Chenggang Wu
Albert Cohen (Eds.)

LNCS 8299

Advanced Parallel Processing Technologies

10th International Symposium, APPT 2013
Stockholm, Sweden, August 2013
Revised Selected Papers

Springer

Lecture Notes in Computer Science 8299

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

Editorial Board

David Hutchison
Lancaster University, UK

Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler
University of Surrey, Guildford, UK

Jon M. Kleinberg
Cornell University, Ithaca, NY, USA

Alfred Kobsa
University of California, Irvine, CA, USA

Friedemann Mattern
ETH Zurich, Switzerland

John C. Mitchell
Stanford University, CA, USA

Moni Naor
Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz
University of Bern, Switzerland

C. Pandu Rangan
Indian Institute of Technology, Madras, India

Bernhard Steffen
TU Dortmund University, Germany

Madhu Sudan
Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos
University of California, Los Angeles, CA, USA

Doug Tygar
University of California, Berkeley, CA, USA

Gerhard Weikum
Max Planck Institute for Informatics, Saarbruecken, Germany

Chenggang Wu Albert Cohen (Eds.)

Advanced Parallel Processing Technologies

10th International Symposium, APPT 2013
Stockholm, Sweden, August 27-28, 2013
Revised Selected Papers

Springer

Volume Editors

Chenggang Wu
Chinese Academy of Sciences, Institute of Computing Technology
State Key Laboratory of Computer Architecture
No. 6 Kexueyuan South Road, Haidian District, 100190 Beijing, China
E-mail: wucg@ict.ac.cn

Albert Cohen
INRIA and École Normale Supérieure
Département d'Informatique
45 rue d'Ulm, 75005 Paris, France
E-mail: albert.cohen@inria.fr

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-45292-5 e-ISBN 978-3-642-45293-2
DOI 10.1007/978-3-642-45293-2
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013955010

CR Subject Classification (1998): C.1.4, I.3.1, D.2, D.1.3, C.2, J.1, H.5, D.4

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

© Springer-Verlag Berlin Heidelberg 2013
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in ist current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

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

With the continuity of Moore's law in the multicore era, the success of cloud computing, and the emerging heterogeneous systems, parallelism pervades almost every domain of computing and information processing. This creates grand challenges to computer architectures and systems, and puts enormous pressure on the design of a new generation of tools and programming methods to harness these systems. These challenges formed the core theme of APPT 2013. Our two-day technical program of APPT 2013 provided an excellent venue capturing some of the state of the art and practice in parallel architecture, parallel software, concurrent and distributed systems, cloud computing, with a highlight on computing systems for big data applications.

We believe this biennial event provides a forum for the presentation of this community's research efforts and exchanging viewpoints. We would like to express our thankfulness to all the colleagues who submitted papers and congratulate those whose papers were accepted. As an event that has taken place for 18 years, APPT aims at providing a high-quality program for all attendees. In all, 62 papers were submitted, a 50% increase over the last conference. We accepted 30 papers: 18 as an oral presentation and 12 as a poster presentation. The regular paper acceptance rate is 29%.

Most submissions were reviewed by three Program Committee(PC) members. An online PC meeting was held during June 9–14. Consensus was reached for each submission.

We would like to thank the authors for submitting their fine work to APPT 2013, and we would also like to show our sincere appreciation to this year's dream-team PC. The 18 PC members did an excellent job in returning high-quality reviews in time and engaging in a constructive online discussion.

We would also like to thank the general chairs (Prof. Yong Dou and Mats Brorsson), and the local organization, publicity and publication chairs for making APPT 2013 possible. Finally, the contributions from our sponsors and supporters were invaluable: We would like to thank the China Computer Federation, the Technical Committee on Computer Architecture of the China Computer Federation, the Royal Institute of Technology in Stockholm, the National Laboratory for Parallel and Distributed Processing, and the State Key Laboratory of High Performance Computing. Our thanks also goes to Springer for its assistance in putting the proceedings together.

August 2013

Albert Cohen
Chenggang Wu

Organization

APPT 2013 was organized by the China Computer Federation.

General Chairs

Yong Dou	National University of Defense Technology, China
Mats Brorsson	KTH Royal Institute of Technology, Sweden

Organization Chairs

Zhonghai Lu	KTH Royal Institute of Technology, Sweden
Xingwei Wang	Northeastern University, China

Program Chairs

Chenggang Wu	Institute of Computing Technology, China
Albert Cohen	Inria, France

Program Committee

David Black-Schaffer	Uppsala University, Sweden
Haibo Chen	Shanghai Jiaotong University, China
Yunji Chen	Institute of Computing Technology, China
Chen Ding	University of Rochester, USA
Lieven Eeckhout	Ghent University, Belgium
Robby Findler	Northwestern University, USA
Georgi Gaydadjiev	University of Gothenburg, Sweden
R. Govindarajan	Indian Institute of Science, India
Wei-Chung Hsu	National Chiao Tung University Hsinchu, Taiwan, China
Wolfgang Karl	Karlsruhe Institute of Technology, Germany
Lawrence Rauchwerger	Texas A&M University, USA
Xipeng Shen	University of William and Mary, USA
Olivier Temam	INRIA, France
Zhenjiang Wang	Institute of ComputingTechnology, China
Youfeng Wu	Intel, USA
Pen-Chung Yew	University of Minnesota, USA
Qing Yi	University of Colorado at Colorado, USA
Yunlong Zhao	Harbin Institute of Technology, China

Publicity Chairs

Gongxuan Zhang	Nanjing University of Science and Technology, China
Hans Vandierendonck	Queen's University Belfast, Queen's University Belfast, UK

Publication Chairs

Junjie Wu	National University of Defense Technology, China
Zhicai Shi	Shanghai University of Engineering Science, China

Sponsor

China Computer Federation China

Supporters

Technical Committee on Computer Architecture of CCF China
KTH Royal Institute of Technology Sweden
Springer-Verlag Germany
National Key Laboratory of Parallel and Distributed Processing China
State Key Laboratory of High Performance Computing China

Table of Contents

Inference and Declaration of Independence in Task-Parallel Programs . . . 1
Foivos S. Zakkak, Dimitrios Chasapis, Polyvios Pratikakis, Angelos Bilas, and Dimitrios S. Nikolopoulos

BDDT: Block-Level Dynamic Dependence Analysis for Task-Based Parallelism . . . 17
George Tzenakis, Angelos Papatriantafyllou, Hans Vandierendonck, Polyvios Pratikakis, and Dimitrios S. Nikolopoulos

A User-Level NUMA-Aware Scheduler for Optimizing Virtual Machine Performance . . . 32
Yuxia Cheng, Wenzhi Chen, Xiao Chen, Bin Xu, and Shaoyu Zhang

Towards RTOS: A Preemptive Kernel Basing on Barrelfish . . . 47
Jingwei Yang, Xiang Long, Xukun Shen, Lei Wang, Shuaitao Feng, and Siyao Zheng

Interrupt Modeling and Verification for Embedded Systems Based on Time Petri Nets . . . 62
Gang Hou, Kuanjiu Zhou, Junwang Chang, Rui Li, and Mingchu Li

Pruning False Positives of Static Data-Race Detection via Thread Specialization . . . 77
Chen Chen, Kai Lu, Xiaoping Wang, Xu Zhou, and Li Fang

G-Paradex: GPU-Based Parallel Indexing for Fast Data Deduplication . . . 91
Bin Lin, Xiangke Liao, Shanshan Li, Yufeng Wang, He Huang, and Ling Wen

Research on SPH Parallel Acceleration Strategies for Multi-GPU Platform . . . 104
Lei Hu, Xukun Shen, and Xiang Long

Binarization-Based Human Detection for Compact FPGA Implementation . . . 119
Shuai Xie, Yibin Li, Zhiping Jia, and Lei Ju

HPACS: A High Privacy and Availability Cloud Storage Platform with Matrix Encryption . . . 132
Yanzhang He, Xiaohong Jiang, Kejiang Ye, Ran Ma, and Xiang Li

ECAM: An Efficient Cache Management Strategy for Address Mappings in Flash Translation Layer 146
Xuchao Xie, Qiong Li, Dengping Wei, Zhenlong Song, and Liquan Xiao

A Performance Study of Software Prefetching for Tracing Garbage Collectors 160
Hao Wu, Zhenzhou Ji, Suxia Zhu, and Zhigang Chen

Adaptive Implementation Selection in the SkePU Skeleton Programming Library 170
Usman Dastgeer, Lu Li, and Christoph Kessler

Automatic Skeleton-Based Compilation through Integration with an Algorithm Classification 184
Cedric Nugteren, Pieter Custers, and Henk Corporaal

Optimizing Program Performance via Similarity, Using a Feature-Agnostic Approach 199
Rosario Cammarota, Laleh Aghababaie Beni, Alexandru Nicolau, and Alexander V. Veidenbaum

Scalable NIC Architecture to Support Offloading of Large Scale MPI Barrier 214
Shaogang Wang, Weixia Xu, Dan Wu, Zhengbin Pang, and Pingjing Lu

Hierarchical Clustering Routing Protocol Based on Optimal Load Balancing in Wireless Sensor Networks 227
Tianshu Wang, Gongxuan Zhang, Xichen Yang, and Ahmadreza Vajdi

An Efficient Parallel Mechanism for Highly-Debuggable Multicore Simulator 241
Xiaochun Ye, Dongrui Fan, Da Wang, Fenglong Song, Hao Zhang, and Zhimin Tang

Data Access Type Aware Replacement Policy for Cache Clustering Organization of Chip Multiprocessors 254
Chongmin Li, Dongsheng Wang, Haixia Wang, Guohong Li, and Yibo Xue

Agent-Based Credibility Protection Model for Decentralized Network Computing Environment 269
Xiaolong Xu, Qun Tu, and Xinheng Wang

A Vectorized K-Means Algorithm for Intel Many Integrated Core Architecture 277
Fuhui Wu, Qingbo Wu, Yusong Tan, Lifeng Wei, Lisong Shao, and Long Gao

Towards Eliminating Memory Virtualization Overhead 295
Xiaolin Wang, Lingmei Weng, Zhenlin Wang, and Yingwei Luo

An Improved FPGAs-Based Loop Pipeline Scheduling Algorithm for Reconfigurable Compiler 307
Zhenhua Guo, Yanxia Wu, Guoyin Zhang, and Tianxiang Sui

ACF: Networks-on-Chip Deadlock Recovery with Accurate Detection and Elastic Credit 319
Nan Wu, Yuran Qiao, Mei Wen, and Chunyuan Zhang

An Auction and League Championship Algorithm Based Resource Allocation Mechanism for Distributed Cloud 334
Jiajia Sun, Xingwei Wang, Keqin Li, Chuan Wu, Min Huang, and Xueyi Wang

Accelerating Software Model Checking Based on Program Backbone 347
Kuanjiu Zhou, Jiawei Yong, Xiaolong Wang, Longtao Ren, Gang Hou, and Junwang Chang

A Cloud Computing System for Snore Signals Processing 359
Jian Guo, Kun Qian, Zhaomeng Zhu, Gongxuan Zhang, and Huijie Xu

Research on Optimum Checkpoint Interval for Hybrid Fault Tolerance 367
Lei Zhu, Jianhua Gu, Yunlan Wang, and Tianhai Zhao

Programming Real-Time Image Processing for Manycores in a High-Level Language 381
Essayas Gebrewahid, Zain-ul-Abdin, Bertil Svensson, Veronica Gaspes, Bruno Jego, Bruno Lavigueur, and Mathieu Robart

Self-adaptive Retransmission for Network Coding with TCP 396
Chunqing Wu, Hongyun Zhang, Wanrong Yu, Zhenqian Feng, and Xiaofeng Hu

Author Index 409

Inference and Declaration of Independence in Task-Parallel Programs

Foivos S. Zakkak[1], Dimitrios Chasapis[1], Polyvios Pratikakis[1], Angelos Bilas[1], and Dimitrios S. Nikolopoulos[2]

[1] FORTH-ICS, Heraklion, Crete, Greece
{zakkak,hassapis,polyvios,bilas}@ics.forth.gr
[2] Queens University of Belfast, Belfast, United Kingdom
d.nikolopoulos@qub.ac.uk

Abstract. The inherent difficulty of thread-based shared-memory programming has recently motivated research in high-level, task-parallel programming models. Recent advances of Task-Parallel models add implicit synchronization, where the system automatically detects and satisfies data dependencies among spawned tasks. However, dynamic dependence analysis incurs significant runtime overheads, because the runtime must track task resources and use this information to schedule tasks while avoiding conflicts and races.

We present SCOOP, a compiler that effectively integrates static and dynamic analysis in code generation. SCOOP combines context-sensitive points-to, control-flow, escape, and effect analyses to remove redundant dependence checks at runtime. Our static analysis can work in combination with existing dynamic analyses and task-parallel runtimes that use annotations to specify tasks and their memory footprints. We use our static dependence analysis to detect non-conflicting tasks and an existing dynamic analysis to handle the remaining dependencies. We evaluate the resulting hybrid dependence analysis on a set of task-parallel programs.

Keywords: Task-Parallelism, Static Analysis, Dependence Analysis, Deterministic Execution.

1 Introduction

The inherent difficulty and complexity of thread-programming has recently led to the development of several task-based programming models [1–4]. Task-based parallelism offers a higher level abstraction to the programmer, making it easier to express parallel computation. Although early task-based parallel languages required manual synchronization, recent task-based systems implicitly synchronize tasks, using a task's memory footprint at compile or at run time to detect and avoid concurrent accesses or even produce deterministic execution [5–9]. In order for such a dependence analysis to benefit program performance, it must (i) be accurate, so that it does not discover false dependencies; and (ii) have low overhead, so that it does not nullify the benefit of discovering extra parallelism.

Static systems detect possibly conflicting tasks in the program code and insert synchronization prohibiting concurrent access to shared memory among all runtime instances of possibly conflicting tasks. As this can be too restrictive, some existing static

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 1–16, 2013.
© Springer-Verlag Berlin Heidelberg 2013

systems speculatively allow conflicting task instances to run in parallel and use dynamic techniques to detect and correct conflicts [6].

Dynamic dependence analysis offers the benefit of potentially discovering more parallelism than is possible to describe statically in the program, as it checks all runtime task instances for conflicts and only synchronizes task instances that actually (not potentially) access the same resources. However, dynamic dependence analysis incurs a high overhead compared to hand-crafted synchronization. It requires a complex runtime system to manage and track memory allocation, check for conflicts on every task instance, and schedule parallel tasks. Often, the runtime cost of checking for conflicts in pessimistic, or rolling back a task in optimistic runtimes becomes itself a bottleneck, limiting the achievable speedup as the core count grows.

This paper aims to alleviate the overhead of dynamic dependence analysis without sacrificing the benefit of implicit synchronization. We develop SCOOP, a compiler that brings together static and dynamic analyses into a hybrid dependence analysis in task-parallel programs. SCOOP uses a static dependence analysis to detect and remove runtime dependence checks when unnecessary. It then inserts calls to the task-parallel runtime dynamic analysis to resolve the remaining dependencies only when necessary. Our work makes the following contributions:

- We present a static analysis that detects independent task arguments and reduces the runtime overhead of dynamic analysis. We implement our analysis in SCOOP, a source-to-source compiler for task-parallel C, using OpenMP-Task extensions to define tasks and their memory footprints.
- We combine our static dependence analysis with an existing dynamic analysis, resulting in an efficient hybrid dependence analysis for parallel tasks. SCOOP uses the static dependence analysis to infer redundant and unnecessary dependence checks and inserts custom code to use the dynamic analysis only for the remaining task dependencies at runtime.
- We evaluate the effect of our analysis using an existing runtime system. On a representative set of benchmarks, SCOOP discovers almost all independent task arguments. In applications with independent task arguments, SCOOP achieved speedups up to 68%.

2 Motivation

Consider the C program in Figure 1. This program has three global integer variables, `a`, `b` and `c` (line 1) and a global pointer `alias` (line 2) that points to `b`. Function `set()` copies the value of its second argument to the first (line 4) and function `addto()` adds the value of its second argument to the value of its first (line 5). The two functions are then invoked in two parallel tasks, to add `c` to `b` (lines 8–9) and to set the value of `a` to the value pointed to by `alias` (lines 11–12). The first task reads and writes its first argument, `b`, and reads from its second argument, `c`. Similarly, the second task writes to its first argument, `a`, and reads from its second argument `alias`. The program then waits at a synchronization point for the first two tasks to finish (line 14) and then spawns a third task that reads from `c` and writes to `a` (lines 16–17).

```
1  int a = 1, b = 2, c = 3;
2  int *alias = &b;
3
4  void set(int *x, int *y) { *x = *y; }
5  void addto(int *x, int *y) { *x += *y; }
6
7  int main() {
8    #pragma task inout(&b) in(&c)
9    addto(&b, &c);
10
11   #pragma task out(&a) in(alias)
12   set(&a, alias);
13
14   #pragma wait all
15
16   #pragma task out(&a) in(&c)
17   set(&a, &c);
18 }
```

Fig. 1. Tasks with independent arguments

To execute this program preserving the sequential semantics, the second task `set` needs to wait until the value of `b` is produced by the first task, i.e., there is a *dependence* on memory location `b`. Note, however, that since the third task cannot be spawned until the first two return, memory location `c` is only accessed by the first task and `a` is only accessed by the second. So, any dependence analysis time spent checking for conflicts on `a` or `c` before it starts the first two tasks is unnecessary overhead that delays the creation of the parallel tasks, possibly restricting available parallelism and thus the scalability of the program. So, the `#pragma task` directive spawning these tasks states that `c` and `a` are *safe* or *independent* arguments, that the analysis does not need to track. For the same reason, both the arguments of the third task are safe, meaning it can start to run without checking for dependencies.

Section 3 describes the static analysis we use to discover independent task arguments like `a` and `c` above. Inferring that a task argument does not need to be checked for dependencies requires verifying that no other task can access that argument. In short, the static analysis infers this independence in three steps. First, we compute aliasing information for all memory locations in the program. Second, we compute which tasks can run in parallel; we do not need to check for conflicting arguments between for example the second and third task in the example of Figure 1, even though `a` is accessed by both, because the barrier prohibits them from running at the same time. Third, we check whether a memory location (through any alias) is never accessed in parallel by more than one task. We can then safely omit checking this location at runtime. We can extend this idea by differentiating between read and write accesses and allowing for concurrent reads without checking for dependencies, as long as no writes can happen in parallel.

$$
\begin{array}{lll}
\text{Values} & v & ::= n \mid \texttt{()} \mid \lambda\, x\,.\, e \\
\text{Expressions} & e & ::= v \mid x \mid e; e \mid e\ e \mid \texttt{ref}\ e \mid !\, e \mid e := e \\
 & & \quad \mid\ \texttt{task}(e_1, \ldots, e_n)\,\{e\} \mid \texttt{barrier} \\
\text{Locations} & \rho & \in \mathcal{L} \\
\text{CFG Points} & \phi & \in \mathcal{F} \\
\text{Tasks} & \pi & \in \mathcal{T} \\
\text{Types} & \tau & ::= \mathit{int} \mid \mathit{unit} \mid (\tau, \phi) \rightarrow (\tau, \phi) \mid \mathit{ref}^{\rho}(\tau) \\
\text{Constraints} & C & ::= \emptyset \mid C \cup C \mid \tau \leq \tau \mid \rho \leq \rho \mid \phi \leq \phi \\
 & & \quad \mid\ \rho \leq \pi \mid \pi \| \pi \mid \phi : \mathsf{Barrier} \mid \phi : \pi \\
\text{Environments} & \Gamma & ::= \cdot \mid \Gamma, x : \tau
\end{array}
$$

Fig. 2. $\lambda_{\|}$: A simple task-based parallel language

3 Static Independence Analysis

This section presents the core algorithm of the independence analysis. To simplify the presentation, we use a small language $\lambda_{\|}$, and do not differentiate between reads and writes. Section 4 describes how we extended our analysis to the C full programming language.

3.1 The Language $\lambda_{\|}$

Figure 2 presents $\lambda_{\|}$, a simple task-parallel programming language. $\lambda_{\|}$ is a simply-typed lambda calculus extended with dynamic memory allocation and updatable references, task creation and barrier synchronization. Values include integer constants n, the unit value `()` and functions $\lambda\, x\,.\, e$. Program expressions include variables x, function application $e_1\ e_2$, sequencing, memory operations and task operations. Specifically, expression `ref` e allocates some memory, initializes it with the result of evaluating e, and returns a pointer to that memory; expression $e_1 := e_2$ evaluates e_1 to a pointer and updates the pointed memory using the value of e_2; and expression $!\, e$ evaluates e to a pointer and returns the value in that memory location. Expression $\texttt{task}(e_1, \ldots, e_n)\,\{e\}$ evaluates each e_i to a pointer and then evaluates the task body e, possibly in parallel. The task body e must always return `()` and can only access the given pointers; if e is evaluated in a parallel task, the task expression immediately returns `()`. Finally, expression `barrier` waits until all tasks issued until this point have been executed.

