}$ serves from the combined cache is not lowered. Thus, replicating larger bandwidth objects serves, in general, to increase the bytes served.

Rule 3. *Among the objects chosen for caching, unreplicate and totally share those with lower bandwidths, subject to not violating the ICCB constraints.*

The rationale for this rule is similar to that for the prior one.

Object-Placement Algorithm. Let $\mathcal{O}$ denote the set of all M objects arranged in non-increasing order of their bandwidths. We start with the set of objects with the highest demand (that is, the largest bandwidth objects), referred to as $\mathcal{O}_C$, that will fit in a cache of size C. (These will be the objects that each node caches in a non-cooperative setting.) These would form the initial set of replicated objects, while the initial shared object set is $\emptyset$.

Let $\bar{\mathcal{O}}_C$ denote the set of objects in $\mathcal{O}$ excluding those in $\mathcal{O}_C$. Since ICCB B is constrained, not all objects can be unreplicated and shared, and by Rule 2, high bandwidth objects should be replicated. Our goal is to identify the boundary at which unreplication and sharing should commence.

Given that ICCB is B, the amount of data that each node serves from the combined cache can be at most B bytes per second higher than the total bandwidth of the objects in $\mathcal{O}_C$. Let $\mathcal{O}_{inc} \subseteq \bar{\mathcal{O}}_C$ denote the set of objects brought into the combined cache when cooperation is enabled. By Rule 3, as many of these objects should be totally shared. If all the objects in $\mathcal{O}_{inc}$ could be totally shared, then the total bandwidth of all the objects in $\mathcal{O}_{inc}$ could be at most B. Furthermore, for every $N-1$ objects brought into

1: b : `array` $1..M$ `of real sorted descending` {object bandwidths}
2: shr_from : `integer` {starting index of unreplicated and shared objects in $\mathcal{O}_C$}
3: L : `integer` {no. of objects in $\mathcal{O}_C$ that are unreplicated and shared}
4: B_{top} : `real` {total bandwidth of unreplicated and shared objects in $\mathcal{O}_C$}
5: B_{inc} : `real` {total bandwidth of unreplicated and shared objects in $\bar{\mathcal{O}}_C$}
6: {Determine the objects with total bandwidth at most $B/(N-1)$ at the tail of the objects in $\mathcal{O}_C$}
7: $i := K$; {K is the no. of objects that can be held in a cache}
8: $B_{top} := 0$;
9: **while** $B_{top} + b[i] \leq B/(N-1)$ **do**
10: $B_{top} := B_{top} + b[i]$;
11: $i := i - 1$;
12: **end while**
13: $shr\_from := i + 1$;
14: $L := K - shr\_from + 1$;
15: /* Select $(N-1) \cdot L$ objects from $\bar{\mathcal{O}}_C$*/
16: $B_{inc} := \sum_{i=K+1}^{K+(N-1)\cdot L} b[i]$
17: $\mathcal{O}_{shared}$:= objects $K - L + 1 \ldots K + (N-1) \cdot L$;
18: **while** $NB/(N-1) > B_{inc} + B_{top}$ **do**
19: /* Check if unreplicating and sharing the next lowest bandwidth object from $\mathcal{O}_C$ can increase the total bandwidth of objects from $\bar{\mathcal{O}}_C$ brought into the combined cache and shared */
20:
21: **if** $b[shr\_from - 1] + \sum_{i=K+(N-1)(L+1)}^{K+(N-1)(L+1)} b[i] + B_{inc} + B_{top} \leq NB/(N-1)$ **then**
22: $shr\_from := shr\_from - 1$; $L := L + 1$;
23: $B_{top} := B_{top} + b[shr\_from]$;
24: $B_{inc} := B_{inc} + \sum_{i=K+(N-1)(L+1)}^{K+(N-1)(L+1)} b[i]$;
25: Update $\mathcal{O}_{shared}$ to include the newly selected objects;
26: **end if**
27: **end while**
28: /* Distribute the objects in $\mathcal{O}_{shared}$ using a balanced fit heuristic such that the total bandwidth of all objects assigned to any node is at most $B/(N-1)$ */
29: **for** each object with index O in $\mathcal{O}_{shared}$ **do**
30: /* consider objects in non-increasing order of their bandwidths */
31: assign O to the node with the largest unused ICCB among those with spare physical slots if such a node exists and unused ICCB at the node is at least $(N-1) \cdot b[O]$;
32: mark O as totally shared from the node caching it;
33: **end for**
34: assign the remaining objects in $\mathcal{O}_{shared}$ to nodes with available physical slots; mark them unshared

Fig. 1. Object Placement Algorithm PA

the combined cache, due to cache capacity constraints, at least one object from $\mathcal{O}_C$ should be unreplicated and shared (to make room for the incoming objects). Thus, if $\ell = |\mathcal{O}_{inc}|$, $L = \lceil \ell/(N-1) \rceil$ of the $\mathcal{O}_C$ objects should be unreplicated and shared.