3.2 Type System

We use a type system to generate a set of constraints C and infer independence of task arguments. Figures 3(a) and 3(b) shows the type language of $\lambda_{\|}$, which includes integer and unit types, function types $(\tau, \phi) \rightarrow (\tau, \phi)$ and reference (or pointer) types $\mathit{ref}^{\rho}(\tau)$. We annotate function and reference types with inference labels ϕ and ρ, and use them to compute the *control flow graph* among ϕ labels and the *points-to graph* among ρ labels, respectively. Specifically, typing the program creates a constraint graph C, which

has three kinds of vertices. Location labels ρ annotating reference types abstract over memory locations, control flow labels ϕ abstract over a control flow point in the program execution, and task labels π abstract over parallel tasks in the program. Typing a program e in $\lambda_{\|}$ creates a constraint graph C. Constraint $\tau_1 \leq \tau_2$ requires τ_1 to be a subtype of τ_2. Constraint $\rho_1 \leq \rho_2$ (ρ_1 "flows to" ρ_2) means abstract memory location ρ_2 references all locations that ρ_1 references. Constraint $\phi_1 \leq \phi_2$ means the execution of control flow point ϕ_2 follows immediately after the execution of ϕ_1. Constraint $\rho \leq \pi$ (ρ "is an argument of" π) means an abstract memory location ρ is in the memory footprint of task π. Constraint $\pi_1 \| \pi_2$ (π_1 "can happen in parallel with" π_2) means there may be an execution where tasks represented by π_1 and π_2 are executed in parallel. Constraint $\phi : \mathsf{Barrier}$ means there is barrier synchronization at control flow point ϕ of all executions. Finally, constraint $\phi : \pi$ means there can be an execution where task π is executed in parallel while control flow reaches point ϕ.

Figure 3(a) shows the type system for $\lambda_{\|}$. Typing judgments have the form $C; \phi; \Gamma \vdash e : \tau; \phi'$, meaning program expression e has type τ under assumptions Γ and constraint set C. Rules [T-INT] and [T-UNIT] and [T-VAR] are standard, with the addition of control flow point ϕ as both starting and ending point. Rule [T-FUN] types function definitions. The function type $(\tau_1, \phi_1) \rightarrow (\tau_2, \phi_2)$ includes the starting and ending control flow point of the function body. As with typing the other values, function definitions do not change control flow point ϕ. Rule [T-SEQ] types sequence, where e_1 must have type $unit$, and the sequence expression has the type of e_2. The ending control flow point ϕ_1 of e_1 is the starting point of e_2. The first two premises in rule [T-APP] type the function expression e_1 and argument e_2 capturing the control flow order, the third premise "inlines" the control flow of the function by setting the function starting point immediately after the evaluation of the argument. The function application ends at ϕ_2. Rule [T-REF] creates a fresh label ρ that represents all memory locations produced by the expression, and annotates the resulting type. Rules [T-DEREF] and [T-ASGN] are straightforward, and type reference read and write expressions respectively.

Rule [T-TASK] types task creation expressions. The first premise types the task argument expressions $e_1, \ldots, e_n$ with reference types in that control flow order. The next three premises create a constraint that task π runs in parallel with control flow point ϕ' of the task-create expression. The fifth premise marks all locations ρ_i of the arguments as the footprint of task π, and the last premise requires the task body to have type $unit$. The task's control flow ends at any control flow point ϕ''. Rule [T-BARRIER] types barrier expressions, marking control flow point ϕ' as a barrier synchronization operation. Finally, rule [T-SUB] is standard subsumption.

3.3 Constraint Resolution

Applying the type system shown in Figure 3(a) generates a set of constraints C. To infer task arguments that are safe to skip during the runtime dependence analysis, we first compute the may-happen-in-parallel relation $\pi_1 \| \pi_2$ among tasks by solving the constraints C. Figure 3(b) shows the constraint resolution algorithm as a set of rewriting rules that are applied exhaustively until C cannot change any further. Here, $\cup\Rightarrow$ rewrites the constraints on the left to be the union of the constraints on both the left and right side.

$$\text{[T-INT]}\quad \frac{}{C;\phi;\Gamma \vdash n : \mathit{int};\phi}$$

$$\text{[T-UNIT]}\quad \frac{}{C;\phi;\Gamma \vdash \texttt{()} : \mathit{unit};\phi}$$

$$\text{[T-FUN]}\quad \frac{\phi_1\text{–}\mathit{fresh} \qquad C;\phi_1;\Gamma, x:\tau_1 \vdash e : \tau_2;\phi_2}{C;\phi;\Gamma \vdash \lambda\, x\,.\, e : (\tau_1,\phi_1) \to (\tau_2,\phi_2);\phi}$$

$$\text{[T-VAR]}\quad \frac{\Gamma(x) = \tau}{C;\phi;\Gamma \vdash x : \tau;\phi}$$

$$\text{[T-SEQ]}\quad \frac{C;\phi;\Gamma \vdash e_1 : \mathit{unit};\phi_1 \qquad C;\phi_1;\Gamma \vdash e_2 : \tau;\phi_2}{C;\phi;\Gamma \vdash e_1;e_2 : \tau;\phi_2}$$

$$\text{[T-APP]}\quad \frac{C;\phi;\Gamma \vdash e_1 : (\tau_1,\phi_1) \to (\tau_2,\phi_2);\phi' \qquad C;\phi';\Gamma \vdash e_2 : \tau_1;\phi'' \qquad C \vdash \phi'' \leq \phi_1}{C;\phi;\Gamma \vdash e_1\ e_2 : \tau_2;\phi_2}$$

$$\text{[T-REF]}\quad \frac{C;\phi;\Gamma \vdash e : \tau;\phi' \qquad \rho\text{–}\mathit{fresh}}{C;\phi;\Gamma \vdash \texttt{ref}\ e : \mathit{ref}^{\rho}(\tau);\phi'}$$

$$\text{[T-DEREF]}\quad \frac{C;\phi;\Gamma \vdash e : \mathit{ref}^{\rho}(\tau);\phi'}{C;\phi;\Gamma \vdash !\,e : \tau;\phi'}$$

$$\text{[T-ASGN]}\quad \frac{C;\phi;\Gamma \vdash e_1 : \mathit{ref}^{\rho}(\tau);\phi' \qquad C;\phi';\Gamma \vdash e_2 : \tau;\phi''}{C;\phi;\Gamma \vdash e_1 := e_2 : \tau;\phi''}$$

$$\text{[T-TASK]}\quad \frac{\forall i \in [1..n]\,.\, C;\phi_i;\Gamma \vdash e_i : \mathit{ref}^{\rho_i}(\tau_i);\phi_{i+1} \qquad \phi',\pi\text{–}\mathit{fresh} \qquad C \vdash \phi_{n+1} \leq \phi' \qquad C \vdash \phi' : \pi \qquad \forall i \in [1..n]\,.\, C \vdash \rho_i \leq \pi \qquad C;\phi';\Gamma \vdash e' : \mathit{unit};\phi''}{C;\phi_1;\Gamma \vdash \texttt{task}(e_1,\ldots,e_n)\,\{e'\} : \mathit{unit};\phi'}$$

$$\text{[T-BARRIER]}\quad \frac{\phi'\text{–}\mathit{fresh} \qquad C \vdash \phi' : \mathsf{Barrier} \qquad C \vdash \phi \leq \phi'}{C;\phi;\Gamma \vdash \texttt{barrier} : \mathit{unit};\phi'}$$

$$\text{[T-SUB]}\quad \frac{C;\phi;\Gamma \vdash e : \tau;\phi' \qquad C \vdash \tau \leq \tau'}{C;\phi;\Gamma \vdash e : \tau';\phi'}$$

(a) Type Inference Rules

$$\begin{aligned}
C \cup \{\mathit{int} \leq \mathit{int}\} &\Rightarrow C \\
C \cup \{\mathit{unit} \leq \mathit{unit}\} &\Rightarrow C \\
C \cup \{(\tau_1,\phi_1) \to (\tau_2,\phi_2) \leq (\tau_1',\phi_1') \to (\tau_2',\phi_2')\} &\Rightarrow \\
&\quad C \cup \{\tau_1' \leq \tau_1, \tau_2 \leq \tau_2', \phi_1' \leq \phi_1, \phi_2 \leq \phi_2'\} \\
C \cup \{\mathit{ref}^{\rho_1}(\tau_1) \leq \mathit{ref}^{\rho_2}(\tau_2)\} &\Rightarrow C \cup \{\rho_1 \leq \rho_2, \tau_1 \leq \tau_2, \tau_2 \leq \tau_1\} \\
C \cup \{\rho \leq \rho', \rho' \leq \rho''\} &\cup\Rightarrow \{\rho \leq \rho''\} \\
C \cup \{\rho \leq \rho', \rho' \leq \pi\} &\cup\Rightarrow \{\rho \leq \pi\} \\
C \cup \{\phi_1 \leq \phi_2, \phi_1 : \pi\} &\cup\Rightarrow \{\phi_2 : \pi\} \text{ when } \{\phi_2 : \mathsf{Barrier}\} \notin C \\
C \cup \{\phi : \pi_1, \phi : \pi_2\} &\cup\Rightarrow \{\pi_1 \| \pi_2\}
\end{aligned}$$

(b) Constraint Solving Rules

Fig. 3. Constraint generation and solving

The first four rules reduce subtyping constraints into edges between abstract labels: we drop integer and unit subtyping; we replace function subtyping with contravariant edges between the starting control flow points and arguments, and covariant edges between the returning control flow points and results; and we replace reference subtyping with equality on the referenced type (note both directions of subtyping) and a flow constraint on the abstract location labels. The fifth rule solves the points-to graph by adding all transitivity edges between abstract memory locations, and the seventh rule

marks any locations aliasing task arguments also as task arguments. The seventh rule amounts to a forwards data-flow analysis on the control flow graph. Namely, for every control flow edge $\phi_1 \leq \phi_2$ we propagate any task π that executes in parallel with ϕ_1 to also execute in parallel with ϕ_2, unless ϕ_2 is a barrier. Finally, the last rule marks any two tasks π_1 and π_2 that both run in parallel with any control flow point ϕ, as also in parallel with one another. We use C^* to represent the result of exhaustively applying the constraint resolution rules on a set C.

3.4 Task Argument Independence

Having solved the constraints C of a program, we can now infer independent task arguments, namely arguments that cannot be accessed concurrently by any two parallel tasks. Formally, we define the dependent set $D_C(\rho)$ of location ρ under constraints C to be the set of tasks that can access ρ in parallel:

$$D_C(\rho) \doteq \{\pi \mid C^* \vdash \rho \leq \pi\}$$

We can now compute independent task arguments, i.e., memory locations that can be at most accessed by one task:

$$C \vdash \mathit{Safe}(\rho) \iff |D_C(\rho)| \leq 1$$

4 Implementation

We have extended the algorithm presented in Section 3 to the full C programming language in a compiler for task-parallel programs with implicit synchronization. To handle the full C language, we make several assumptions concerning data- and control-flow. Our pointer analysis assumes that all allocation in the program is done through the `libc` memory allocator functions and that no pointers are constructed from integers. We treat unsafe casts and pointer arithmetic conservatively and conflate the related memory locations. We perform field-inference for structs and type-inference for `void*` pointers to increase the precision of the pointer analysis. We currently assume there is no `setjmp/longjmp` control-flow.

The compiler is structured in three phases. The first extends the C front-end with support for OpenMP-like `#pragma` directives to define tasks and task footprints. We have chosen to mark task creation at the calling context, instead of marking a function definition and have every invocation of the function create a parallel task for better precision; this way we are able to call the same function both sequentially or as a parallel task without rewriting it or creating wrapper functions. The syntax for declaring task footprints supports strided memory access patterns, so that we can describe multidimensional array tiles as task arguments. When not explicitly given, we assume that the size of a task argument is the size of its type.

The second phase uses a type-system to generate points-to and control flow constraints and solves them to infer argument independence, as described in Section 3. In Section 3, however, we have made several simplifying assumptions to improve the presentation of the algorithm, that must be addressed when applying the analysis on the full C language.

Although in the formal presentation we do not differentiate between read and write effects in the task footprint, we actually treat input and output task arguments differently. In particular, we match the behavior of the runtime system, which allows multiple reader tasks of a memory location to run in parallel. Thus, we also mark task arguments that are only read in parallel as independent.

To increase the analysis precision, we use a context-sensitive, field sensitive points-to analysis, and a context-sensitive control flow analysis. In both cases, context sensitivity is encoded as CFL-reachability, with either points-to or control flow edges that enter or exit a calling context marked as special *open* or *close parenthesis* edges [10].

Finally, in several benchmarks tasks within loops access disjoint parts of the same array. However, the points-to analysis treats all array elements as one abstract location, producing false aliasing and causing such safe arguments to be missed. To rectify this, in part, we have implemented a simple loop-dependence analysis that discovers when different loop iterations access non-overlapping array elements. This (orthogonal) problem has been extensively studied in the past [11, 12], resulting in many techniques that can be applied to improve the precision of this optimization.

The final phase transforms the input program to use the runtime system to create tasks and perform dependence checks for task arguments not inferred or declared independent. As an optimization, the compiler produces custom code to interact with the runtime structures instead of using generic runtime API calls. In particular, for each `#pragma task` call, the compiler generates custom code that creates a task descriptor (closure) with the original function as task body, registers the task arguments with the runtime dependence analysis, and replaces the specified function call with the generated code.

We ran the resulting programs using the BDDT runtime [8] to perform dynamic dependence analysis and check for any dependencies that are not ruled out statically. The BDDT runtime system maintains a representation of every task instance and its footprint at run time, and uses these to check for overlap among task arguments and compare their access properties to detect task dependencies. To do that, BDDT splits task arguments into virtual memory blocks of configurable size and analyzes dependencies between blocks. Similarly to whole-object dependence analysis used in tools such as SMPSs, SvS, and OoOJava, block-based analysis detects true read-after-write (RAW) dependencies, or write-after-write (WAW) and write-after-read (WAR) anti-dependencies between blocks, by comparing block starting addresses and checking their access attributes. We selected BDDT for our experiments due to its good performance and because it is easy to disable specific dynamic checks on specific task instances using its API. However, the SCOOP static independence analysis can be used to remove unnecessary dynamic checks from other task-parallel runtimes with implicit synchronization.

We used a region-based allocator to support dynamic memory allocation in tasks, and allow for tasks that operate on complex data structures. This way, we extend BDDT to handle task footprints that include dynamic regions specially: the task footprint language allows several task arguments to belong to a dynamic region; the task footprint then includes the region instead of the individual arguments, and SCOOP registers only the region descriptor with the dependence analysis.

Table 1. Benchmark description and analysis performance

Benchmark	LOC	Tasks	Total Args	Scalar Args	Analysis (s)	Graph Nodes	Safe Args
Black-Scholes	3564	1	8	1	3.17	1790	6
Ferret	30145	1	2	0	699.05	85128	2
Cholesky	1734	4	16	8	1.06	7571	0
GMRES	2661	18	72	20	2.21	7957	9
HPL	2442	11	59	35	1.47	9330	0
Jacobi	1084	1	6	0	0.74	3980	0
FFT	2935	4	12	4	1.72	9750	3
Multisort	1215	2	8	4	1.02	4016	0
Intruder	6452	1	5	1	19.89	16855	3

5 Evaluation

We evaluated the effect of the static independence analysis in SCOOP on a set of representative benchmarks, including several computational kernels and small-sized parallel applications. We ran the experiments on a Cray XE6 compute node with 32GB memory and two AMD Interlagos 16-core 2.3GHz dual-processors, a total of 32 cores. We compiled all benchmarks with GNU GCC 4.4.5 using the -O3 optimization flag. As is standard in evaluation of task-parallel systems in the literature, we measured the performance of the parallel section of the code, excluding any initialization and I/O at the start and end of each benchmark. We used barriers to separate the initialization phase from the measured computation. Finally, to minimize variation among different runs, we report the average measurements over twenty runs for each benchmark.

We use the following benchmarks in our evaluation, listed in Table 1. *Black-Scholes* is a parallel implementation of a mathematical model for price variations in financial markets with derivative investment instruments, taken from the PARSEC [13] benchmark suite. *Ferret* is a content-based similarity search engine toolkit for feature rich data types (video, audio, images, 3D shapes, etc), from the PARSEC benchmark suite. *Cholesky* is a factorization kernel used to solve normal equations in linear least squares problems. *GMRES* is an implementation of the iterative Generalized Minimal Residual method for solving systems of linear equations. *HPL* solves a random dense linear system in double precision arithmetic. *Jacobi* is a parallel implementation of the Jacobian method for solving systems of linear equations. *FFT* is a kernel implementing a 2-dimensional Fourier algorithm, taken from the SPLASH-2 [14] benchmark suite. It is implemented in alternating transpose and computation phases. On each transpose the data gets reordered, creating irregular dependencies between the two phases. *Multisort* is a parallel implementation of Mergesort. Multisort is an alternative implementation of the Cilksort test from Cilk [1]. It has two phases: during the first phase, it divides the data into chunks and sorts each chunk. During the second phase, it merges those chunks. *Intruder* is a Signature-based network intrusion detection systems (NIDS), from the STAMP benchmark suite [15]. It processes network packets in parallel in three phases: capture, reassembly, and detection. The reassembly phase uses a dictionary that contains linked lists of packets that belong to the same session. The lists are allocated using

Table 2. Impact of the analysis on performance

Benchmark	Task Instances	Dynamic (ms)	Standard Deviation	Static & Dynamic (ms)	Standard Deviation	Speedup
Black-Scholes	234375	1618	5.83 %	963	4.51 %	1.68
Ferret	1000	3344	1.10 %	3344	1.33 %	1.00
Cholesky	45760	983	0.23 %	981	0.32 %	1.00
GMRES	5170	16640	4.42 %	13947	2.65 %	1.19
HPL	28480	1628	0.44 %	1574	0.37 %	1.03
Jacobi	204800	11499	0.39 %	11588	0.53 %	0.99
FFT	28864	2028	0.10 %	1849	0.29 %	1.10
Multisort	11264	3683	15.77 %	3446	15.15 %	1.07
Intruder	> 4M	12572	1.23 %	10332	1.76 %	1.22

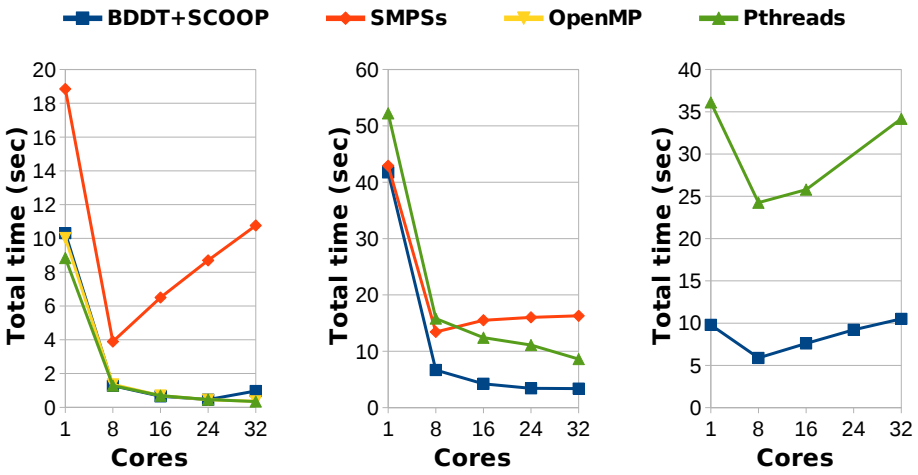
dynamic regions. Intruder issues a new task for each packet it receives. When a task reassembles the last packet of a session it also executes the detection algorithm.

The second column (*LOC*) of Table 1 shows the size of each benchmark in lines of code[1]. The third column (*Tasks*) shows the number of task invocations in the code. The fourth column (*Total Args*) shows the total number of arguments of all task invocations. We report the total number of arguments as each such argument incurs the additional overhead of a runtime dependency check. The fifth column (*Scalar Args*) shows how many of those arguments are scalars passed by value, since it is trivial for either the programmer or the analysis to find them, thus we do not count them as *independent* arguments discovered by the static analysis.

The last three columns of Table 1 show the performance and precision of the static analysis. Namely, the sixth column (*Analysis*) shows the total running time of the static dependence analysis in seconds. The seventh column (*Graph Nodes*) shows the number of nodes in the constraint graph. The last column (*Safe Args*) shows the number of independent task arguments inferred by the analysis. Note that Ferret, the largest benchmark, creates the largest constraint graph, causing an analysis time of over 11 minutes. This is because the context sensitive analysis has cubic complexity in the size of the constraint graph.

Table 2 shows the effect of the optimization on the total running time of all benchmarks, on 32 cores. Specifically, the second column (*Task Instances*) shows the total number of task spawns by each benchmark during execution. The third column (*Dynamic*) shows the total running time in milliseconds for each benchmark, without using the static analysis. Here, we use BDDT to perform runtime dependence analysis on *all* the non-scalar task arguments of all tasks. The fourth column (*Standard Deviation*) shows the standard deviation of the *Dynamic* total running time for twenty runs. The fifth column (*Static & Dynamic*) shows the total running time in milliseconds for each benchmark compiled with SCOOP, where the runtime dependence analysis is disabled for any arguments found safe by the static analysis. The sixth column (*Standard Deviation*) shows the standard deviation of the *Static & Dynamic* total running time. Finally,

[1] We count lines of code not including comments after merging all program sources in one file.



(a) Black-Scholes **(b)** Ferret **(c)** Intruder

Fig. 4. Comparison with alternative runtimes

the last column shows the speedup factor gained by removing redundant checks for arguments found independent by the static analysis, compared to always checking arguments dynamically. Note that even though the static analysis does not infer independent task arguments in Multisort and HPL, we observe a speedup of 1.07 and 1.03 respectively when compiling with SCOOP compared to the dynamic-only execution. This happens because SCOOP generates code to interface directly with the BDDT runtime internals, whereas the BDDT API may perform various checks, e.g., on scalar arguments. We consider the 0.99 speedup (slowdown) in Jacobi to be well within the noise due to cache effects, other processes executing, etc., as seen by the deviation observed among twenty runs.

The dependence analysis is able to infer safe task arguments only in Black-Scholes, Ferret, GMRES, FFT and Intruder. In these benchmarks, inferring independent arguments has a large impact on the overhead and scalability of the dependence analysis, producing substantial speedup over the original BDDT versions for four benchmarks. The reduction of dependence analysis overhead is not noticeable in Ferret because the tasks are very coarse grain. On the rest of the benchmarks (Cholesky, HPL, Jacobi and Multisort) the dependence analysis fails to find any safe arguments. We examined all benchmarks manually and found that there are no safe arguments.

For reference, we compare the three largest benchmarks with related parallel runtimes. We have ported each benchmark to all runtimes so that they are as comparable as possible and express the same parallelism. Figure 4 shows the results. Specifically, Figure 4a compares four parallel implementations of Black-Scholes: the original Pthreads and OpenMP implementations from the PARSEC benchmark suite, as well as two ports of the OpenMP version into SMPSs and BDDT, using SCOOP. Note that SCOOP finds and removes the redundant dependency checks. As Black-Scholes is a data-parallel application, this removes most of the runtime overhead and matches in performance the

fine-tuned Pthreads and OpenMP implementations. In comparison, the SMPSs runtime scales up to 8 cores, mainly due to the overhead caused by redundant checks on independent task arguments.

Figure 4b compares the original Pthreads implementation of Ferret with the SMPSs and BDDT runtime. The Pthread version uses one thread to run each computation phase, causing load imbalance. In comparison, BDDT and SMPSs perform dynamic task scheduling that hides load imbalance and distributes computation to processors more evenly. Although SCOOP detects and removes redundant runtime checks, Ferret tasks are computationally heavy and coarse-grain, hiding the effect of the optimization. The difference in performance between SMPSs and BDDT is mainly due to constant-factor overheads in task scheduling.

Figure 4c compares the original Pthreads implementation of Intruder from the STAMP benchmark, with a port for BDDT. The STAMP implementation uses software transactional memory to synchronize threads, which causes high contention effects above 8 cores, limiting performance. In comparison, BDDT incurs lower overheads and uses pessimistic synchronization that also removes the cost of rollbacks.

6 Related Work

Task Parallelism: There are several programming models and runtime systems that support task parallelism. Most, like OpenMP [2], Thread Building Blocks [16], Cilk [1], and Sequoia [3], use tasks to express recursive or data parallelism, but require manual synchronization in the presence of task dependencies. That usually forces programmers to use locks, barriers, or other synchronization techniques that are not point-to-point, and result in loss of parallelism even among task instances that do not actually access the same memory.

Some programming models and languages aim to automatically infer synchronization among parallel sections of code. Transactional Memory [17] preserves the atomicity of parallel tasks, or transactions, by detecting and retrying any conflicting code. Static lock allocation [18] provides the same serializability guarantees by automatically inferring locks for atomic sections of code. These attempts, however, allow non-deterministic parallel executions, as they only enforce race freedom or serializability, not ordering constraints among parallel tasks.

Jade [19] is a parallel language that extends C with parallel coarse-grain tasks. Similarly, StarSs, SMPSs and OpenMP-Ss [9, 20] are task-based programming models for scientific computations in C that use annotations on task arguments to dynamically detect argument dependencies between tasks. All of these runtimes could benefit from independencies discovered by SCOOP to reduce the overhead of runtime checks.

Static analysis has been used in combination with dynamic analysis in parallel programs in the past. SvS [7] uses static analysis to determine possible argument dependencies among tasks and drive a runtime-analysis that computes task dependencies with overlapping approximate footprints. SvS assumes all tasks to be commutative and does not preserve the original program order as SCOOP and BDDT. Prabhu et al. [21] define sets of commutative tasks in parallel programs. The compiler uses this information to

allow more possible orderings in a program and extract parallelism. As with all compiler-only parallelization techniques, this approach is limited by over-approximation in static pointer and control flow analyses that might cause many tasks to be run sequentially, because only two instances have clearly disjoint memory footprints. To avoid this, CommSets uses optimistic transactional memory, which is not suitable for programs with high contention or effects that cannot be rolled back.

Deterministic parallelism: Recent research has developed methods for the deterministic execution of parallel programs. Kendo [22] enforces a deterministic execution for race-free programs by fixing the lock-acquisition order, using performance counters. Grace [23] produces deterministic executions of multithreaded programs by using process memory isolation and a strict sequential-order commit protocol to control thread interactions through shared memory. DMP [24] uses a combination of hardware ownership tracking and transactional memory to detect thread interactions through memory. Both systems produce deterministic executions, even though they may not be equivalent to the sequential program. Instead, they enforce the appearance of the same arbitrary interleaving across all executions.

Out-of-Order Java [5] and Deterministic Parallel Java [6], task-parallel extensions of Java. They use a combination of data-flow, type-based, region and effect analyses to statically detect or check the task footprints and dependencies in Java programs. OoOJava then enforces mutual exclusion of tasks that may conflict at run time; DPJ restricts execution to the deterministic sequential program order using transactional memory to roll back tasks in case of conflict. As task footprints are inferred (OoOJava) or checked (DPJ) statically in terms of objects or regions, these techniques require a type-safe language and cannot be directly applied on C programs with pointer arithmetic and tiled array accesses.

Chimera [25] proposes a hybrid system that detects and transforms races, so that the runtime system can then enforce deterministic execution.

Static and Dynamic Dependence Analysis: Static dependence analysis is often employed in compilers and tools that optimize existing parallel programs or for automatic parallelization. Early parallelizing compilers used loop dependence analysis to detect data parallelism in loops operating on arrays [12, 26], and even dynamic dependence analysis to automatically synchronize loops [27] These systems, however do not handle inter-loop dependencies and do not work well in the presence of pointers. Recently, Holewinski et al. [28] use dynamic analysis of the dynamic dependence graph of sequential execution to detect SIMD parallelism.

Several pointer analyses have been used to detect dependencies and interactions in parallel programs. Naik and Aiken [29] extend pointer analysis with *must-not-alias* analysis to detect memory accesses that cannot lead to data races. Pratikakis et al. [30, 31] use a context-sensitive pointer and effect analysis to detect memory locations accessed by many threads in the Locksmith race detector. SCOOP uses the pointer analysis in Locksmith to detect aliasing between task footprints, and extends Locksmith's flow-sensitive dataflow analysis to detect tasks that can run in parallel.

7 Conclusions

This paper presents SCOOP, a compiler for a task-parallel extension of C with implicit synchronization. SCOOP targets task parallel runtimes such as BDDT and OpenMP-Task, that use dynamic dependence analysis to automatically synchronize and schedule parallel tasks. SCOOP uses static analysis to infer safe task arguments and reduce the runtime overhead for detecting dependencies. We have tested SCOOP using the BDDT runtime system on a set of parallel benchmarks, where it finds and removes unnecessary runtime checks on task arguments. Overall, we believe that task dependence analysis is an important direction in parallel programming abstractions, and that using static analysis to reduce its overheads is a major step in its practical application.

Acknowledgements. The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7), under the ENCORE Project in Computing Systems (www.encore-project.eu), grant agreement N° 248647, and the HiPEAC Network of Excellence (www.hipeac.net), grant agreement N° 287759.

References

1. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. In: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 207–216. ACM, New York (1995)
2. Dagum, L., Menon, R.: OpenMP: An industry-standard api for shared-memory programming. IEEE Comput. Sci. Eng. 5(1), 46–55 (1998)
3. Fatahalian, K., Horn, D.R., Knight, T.J., Leem, L., Houston, M., Park, J.Y., Erez, M., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: programming the memory hierarchy. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. ACM, New York (2006)
4. Pop, A., Cohen, A.: OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. ACM Transactions on Architecture and Code Optimization 9(4), 53:1–53:25 (2013)
5. Jenista, J.C., Eom, Y.H., Demsky, B.C.: OoOJava: software out-of-order execution. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, pp. 57–68. ACM, New York (2011)
6. Bocchino Jr., R.L., Heumann, S., Honarmand, N., Adve, S.V., Adve, V.S., Welc, A., Shpeisman, T.: Safe nondeterminism in a deterministic-by-default parallel language. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 535–548. ACM, New York (2011)
7. Best, M.J., Mottishaw, S., Mustard, C., Roth, M., Fedorova, A., Brownsword, A.: Synchronization via scheduling: techniques for efficiently managing shared state. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 640–652. ACM, New York (2011)
8. Tzenakis, G., Papatriantafyllou, A., Vandierendonck, H., Pratikakis, P., Nikolopoulos, D.S.: BDDT: Block-level dynamic dependence analysis for task-based parallelism. In: International Conference on Advanced Parallel Processing Technology (2013)
9. Perez, J.M., Badia, R.M., Labarta, J.: Handling task dependencies under strided and aliased references. In: Proceedings of the 24th ACM International Conference on Supercomputing, pp. 263–274. ACM, New York (2010)

10. Rehof, J., Fähndrich, M.: Type-base flow analysis: from polymorphic subtyping to cfl-reachability. In: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 54–66. ACM, New York (2001)
11. Wolf, M.E.: Improving locality and parallelism in nested loops. PhD thesis, Stanford University, Stanford, CA, USA (1992) UMI Order No. GAX93-02340
12. Maydan, D.E., Hennessy, J.L., Lam, M.S.: Efficient and exact data dependence analysis. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, pp. 1–14. ACM, New York (1991)
13. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The parsec benchmark suite: characterization and architectural implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 72–81. ACM, New York (2008)
14. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: Characterization and methodological considerations. In: International Symposium on Computer Architecture, pp. 24–36. ACM (1995)
15. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IEEE International Symposium on Workload Characterization, pp. 35–46. IEEE (2008)
16. Reinders, J.: Intel threading building blocks, 1st edn. O'Reilly & Associates, Inc., Sebastopol (2007)
17. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 289–300. ACM, New York (1993)
18. Cherem, S., Chilimbi, T., Gulwani, S.: Inferring locks for atomic sections. In: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 304–315. ACM, New York (2008)
19. Rinard, M.C., Lam, M.S.: The design, implementation, and evaluation of Jade. ACM Transactions on Programming Languages and Systems 20(3), 483–545 (1998)
20. Planas, J., Badia, R.M., Ayguadé, E., Labarta, J.: Hierarchical task-based programming with StarSs. International Journal of High Performance Computing Applications 23(3), 284–299 (2009)
21. Prabhu, P., Ghosh, S., Zhang, Y., Johnson, N.P., August, D.I.: Commutative set: a language extension for implicit parallel programming. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 1–11. ACM, New York (2011)
22. Olszewski, M., Ansel, J., Amarasinghe, S.: Kendo: efficient deterministic multithreading in software. In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 97–108. ACM, New York (2009)
23. Berger, E.D., Yang, T., Liu, T., Novark, G.: Grace: safe multithreaded programming for c/c++. In: Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, pp. 81–96. ACM, New York (2009)
24. Devietti, J., Nelson, J., Bergan, T., Ceze, L., Grossman, D.: RCDC: a relaxed consistency deterministic computer. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 67–78. ACM, New York (2011)
25. Lee, D., Chen, P.M., Flinn, J., Narayanasamy, S.: Chimera: hybrid program analysis for determinism. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 463–474. ACM, New York (2012)
26. Pugh, W.: The omega test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM 8, 4–13 (1992)

27. Rauchwerger, L., Padua, D.: The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. In: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, pp. 218–232. ACM, New York (1995)
28. Holewinski, J., Ramamurthi, R., Ravishankar, M., Fauzia, N., Pouchet, L.N., Rountev, A., Sadayappan, P.: Dynamic trace-based analysis of vectorization potential of applications. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 371–382. ACM, New York (2012)
29. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 327–338. ACM, New York (2007)
30. Pratikakis, P., Foster, J.S., Hicks, M.: Locksmith: Practical static race detection for c. ACM Transactions on Programming Languages and Systems 33(1), 3:1–3:55 (2011)
31. Pratikakis, P., Foster, J.S., Hicks, M.: Locksmith: context-sensitive correlation analysis for race detection. In: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 320–331. ACM, New York (2006)

BDDT: Block-Level Dynamic Dependence Analysis for Task-Based Parallelism

George Tzenakis[1], Angelos Papatriantafyllou[3], Hans Vandierendonck[1], Polyvios Pratikakis[2], and Dimitrios S. Nikolopoulos[1]

[1] Queen's University of Belfast, Belfast, United Kingdom
{gtzenakis01,h.vandierendonck,d.nikolopoulos}@qub.ac.uk
[2] FORTH-ICS, Heraklion, Crete, Greece
polyvios@ics.forth.gr
[3] TU Wien, Vienna, Austria
papatriantafyllou@par.tuwien.ac.at

Abstract. We present BDDT, a task-parallel runtime system that dynamically discovers and resolves dependencies among parallel tasks. BDDT allows the programmer to specify detailed task footprints on any memory address range, multidimensional array tile or dynamic region. BDDT uses a block-based dependence analysis with arbitrary granularity. The analysis is applicable to existing C programs without having to restructure object or array allocation, and provides flexibility in array layouts and tile dimensions.

We evaluate BDDT using a representative set of benchmarks, and we compare it to SMPSs (the equivalent runtime system in StarSs) and OpenMP. BDDT performs comparable to or better than SMPSs and is able to cope with task granularity as much as one order of magnitude finer than SMPSs. Compared to OpenMP, BDDT performs up to 3.9× better for benchmarks that benefit from dynamic dependence analysis. BDDT provides additional data annotations to bypass dependence analysis. Using these annotations, BDDT outperforms OpenMP also in benchmarks where dependence analysis does not discover additional parallelism, thanks to a more efficient implementation of the runtime system.

Keywords: Compilers and runtime systems, Task-parallel libraries, Middleware for parallel systems, Synchronization and concurrency control.

1 Introduction

Task-parallel programming models [2,8,11] offer a more abstract, more structured way for expressing parallelism than threads. In these systems the programmer only describes the parts of the program that can be computed in parallel, and does not have to manually create and manage the threads of execution. This lifts a lot of the difficulty in describing parallel, independent computations compared to the threading model, but still requires the programmer to manually find and enforce any ordering or memory dependencies among tasks. Moreover, these models maintain the inherent nondeterminism found in threads, which makes them hard to test and debug, as some executions may not be easy to reproduce.

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 17–31, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Programming models with implicit parallelism [4,10,13,14] extend task-parallel programming models with automatic inference of dependencies, requiring the programmer to only describe the memory resources required in each task. They are easier to use, as programmers need not discover and describe parallelism —which might be unstructured and dynamic— but can instead annotate the program using compiler directives [10,14] or language extensions [4,6]; the compiler and runtime system then discover parallelization and manage dependencies transparently.

Dynamic dependence analysis can discover more parallelism than possible to describe statically in the program, as it only synchronizes tasks that actually (not potentially) access the same resources. In order for a dynamic dependence analysis to benefit program performance, it must (i) be accurate, so that it does not discover false dependencies; and (ii) have low overhead, so that it does not nullify the benefit of discovering extra parallelism. Most existing such systems require the programmer to restrict task footprints into either whole and isolated program objects, one-dimensional array ranges, or static compile-time regions. This may cause false dependencies in programs where tasks have partially overlapping or unstructured (irregular) memory footprints, or disallow tasks that operate on a multidimensional tile of a large array or on dynamic linked data structures.

SMPSs [12], a state-of-the-art implementation of the StarSs programming model for shared-memory multicores with implicit parallelism, supports non-contiguous array tiles and non-unit strides in task arguments. This is, however, at the cost of reduced parallelism due to overapproximation of memory address ranges and high overhead for maintaining a complex data structure used to discover partial overlaps.

This paper presents BDDT, a task-parallel runtime system that dynamically discovers and resolves dependencies deterministically among parallel tasks, producing executions equivalent to a sequential program execution. BDDT supports a provably deterministic task-based programming model [15]. Lifting the above restrictions of existing systems, BDDT allows the programmer to specify detailed task footprints on any, potentially non-contiguous, memory address range, multidimensional array tile, or dynamic region. To allow this, we use a block-based dependence analysis with arbitrary granularity, making it easier to apply to existing C programs without having to restructure object or array allocation, introduce buffers and marshaling, or change the granularity of task arguments.

Overall, this paper makes the following contributions:

- We present a novel technique for block-based, dynamic task dependence analysis that allows task arguments spanning arbitrary memory ranges, partial argument overlapping across tasks, dependence tracking at configurable granularity, and dynamic memory management in tasks. The analysis is tunable to balance accuracy and performance.
- We implement this dependence analysis in BDDT, a runtime system for scheduling parallel tasks. Our implementation is adaptive, the programmer can enable or disable the dependence analysis for each task argument independently to minimize overhead when the analysis is not necessary.
- We evaluate the performance of our runtime system. On a representative set of benchmarks, BDDT performs comparable to or better than SMPSs and handle

arbitrary tile sizes and array dimensions. In several benchmarks, dynamic dependence analysis in BDDT discovers additional parallelism, producing speedups of up to 3.9× compared to OpenMP using barriers. BDDT outperforms OpenMP on embarrassingly parallel tasks without dependencies, by using hand-added annotations to disable the dependence analysis.

2 Dataflow Execution Engine Design

BDDT uses a dataflow execution engine based on block-level dependence analysis for identifying parallel tasks. Task arguments are annotated —at task-issue time— with data access attributes, corresponding to three access patterns: read (**`in`**), write (**`out`**) and read/write (**`inout`**). The runtime system detects dependencies between tasks by comparing the access properties of arguments of different tasks that overlap in memory. To do that, BDDT splits arguments into virtual memory blocks of configurable size and analyzes dependencies between blocks. Similarly to whole-object dependence analysis used in tools such as SMPSs and SvS, block-based analysis detects true (RAW) or anti- (WAW, WAR) dependencies between blocks by comparing block starting addresses and checking their access attributes. Block-based analysis can also detect dependencies between tasks that whole object analysis does not: Partially overlapping arguments are dependencies if the overlapping part is written by at least one task. Furthermore, tasks can have arguments that are non-contiguous in memory, such as a tile of a multidimensional array or a collection of objects in random memory locations.

There are two potential drawbacks to block-based dependence analysis. First, as the dependence analysis is performed per block, the runtime system must sometimes repeat the same action across all blocks in an argument, increasing overhead. In contrast, whole-object dependence tracking must perform each action only once per argument. Second, false positives may occur when data structures are not properly laid out or when the block size is too large. BDDT overcomes both problems. A custom memory allocator integrates the metadata with the application data to eliminate the overhead of metadata lookup, and allows the sharing of metadata between blocks. Moreover, BDDT allows the user to adjust block granularity, to be coarse enough to amortize overhead, yet fine enough to avoid false positives. In our experience, selecting an appropriate block size is quite straightforward.

Each task in the program goes through four stages: *task issue* performs dependence analysis, queuing the task if any pending dependencies are unresolved; *task scheduling* releases a task for execution when all its dependencies are resolved, selects a worker's queue and inserts the task; *task execution* executes it; and *task release* resolves pending dependencies of an executed task, potentially releasing new tasks for execution. The dynamic dependence analysis induces overheads in the issue and release stages, for checking dependencies and task wakeup, respectively. We design the data structures used in the dependence analysis specifically to minimize these overheads.

Retrieving the metadata that track dependencies for each byte, object, or block of memory accessed by a task can be expensive. A general solution to this is to maintain a hash from memory addresses to metadata [12], although this incurs a large overhead per access. A faster way is to attach the metadata directly to the actual data payload [1,18],

but this may make metadata visible to the programmer, require significant additional changes to the program source and memory layout. We achieve the best of both solutions using a custom memory allocator that allows for fast lookup of metadata, while still hiding metadata management in the runtime system.

The dependence analysis on blocks is quite similar to dependence tracking on whole objects. There can be, however, extra overhead, as a task argument may consist of multiple blocks, and dependencies must be tracked on each such block. BDDT allows multiple blocks to share the same metadata information. Then, critical dependence tracking operations operate on one metadata element instead of multiple, reducing the overhead of dependence tracking. We use this mechanism in particular to track dependencies on strided arguments—usually multidimensional array tiles: while dependencies are tracked on each block individually, the runtime system registers a single metadata element for all the blocks in the tile.