By the ICCB constraint B, at each node, the total bandwidth of all the objects that are unreplicated and totally shared cannot exceed $B/(N-1)$. Thus, the total bandwidth of all the unreplicated and totally shared objects in the combined cache cannot exceed $NB/(N-1)$. Our objective of maximizing the total bytes served from the combined cache thus reduces to the following:

Phase 1. Choosing L objects from $\mathcal{O}_C$ and ℓ objects from $\bar{\mathcal{O}}_C$ for sharing such that the total bandwidth of the objects from $\bar{\mathcal{O}}_C$ is maximized and the constraints below hold.

(C1) $\ell \leq (N-1) \cdot L$

(C2) The total bandwidth of the $L + \ell$ objects chosen is at most $NB/(N-1)$

Phase 2. Partitioning the $L + \ell$ chosen objects among the N nodes such that the total bandwidth of the objects assigned to each node is at most $B/(N-1)$.

Listing for an algorithm, denoted PA, that accomplishes the above is provided in Fig. 1. Choosing objects that should be unreplicated and totally shared is performed in the first phase in lines 7–27. In this phase, L is initially set to the number of the lowest bandwidth objects in $\mathcal{O}_C$ with total bandwidth at most $B/(N-1)$ (lines 7–17). If the combined bandwidth of the first $(N-1) \cdot L$ objects from $\bar{\mathcal{O}}_C$, B_{inc}, and the L objects from $\mathcal{O}_C$, B_{top}, is at least $NB/(N-1)$, then the algorithm moves to the second phase. Since objects are arranged in non-increasing order of bandwidths, $B_{inc} \leq (N-1) \cdot B_{top}$ holds at every step.

On the other hand, if the combined bandwidth is less than $NB/(N-1)$, then the **while** loop in line 18 is entered. L is incremented by one and ℓ by $N-1$ as long as the combined bandwidth of the chosen objects remains less than $NB/(N-1)$. The first phase ends when no more objects can be brought in from $\bar{\mathcal{O}}_C$. At its end, NL objects are marked for sharing.

In the second phase, the objects chosen for sharing are partitioned among the nodes. In the first step of this phase in lines 29–32, objects are distributed such that the total bandwidth of all the objects assigned and totally shared from a node is at most $B/(N-1)$. Since distributing objects without violating the ICCB constraint is a bin packing problem, for which feasible solutions are known to *not exist* for all instances, not all objects can be expected to be successfully assigned. The remaining objects are filled in the available slots of all the caches in the next step in line 34. Exactly $K - L$ objects are fully replicated while no object is partially replicated. Hence, each cache can hold exactly L more objects for a total of NL objects in all the caches. Thus, since the total number of objects chosen for distribution is NL, all objects will be successfully assigned to some cache but not all may necessarily be shared.

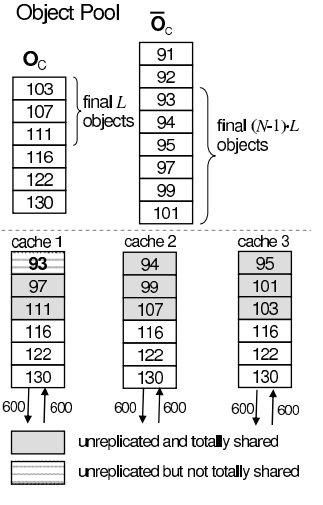


Fig. 2. Placement example

Example. To better understand the algorithm, consider the example in Fig. 2. Here $N = 3$ and $C = 6S$ so that $K = 6$. $M = 14$ objects, and their bandwidths are indicated in the boxes. The objects in $\mathcal{O}_C$ and $\bar{\mathcal{O}}_C$ are as indicated. ICCB B is 600. Since $B = 600$, in the first phase, $L = 3$ objects and $\ell = 2L = 6$ objects as marked in the figure could be selected for sharing from $\mathcal{O}_C$ and $\bar{\mathcal{O}}_C$, respectively. B_{top} is 321, B_{inc} is 579, and $B_{top} + B_{inc} = 900 = NB/(N-1)$.