To detect task dependencies, we also allow multiple metadata elements to describe the same block, capturing the task order. Specifically, each written task argument (**`out`** or **`inout`** footprint) creates a new metadata element to describe the argument blocks, and each read (**`in`**) task argument creates one or more metadata element to describe the argument blocks. Read arguments may result in multiple metadata elements, if the relevant blocks were described by more than one metadata elements (fragmentation) before the new task is created; this captures the scenario of a consumer task waiting for multiple producers. This design allows for an efficient dependence analysis while limiting the complexity of accessing and updating the same data structures for every block.

Using solely memory address ranges to describe the memory footprint of a task restricts tasks to finite footprints. Moreover, it prohibits tasks from allocating memory, as the new range is yet unknown at task-invocation time, and thus cannot be part of the footprint. To address these issues and allow tasks to operate on dynamic data structures, we also use a region-based allocator [17]. A dynamic region (or zone) is an isolated heap in which objects can be dynamically allocated. A BDDT task footprint can include dynamic regions, meaning that all memory allocated in a region is in the task's footprint, without having to enumerate the actual ranges. Moreover, a task can then allocate new memory inside a region in its footprint (when it has an `out` or `inout` effect). Dynamic regions are directly linked in the dependence analysis by allocating exactly one metadata element per region, and treating it as exactly one memory location.

3 Implementation

BDDT constructs a task dependence graph dynamically, by deducing task dependencies from the access modes and blocks in task footprints. We design the system so that identifying and retrieving dependent tasks causes minimal overhead.

The runtime system consists of (i) a custom block-based memory allocator; (ii) metadata structures for dependence analysis; and (iii) a task scheduler. The two metadata types include (i) task elements; and (ii) block elements. The memory allocator is designed to facilitate fast lookup of block elements that model the outstanding and executing tasks operating on these blocks, and to coalesce runtime system operations on

blocks that are used in the same way, e.g., consecutive blocks forming a single task argument. This key optimization in the design in the runtime reduces redundancy and saves both time and space.

3.1 Task Elements

Each task metadata element represents a dynamic instance of a task. Task elements contain all the essential metadata that is necessary to execute a task, including the closure: a function pointer, the number of arguments, and the address, size and access attributes for each argument. BDDT supports strided arguments specified as a base address, the size and number of elements, and the stride (in bytes) between consecutive elements. Arguments consisting of multiple contiguous blocks are also considered as strided arguments by setting the stride equal to the block size.

Each task element contains a list of dependent tasks: tasks that wait for this task to finish. New task elements are appended to this list whenever a new dependent task is issued. The dependent task list is also used during task release to check whether any of the dependent tasks becomes ready to execute. We implement this check using an atomically updated join counter, which tracks the number of task arguments that are not yet ready. As a task finishes execution and is released, it reduces the counters of all task elements in its dependent list.

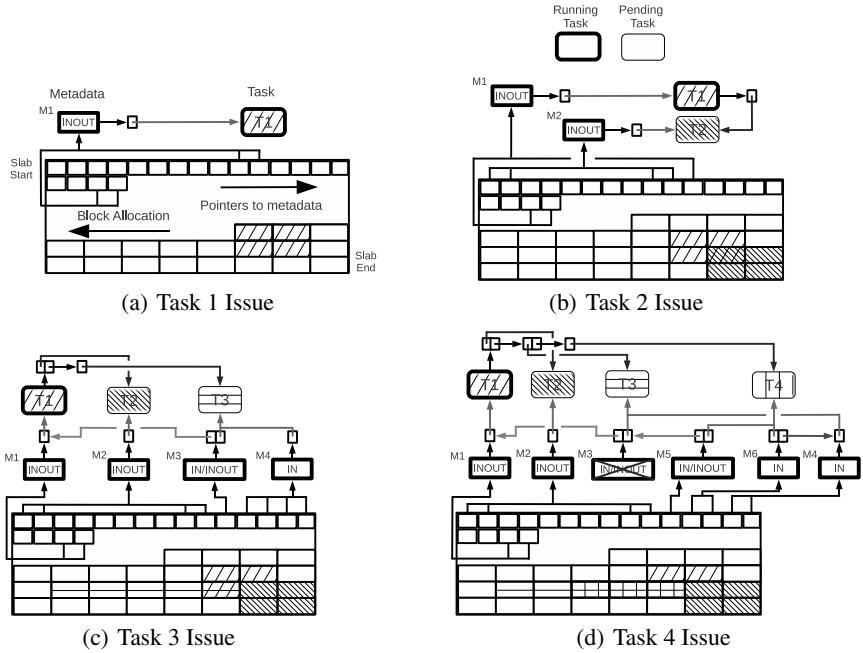
3.2 Block Elements

Block metadata elements capture the running and outstanding tasks that operate on a particular collection of data blocks. They facilitate the construction of the task graph as dependencies between tasks are derived from the data blocks that they access. A BDDT block metadata element may represent a collection of blocks; in contrast, systems implementing object-based dependence tracking would use one block metadata element per object. Conversely, a collection of blocks may correspond to multiple block elements describing operations on that collection. A new block metadata element is allocated for each task with a write effect (i.e., **`out`** or **`inout`**). These block elements are strictly ordered from the youngest (most recently issued) tasks to the oldest (least recently issued) tasks. Moreover, every block metadata element contains a list of task elements that take the corresponding collection of blocks as a task argument. This list is used at task issue to link the newly issued task on the dependent task list of older tasks that have overlapping arguments.

Block metadata elements conceptually include information about the access mode of the collection (**`in`**, **`inout`**, **`out`**). In fact, for space optimization, there can be up to two access modes per block metadata element: an **`in`** mode, followed by an **`out`** or **`inout`** mode. All task elements with **`in`** effect in the task list of a block metadata element store an additional pointer to the first non-**`in`** task in the list. The reason for this is that, in the common case, a series of tasks with **`in`** mode is always followed by a single task with **`out`** or **`inout`** mode. By construction, the metadata elements reveal the parallelism between tasks: tasks listed in the same block metadata element may execute in parallel, while tasks in a younger block element must wait until all tasks in an older block element have finished execution.

3.3 Memory Allocator

BDDT uses a custom memory allocator to embed dependence analysis metadata in the allocator's metadata structures[1]. The allocator partitions the virtual address space in *slabs* and services memory allocation requests from such slabs. Memory allocators typically manage multiple slabs and allocate chunks of the same size in the same slab. BDDT divides the slabs in blocks of configurable but fixed size. For every data block in a slab, there is also room provisioned to store a pointer to index the metadata elements, as discussed in Section 3.2.



(a) Task 1 Issue (b) Task 2 Issue

(c) Task 3 Issue (d) Task 4 Issue

Fig. 1. Memory allocation and dependence analysis metadata

Figure 1(a) shows the structure of such a slab. While data blocks are allocated starting from one end of the slab, pointers to the metadata for these blocks are allocated starting from the opposite end of the slab. Thus, there may be fragmented (unusable) memory in the slab, the amount of which is bounded by the block size. All shared memory and metadata is bulk-deallocated upon completion of all tasks. As such, BDDT does not need special handling for fragmentation. Moreover, by using slabs of fixed size and alignment, we can calculate the address of a block's metadata through very efficient

[1] Several parallel runtime systems implement custom memory allocators for performance reasons, e.g. Cilk++ and Intel TBB. This is not a limitation of the usability of the programming model.

integer arithmetic on the block address. This also increases locality, as the metadata pointers of consecutive blocks are located at neighboring addresses.

The metadata pointers stored in the slab implement collections of blocks: a collection of blocks is a group of blocks that are operated on by the same task and will be available as a task argument together. We optimize dependence tracking by mapping all blocks in a collection to the same metadata elements. BDDT implements merging of collections of blocks simply by assigning a pointer to the same metadata element to all blocks, and splits collections of blocks by assigning a new pointer to a subset of the blocks.

For example, Figure 1(a) shows the state of memory after issuing task T1, which accesses 4 different blocks as an **`inout`** strided argument. When issuing T1, BDDT registers one metadata element (M1) for the four blocks and sets the slab pointers of the blocks to M1. In addition, T1's task element is inserted in M1's linked list of tasks. In the state shown, task T1 is executing or pending to execute.

3.4 Task Issue

During task issue, BDDT identifies dependencies between the new task and older tasks by scanning all data blocks in the task arguments and analyzing the corresponding metadata elements. Note that blocks operated on in the same way are mapped to the same metadata element. As such, a task with a large memory footprint may still require only a few of the following actions. Depending on the access mode (**`in`**, **`out`**, or **`inout`**), and any outstanding tasks that access the same data, BDDT either immediately schedules the task, or stores it for later scheduling. If the task touches a memory block for the first time, BDDT creates an empty block metadata element for each collection of blocks with the same access mode.

*Handling **`in`** arguments:* If the most recent block metadata element contains writer tasks, then BDDT iterates through the metadata's list and registers the new task in the list of dependent tasks of all the linked task elements. It also increments the join counter by one in every task element it finds. Next, BDDT creates a new metadata element in the youngest position of the metadata element list for the current collection of blocks, and adds the new task to the new metadata element's task list. Alternatively, if the most recent metadata element contains only reader tasks, then the new task element is simply added to its task list. Note that the operations on the metadata elements are performed only once for all blocks sharing the same metadata elements, i.e., they have equal pointers in the memory allocators slab at the start of task issue. The equality of slab pointers is maintained after task issue for all blocks accessed by the new task. If the collection contains blocks that are not accessed by the new task, then their slab pointers are not updated. This results in a split of the collection.

*Handling **`inout`** and **`out`** arguments:* Such arguments similarly benefit from the optimization of operating on collections of blocks that have the same slab pointer. If the most recent metadata element contains writer tasks, then BDDT iterates through the metadata's task list and adds the new task to the dependent list of all the task elements. It also increments the join counter by one for every task element on the list, creates a

new metadata element, and inserts the new task in its task list. If the most recent metadata element contains readers but no writers, BDDT again adds a new metadata element. This is necessary because all blocks in the collection are mapped to this new metadata element. The new task is again inserted in the task list in the metadata element, and in the dependent task lists of each task in the previous metadata element.

Collections of blocks are merged when blocks with different metadata elements are passed as part of the same **`out`** or **`inout`** argument. In this case, a new metadata element is added and the slab pointers for each block are set to point to the new metadata element. This results in a merge of blocks in a collection and accelerates future dependence analysis.

As an example, in Figure 1(b), assume that while task T1 is running, the program spawns new tasks T2. Task T2 reads and writes four blocks in **`inout`** mode, where one block overlaps with the footprint of T1. To issue T2, BDDT creates a new metadata element (M2), and iterates through the linked list of M1 to place T2 in T1's dependence list. T2 becomes the first node in the linked list of M2. Finally, BDDT alters all slab-pointers corresponding to the blocks in T2's footprint (including that of the overlapping block) to point to M2.

Continuing the example, Figure 1(c) shows the issue of task T3. Task T3 reads five contiguous blocks in **`in`** mode. These blocks partially overlap with the memory footprint of T1. Two new metadata elements are created: M3 that models accesses to the block accessed by both T1 and T3, and M4 that models accesses to the remaining blocks. The slab pointers are updated accordingly, splitting the collection of blocks accessed by T1 to reflect different subsequent usage. Task T3 is linked in the dependent tasks list of T1.

Finally, Figure 1(d) shows the issue of task T4. T4 reads 3 contiguous blocks in **`in`** mode. This argument overlaps with the T1/T3 footprint intersection (M3) and it partially overlaps with the collection of blocks that is accessed uniquely by T3 (M4). Consequently, two new metadata elements are created. M5 complements the M3 metadata element while M6 models accesses to part of M4. M4 persists and models the blocks accessed by T3 but not by T4. T4 is inserted in the list of dependent tasks of T1 because it has a dependence with T1.

Note that metadata elements are recycled when they are no longer used: when the last slab pointer to a metadata element is removed, the metadata element is freed, as is the case for M3 in the example. Note also that, in total, 11 blocks are accessed, but due to the coalescing of metadata elements between blocks that are accessed in the same way, only 6 metadata elements are allocated.

3.5 Task Release and Scheduling

BDDT is based on a master-worker program model. The master is responsible for task issue and dependence analysis. The workers concurrently perform task scheduling, execution and release. The master can also operate as a worker, as discussed below. On task completion, the finished task walks through its dependence list and decrements by one the dependence counter of every dependent task. Tasks with no pending dependencies are pushed for execution.

BDDT schedules a task for execution whenever all its dependencies are satisfied. Each worker thread has its own queue of ready tasks. Queues have finite length and are implemented efficiently, as concurrent arrays. The master thread has its own task queue and can operate as a worker when the queues of all workers are full.

The master issues ready tasks to worker queues round-robin. Workers issue tasks to their own task queues to preserve memory locality. If a worker's task queue becomes full, the worker issues tasks to task queues of other workers round-robin. In case there is no empty slot in any task queue, the task is executed synchronously by the issuing thread. Any thread can steal tasks from any other thread's task queue in case its own task queue is empty.

Task queues are allocated in a NUMA aware, first-touch policy. NUMA aware allocation is important to reduce remote memory accesses inside the critical path of the worker thread. Ready task queues support lock-free dequeuing with the utilization of atomic primitives. A bit vector indicates the free slots in the queue. We use the atomic "*bit scan forward*" and "*bit test and set/reset*" instructions of x86 to manipulate this vector. The queue allows any number of dequeue operations and up to one enqueue operation to occur concurrently. Enqueue operations must therefore be mutually exclusive by means of a spin-lock. Each task queue has a fixed size of 32 slots which is imposed by the atomic primitives used to implement the task queues.

3.6 Complexity Analysis and Discussion

BDDT incurs overhead for task issue and task release. Task release overhead depends on the shape of the task graph: The runtime system receives a finished task from the scheduler and inspects its dependent task list to locate ready tasks. We assume that the average out-degree in the task graph is d_{out}, at least in the part of the task graph that is dynamically generated. Task release then takes O(d_{out}) operations. Note that BDDT shares metadata elements between blocks in the same collection, so the dependent task list is scanned only once per collection. For the remaining blocks, only the slab pointers may have to be updated. Assuming an average collection size of C blocks and N blocks per task, then task release takes O($d_{out}\frac{N}{C}$) operations on average. In practice, sharing of metadata elements reduces task release overhead by more than 50% for arguments having more than 64 blocks.

Task issue has similar complexity. If prior producers of a block are still executing, then the runtime system locates the metadata of a block with a single operation on the bits of the block address in O(1) time and a new metadata element is created. The issued task is linked to all tasks in the last-issued task list of the prior metadata element, taking O(d_{in}) operations assuming an average in-degree d_{in} in the task graph. Furthermore, the slab pointers of all blocks in a collection are updated. In total, task issue takes O($d_{in}\frac{N}{C}$) operations on average. Note that the overhead of merging and splitting collections is included in the presented formulas as they are realized by setting the slab pointers.

To put the overheads in perspective, we compare BDDT to SMPSs [12]. The latest version of SMPSs that we use as a comparison has less functionality than BDDT: it handles only multi-dimensional array tiles, encoded with a binary representation, thus disallowing arbitrary pointer arithmetic. The representation is approximate and subject

to aliasing and alignment constraints, which restricts the acceptable tile and array sizes to powers of two and is prone to false positives.

Dependence detection in SMPSs requires encoding of tiles in their binary representation, taking a number of operations proportional to the number of bits in an address. SMPSs walks a tree data structure to detect overlap with other tiles, updates the tree by adding the tile or updates the metadata of an already existing tile. These operations take O($d_{in}\frac{N}{R}\log T$), where R is the average tile size expressed in blocks and T is the number of tiles in the tree. The tile size may be less than the argument size N because tiles must be split to eliminate false positives, with $R = O(N)$ in the worst case.

Although comparisons between block size and array length, and between average collection size C and average tile size R are not trivial, we can conclude that BDDT has the advantage that the appropriate metadata elements are identified in O(1), while SMPSs requires O($\log T$) time to locate metadata elements of overlapping task arguments.

4 Experimental Analysis

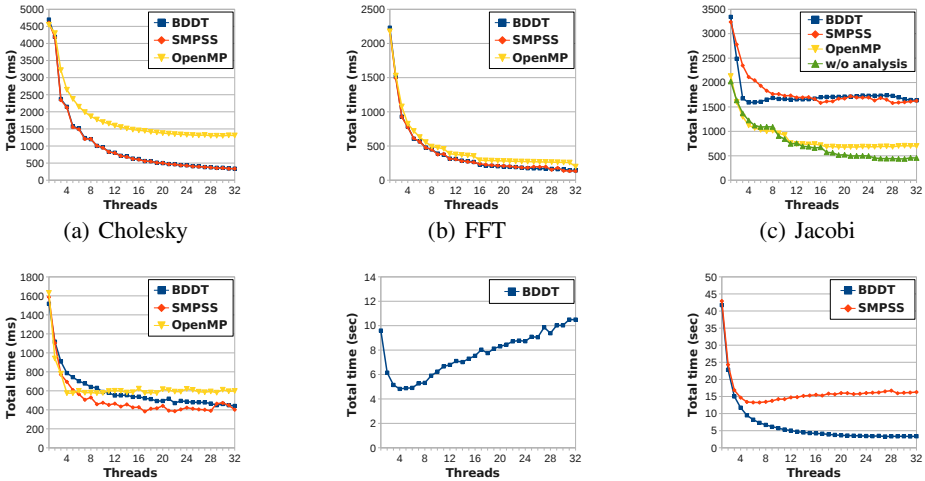
We ran all experiments on a Cray XE6 compute node with 32GB memory and two AMD Interlagos 16-core 2.3GHz dual-processors, i.e. a node with a total of 32 cores and 8 cores per processor. Every pair of cores shares one FPU, possibly reducing floating point arithmetic performance. Each 8-core processor has its own NUMA partition, yielding a total of 4 NUMA partitions with 8 GB of DRAM per partition. To uniformly distribute application data on all NUMA nodes, we initialize input data in parallel. Each core allocates and touches a part of the input array(s) used in each benchmark, so that all NUMA partitions perform approximately the same number of off-chip memory accesses during execution.

4.1 Benchmarks

We use a set of task-based benchmarks to evaluate BDDT. All benchmarks use row-major (C-language) array layout. Ferret is taken from the PARSEC benchmark suite [5] and Intruder is from the STAMP benchmark suite [7]. Cholesky, FFT, and Jacobi are SMPSs benchmarks [12] and porting them to BDDT requires only trivial changes.

We compare the performance of the task-based benchmarks with equivalent OpenMP implementations when available, so that both use the same parallelization strategy and parameters, modulo the removal of barriers in the task-based version. We compare against OpenMP in two contexts: First, we measure the performance gain of dynamic dependence analysis in applications where OpenMP requires barriers to enforce dependencies. Second, we measure the overhead cost of the dynamic dependence analysis using applications with ample task parallelism and few or no dependencies.

The block size used in BDDT to partition task arguments affects the overhead and accuracy of dynamic dependence analysis. For the block linear algebra benchmarks that work on two-dimensional tiles of the input array, we set the block size to the row size of a tile. Multisort recursively splits an array until a certain threshold, which we set as the BDDT block size.



(a) Cholesky (b) FFT (c) Jacobi

(d) Multisort (e) Intruder (f) Ferret

Fig. 2. BDDT, SMPSs and OpenMP on Interlagos

Cholesky is a factorization kernel that solves normal equations in linear least squares problems. The kernel can be decomposed into four tile operations, each of which corresponds to a task in the benchmark. Dependencies among tasks create an irregular task graph, requiring the OpenMP implementation to use barriers between phases. This limits parallelism across outermost iterations of the code. Both SMPSs and BDDT overcome this limitation using dynamic dependence analysis. Figure 2(a) shows the performance of Cholesky for a 4096×4096 double precision matrix and 128×128 tiles. BDDT performs 3.9× better than OpenMP on 32 cores, due to the extraction of additional parallelism. Moreover, BDDT matches the SMPSs performance with less than 2% deviation. Both BDDT and SMPSs versions of the benchmark achieve a top speedup of 14 on 32 cores, whereas the top speedup of the OpenMP version is 3.5.

FFT involves alternating phases of transposing a two-dimensional array and computing one-dimensional FFT. We use the FFTW library for the 1-D FFT computations; FFTW requires a row-wise layout in memory for the input array, which forces each FFT calculation task to operate on an entire row of the array. In contrast, transposition phases can break the array into tiles, so the transpose tasks' arguments are non-contiguous array tiles. Because of this difference in the memory layout of task arguments, the OpenMP version must use barriers between phases to ensure correctness. Dynamic dependence analysis in BDDT overcomes this limitation and exploits parallelism across phases, thus permitting transpose and FFT tasks to overlap. Figure 2(b) shows the performance of a 2-D FFT on 16M complex double-precision elements with BDDT and OpenMP. The input array is 4096×4096 elements and the transpose tile size is 128×128. OpenMP outperforms BDDT by up to 3% when the code runs with up to 4 cores, due to the cost of dynamic dependence analysis. Using more than 4 cores, BDDT extracts more parallelism than OpenMP and performs up to 50% better on 32 cores. BDDT still manages an overall speedup of 16× using 32 cores, whereas OpenMP achieves a maximum

speedup of 11×. Furthermore, SMPSs has a performance advantage over BDDT by 3% on 32 cores.