In the second phase, eight of the objects selected in the first phase could be distributed among the three caches using the heuristic in lines 29–32 such that ICCB is respected. The final object (with lowest bandwidth) is assigned to the first cache but is marked unshared as otherwise ICCB would be violated. It can be verified that the objects cannot be partitioned among the caches such that the constraints are satisfied.

Byte hit rate after cooperation increases by 579 Bps for Cache 1 and 486 Bps for each of the other two nodes. Hit rates for the latter two can be increased by $93/2 = 46.5$ Bps by serving half their requests to the ninth object from Cache 1.

Algorithm complexity: If the objects are sorted by their bandwidths, the complexity of the algorithm can easily be seen to be $O(NK)$. Otherwise, it is $O(NK + M \log M)$, which is $O(M(K + \log M))$.

3.3 Approximation Ratio

We now derive an approximation ratio for Algorithm PA, assuming Zipf-like distribution [3] or its generalization, the MZipf distribution [8], for object popularity distributions. The approximation ratio would hold as long as the ratio of the probability of accesses of objects with ranks i and $i+1$ is at most $\frac{i+1}{i}$.

PA chooses L and $(N-1)L$ contiguous objects from the tail and head of $\mathcal{O}_C$ and $\bar{\mathcal{O}}_C$, respectively, such that their total bandwidth is maximized subject to not exceeding $NB/(N-1)$. Let the two subsets be denoted $\mathcal{O}_C^{PA}$ and $\bar{\mathcal{O}}_C^{PA}$, respectively, and let $\mathcal{BW}(.)$ denote the bandwidth function. The increase in hit rate achieved by PA per node is therefore at most $\mathcal{BW}(\bar{\mathcal{O}}_C^{PA})$. (It would be less than $\mathcal{BW}(\bar{\mathcal{O}}_C^{PA})$ if the objects $\mathcal{O}_C^{PA}$ and $\bar{\mathcal{O}}_C^{PA}$ cannot be partitioned among the nodes.) The hit rate per node obtained by an optimal algorithm may be higher by less than the bandwidth corresponding to the next $N-1$ objects from $\bar{\mathcal{O}}_C$. To see this note that since PA could not choose the next $N-1$ objects, the total bandwidth of those objects and an additional lightest object from $\mathcal{O}_C$ along with objects in $\mathcal{O}_C^{PA}$ and $\bar{\mathcal{O}}_C^{PA}$ exceeds $NB/(N-1)$. Hence, choosing any other object from $\mathcal{O}_C$ would not enable choosing $N-1$ objects from $\bar{\mathcal{O}}_C$ with larger bandwidth than the next $N-1$.

Let the maximum bandwidth $\mathcal{BW}_{\max}$ of any object in $\mathcal{O}_C^{PA} \cup \bar{\mathcal{O}}_C^{PA}$ be at most $f \cdot \frac{B}{N-1}$, where $0 < f < 1$, and let $R = \lfloor \frac{1}{f} \rfloor$. In general, for $\frac{1}{n+1} < f \leq \frac{1}{n}$, $R = n$ holds. Also, $R = \lfloor \frac{1}{f} \rfloor \Rightarrow R \leq \frac{1}{f}$, and hence,

$$f \leq \frac{1}{R}. \tag{7}$$

To determine an approximation ratio, we need to determine a lower bound on the increase to hit rate achieved by PA. As discussed above, it would be less than $\mathcal{BW}(\bar{\mathcal{O}}_C^{PA})$ if objects in $\mathcal{O}_C^{PA} \cup \bar{\mathcal{O}}_C^{PA}$ are not all fully shared. If β denotes the bandwidth of objects in $\mathcal{O}_C^{PA} \cup \bar{\mathcal{O}}_C^{PA}$ that are not fully shared, then the increase in hit rate achieved by PA is

given by $\mathcal{BW}(\bar{\mathcal{O}}_{\mathcal{C}}^{\mathsf{PA}}) - \beta$. The following lemma provides a lower bound on the sharing achieved by PA.

Lemma 1. *The total bandwidth of the objects that are unreplicated and fully shared at the end of Phase 2 of* PA *is at least* $\frac{R}{R+1}(\mathcal{BW}(\mathcal{O}_{\mathcal{C}}^{\mathsf{PA}}) + \mathcal{BW}(\bar{\mathcal{O}}_{\mathcal{C}}^{\mathsf{PA}}))$.

Proof: The total number of objects chosen for sharing is NL. These objects may be assigned to one of the N nodes and the maximum number of objects assigned to a node cannot exceed L. An object assigned to a node may be fully shared if the total bandwidth of all the objects assigned previously to the same node and the new object is at most $B/(N-1)$.

During the partition phase of PA, let Ω be the first object that could not be assigned in a fully-shared manner, and let w denote its bandwidth. Then for each node $\mathcal{C}$, one of following conditions hold: **(1)** The total assigned bandwth in $\mathcal{C}$ is at least $(B/(N-1) - w)$. **(2)** $\mathcal{C}$ is full with L assigned objects (has no empty slots), but the total bandwidth of the objects assigned to it is less than $B/(N-1)$.

Let n of the N nodes be of type 1, with condition 1 holding, and the remaining $N-n$, of type 2. Since objects are assigned in monotonically decreasing order of their bandwidths, the bandwidth of every object assigned to a node of type 2 is at least w. Then, the total bandwidth $\mathcal{B}$, of all the objects before Ω assigned to the N nodes is at least $n(\frac{B}{N-1} - w) + (N-n)wL$. By (7) and the definition of f, the bandwidth of any object is at most $B/(R(N-1))$. Hence, if $L \leq R$, the total bandwidth of any subset of L objects is at most $B/(N-1)$. Thus, the NL objects in $\mathcal{O}_{\mathcal{C}}^{\mathsf{PA}} \cup \bar{\mathcal{O}}_{\mathcal{C}}^{\mathsf{PA}}$ can be partitioned among the N nodes such that each is fully shared. Hence, for the rest of the proof take $L > R$. Since the total bandwidth expression is an increasing function of L, $\mathcal{B}$ is at least $\frac{nB}{N-1} + ((R+1)N - (R+2)n)w$. We consider the following cases.

Case 1: $n \leq \frac{1}{R+1}N$**.** In this case at least $R/(R+1)$ of the nodes are of type 2, which are full. Thus, at least a fraction $R/(R+1)$ of the objects are fully shared. Since the objects are assigned in the order of decreasing bandwidths, the total shared bandwidth is at least a fraction $R/(R+1)$ of the bandwidth of the objects in $\mathcal{O}_{\mathcal{C}}^{\mathsf{PA}}$ and $\bar{\mathcal{O}}_{\mathcal{C}}^{\mathsf{PA}}$.

Case 2: $\frac{1}{R+1}N < n \leq \frac{R}{R+1}N$**.** The proof for this case is a little involved and hence omitted due to space constraints. It will be made available in a longer version of the paper.

Case 3: $\frac{R}{R+1}N < n \leq \frac{R+1}{R+2}N$**.** Since $(R+1)N - (R+2)n \geq 0$ holds and $\mathcal{B} \geq \frac{nB}{N-1} + ((R+1)N - (R+2)n)w$, $\mathcal{B} \geq \frac{R}{R+1}NB(N-1)$.

Case 4: $n > \frac{R+1}{R+2}N$**.** In this case, $(R+1)N - (R+2)n < 0$, and hence, $\frac{nB}{N-1} + ((R+1)N - (R+2)n)w$ is a decreasing function of w. If $w > \frac{1}{R+1}\frac{B}{N-1}$, then since objects are assigned in decreasing bandwidth order and each of the N nodes has at least R objects assigned (because as discussed above $R < L$ and the bandwidth of any object is at most $\frac{1}{R}\frac{B}{N-1}$), $\mathcal{B} \geq R \cdot N \cdot \frac{1}{R+1}\frac{B}{N-1} \geq \frac{R}{R+1}(\mathcal{BW}(\mathcal{O}_{\mathcal{C}}^{\mathsf{PA}}) + \mathcal{BW}(\bar{\mathcal{O}}_{\mathcal{C}}^{\mathsf{PA}}))$. If not $\mathcal{B}$ is minimized for $w = \frac{1}{R+1}\frac{B}{N-1}$ and hence $\mathcal{B} \geq \frac{nB}{N-1} + ((R+1)N - (R+2)n)\frac{1}{R+1}\frac{B}{N-1}$, which simplifies to $\frac{NB}{N-1} - \frac{n}{R+1}\frac{B}{N-1}$. This expression is minimized at $n = N$, yielding $\mathcal{B} \geq \frac{R}{R+1}\frac{NB}{N-1}$. ■