Jacobi is a common method for solving linear equations. Each task in Jacobi works on a tile of the array. The kernel is an iterative method, so we keep an input and an output array and swap arrays from one iteration to the next. Dynamic dependence analysis allows tasks from consecutive iterations to execute in parallel by keeping two arrays as input and output and swapping them from one iteration to the next. In contrast, the OpenMP implementation must issue a barrier between outermost iterations of the kernel. We tested Jacobi using a 4096×4096 array and 128×128 tile size. The kernel is data parallel, communication bound and memory intensive, thus highlighting the analysis overhead. In BDDT and SMPSs, overheads dominate execution time, yielding a 2.3× slowdown compared to OpenMP on 32 cores. The scalability of the BDDT version of the code is also inferior to that of OpenMP: maximum speedup with BDDT reaches 2.1 vs. 3.1 with OpenMP. BDDT allows the programmer to selectively apply or turn off the dependence analysis per task argument. The "w/o analysis" line in Figure 2(c) shows the performance of BDDT with dependence analysis disabled for all arguments, via data annotations. BDDT performs identical to OpenMP on up to 16 cores. For 16 cores or more, BDDT outperforms OpenMP by 15% to 45%, a difference that increases with the core count. The result indicates that BDDT's implementation of the runtime system is efficient, scalable, and can be used by both conventional task-based models and advanced models with out-of-order task execution capabilities.

Multisort is a parallel sorting algorithm from the Cilk distribution [11]. The algorithm is a parallel extension of ordinary Mergesort. Multisort recursively divides an array in halves up to a threshold, sorts each half, and merges the sorted halves, with each merge task working on overlapping parts of the array. OpenMP requires barriers between merge phases of the algorithm. Figure 2(d) shows the performance of Multisort on an array of 32M integers, with a threshold of 128K elements for stopping recursive subdivision. BDDT extracts more parallelism than OpenMP and achieves up to 35% better performance on 32 cores. Specifically, BDDT is 3.5× faster on 32 cores, while the top speedup of the OpenMP version is 2.7. SMPSs presents a performance advantage of 20% on average for small number of cores but it deteriorates for higher core counts, falling to 5% on 32 cores.

Intruder is a signature-based network intrusion detection system. It processes network packets in parallel in three phases: capture, reassembly, and detection. The reassembly phase uses a dictionary that contains linked lists of packets that belong to the same session. The lists are allocated in BDDT dynamic regions, allowing task footprints to include whole lists and tasks to allocate new elements. Each packet issues a task that inserts the packet into a list and possibly packs the list. The task footprint contains the whole region where the list is allocated. Figure 2(e) shows the performance of Intruder on 16384 sessions with max 512 packets per session. The figure only contains BDDT data, because SMPSS cannot express task footprints that contain dynamically linked lists. Intruder scales up to 2 on 4 cores and then speedup falls to 0.9 on 32 cores. Note that while the Intruder port to BDDT outperforms the sequential code by up to a factor of 2×, the original software transactional memory implementation [7] fails to get any speedup over sequential runs.

Ferret is an image similarity search engine. We issue parallel queries to the search engine, where each query corresponds to a task. We used 1,000 images to issue queries to a database which contains 59,695 images. Figure 2(f) shows the performance of Ferret. BDDT scales up to 15.5× on 32 cores while SMPSS reaches maximum scalability of 3.4× on 6 cores.

5 Related Work

Task parallel programming models offer a more structured alternative to parallel threads, allowing the programmer to easily specify scoped regions of code to be executed in parallel. OpenMP [2] is an API for parallelization of sequential code, where the programmer introduces a set of directives in an otherwise sequential program, to express shared memory parallelism for loops and tasks. OpenMP implements these directives in a runtime system that hides the thread management required, although the programmer is still responsible for avoiding races and inserting all necessary synchronization.

Cilk [11] is a parallel programming language that extends C++ with recursive parallel tasks. Cilk tasks can be fine-grained with little overhead, as Cilk creates parallel tasks only when necessary, using a work-stealing scheduler; and "inlines" all other tasks at no extra cost. The programmer must use *sync* statements to avoid data races and enforce specific task orderings.

Sequoia [3,8] is a parallel programming language similar to C++, which targets both shared memory and distributed systems. In Sequoia, the programmer describes (i) a hierarchy of nested parallel tasks by defining atomic *Leaf* tasks that perform simple computations, and *inner* tasks that break down the computation into smaller sub-tasks; (ii) a machine description of the various levels in the memory hierarchy and any implicit (coherency) or explicit communication (data transfer) among memories; and (iii) a mapping file that describes how data should be distributed among task hierarchies, which tasks should run at each level in the memory hierarchy, and when computation workload should be broken into smaller tasks. Sequoia inserts implicit barriers following the completion of each group of parallel tasks at a given level of the memory hierarchy.

Several programming models and languages aim to automatically infer synchronization between parallel computations. Transactional Memory [9] preserves the atomicity of parallel tasks, or transactions, by detecting conflicting memory accesses and retrying the related transactions. Jade [16] is a parallel language that extends C with coarse-grain tasks. In Jade, the programmer must declare and manage local- and shared-memory objects and define task memory footprints in terms of objects. The runtime system then detects dependencies on objects and enforces program order on conflicting tasks. Jade requires task arguments to be whole objects, and maintains per-object metadata for the dependence analysis.

StarSs [14] is a task-based programming model for scientific computations that uses annotations on task arguments to dynamically detect argument dependencies between tasks. SMPSs [12] is a runtime system that implements a subset of StarSs for multicore processors with coherent shared memory. Similarly to BDDT, in SMPSs each task invocation includes the task memory footprint, used to detect dependencies among tasks and order their execution according to program order. SMPSs describes array-tile arguments

using a three-value-bit vector representation to encode memory address ranges. This representation, however, causes aliasing and over-approximation of memory ranges when the array base address, row-size and stride are not powers of 2. Aliasing in turn creates false dependencies which reduce parallelism, and also a high overhead for maintaining and querying a global trie-structure that detects overlapping memory ranges. In comparison, BDDT uses a transparent block-level dependence analysis with constant-time overhead per block that, with proper choice of block size, eliminates aliasing and false dependencies.

SvS [4] is a task-based programming model that uses static analysis to determine possible argument dependencies among tasks and drive a runtime-analysis that computes reachable objects for every task, using an efficient approximate representation of the reachable object sets, resembling Bloom filters. It then detects possible conflicts and enforces mutual exclusion between tasks. SvS assumes all tasks to be commutative and does not preserve the original program order as BDDT. Moreover, it tracks task dependencies at the object level, restricting SvS on type-safe languages. Finally, SvS object reachability sets are approximate, and may include many reachable objects in the program, regardless of whether they are accessed by a task or not. This may hinder the available parallelism, and fails to take advantage of programmer knowledge about the memory footprint of each task.

Out-of-Order Java [10] and Deterministic Parallel Java [6] are task-parallel extensions of Java. They use a combination of data-flow, type-based, region and effect analyses to statically detect or check the task footprints and dependencies in Java programs. OoOJava then enforces mutual exclusion of tasks that may conflict at run time; DPJ restricts execution to the deterministic sequential program order using transactional memory to roll back tasks in case of conflict. As task footprints are inferred (OoOJava) or checked (DPJ) statically in terms of objects or regions, these techniques require a type-safe language and cannot be directly applied on C programs with pointer arithmetic and tiled array accesses.

6 Conclusions

We presented BDDT, a runtime system for dynamic dependence analysis in task-based programming models. BDDT performs dependence analysis among tasks with memory footprints spanning arbitrary ranges or dynamic data structures, and preserves program order for dependent tasks. BDDT outperforms OpenMP by up to a factor of 3.8× in benchmarks where dynamic dependence analysis can exploit distant parallelism beyond barriers, and similarly or better than OpenMP in data-parallel benchmarks when dynamic dependence analysis is deactivated. Compared to SMPSs, a state-of-the-art task-based model based on dynamic dependence analysis, BDDT has lower overhead, supports dynamic memory management in tasks, and allows the dependence analysis to be applied (or disabled) on individual task arguments.

Acknowledgments. We thankfully acknowledge the support of the European Commission under the 7th Framework Programs through the TEXT (FP7-ICT-261580) project. Hans Vandierendonck is supported by the People Programme (Marie Curie Actions)

of the European Union's Seventh Framework Programme (FP7/2007-2013) under REA grant agreement no. 327744. This research is also supported by EPSRC through the GEMSCLAIM project (grant EP/K017594/1).

References

1. Augonnet, C., Thibault, S., Namyst, R.: StarPU: a runtime system for scheduling tasks over accelerator-based multicore machines. Tech. Report RR-7240, INRIA (March 2010)
2. Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., Zhang, G.: The Design of OpenMP Tasks. TPDS 20(3), 404–418 (2009)
3. Bauer, M., Clark, J., Schkufza, E., Aiken, A.: Programming the Memory Hierarchy Revisited: Supporting Irregular Parallelism in Sequoia. In: PPoPP (2011)
4. Best, M.J., Mottishaw, S., Mustard, C., Roth, M., Fedorova, A., Brownsword, A.: Synchronization via Scheduling: Techniques for Efficiently Managing Shared State. In: PLDI (2011)
5. Bienia, C., Kumar, S., Pal Singh, J., Li, K.: The PARSEC benchmark suite: Characterization and architectural implications. In: PACT (October 2008)
6. Bocchino, R., Adve, V.S., Dig, D., Adve, S.V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., Vakilian, M.: A type and effect system for deterministic parallel Java. In: OOPSLA (2009)
7. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC (September 2008)
8. Fatahalian, K., Horn, D.R., Knight, T.J., Leem, L., Houston, M., Park, J.Y., Erez, M., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: Programming the Memory Hierarchy. In: SC (2006)
9. Herlihy, M., Moss, J.E.: Transactional memory: Architectural support for lock-free data structures. In: ISCA (1993)
10. Jenista, J.C., Eom, Y.H., Demsky, B.: OoOJava: Software Out-of-Order Execution. In: PPoPP (2011)
11. Leiserson, C.E.: The Cilk++ concurrency platform. TJS 51(3), 244–257 (2010)
12. Pérez, J.M., Badia, R.M., Labarta, J.: Handling Task Dependencies under Strided and Aliased References. In: ICS (2010)
13. Pérez, J.M., Bellens, P., Badia, R.M., Labarta, J.: CellSs: Making it Easier to Program the Cell Broadband Engine Processor. IBMRD 51(5), 593–604 (2007)
14. Planas, J., Badia, R.M., Ayguadé, E., Labarta, J.: Hierarchical Task-Based Programming With StarSs. IJHPCA 23(3), 284–299 (2009)
15. Pratikakis, P., Vandierendonck, H., Lyberis, S., Nikolopoulos, D.S.: A programming model for deterministic task parallelism. In: MSPC, pp. 7–12 (2011)
16. Rinard, M.C., Lam, M.S.: The Design, Implementation, and Evaluation of Jade. TOPLAS 20(3), 483–545 (1998)
17. Tofte, M., Talpin, J.-P.: Region-based memory management. Inf. Comput. 132(2) (1997)
18. Vandierendonck, H., Pratikakis, P., Nikolopoulos, D.S.: Parallel programming of general-purpose programs using task-based programming models. In: HotPar (2011)

A User-Level NUMA-Aware Scheduler for Optimizing Virtual Machine Performance

Yuxia Cheng, Wenzhi Chen, Xiao Chen, Bin Xu, and Shaoyu Zhang

College of Computer Science and Technology, Zhejiang University, Hangzhou, China
{rainytech,chenwz,chunxiao,xubin,zsy056}@zju.edu.cn

Abstract. Commodity servers deployed in the data centers are now typically using the Non-Uniform Memory Access (NUMA) architecture. The NUMA multicore servers provide scalable system performance and cost-effective property. However, virtual machines (VMs) running on NUMA systems will access remote memory and contend for shared on-chip resources, which will decrease the overall performance of VMs and reduce the efficiency, fairness, and QoS that a virtualized system is capable to provide. In this paper, we propose a "Best NUMA Node" based virtual machine scheduling algorithm and implement it in a user-level scheduler that can periodically adjust the placement of VMs running on NUMA systems. Experimental results show that our NUMA-aware virtual machine scheduling algorithm is able to improve VM performance by up to 23.4% compared with the default CFS (Completely Fair Scheduler) scheduler used in KVM. Moreover, the algorithm achieves more stable virtual machine performance.

Keywords: NUMA, virtual machine, scheduling, memory locality.

1 Introduction

Multicore processors are commonly seen in today's computer architectures. However, a high frequency (typically 2-4 GHz) core often needs an enormous amount of memory bandwidth to effectively utilize its processing power. Even a single core running a memory-intensive application will find itself constrained by memory bandwidth. As the number of cores becomes larger, this problem becomes more severe on Symmetric Multi-Processing (SMP) systems, where many cores must compete for memory controller and bandwidth in a Uniform Memory Access (UMA) manner. The Non-Uniform Memory Access (NUMA) architecture is then proposed to alleviate the constrained memory bandwidth problem as well as to increase the overall system throughput.

Commodity servers deployed in today's data centers are now typically using the Non-Uniform Memory Access (NUMA) architecture. The NUMA system links several small and cost-effective nodes (known as NUMA nodes) via the high-speed interconnect, where each NUMA node contains processors, memory controllers, and memory banks. The memory controller on a NUMA node is responsible for the local NUMA node memory access. An application accessing

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 32–46, 2013.
© Springer-Verlag Berlin Heidelberg 2013

remote NUMA node memory requires the remote memory controller to fetch the data from remote memory banks and send back the data through the high-speed interconnect, thus the latency of accessing remote node memory is larger than accessing local node memory.

The difference of memory access latency between local NUMA node and remote NUMA node will severely impact an application's performance, if the application is running on one NUMA node while its memory is located in another NUMA node. For example, the Linux default task scheduler CFS takes little consideration of the underlying NUMA topologies and will scheduling tasks to different cores depending on the CPU load balance, which eventually will result in applications running on different cores and their memory being distributed on different NUMA nodes. Especially for memory sensitive applications, the remote memory access latency will greatly impact the overall application performance.

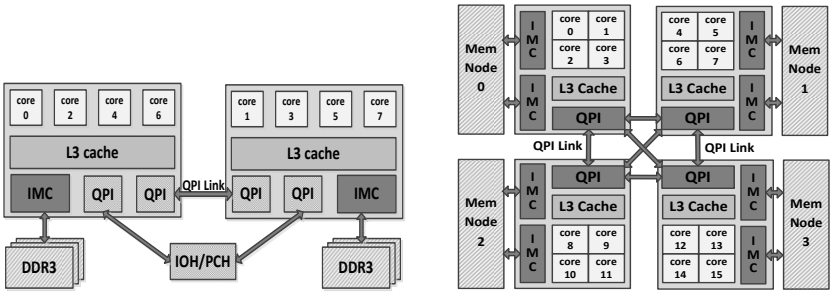
Virtualization poses additional challenges on performance optimizations of the NUMA multicore systems. Existing virtual machine monitors (VMMs), such as Xen [6] and KVM [12], are unaware of the NUMA multicore topology when scheduling VMs. The guest operating system (OS) running in a virtual machine (VM) also have little knowledge about the underlying NUMA multicore topology , which makes application and OS level NUMA optimizations working ineffectively in virtualized environment. As a result, the VMs running both in Xen and KVM are frequently accessing remote memory on the NUMA multicore systems, and this lead to sub-optimal and unpredictable virtual machine performance on NUMA servers.

In this paper, we propose a "Best NUMA Node" based virtual machine scheduling algorithm and implement it in a user-level scheduler that can periodically adjust the placement of VMs running on NUMA systems and make NUMA-aware scheduling decisions. Our solution not only improves VM's memory access locality but also maintains system load balance. And each VM achieves more stable performance on NUMA multicore systems.

The rest of this paper is organized as follows: the NUMA performance impact is analyzed in section 2. We present the proposed NUMA-aware scheduling algorithm and describe the implementation of the user-level scheduler in section 3. In section 4, the performance evaluation of the proposed algorithm is presented. Finally, we discuss the related work in section 5 and draw our conclusion in section 6.

2 NUMA Performance Impact

The NUMA architecture introduces more complex topology than UMA (Uniform Memory Access) systems. Applications (especially for long-running applications such as VMs) may have a high probability of accessing memory remotely on NUMA systems. The CPU, memory bandwidth, and memory capacity load balance among NUMA nodes put much burden on OS and VMM schedulers to properly take advantage of the NUMA architecture. The main focus of these schedulers is to load balance CPU processing resource and seldom consider the



(a) 2-NUMA-node Westmere-EP System

(b) 4-NUMA-node Westmere-EX System

Fig. 1. Dual Socket NUMA Multicore System Performance Impact

NUMA memory effect. In this section, we conduct some experiments to show that the existing VM scheduler CFS (Completely Fair Scheduler) used in KVM [12] will schedule VMs onto different NUMA nodes which results in VMs' remote memory access, and we also evaluate the performance degradation caused by remote memory access on NUMA systems.

Table 1. Hardware Configuration of multicore NUMA servers

Server models	Dell R710	Dell R910
Processor type	Intel Xeon E5620	Intel Xeon E7520
Number of cores	4 cores (2 sockets)	4 cores (4 sockets)
Clock frequency	2.4 GHz	1.87 GHz
L3 cache	12MB shared, inclusive	18MB shared, inclusive
Memory	2 memory nodes, each with 16GB	4 memory nodes, each with 16GB

We use two experimental systems for evaluation. One is a two-NUMA-node Dell R710 server, the other is a four-NUMA-node Dell R910 server. The detailed hardware configuration is shown in table 1. Both servers are commonly seen in today's data centers. The R710 server has two 2.40 GHz Intel (R) Xeon (R) CPU E5620 processors based on the Westmere-EP architecture (shown in Fig.1 (a)). Each E5620 processor has four cores sharing a 12MB L3 cache. The R710 server has a total of 8 physical cores and 16GB memory, with each NUMA node having 4 physical cores and 8 GB memory. The R910 server has four 1.87 GHz Intel (R) Xeon (R) CPU E7520 processors based on the Nehalem-EX architecture (shown in Fig.1 (b)). Each E7520 processor has four cores sharing a 18MB L3 cache. The R910 server has a total of 16 physical cores and 64 GB memory, with each NUMA node having 4 physical cores and 16 GB memory.

We briefly describe the NUMA architecture of our evaluation platforms. The 2-NUMA-node Intel Xeon Westmere-EP topology is shown in Fig.1 (a). In the

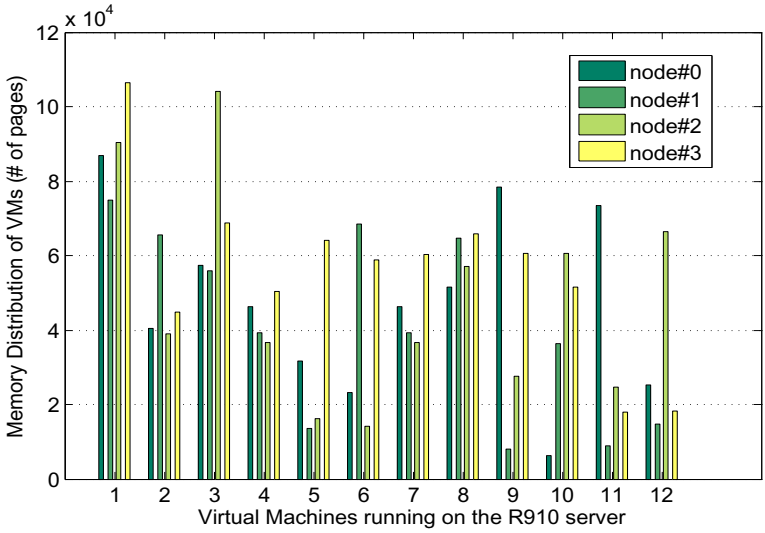


Fig. 2. Memory Distribution of VMs running on NUMA Systems

Westmere-EP architecture, there are usually four or six cores sharing the Last Level Cache (LLC, or L3 cache) in a socket, while each core has its own private L1 and L2 cache. Each socket has the Integrated Memory Controller (IMC) connected to the local three channels of DDR3 memory. Accessing to the physical memory connected to a remote IMC is called the remote memory access. The Intel QuickPath Interconnect (QPI) interfaces are responsible for transferring data between two sockets. And the two sockets communicate with I/O devices in the system through IOH/PCH (IO Hub / Platform Controller Hub) chips. Fig.1 (b) shows a 4-NUMA-node Intel Xeon Westmere-EX topology, there are four NUMA nodes interconnected by the QPI links in the system, and each node has four cores sharing one LLC with two IMCs integrated in the socket. Although other NUMA multicore processors (e.g., AMD Opteron) may differ in the configuration of on-chip caches and the cross-chip interconnect techniques, they have similar architectural designs.

2.1 Memory Distribution on NUMA Nodes

We use KVM as our experimental virtualization environment. The default Linux memory allocation strategy is allocating memory on local node as long as the task is running on that node and the node has enough free memory. Therefore, after a long period of running, a VM will migrate from one NUMA node to another NUMA node due to the CPU load balance of the CFS scheduler. Eventually, the VM's memory will be scattered on all NUMA nodes. Fig.2 shows the memory distribution of virtual machines running on the NUMA system.

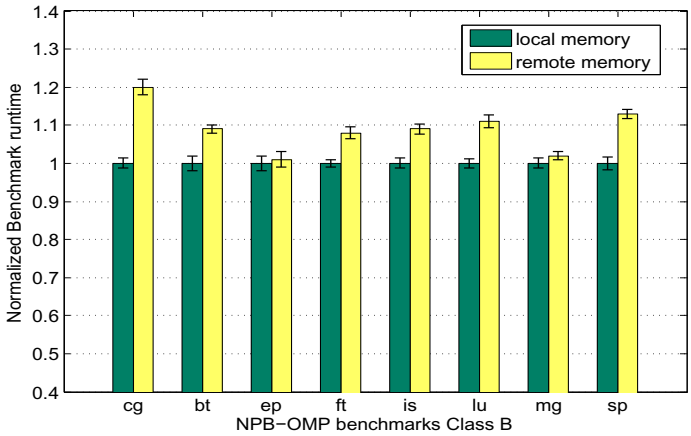


Fig. 3. Remote Memory Access Penalty on Virtual Machine Performance

The data is collected from the R910 server (described in Section 2.1) with KVM virtualization environment, twelve VMs are running on the server, and each VM is configured with 4 VCPUs and 4 GB memory.

The memory of VMs scattered on all NUMA nodes will lead to high memory access latency due to a large proportion of memory access from remote NUMA node. In section 2.2, we will study the VM's remote memory access penalty on NUMA systems.

2.2 Remote Memory Access Penalty

We run a single virtual machine on the R710 server to distinguish the remote memory access performance impact on NUMA systems. In the experimental evaluation, we first run the local memory access case that the VM's VCPUs and memory are all located in the same NUMA node. Then, we run the remote memory access case that the VM's VCPUs are pinned to one node and its memory is bound to another node (using the virsh VM configuration file). In the two cases, we record the average runtimes of NPB benchmarks (a total of five runs for each benchmark) running inside the VM.

Fig.3 shows the benchmarks' local performance compared with remote performance. The average runtime of benchmarks in the remote memory access case is normalized to the local memory access case. As the result shows, some benchmarks (such as *cg*, *lu*, *sp*) have significant performance degradation due to remote memory access latency. But there has little performance impact on other benchmarks (such as *ep* and *mg*) due to the NUMA memory effect.

From the experimental result, we find that even in a two-NUMA-node system, the remote memory access penalty is obvious, especially for NUMA sensitive workloads. Therefore, it is beneficial to improve virtual machine memory access

locality on NUMA systems via properly using NUMA-aware scheduling methods. In section 3, we present our NUMA-aware VM scheduling policy.

3 The NUMA-Aware VM Scheduler

3.1 Main Idea

Virtual machines running on multicore NUMA systems will benefit from local node execution that a VM's VCPUs are running on one NUMA node and its memory is also located on the same NUMA node. VMs running on their local nodes will reduce remote memory access latency. What's more, all VCPUs of a VM running on one NUMA node will reduce the last level cache (LLC) coherency overhead among LLCs of different NUMA nodes. If the VCPUs of a VM is running on different NUMA nodes that have separate LLCs, when they access shared data it will cause LLC coherency overhead. Finally, VMs local node execution will also reduce the interconnect contention (e.g. contention for QPI links in Intel Xeon processors). Although scheduling one VM's VCPUs on the same NUMA node will increase shared on-chip resources contention, we try to schedule different VMs onto separate NUMA nodes with best effort to mitigate shared resources contention and to maximize system throughput with a balanced memory bandwidth usage.

However, it is a big challenge to make all the VMs execute on their local NUMA node and at the same time fully utilize the scalable NUMA architecture. One simple solution is to manually bind the VMs onto NUMA nodes, so all VMs will have local node execution. But the manually bind solution lacks flexibility and may lead to system load imbalance. Some heavily loaded NUMA nodes may become the performance bottlenecks. System load imbalance will greatly impact the VMs overall performance and can not effectively and efficiently take advantage of the multicore NUMA architecture.

To improve virtual machine memory access locality and at the same time to achieve system load balance, we propose a user-level NUMA-aware VM scheduler that periodically scheduling the VCPUs onto certain NUMA nodes according to the CPU and memory usage of all VMs in the virtualized system. The NUMA-aware scheduling algorithm properly selects a "best NUMA node" for a VM that is worth scheduling onto this "best NUMA node" to improve memory access locality as well as to balance system load. Our "Best NUMA Node" based virtual machine scheduling algorithm (short for BNN algorithm) dynamically adjust the placement of VMs running on NUMA nodes as the workload behaviors of the VMs change during execution. In section 3.2, we discuss the design motivation and show a detailed description of the BNN algorithm. We implemented the BNN algorithm in our user-level VM scheduler, and the implementation of the user-level VM scheduler is presented in section 3.3.

3.2 The BNN Scheduling Algorithm

The "Best NUMA Node" based virtual machine scheduling algorithm (short for BNN algorithm) is mainly composed of three parts: (i) selecting the VMs that

are worth scheduling to improve their memory access locality; (ii) finding the "Best NUMA Nodes" for the VMs selected by the previous step; (iii) scheduling the VCPUs of the selected VMs to their "Best NUMA Node".

(i) Selecting Proper VMs

We first select proper VMs that are worth scheduling to improve the virtual machine memory access locality and the overall NUMA system load balance. To select the most actively running VMs, the CPU load of each VM is calculated online (the CPU load calculation of each VM is presented in section 3.3). VMs are then sorted by their CPU load in descending order. Then, we select the topmost k VMs as the proper VMs that are worth scheduling, such that the value k satisfies the following equation:

$$\sum_{i=1}^{k} \sum_{j=1}^{N(VM_i)} Load(V_j) > \frac{4}{5} \sum_{i=1}^{m} \sum_{j=1}^{N(VM_i)} Load(V_j) \tag{1}$$

where $N(VM_i)$ represents the number of VCPUs of VM_i, $Load(V_j)$ represents the CPU load of $VCPU_j$, and m represents the total number of VMs in the system. As denoted in the equation (1), we select the topmost k active VMs as our target scheduling VMs (the total CPU load of these k VMs occupies 80% ($\frac{4}{5}$) CPU usage of all VMs in the system) and let the default CFS scheduler take over the rest of the VMs in the system to do fine-grained load balancing job. We observe that active VMs suffer from NUMA effect more than less active VMs, therefore we select the topmost k active VMs as our target NUMA-aware scheduling VMs and the value of 80% CPU usage of all VMs is tuned by experimental results. By scheduling the most active VMs into proper NUMA nodes through our user-level scheduler and scheduling the remaining less active VMs through the system default CFS scheduler, we can effectively address the challenges of virtual machine memory access locality and system load balance on NUMA multicore systems.

(ii) Finding the "Best NUMA Node"

After selecting the proper VMs for NUMA-aware scheduling, we try to find the "Best NUMA Node" for every selected VMs. First, we examine the memory distribution of each selected VMs. The memory footprint of VMs in each NUMA node is gathered online (the calculation of memory footprint of each VM is presented in section 3.3). According to the memory footprint of the VM, we select the "Best NUMA Node" candidates (short for BNN candidates) for the VM. BNN candidates for the VM satisfy the following equation:

$$M_i > \frac{1}{n} \sum_{j=1}^{n} M_j, \quad (1 \leq i \leq n) \tag{2}$$

where M_i represents the memory footprint of the VM in NUMA node i, n represents the total number of NUMA nodes in the system. Equation (2) means the NUMA node i is selected as the BNN candidates as long as the memory footprint in NUMA node i is larger than the average memory footprint of the VM in all NUMA nodes. We select these NUMA nodes that have relatively large memory

Finding ``best NUMA node`` for each selected VMs
Input: List of selected VMs L_{vm}. The list is sorted in descending order of the VMs' CPU load. Each VM's BNN candidates set. **Output:** A mapping M_{BNN} of VMs to ``best NUMA nodes``.
Variables: the number of NUMA nodes n; the number of total VMs m; VCPU Resource VR; VM_t's BNN candidates set $BNNC_{VM_t}$;
1: Initialize VCPU Resource VR_j of each NUMA node j. 2: $VR_j = \frac{1}{n}\sum_{i=1}^{m} N(VM_i), \quad (j = 1, \ldots, n)$ 3: $M_{BNN} \leftarrow \emptyset, \quad VM_t \leftarrow$ pop_front (L_{vm}); 4: **while** $VM_t \neq$ NULL **do** 5: max = node i in $BNNC_{VM_t}$ candidates set that has the largest VR value 6: **if** ($VR_{max} - N(VM_t) \geq 0$) 7: $BNN = max$; 8: **else** 9: BNN = node j that has the largest VR value among all nodes; 10: **end if** 11: $VR_{BNN} = VR_{BNN} - N(VM_t)$, push_back (M_{BNN}, (VM_t, BNN)) 12: $VM_t \leftarrow$ pop_front (L_{vm}); 13: **end while**

Fig. 4. The algorithm of finding the BNN node for each VM

footprint of the VM as its BNN candidates. Because the VM scheduling into BNN candidate nodes will have a higher probability of accessing local memory.

Then, we find the "Best NUMA node" from the BNN candidates set for each VM. Figure 4 shows the algorithm of finding the BNN node for each VM. First, the VCPU resource of each NUMA node is initialized (Line 1-2) as follows:

$$VR_j = \frac{1}{n}\sum_{i=1}^{m} N(VM_i), \qquad (j = 1, ..., n) \tag{3}$$

where n represents the total number of NUMA nodes in the system, m represents the total number of VMs in the system, $N(VM_i)$ represents the number of VCPUs of VM_i. VR_j means the VCPU resource of NUMA node j, that is the number of VCPUs allocated in NUMA node j. We suppose that each NUMA node should have equal number of VCPUs to achieve system load balance and maximize system throughput, so we equalize VR_j as equation (3) shows.

After initializing VR_j, we design an approximate bin packing algorithm to find the "Best NUMA Node" for each selected VM. In the beginning, each node j has the VCPU resource capacity of VR_j. Every time, we pick up a VM_t from the sorted VM list (L_{vm}). We select a node that has the largest VR (VCPU resource) value from the BNN candidates set of the VM_t, and record the node

id as max (Line 5). If the max node has sufficient VCPU resource capacity to hold the VM_t (Line 6-7), then we select max as the VM_t's BNN node (good memory locality for VM_t and predictable load balance). Other wise, we try to find a node that has large VCPU resource to hold VM_t to maintain system load balance, so we select the node that has the largest VR value among all nodes in the system as the VM_t's BNN node. By heuristically selecting relatively large VR value node each time, we can achieve good system balance when assigning VMs to their BNN nodes. After selecting the BNN node for VM_t, we decrease the VR capacity of the BNN node and save the VM_t's BNN node mapping strategy in the mapping list M_{BNN}. Then, we find the BNN node for the next VM from the VM list L_{vm} until all selected VMs are mapped to their BNN nodes.

In the BNN algorithm, we assume that the number of a VM's VCPUs is smaller than the number of physical cores in one NUMA node and a VM's memory size is no larger than the physical memory size of one NUMA node. Therefore, we can assign each VM a BNN node to hold VMs. If the VCPU number and memory size of a VM are larger than a physical NUMA node (called huge VMs), we can configure these huge VMs with several small virtual NUMA nodes using the qemu-kvm's VNUMA functionality and make sure each virtual NUMA node of the huge VM is smaller than a physical NUMA node. Then, we can use the BNN algorithm to schedule these virtual NUMA nodes just like scheduling small VMs.

(iii) Scheduling VCPUs to BNN Nodes

After finding the "Best NUMA node" for each selected VMs, the scheduler migrates the VMs' VCPUs to their "Best NUMA nodes" according to the BNN mapping list M_{BNN}. We use the sched_ setaffinity() system call to schedule VCPUs to the proper NUMA nodes. After the VCPUs' affinities are set to their BNN nodes, the job of scheduling VCPUs within nodes is automatically done by the CFS scheduler. The unselected VMs (the less active VMs) will also be scheduled by the CFS scheduler to achieve more fine-grained system load balance.

As the VMs' workload behavior will change over time, our NUMA-aware VM scheduler will periodically execute the above three steps to dynamically adjust the BNN nodes for the selected VMs. The adjustment period is now heuristically set to 60s.

3.3 Implementation of User-Level Scheduler

The NUMA-aware VM scheduler is a user-level process that is designed to test the effectiveness of scheduling algorithms on real NUMA multicore systems. It is able to monitor the virtual machine execution online, gather VM's runtime information for making scheduling decisions, pass it to the scheduling algorithm and enforce the algorithm's decisions. The NUMA-aware VM scheduler has three major phases of execution: (i) Gathering system information online; (ii) Executing scheduling algorithm; (iii) Migrating VM's memory.

Table 2. Gathering system information for scheduling

Gathered information	Description of gathering methods
Information about the system's NUMA topology	The NUMA topology can be obtained via sysfs by parsing the contents of /sys/devices/system/node. The CPU topology can be obtained by parsing the contents of /sys/devices/system/cpu.
Information about the CPU load of each VM	VMs created by the KVM are regarded as general processes in the Linux system. VCPUs of a VM are regarded as threads of the VM process. The CPU load of the threads can be obtained via /proc/<PID>/task/<TID>/stat file (columns 13th and 14th represent the number of jiffies[1] during which the given thread are running in user and kernel mode respectively).
Information about the memory footprint of each VM in each NUMA node	The memory footprint of a VM in each NUMA node is the amount of memory stored on each NUMA node for the given VM process. The file /proc/<PID>/numa_maps contains the node distribution information for each virtual memory area assigned to the process in number of pages.

(i) Gathering system information

The detailed description of gathering system information through user-level scheduler online is shown in table 2. There are three kinds of information obtained online via parsing pseudo file systems (*proc* and *sysfs*).

(1) Information about the system's NUMA topology is obtained once the scheduler starts running.

(2) Information about the CPU load of each VM is calculated periodically. We calculate the average CPU load of each VM during one scheduling epoch, and we sort the VMs using their average CPU load in descending order.

(3) Information about the memory footprint of VMs in NUMA nodes can be obtained from the numa_map files as shown in table 2. The memory footprint of each VM is calculated when the scheduling algorithm selects BNN candidate nodes for the VM.

(ii) Executing scheduling algorithm

The user-level scheduler monitors the VMs workload behavior (the CPU load and memory distribution information), passes the gathered information to the NUMA-aware VM scheduling algorithm and enforces algorithm's decision on migrating VCPU threads onto their proper NUMA nodes. The NUMA-aware scheduling algorithm is periodically executed and the scheduling decisions are enforced using sched_ setaffinity() system call.

[1] One jiffy is the duration of one tick of the system timer interrupt.

(iii) Migrating VM's memory

The user-level scheduler also provides the function of migrating memory to a specified NUMA node using the move_pages() system call. Our NUMA-aware VM scheduler adopts two memory migration strategies to migrate a VM's proper memory pages to its BNN node. The two memory migration strategies are as follows:

(1) If a VM's BNN node is changed to another NUMA node and the VM's VCPUs are scheduled onto its new BNN node. We then use the Intel PEBS (Precise Event-Based Sampling) functionality [2] of sampling memory instructions to get the memory address of the VM. If the sampled memory address is located in the remote NUMA node, we uses the move_pages() system call to migrate the pages around the sampled address to the BNN node. The sampled addresses are considered as frequently accessed memory addresses which have a higher probability to be sampled by PEBS than those less frequently accessed addresses. In this way, we migrate the frequently accessed memory pages from remote node to the BNN node.

(2) When the system load is below a certain threshold (for example 1/p CPU usage of the total system, where p is the total number of physical cores), the scheduler will begin a memory migration phase. In each memory migration phase, the scheduler randomly selects one VM and migrates the VM's memory pages that reside in other nodes to its BNN node. Once the system load is below the previously defined threshold, the memory migration phase will restart memory migration phase.

4 Performance Evaluation

In this section, we evaluate the proposed BNN algorithm using the real-world parallel workloads. We compare the performance of BNN with KVM's default CFS (Completely Fair Scheduler) scheduler and a manually VM binding strategy in Section 4.1. Then we show the improvement of performance stability of the BNN scheduler in Section 4.2. Finally, we analyze the BNN's runtime overhead in Section 4.3.

We run the experiments on the R910 server described in table 1. The server is configured with 32 logical processors with hyperthreading enabled. In order to isolate the NUMA effect from other factors that affect VMs performance, we disable the Intel Turbo Boost in BIOS and set the processors to the maximum frequency. We ran VMs in qemu-kvm (version 0.15.1). Both the host and guest operating systems used in the experiments are SUSE 11 SP2 (the Linux kernel version 3.0.13). The proposed NUMA-aware VM scheduler runs in the host OS. We use the NAS Parallel Benchmark (NPB 3.3) [1] to measure virtual machine performance. The NPB benchmark suite is a set of benchmarks developed for the performance evaluation of parallel applications.

We simultaneously run 8 VMs on R910 server. Each VM is configured with 4 VCPUs and 8 GB memory. Inside each VM, we run one 4-threaded NPB-OMP benchmark. For example, a 4-threaded *bt* benchmark runs in VM1, a 4-threaded *cg* benchmark runs in VM2, and a 4-threaded *sp* benchmark runs in VM8.

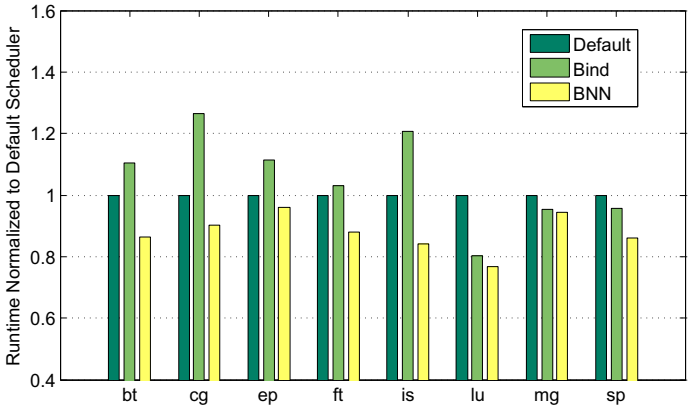


Fig. 5. Virtual machines performance compared with default scheduler

4.1 Improvement on VM Performance

Figure 5 shows the runtime of benchmarks under three different strategies: the default CFS scheduler (Default), the manually bind VMs strategy (Bind), and the BNN scheduler (BNN). We simultaneously run 8 VMs described above on R910 server. Each runtime is the average of five runs under the same strategy and is normalized to the runtime under default KVM CFS scheduler. In the Bind strategy, we manually bind two VMs into one NUMA node, and the eight VMs are evenly distributed across four NUMA nodes on R910 server with two VMs bound to every NUMA node. In the Bind strategy, VMs running *bt* and *cg* are bound to node 0, VMs running *ep* and *ft* are bound to node 1, VMs running *is* and *lu* are bound to node 2, and VMs running *mg* and *sp* are bound to node 3.

From the experimental results, we observe that BNN outperformed *Default* by at least 4.1% (ep) and by as much as 23.4% (lu). Since the *Bind* strategy is unable to adjust to the workload changes, the performance of some benchmarks degrade significantly (the performance degradation of *cg* and *is* is up to 26.4% and 20.7% respectively) compared with *Default*. From the figure, we also find that BNN is more effective for benchmarks that are more sensitive to the NUMA remote memory access latency. For example, BNN significantly improves the performance of *lu*, *bt*, and *sp* by 23.4%, 14.5%, and 14.9% respectively, while only improves the performance of *ep* and *mg* by 4.1% and 5.7%. From previous experiment in Fig.3., we can find that *lu*, *bt*, and *sp* are NUMA sensitive benchmarks, while *mg* and *ep* are insensitive to NUMA effect. The BNN scheduler considers both NUMA effect and system load balance, so BNN achieves better performance than both *Default* and *Bind*.