Thus, the bandwidth of objects not fully shared is at most $\frac{1}{R+1}(\mathcal{BW}(\mathcal{O}_C^{\mathsf{PA}}) + \mathcal{BW}(\bar{\mathcal{O}}_C^{\mathsf{PA}}))$ and the increase in hit rate is at least $\frac{R}{R+1}\mathcal{BW}(\bar{\mathcal{O}}_C^{\mathsf{PA}}) - \frac{1}{R+1}\mathcal{BW}(\mathcal{O}_C^{\mathsf{PA}})$.

The lemma below provides a lower bound on $\mathcal{BW}(\bar{\mathcal{O}}_C^{\mathsf{PA}})$. Recall that $K = C/S$ denotes the number of objects that fit in a cache.

Lemma 2. *If the object popularity distribution follows Zipf-like or MZipf, then* $\mathcal{BW}(\bar{\mathcal{O}}_C^{\mathsf{PA}}) \geq \frac{\ln \frac{K+(N-1)L}{K+1}}{\ln \frac{K+1}{\max(K-L,1)}} \times \mathcal{BW}(\mathcal{O}_C^{\mathsf{PA}})$.

Proof: Follows from the facts that the probability of accessing an object at rank i is proportional to $1/i$ for the Zipf object popularity distribution and lower than $1/i$ for Zipf-like and MZipf distributions, and $\sum_{i=1}^{n_1} \frac{1}{i} - \ln(n_1) \geq \sum_{i=1}^{n_2} \frac{1}{i} - \ln(n_2+1)$, for all $n_1, n_2 \geq 1$. ■

We are now ready to provide an approximation ratio. Let $\alpha = L/K$.

Theorem 2. *The approximation ratio of* PA *when object popularities conform to the Zipf, Zipf-like or MZipf distributions is given by* $\frac{L+1}{L}\frac{(R+1)\ln(\frac{K+(N-1)L}{K+1})}{R\ln(\frac{(K+(N-1)L)}{K+1}-\ln(\frac{K+1}{\max K-L,1})}$, *which is,* $\frac{L+1}{L}\frac{(R+1)\ln(\frac{1+(N-1)\alpha}{1+\frac{1}{K}})}{R\ln(\frac{1+(N-1)\alpha}{1+\frac{1}{K}})+\ln(\frac{\max(1-\alpha,\frac{1}{K})}{1+\frac{1}{K}})}$, *which is* $\frac{L+1}{L}\frac{(R+1)\ln(1+(N-1)\alpha)}{R\ln(1+(N-1)\alpha)+\ln(\max(1-\alpha,\epsilon))}+\delta$.

Proof: As discussed earlier, by Lemma 1, the effective increase to hit rate achieved by Algorithm PA is at least $(R \cdot \mathcal{BW}(\bar{\mathcal{O}}_C^{\mathsf{PA}}) - \mathcal{BW}(\mathcal{O}_C^{\mathsf{PA}}))/(R+1)$. Letting $\mathcal{BW}(\bar{\mathcal{O}}_C^{\mathsf{PA}}) = \kappa \cdot \mathcal{BW}(\mathcal{O}_C^{\mathsf{PA}})$, the increase to hit rate achieved by PA is at least $(R\kappa-1)\cdot\mathcal{BW}(\mathcal{O}_C^{\mathsf{PA}})/(R+1)$.

To determine the approximation ratio, we need to determine a bound on the increase in hit rate achieved by an optimal algorithm. As discussed at the beginning of this subsection, this value is at most the bandwidth of an additional $N-1$ objects from $\bar{\mathcal{O}}_C$. Since objects are arranged in decreasing order of bandwidths, the total bandwidth of these additional objects would be at most the bandwidth of any $N-1$ objects in $\bar{\mathcal{O}}_C^{\mathsf{PA}}$. Since $|\bar{\mathcal{O}}_C^{\mathsf{PA}}| = (N-1)L$, the increase in hit rate achieved by an optimal algorithm would be at most $\frac{(N-1)(L+1)}{(N-1)L}$ times $\mathcal{BW}(\bar{\mathcal{O}}_C^{\mathsf{PA}}) = \frac{(L+1)}{L} \times \kappa \cdot \mathcal{BW}(\mathcal{O}_C^{\mathsf{PA}})$. The approximation ratio, given by the ratio of the optimal increase in hit rate to the effective increase achieved by PA is hence $\frac{(R+1)\kappa}{R\kappa-1} \cdot \frac{L+1}{L}$. Using the lower bound provided by Lemma 2 for κ and simplifying, we obtain the approximation ratios in the lemma. ■

Discussion: The approximation ratio ρ in Thm. 2 decreases with increasing N and R. For a given N and R, ρ initially decreases with α and then increases. ρ is valid for all values except when N, R, and L are all at most 3, in which case, cooperation is of very little or no use anyway. When K is at least 1000, for $\alpha \leq 0.1$, that is, when at most 10% of the objects in cache are chosen for sharing, $\rho < 2.96$ for $N \geq 5$, for all R, and $\rho < 2.7$ for $R \geq 3$, for all N. In general, ρ is small if α is not large (≤ 0.7) or one of R and N is not too small, with values at least 3 and 5, respectively, which are very reasonable. Recall that the achievable increase in hit rate depends on the ICCB B and for an appreciable increase, say $x\%$, B should be at least $x\%$ of the total bandwidth of all the objects. For the Zipf distribution, the bandwidth of the object with rank n is $1/(n \cdot ln(K))$ of the total object bandwidth. Hence, for $x \geq 10$, the bandwidths of objects with rank 10 and higher is less than $1/90^{\text{th}}$ the total bandwidth for a modest

$M = 10000$. Thus, $R \geq 9$, and in practice can be expected to be much higher. For $R \geq 10$, $\rho \leq 1.76$ and $\rho \leq 3.48$ for α as high as 0.99 and 0.9, respectively, for all N. The approximation ratio of the placement algorithm PA can thus be taken to be a small constant for all practical purposes.

3.4 Extensions to Algorithm PA

Hierarchical Caching: Suppose a second-level parent cache of K' objects is provided in the path of the object requests, *e.g.*, at the CC in the wireless backhaul. In the absence of cooperation among child nodes, the most popular K objects will be replicated at the children, while the next K' popular objects would be placed at the parent. Which objects to unreplicate and share at the children when cooperation is enabled would depend on the limit on the amount by which the content served from the parent node may be increased. If this traffic need not be limited as long as the total hit rate increases, then an object placement may be obtained by assuming a cache of $K + K'$ objects at each child and applying algorithm PA, but restricting L to at most K. The $K - L$ most popular objects should then be replicated at all the child nodes, the next K' objects placed at the parent, and the next NL objects placed in one of the children as specified by PA. The reason for considering $K + K'$ objects as opposed to K during placement is to reduce the mean bandwidth of the objects that are shared, and thereby improve the efficiency of partitioning them in a fully shared manner without violating ICCB. If there is a restriction on the traffic that may be increased on the parent to child links, the restriction should be used to determine the cache size that PA should assume to determine a placement.

Relaxing the assumptions: The most restrictive of the similarity characteristics assumed for the caches and objects is that of identical sizes for all the objects. Identical object popularity distributions can be expected in a cluster of a few tens of BSs, which as discussed in Sec. 4, are sufficient in practice to achieve close to ideal hit rates. This is because at least a couple of thousand BSs are typically deployed in a mid-size city and hence 20-30 BSs can be expected to cover just a fraction of a city with somewhat homogeneous object access patterns. Homogeneity would also be enhanced by the larger expected mobility within a smaller region. The assumption of identical cache sizes may be expected to hold for the same reason that the number of BSs needed for good hit rates can be found within a small geography. If the assumption does not hold, it may easily be overcome by running PA with the smallest of the cache sizes. The additional capacity in the larger caches may simply be used for storing additional objects beyond those specified by PA for higher hit rates at the larger caches. Handling non-identical object sizes is discussed below.

Non-Identical Object Sizes: If object sizes are not identical, then since it is the bandwidth per unit size that matters, objects should be ordered by their popularity instead of bandwidths. Next, instead of choosing $N - 1$ objects from $\bar{\mathcal{O}}_C$ for every object chosen from $\mathcal{O}_C$, Algorithm PA should be modified to choose the maximum number of objects whose combined size does not exceed $N - 1$ times the size of an object chosen from $\mathcal{O}_C$. While non-homogeneity in object sizes can lead to inefficiencies in object selection and distribution, they can be expected to be minimal when the object pool is large.