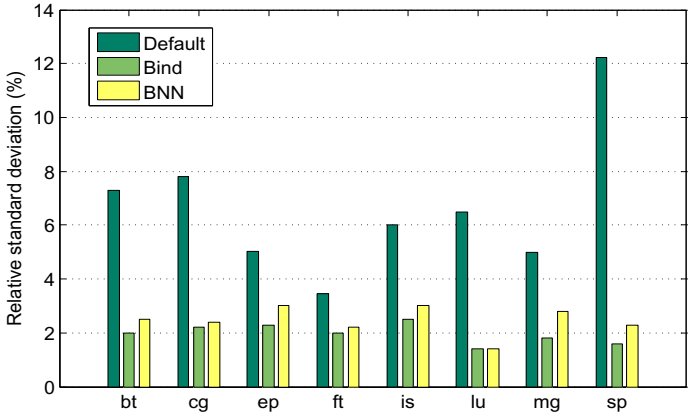


Fig. 6. Comparison of runtime variations among Default, Bind and BNN strategies

4.2 Improvement on Performance Stability

Figure 6 shows the performance stability comparison of *Default*, *Bind*, and *BNN* strategies in terms of benchmarks runtime variations. We calculated the Relative Standard Deviations (RSD) for a set of five runs of each benchmarks under different strategies. RSD measures the extent of stability across program executions. The smaller the RSD value, the more stable and consistent program performance. As expected, the manually bind strategy achieved small RSD values in all workloads with no more than 3% variations. The default CFS scheduler (that only considers CPU load when scheduling VCPUs to cores) caused much more variations than the *Bind* strategy. For the NUMA sensitive *sp* benchmark, the variations can be as high as 12.4%. In comparison, BNN achieves performance stability close to the *Bind* strategy and has significant improvement on performance stability than the *Default* strategy.

4.3 Overhead Analysis

The time complexity of BNN algorithm is O(nlgn). Sorting VMs according to their CPU load has O(nlgn) time complexity, and finding the "best NUMA node" for each VM has O(n) time complexity. As our scheduler executes the BNN algorithm every 60s, so the total overhead of BNN scheduling algorithm is very low. Our experimental results show that the proposed NUMA-aware scheduler incurs less than 0.5% CPU overhead in the system.

5 Related Work

There has been great research interest in performance optimizations of NUMA-related multicore systems. Many research efforts aim at improving application

throughput, fairness, and predictability on NUMA multicore systems. Existing work has tried to address these issues via thread scheduling and memory migration.

In UMA (Uniform Memory Access) multicore systems, thread scheduling methods have been studied to avoid the destructive use of shared on-chip resources [7,16,13] or to use the shared resources constructively [4,8]. The NUMA (Non-Uniform Memory Access) architecture introduces another performance impact factor, the memory locality factor, to be considered when scheduling threads[15]. Researchers proposed the profile-based [5] or dynamic memory migration techniques [9] to improve memory locality on NUMA systems. [7] and [13] considered both shared on-chip resources and memory locality factors to optimize applications performance on NUMA multicore systems. [10] proposed a user-level scheduler on NUMA systems to help design NUMA-aware scheduling algorithms.

Virtualization poses additional challenges on performance optimizations of NUMA multicore systems. [3] proposed a technique that allows a guest OS to be aware of its virtual NUMA topology by reading the emulated ACPI (Advanced Configuration and Power Interface) SRAT (Static Resource Affinity Table). [14] presented a method that allows the guest OS to query the VMM via para-virtualized hypercalls about the NUMA topology. [11] proposed another approach that does not assume any program or system-level optimizations and directly works in the VMM layer by using Performance Monitoring Unit (PMU) to dynamically adjust VCPU-to-core mappings on NUMA multicore systems.

In contrast, our NUMA-aware virtual machine scheduler uses the novel BNN algorithm to dynamically find the "Best NUMA Node" for each active VM and allows these VMs running on their BNN nodes and their memory also allocated in their BNN nodes. Our approach does not need modify the VMM or guest OS, and has a low overhead that only uses system runtime information available from the Linux pseudo file systems to make scheduling decisions.

6 Conclusion

In this paper, we proposed a "Best NUMA Node" based virtual machine scheduling algorithm and implemented it in a user-level scheduler in the KVM virtualized systems. The experimental results show that the BNN algorithm improves virtual machine performance. Optimizing virtual machine performance on NUMA multicore systems faces a lot of challenges, our solution tries to improve memory access locality and at the same time maintain system load balance. In the future work, we try to (1) find metrics for predicting data sharing among VMs and using these metrics to aid VM scheduling on NUMA systems; and (2) design a more adaptive memory migration strategy to further improve memory access locality on NUMA systems.

References

1. The NAS Parallel Benchmarks, http://www.nas.nasa.gov/publications/npb.html
2. Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3: System Programming Guide
3. Ali, Q., Kiriansky, V., Simons, J., Zaroo, P.: Performance Evaluation of HPC Benchmarks on VMware's ESXi Server. In: Alexander, M., et al. (eds.) Euro-Par 2011, Part I. LNCS, vol. 7155, pp. 213–222. Springer, Heidelberg (2012)
4. Bae, C.S., Xia, L., Dinda, P., Lange, J.: Dynamic Adaptive Virtual Core Mapping to Improve Power, Energy, and Performance in Multi-socket Multicores. In: HPDC (2012)
5. Marathe, J., Mueller, F.: Hardware Profile-Guided Automatic Page Placement for ccNUMA Systems. In: PPoPP (2006)
6. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the Art of Virtualization. In: ACM SIGOPS Operating Systems Review (2003)
7. Blagodurov, S., Zhuravlev, S., Dashti, M., Fedorova, A.: A Case for NUMA-Aware Contention Management on Multicore Systems. In: USENIX ATC (2011)
8. Ghosh, M., Nathuji, R., Lee, M., Schwan, K., Lee, H.S.: Symbiotic Scheduling for Shared Caches in Multi-core Systems using Memory Footprint Signature. In: ICPP (2011)
9. Ogasawara, T.: NUMA-Aware Memory Manager with Dominant-Thread-Based Copying GC. In: OOPSLA (2009)
10. Blagodurov, S., Fedorova, A.: User-Level Scheduling on NUMA Multicore Systems under Linux. In: Proceedings of Linux Symposium (2011)
11. Rao, J., Wang, K., Zhou, X., Xu, C.: Optimizing Virtual Machine Scheduling in NUMA Multicore Systems. In: HPCA (2013)
12. Kivity, A., Kamay, Y., Laor, D., Lublin, U., Liguori, A.: KVM: the Linux Virtual Machine Monitor. In: Proceedings of the Linux Symposium (2007)
13. Majo, Z., Gross, T.R.: Memory Management in NUMA Multicore Systems: Trapped between Cache Contention and Interconnect Overhead. ACM SIGPLAN Notices (2011)
14. Rao, D.S., Schwan, K.: vNUMA-mgr: Managing VM Memory on NUMA Platforms. In: HiPC (2010)
15. Tang, L., Mars, J., Vachharajani, N., Hundt, R., Soffa, M.: The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In: ISCA (2011)
16. Zhuravlev, S., Blagodurov, S., Fedorova, A.: Addressing Shared Resource Contention in Multicore Processors via Scheduling. In: ASPLOS (2010)

Towards RTOS: A Preemptive Kernel Basing on Barrelfish*

Jingwei Yang, Xiang Long, Xukun Shen, Lei Wang,
Shuaitao Feng, and Siyao Zheng

Beihang University, Beijing 100191, PRC
{yangjingwei,fengshuaitao,zhengsiyao}@les.buaa.edu.cn,
{long,xkshen,wanglei}@buaa.edu.cn

Abstract. Multi-core or even many-core systems are becoming common. New operating system architectures are needed to meet the ever increasing performance requirements of multi-/many-core based embedded systems. Barrelfish is a multi-/many-core oriented open-source operating system built by ETH Zurich and Microsoft Research in a multi-kernel style. Every kernel runs on a dedicated core with local interrupts disabled, which may result in a failure in real-time systems. To make it preemptive, new kernel state and private kernel stack are added. Capability is the central mechanism of Barrelfish, lots of analyses and modifications have been done to make the duration interrupts are disabled as short as possible when kernel objects in capability space are accessed. As a result, meaningful real-time performance have been obtained. Combined with the inherent scalability of multi-kernel design in Barrelfish, the work in this paper could be a useful reference for multi-/many-core based embedded systems.

Keywords: embedded system, real-time, multi-/many-core, multi-kernel, preemption.

1 Introduction

Multi-/Many-core based embedded systems are widely used in many fields such as network routers, automobile electronics, large scale complex control systems and so on. It's believed that processors with hundreds or even thousands of cores(hard-threads) will be used in the near future[1][2][3][4]. As a result, operating system could become a bottle-neck for computers with large number of cores because of the poor scalability[5][6][8][19]. Many-core based embedded systems may suffer more from operating system[13][18]. Predictability is often one of the most important issues faced by embedded systems, especially for hard real-time systems[9][10], multi-/many-core processors make it a great challenge for the system to be predictable. Under shared memory programming paradigm, multi-/many-core based embedded systems often perform poorly with respect to scalability and predictability because of reasons as following:

* The work is supported by the National High Technology Research and Development Program(No. 2011AA01A204).

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 47–61, 2013.
© Springer-Verlag Berlin Heidelberg 2013

1. scalability:

   - ping-ponging and false-sharing effects of cache-coherence increase system overhead as the number of cores increases.
   - the increase of communication speed among cores and memory modules doesn't quite follow the increase of cores.
   - accesses to shared variables protected by locks must follow a serial pattern although lots of cores exist in the system.

2. predictability:

   - optimized processor designs such as super-scaler architecture, pipelines, and branch prediction make CPI(Cycles Per Instruction) varies.
   - the complex cache unit and poor associated control capability make it hard to predict the latency of cache access.
   - it's difficult to control or predict the precise time when each core gets access to the shared bus and memory modules.
   - the time it takes to access a memory unit is greatly affected by when and where the transaction occurs.
   - prediction of the time needed to finish access to shared memory space in a multi-/many-core environment is also a challenge.

2 Multi-/Many-core Oriented Operating System

To take up the challenges mentioned above, solutions of both hardware and software are provided. In this paper, a software solution is discussed. Barrelfish[5] is a multi-/many-core oriented open-source operating system built by ETH Zurich and Microsoft Research. The multi-kernel philosophy of Barrelfish mainly focus on the scalability of operating system on multi-/many-core platforms, while it also provides excellent predictability for real-time systems. Each kernel in Barrelfish runs on a dedicated core(hard-thread) and shares almost nothing with others except for some necessary global variables. Lots of factors that result in poor scalability and predictability can be avoided. The kernel schedules and runs dispatchers, which are implementation of scheduler activations. It follows a message passing programming paradigm in Barrelfish kernel, inter-dispatcher communication is realized by sending messages[7].

The kernel is also called CPU driver[5], and it provides only a limited number of facilities such as scheduling, interrupt management, capability[16], memory mapping, inter-dispatcher communication and so on. Barrelfish uses the micro-kernel model, most of the architecture independent modules(memory server, monitor, device driver etc.) are moved up to user space, and the whole system can run on a heterogeneous multi-/many-core system with the support of CPU divers[6]. This is good for embedded systems using both CPU and DSP. Explicit message-passing, user space drivers and system services make it easier to model the timing characteristic of Barrelfish, which is meaningful for real-time

applications[12]. What's more, every domain can run an application specific run-time systems or even operating systems in user space, to better satisfy the particular design constraint of application.

Despite so many characteristics suitable for multi-/many-core real-time system, Barrelfish is not meant for it particularly. Preemption[11], which is an important feature in most real-time operating system, is not supported in Barrelfish. Functions in kernel control path such as system calls, and interrupt handlers run in a single core environment with interrupts disabled until it returns to user mode, which may prevent dispatchers and interrupt handlers with higher priority from preempting current kernel control path. What's more, every kernel function shares the same kernel stack in a time-division manner, context switch for dispatcher in kernel mode will arise an error.

3 Analysis on Interrupt Latency

Interrupt latency is one of the most important performance indexes in real-time system. The time it consumes before handling an interrupt is affected not only by the hardware unit in processor, but also by the duration when interrupt is disabled. The WCET(Worst Case Execution Time)[12] of interrupt latency depends not on the duration interrupt is allowed, but on the duration interrupt is not allowed. As all of the kernel control path of Barrelfish execute with interrupt disabled, there is lot of space to improve the interrupt latency.

Interrupt latency consists of hardware latency and kernel latency. L_H stands for hardware latency and L_K stands for kernel latency, then the whole interrupt latency can be represents as:

$$L = L_H + L_K \tag{1}$$

Usually, hardware latency depends on the interrupt logic of processor, which is supposed to be constant. Kernel latency is the difference between the time interrupt is detected by processor and the time kernel begins to handle it. In real-time system, instruction steam executed by every processor core can be divided into N sections(see Figure 1), where $1 \leq N < \infty$, every section consists of two subsections: kernel subsection and user subsection, which mean instructions run in kernel mode and instructions run in user mode respectively. We assume that the time it consumes to execute N sections of instructions is T, and the time needed to execute the $i_{th}(1 \leq i \leq N)$ section is Ts_i, which equals $Tk_i + Tu_i$, where Tk_i stands for kernel subsection execution time, and Tu_i stands for user subsection execution time.

When interrupt occurs, the probability it disturbs the i_{th} user subsection and kernel subsection are Tu_i/T and Tk_i/T respectively, so the probability it disturbs the i_{th} section is $(Tk_i + Tu_i)/T$. In user subsection, interrupt can be handled immediately if interrupt is enabled, no matter the kernel is preemptive or not. In this situation, the kernel latency is 0 in theory.

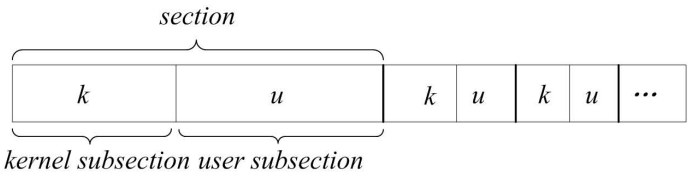


Fig. 1. Sections of instruction stream

3.1 Non-preemptive Barrelfish

In kernel mode of Barrelfish, an interrupt which happens in the i_{th} kernel subsection won't be handled until the system returns to user mode, so the kernel latency is the time it takes to finish the kernel subsection after interrupt has been detected. In every section, the instant at which interrupt occurs is supposed to conform to uniform distribution, and handling of every interrupt happened in kernel mode experiences an unique latency while handling of interrupt happened in user mode experiences no latency, then the value of kernel latency can be described by the following cumulative distribution function:

$$Fs_i(x) = \begin{cases} 0, & x < 0 \\ x/Ts_i + Tu_i/Ts_i, & 0 \leq x \leq Tk_i \\ 1, & x > Tk_i \end{cases} \tag{2}$$

During the whole execution time of T, which consists of N sections, the kernel latency can be described as:

$$F(x) = Ps_1 * Fs_1(x) + Ps_2 * Fs_2(x) + ... + Ps_N * Fs_N(x) = \sum_{i=1}^{N} Ps_i * Fs_i(x) \tag{3}$$

Where Ps_i stands for the probability interrupt detected in the i_{th} section, it can be calculated by Ts_i/T. The probability density function of kernel latency in the i_{th} section is $fs_i(t)$ and:

$$Fs_i(x) = \int_{-\infty}^{x} fs_i(t)dt \tag{4}$$

So we can get that:

$$F(x) = \sum_{i=1}^{N} Ps_i * Fs_i(x) = \sum_{i=1}^{N} Ps_i * \int_{-\infty}^{x} fs_i(t)dt = \int_{-\infty}^{x} \sum_{i=1}^{N} Ps_i * fs_i(t)dt \tag{5}$$

Then the density function of kernel latency during T is:

$$f(t) = \sum_{i=1}^{N} Ps_i * fs_i(t) \tag{6}$$

The expected value of kernel latency during Ts_i is:

$$Es_i(x) = \int_{-\infty}^{+\infty} xfs_i(x)dx \tag{7}$$

And expected value of kernel latency during T can be obtained:

$$E(x) = \int_{-\infty}^{+\infty} xf(x)dx = \sum_{i=1}^{N} Ps_i * \int_{-\infty}^{+\infty} xfs_i(x)dx = \sum_{i=1}^{N} Ps_i * Es_i(x) \tag{8}$$

3.2 Preemptive Barrelfish

In preemptive kernel, every kernel subsection can be divided into several regions, every of which consists of two parts: critical subregion and non-critical subregion(see Figure 2). Interrupts detected in kernel subsection can be handled in non-critical subregion, but not in critical subregion in case race condition happens. Provided that the number of regions in every kernel subsection is M, where $1 \leq M < \infty$, the time it takes to execute the $j_{th}(1 \leq j \leq M)$ critical subregion and non-critical sub region are Tc_j, Tnc_j respectively, the total time needed to run the j_{th} region in a kernel subsection is $Tc_j + Tnc_j$, which is represented by Tr_j, the distribution of kernel latency in every region is described as:

$$Fr_j(x) = \begin{cases} 0, & x < 0 \\ x/Tr_j + Tnc_j/Tr_j, & 0 \leq x \leq Tc_j \\ 1, & x > Tc_j \end{cases} \tag{9}$$

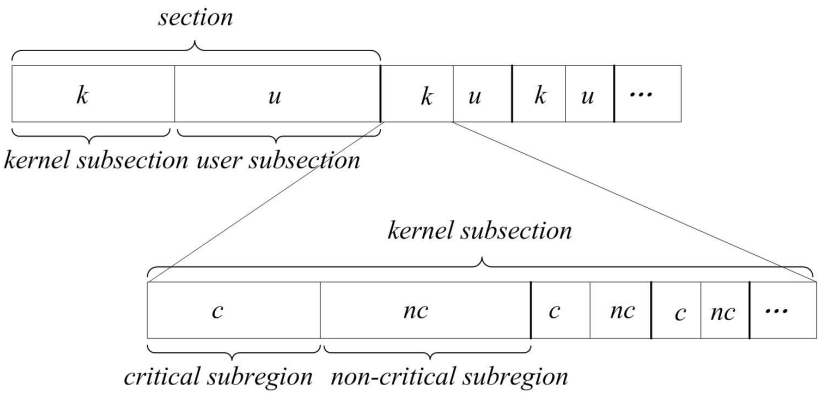


Fig. 2. Regions in kernel subsection

During the execution of the i_{th} kernel subsection, which includes M regions, the kernel latency can be represented as:

$$Fk_i(x) = Pr_1 * Fr_1(x) + Pr_2 * Fr_2(x) + ... + Pr_M * Fr_M(x) = \sum_{j=1}^{M} Pr_j * Fr_j(x) \tag{10}$$

Where Pr_j stands for the conditional probability that interrupts are detected in the j_{th} region of a kernel subsection, which can be calculated by Tr_j/Tk_i. The probability density function of kernel latency in the j_{th} region of a kernel subsection is $fr_j(t)$ and:

$$Fr_j(x) = \int_{-\infty}^{x} fr_j(t)dt \tag{11}$$

Then:

$$Fk_i(x) = \sum_{j=1}^{M} Pr_j * Fr_j(x) = \sum_{j=1}^{M} Pr_j * \int_{-\infty}^{x} fr_j(t)dt = \int_{-\infty}^{x} \sum_{j=1}^{M} Pr_j * fr_j(t)dt \tag{12}$$

So the probability density of the i_{th} kernel subsection is:

$$fk_i(t) = \sum_{j=1}^{M} Pr_j * fr_j(t) \tag{13}$$

The expected kernel latency in j_{th} region is:

$$Er_j(x) = \int_{-\infty}^{+\infty} xfr_j(x)dx \tag{14}$$

So the expected kernel latency in i_{th} kernel subsection is:

$$Ek_i(x) = \int_{-\infty}^{+\infty} xfk_i(x)dx = \sum_{j=1}^{M} Pr_j * \int_{-\infty}^{+\infty} xfr_j(x)dx = \sum_{j=1}^{M} Pr_j * Er_j(x) \tag{15}$$

3.3 Preemptive Barrelfish vs. Non-preemptive Barrelfish

Provided the i_{th} instruction section that locates in a time zone $[a, b]$, there exists an instant c, where $a < c \leq b$, system runs in kernel mode before c and returns to user mode after c in the zone.

In non-preemptive Barrelfish, according to equation (2), the distribution function of kernel latency in zone $[a, b]$ is:

$$Fs_i(x) = \begin{cases} 0, & x < 0 \\ Fk_i(x) * (c-a)/(b-a) + (b-c)/(b-a), & 0 \leq x \leq (c-a) \\ 1, & x > (c-a) \end{cases} \tag{16}$$

$Fk_i(x)$ is the distribution function of kernel latency in kernel subsection:

$$Fk_i(x) = \begin{cases} 0, & x < 0 \\ x/(c-a), & 0 \le x \le (c-a) \\ 1, & x > (c-a) \end{cases} \tag{17}$$

And expected value of kernel latency in kernel subsection is:

$$Ek_i(x) = (c-a)/2 \tag{18}$$

The density function of kernel latency in the i_{th} section is:

$$fs_i(t) = \begin{cases} 1/(b-a), & 0 < t < (c-a) \\ 0, & t \le 0, t \ge (c-a) \end{cases} \tag{19}$$

So the expected kernel latency is:

$$Es_i(x) = \int_0^{c-a} x/(b-a)dx = (c-a)^2/(2*(b-a)) = (c-a)/(b-a)*Ek_i(x) \tag{20}$$

In non-preemptive Barrelfish, the expected kernel latency in a section is just represented by equation (20). In preemptive Barrelfish, the expected kernel latency in a section can be represented by equation (20) too, but the kernel subsection $[a, c]$ can be divided to M regions, so $Es_i(x)$ can be transformed according to equation (15):

$$Es_i(x) = (c-a)/(b-a) * Ek_i(x) = (c-a)/(b-a) * \sum_{j=1}^{M} Pr_j * Er_j(x) \tag{21}$$

Assuming that the j_{th} region in kernel subsection $[a, c]$ starts at a_j and stops at b_j, in every region, kernel runs in critical subregions before c_j and non-critical regions after c_j, where $a_j < c_j \le b_j$, $a_1 = a$, $a_{j+1} = b_j$, and $b_M = c$.