4 Empirical Evaluation

In this section, we present the results of simulations conducted to evaluate the performance of our placement algorithm. We conducted experiments for varying values of total objects M, nodes N, and cache size C. Object size S was set to 1GB. Taking the total bandwidth served by BSs into account, the total object bandwidth was set to 20 Mbps, yielding a request rate of 0.0025/sec. ICCB B was set to 5 Mbps, for a maximum achievable increase of 25% to the hit rate. The M-Zipf distribution [8], in which the probability of accessing object of rank i is proportional to $1/(i+q)^{\gamma}$, was used for object popularities, with $q = 50$ $\gamma = 0.75$.

Representative results with $M = 20,000$, that is a total corpus size of 20 TB, and varying cache sizes as indicated are plotted in Fig. 3. Inset (a) plots the increase in hit rate as a % of the total object bandwidth of 20 Mbps. Since B is 25% of the total object bandwidth, the maximum achievable increase to the hit rate is 25%. The hit rate increase is rather low for small values of N. We also determined an upper bound to the optimal achievable increase for all the cases. The plots of the optimal increase almost coincide with the observed increase and hence have been omitted. The low hit rates for small N are therefore not due to the partitioning inefficiency of PA. This is because for $M = 20,000$, the bandwidth due to the most popular object is roughly 0.001% of the total bandwidth, hence R is quite large, easily exceeding 200. The approximation ratio as given by Thm. 2 is therefore close to 1. The low hit rates for small N are rather due to the larger values for $B/(N-1)$, and hence a large L, as indicated by the plots of $\alpha = L/K$ in Fig. 3(b). For large L, the ratio of the mean bandwidth of objects in $\mathcal{O}_C^{\mathsf{PA}}$ and $\bar{\mathcal{O}}_C^{\mathsf{PA}}$ is high. Since the total bandwidth of the two subsets is constrained to be at most $NB/(N-1)$, $\mathcal{BW}(\bar{\mathcal{O}}_C^{\mathsf{PA}})$, as a fraction of the total bandwidth, is low. Recall that the optimal increase in hit rate only slightly exceeds $\mathcal{BW}(\bar{\mathcal{O}}_C^{\mathsf{PA}})$, and hence when N is small, the increase, both optimal and observed, are low. The hit rate

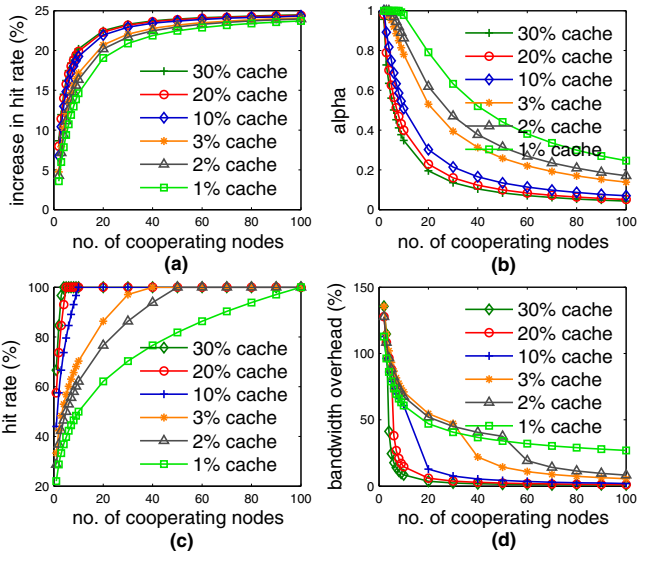


Fig. 3. Performance evaluation results for the placement algorithm PA. **(a)** Additional hit rates for varying % of cache sizes. **(b)** Values of $\alpha = L/K$ for the runs in (a). **(c)** Hit rates with varying N when ICCB is not a limitation. **(d)** Bandwidth overhead for the runs in (c). The legend entries are in the curve order in insets (a) and (c) and reverse order in insets (b) and (d).

increases with increasing N and is quite good for $N > 10$. The difference in the increase achieved with the largest (30% cache) and smallest caches (1% cache) is around 6% for $N \leq 10$, with the maximum of 6.83% seen for $N = 5$. The gap narrows for higher values of N. Similar trends were observed for varying M and object size S. The results indicate that a few tens of cooperating nodes are sufficient to achieve adequate hit rates. Also, the number of nodes needed to achieve a given increase to hit rate increases with decreasing cache size, by a factor of around two for an order of magnitude smaller cache. This is despite the fact that the base hit rate of a larger cache is higher.

Inset (c) plots the maximum cumulative hit rate achieved by PA when ICCB is not constrained. In this case, we also determined the minimum ICCB needed to achieve the observed hit rate. Bandwidth overhead %, given by $\frac{\text{minimum needed ICCB} - \text{increase in hit rate}}{\text{increase in hit rate}} \times 100\%$ is plotted in inset (d). It can be noted from inset (c) that a hit rate of 100% is reached at $N = N_{100\%} = \lceil \frac{\text{total corpus size}}{\text{cache size}} \rceil$, which is as expected. However, the bandwidth overhead at $N \leq N_{100\%}$ is quite high. This is because when $N \leq N_{100\%}$, the caching system is space constrained, and hence, all the objects are unreplicated and fully shared, including the popular, high-bandwidth objects. As N increases beyond $N_{100\%}$, the number of high bandwidth objects that are replicated increases, bringing down the bandwidth overhead. N needed to achieve 100% hit rate with minimal overhead is roughly an order of magnitude larger for a cache that is an order of magnitude smaller.

5 Conclusion

We have explored cooperative caching among the edge elements of a wireless infrastructure to ease the traffic stress expected in the wireless backhaul and core due to the manifold increase in video traffic. We have proposed an efficient object placement algorithm for cooperative caching that has a constant factor approximation ratio under practical conditions. Our simulation studies show that, in practice, the performance of the algorithm is very close to the optimal, and enabling cooperation among a few 10s of nodes may be sufficient to reap significant benefits. The viability of converting the proposed algorithm to a distributed one will be considered as part of future work.

References

1. Applegate, D., Archer, A., Gopalakrishnan, V., Lee, S., Ramakrishnan, K.K.: Optimal content placement for a large-scale vod system. In: ACM Co-NEXT (2010)
2. Borst, S., Gupta, V., Walid, A.: Distributed caching algorithms for content distribution networks. In: INFOCOM (2010)
3. Breslan, L., Cao, P., Fan, L., Phillips, G., Shenker, S.: Web caching and zipf-like distributions: Evidence and implications. In: Proceedings of IEEE INFOCOM, pp. 126–134 (1999)
4. Chankhunthod, A., Danzig, P., Neerdaels, C., Schwartz, M., Worrell, K.: A hierarchical internet object cache. In: USENIX Annual Technical Conference, pp. 153–163 (September 1996)
5. Cisco Systems Inc. Cisco visual networking index: Global mobile data traffic forecast update (2009-2014)

6. Fan, L., Cao, P., Almeida, J., Broder, A.: Summary cache: A scalable wide-area web cache sharing protocol. In: SIGCOMM, pp. 254–265 (September 1998)
7. Garey, M., Johnson, D.: Computers and Intractability: a Guide to the Theory of NP-Completeness, vol. ch. 4. W. H. Freeman and company, NY
8. Hafeeda, M., Saleh, O.: Traffic modeling and proportional partial caching for peer-to-peer systems. IEEE/ACM Transactions on Networking 16(6), 1447–1460 (2008)
9. Han, D., Andersen, D., Kaminsky, M., Papagiannaki, D., Seshan, S.: Hulu in the neighborhood. In: COMSNETS (2011)
10. Kangasharju, J., Roberts, J.W., Ross, K.W.: Object replication strategies for content distribution networks. Computer Communication Journal 25(4), 376–383 (2002)
11. Korupolu, M.R., Plaxton, C.G., Rajaraman, R.: Placement algorithms for hierarchical cooperative caching. In: SODA, pp. 586–595 (1998)
12. Song, Y., Ramasubramanian, V., Sirer, E.: Cobweb: a proactive analysis-driven approach to content distribution. In: SOSP, Poster (2005)
13. Venkataramani, A., Weidmann, P., Dahlin, M.: Bandwidth constrained placement in a wan. In: Principles of Distributed Computing, pp. 134–143 (2001)
14. Xu, Z., Bhuyan, L.: Qos-aware object replica placement in cdns. In: GLOBECOM (2005)

Author Index