The expected kernel latency in a region is calculated similarly with that in a section:

$$Er_j(x) = \int_0^{c_j-a_j} x/(b_j-a_j)dx = (c_j-a_j)^2/(2*(b_j-a_j)) = (c_j-a_j)/(b_j-a_j)*Ec_j(x) \tag{22}$$

Where $Ec_j(x) = (c_j - a_j)/2$.

So the expected kernel latency in a section is:

$$Es_i(x) = (c-a)(b-a) * \sum_{j=1}^{M} Pr_j * (c_j - a_j)/(b_j - a_j) * Ec_j(x) \tag{23}$$

Compare equation (23) with equation (20) we can get that preemptive Barrelfish outperforms non-preemptive Barrelfish in aspect of kernel latency because:

$$\begin{aligned}
&(c-a)/(b-a) * \textstyle\sum_{j=1}^{M} Pr_j * (c_j - a_j)/(b_j - a_j) * (c_j - a_j)/2 \\
&\leq (c-a)/(b-a) * \textstyle\sum_{j=1}^{M} Pr_j * (c_j - a_j)/(b_j - a_j) * (b_j - a_j)/2 \\
&\leq (c-a)/(b-a) * \textstyle\sum_{j=1}^{M} (b_j - a_j)/2 \\
&= (c-a)/(b-a) * (b_M - a_1)/2 \\
&= (c-a)/(b-a) * (c-a)/2 \\
&= (c-a)^2/(2 * (b-a))
\end{aligned} \tag{24}$$

Which means that $Es_i(x)$ in preemptive Barrelfish is less than or equal to $Es_i(x)$ in non-preemptive Barrelfish, then during T, kernel latency in preemptive Barrelfish is less than or equal to that in non-preemptive Barrelfish.

All of the equations above only describe the average case of kernel latency, in real-time system, the worst case matters too. In non-preemptive Barrelfish, kernel latency in section $[a, b]$ will reach the largest possible value, which is $b - a$, when $c = b$, while in preemptive Barrelfish, kernel latency in section $[a, b]$ depends the largest critical subregion $[a_j, b_j]$, which is at most $b_j - a_j$ when $c_j = b_j$. It's known that $(b_j - a_j) \leq (b - a)$ because region $[a_j, b_j]$ locates in section $[a, b]$, which means $a_j \geq a$ and $b_j \leq b$. So, in worst case, preemptive Barrelfish still outperforms non-preemptive Barrelfish in aspect of kernel latency.

4 Preemption Support of Barrelfish

4.1 Analysis and Design for Preemption

When interrupt is handled in user mode, kernel of Barrelfish saves the register snapshot of current dispatcher in area shared between kernel and user space. There are two kinds of register saving area: enabled saving area and disabled saving area, it't up to the state of current dispatcher that kernel chooses which one to save register snapshot in. A boolean type variable *disabled* in DCB(Dispatcher Control Block) indicates the state of the dispatcher. If *disabled* == **false**, register snapshot will be saved in enabled saving area, while *disabled* == **true**, register snapshot will be saved in disabled saving area. When a dispatcher is chosen to run after scheduling or interrupt handling, the kernel will choose one of the snapshots in enabled saving area or disabled saving area to reset/restore the dispatcher's registers.

The time-division shared kernel stack in Barrelfish is not used to save context in user mode, but that in kernel mode. To be preemptive, each dispatcher should have a private kernel stack to save context of kernel functions executing in its address space in case it's switched out. What's more, a kernel saving area is added to DCB, which is used to save the register snapshot the first time dispatcher executing in kernel mode is preempted by an interrupt handler. Just like the variable of *disabled*, a new boolean type variable *kernel* is cited to indicate if the current dispatcher executing in kernel mode when preempted. In this new design, a dispatcher could be in one of the three states such as **kernel**, **disabled**,

enabled. The state of **kernel** has a higher priority than the other two, when *kernel* == **true**, dispatcher is definitely in kernel mode, no matter what *disabled* equals, otherwise, the dispatcher is either in **disabled** or **enabled** state, which are mutual exclusive to each other, according to the variable *disabled*.

In preemptive kernel, interrupt can be allowed when kernel control path such as system calls, exception handlers, and interrupt handlers are executing, context switch is not allowed when handling interrupts. An integer type variable *preempt_count* is used to count the number of interrupts under handling, it increases by one before enter interrupt handlers while decreases by one after leave interrupt handlers. If dispatcher runs in user mode, the register snapshot is saved in enabled or disabled saving area by the common IRQ handler at first, meanwhile, stack pointer is set to the top of private kernel stack of current dispatcher. The same operations on stack pointer will be carried out when system calls and exceptions occurs in user mode. If dispatcher runs in kernel mode, which may result from system calls or exceptions, the register snapshot is saved in kernel saving area the first time interrupt occurs, and private kernel stack will still be used to save context of kernel functions. If interrupt arrives when other interrupts are being handled, the register snapshot will be saved in the current kernel stack instead of kernel saving area, current kernel stack will still be used too. Notice that there exists a special kernel state, in which no dispatcher is runnable and the system stops to execute **hlt** instruction in kernel mode, once interrupt arrives, the dedicated handler will run. In this situation, no private kernel stack but the time-division shared kernel stack in original design is used to save context of kernel functions.

Table 1. Rules related with kernel stack, context switch and snapshot area

	user mode		kernel mode			
	enabled	disabled	syscall	exception	interrupt	waiting
kernel stack	private	private	private	private	private/shared	shared
saving area	enabled	disabled	kernel	kernel	private/shared	N/A
context switch	allowed	allowed	allowed	allowed	not allowed	allowed
restoring area	enabled	disabled	kernel	kernel	private/shared	N/A

1.private: private kernel stack 2.shared: shared kernel stack
3.enabled: enabled saving area 4.disabled: disabled saving area 5.kernel: kernel saving area

Scheduling and context switch is only allowed in the context of systems calls, exception handlers or at the end of interrupt handling. An interrupt handler must check if it's nested in other interrupt handlers before scheduling or switching to another dispatcher, only when all the interrupt handlers have been finished(*preempt_count* == *0*) should scheduling or context switching be carried out. Every dispatcher gets to run again by means of *dispatch*, a function carrying out context switch. *dispatch* chooses the proper register saving area to reset/restore target dispatcher registers. When interrupt arrives, first the register snapshot of current dispatcher should be saved into dedicated saving area,

and then the kernel stack should be chosen. The rules for setting kernel stack and register saving area are shown in table 1. When interrupt handler finished, either another interrupt handler or dispatcher get to run. The rules for context switching and choosing the right target saving area to restore is shown in table 1 too.

4.2 Protection in Critical Region

Preemption only makes it possible to handle interrupt in kernel mode, it's sure that interrupt is not allowed all along the kernel control path. In Barrelfish, every kernel(CPU driver) run in a single core environment, critical region should be avoided when interrupt is allowed. The mostly invoked kernel code is related with capability operation, capability is a central security mechanism and most of the kernel functions such as system calls, message passing, and interrupt handling are invoked by means of capability reference. Call graph about capability is fully analyzed, race conditions should never occur in any section of code. Interrupt is disabled before entering critical region and restored when leaving. Capability functions such as *insert_after*, *insert_before*, scheduling list operation functions *dispatch*, *scheduling*, *make_runnale* operating on shared data structure, and operations on hardware which may be disturbed should all be called in a context where interrupt is not allowed.

5 Experiment

To evaluate the preemptive kernel basing on Barrelfish, we have measured the kernel latency and kernel overhead[13][14]. According to the result obtained in section 3, kernel latency depends heavily on the portion of instructions executed in kernel mode. If only user mode code is executed all the time, there will almost be no improvement in kernel latency because nearly all of the interrupts can be handled immediately in both preemptive and non-preemptive kernel. If code executed is full of I/O operations or system calls, then preemptive kernel would outperform non-preemptive kernel. In general, both of the kernel overhead and kernel latency should be measured to evaluate the performance of preemptive kernel in this paper.

5.1 Experiment Environments

In this experiment, preemptive barrelfish executes on a platform basing on 2 quad-core Xeon-5606 processors, which run at 2.13GHz, the capacity of main memory is 8GB. In order to simulate an external interrupt which triggers in a predictable interval, local APIC(Advanced Programmable Interrupt Controller) timer is set to expire at a dedicated instant in one-shot mode[20][21], then the timer interrupt handler will get invoked in a few time, which is treated as kernel latency here, after the expiring. The experiment is carried out in environments of CPU-bound and I/O-bound workloads separately, which are the two extreme of a real-world scenario.

- CPU-bound workloads: characterized by pure arithmetic logical computing tasks that keep processor core busy, there is almost no critical region during execution, except for some interrupt handling and process scheduling.
- I/O-bound workloads: characterized by a large number of I/O operations such as ethernet transfer, communication port access, where interrupts are disabled to maintain the consistency of system.

Kernel overhead equals the difference between real time needed to execute a CPU-bound workload and the wall time consumed to execute it. The real execution time can be measured when the program runs in kernel mode with interrupt disabled, and the wall time consumed is measured when it runs in user mode with interrupt enabled.

5.2 Evaluating Kernel Latency

In Figures 3(a)-(d), x-axis illustrates the number of times every experiment is carried out. Figure 3(a) and Figure 3(b) represent the kernel latency of non-preemptive Barrelfish and preemptive Barrelfish respectively, each sample represents a single measurement. In I/O-bound workload, the system executes in kernel mode with a higher probability, then preemptive kernel has a lower kernel latency in both average case and worst case, which matches the result in section 3. As shown in Figure 3(b), the worst case kernel latency in preemptive kernel is less than 2500 cycles while that in non-preemptive kernel is as much as 3500 cycles in preemptive kernel as Figure 3(a), it's a encouraging improvement for hard real-time system. In addition, the average case kernel latency is also improved approximately from 2400 cycles to 2200 cycles, which is meaningful for soft real-time system.

In the CPU-bound workload represented in Figure 3(c) and Figure 3(d), system executes in user mode with higher probability, preemptive kernel doesn't outperform non-preemptive too much, interrupt can both be handled immediately after it occurs because CPU-bound workloads run in kernel mode seldom. The average kernel latency are almost the same in preemptive and non-preemptive kernel. Only in few case that non-preemptive kernel has larger kernel latency, just as shown in 3(c) and 3(d), the worst kernel latency is more than 2500 cycles in non-preemptive kernel and less than 2400 cycles in preemptive kernel. (e)-(h) in Figure 3 also represent the different kernel latency between non-preemptive Barrelfish and preemptive Barrelfish, they are the cumulative distribution function of kernel latency. In all of the four figures, probability reaches to 1 from 0 rapidly during a narrow domain in x-axis, which represents the kernel latency. The steeper the curve is, which means less standard deviation, the better the kernel performs with respect to kernel latency. It's clear that preemptive kernel has a better kernel latency when running I/O-bound workload, as shown in (e) (f) of Figure 3. Figure 3(g) and Figure 3(h) show the equivalent result to Figure 3(c) and Figure 3(d).

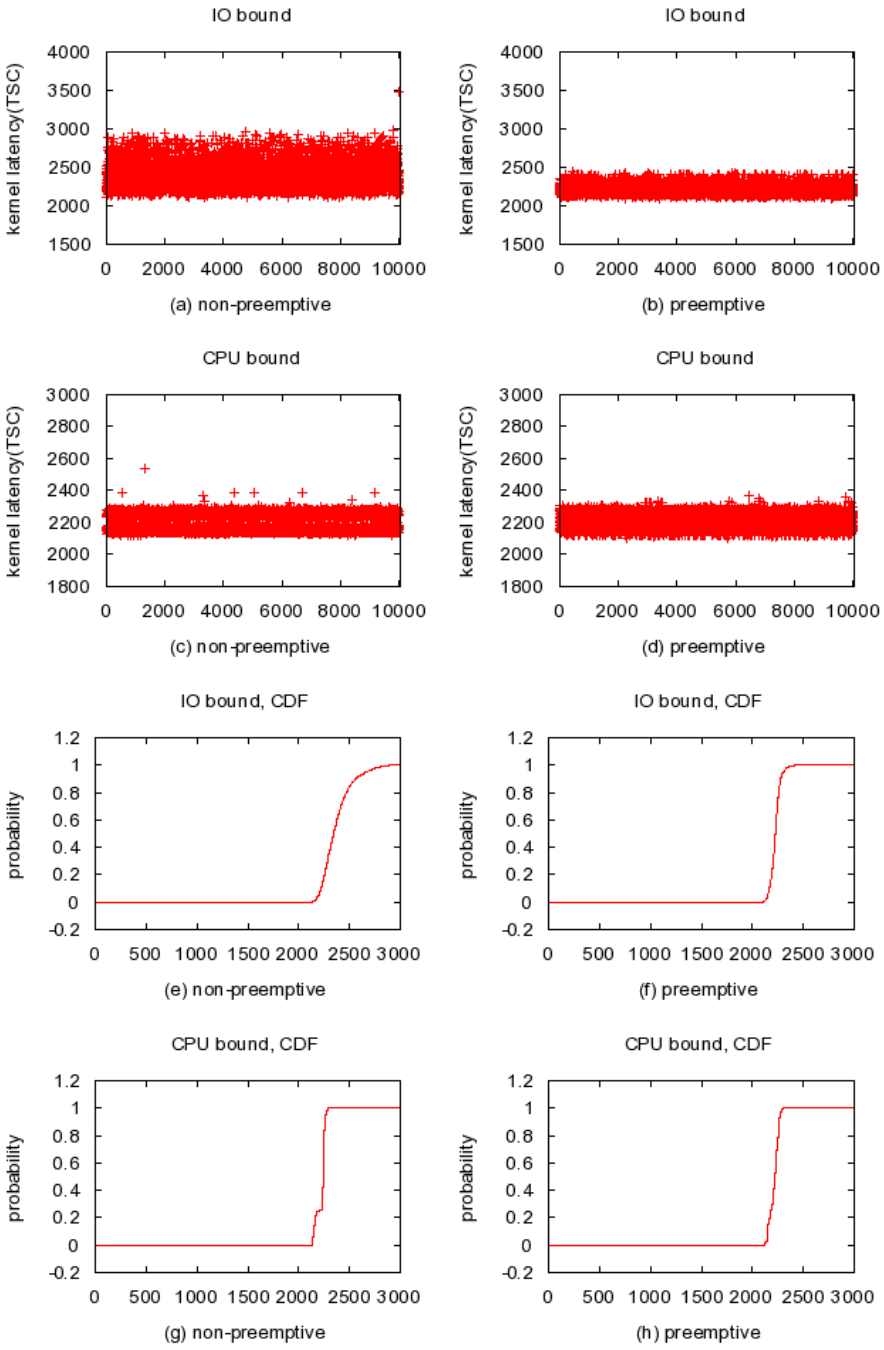


Fig. 3. Experiment data of kernel latency

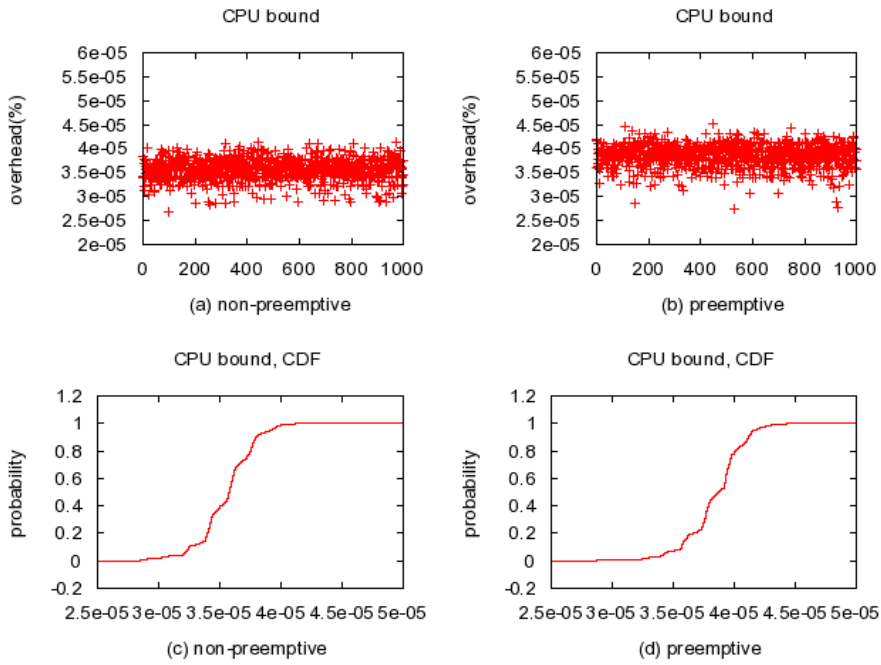


Fig. 4. Experiment data of kernel overhead

5.3 Evaluating Kernel Overhead

The kernel overhead of Barrelfish is also measured in this paper(see Figure 4). x-axis indicates number of experiments and overhead(cycles) respectively in Figure 4(a)-(b) and Figure 4(c)-(d). When running CPU-bound workload, preemptive kernel in this paper may even performs a little worse than non-preemptive kernel in aspect of average kernel latency(see Figure 3(c)-(d)), the main reason is that preemptive kernel may execute more code related with preemption prior to interrupt handling. When it comes to kernel overhead, preemptive kernel is also a little worse than non-preemptive kernel because of the same reason. Figure 4(a) and Figure 4(b) show the result. But Figure 4(c) and Figure 4(d) give a clear representation that the worse overhead in preemptive kernel is so negligible that it's completely acceptable.

6 Conclusions and Future Work

In this paper we have introduced a multi-/many-core oriented operating system: Barrelfish. The multi-kernel architecture of Barrelfish makes it perform well in both aspects of scalability and predictability, it could be a valuable choice for embedded systems basing on multi-/many-core platforms if preemption is supported.

We first analysis the architecture of Barrelfish, and gives a model of the kernel control path. Then, we prove that preemption support in Barrelfish kernel will give rise to a better kernel latency. According the model, we redesign some component of Barrelfish and realize a preemptive kernel. Finally, experiments show that the preemptive kernel in this paper can improved the performance of kernel latency as much as 20% or even more(Figure 3(a)(b)), which is an encouraging result.

Next, a deep optimization and research will be carried out, we are also planning to port Barrelfish to the MIC(Many Integrated Core) platform from Intel, which consists of as many as 60 cores(240 hard-threads). Basing on the architecture of MIC, topology aware task mapping, real-time scheduling, power management and optimized message passing are all meaningful topics to engage in.

References

1. Asanovic, K., Bodik, R., Catanzaro, B., Gebis, J., Husbands, P., Keutzer, K., Patterson, D., Plishker, W., Shalf, J., Williams, S., Yelick, K.: The Landscape of Parallel Computing Research: A View from Berkeley. Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, 18(2006-183):19 (December 2006)
2. Vajda, A., Brorsson, M.: Programing Many-Core Chips. Springer (2011) ISBN 144 1997385, 9781441997388
3. Held, J.: "Single-chip cloud computer", an IA tera-scale research processor. In: Proceedings of the 2010 Conference on Parallel Processing (May 2011)
4. Liu, B.-W., Chen, S.-M., Wang, D.: Survey on Advance Microprocessor Architecture and Its Development Trends. Application Research of Computers (2007)
5. Baumann, A., et al.: The Multi-kernel: A new OS architecture for scalable multicore systems. In: Proceedings of the 22nd ACM Symposium on OS Principles, Big Sky, MT, USA (October 2009)
6. Schüpbach, A., Peter, S., Baumann, A., Roscoe, T., Barham, P., Harris, T., Isaacs, R.: Embracing diversity in the Barrelfish manycore operating system. In: Proceedings of the 1st Workshop on Managed Multi-Core Systems (2008)
7. Baumann, A., et al.: Your computer is already a distributed system. Why isn't your OS? In: Proceedings of the 12th Workshop on Hot Topics in Operating Systems, Monte Verità, Switzerland (May 2009)
8. Bryant, R., Hawkes, J.: Linux scalability for large NUMA systems. In: Ottawa Linux Symp., Ottawa, Canada (July 2003)
9. Liu, C.L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. J. ACM 20(1), 46–61 (1973)
10. Davis, R.I., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM Comput. Surv. 43(4), Article 35, 44 pages (2011)
11. Burns, A.: Scheduling hard real-time systems: a review. Software Engineering Journal (May 1991)
12. Wilhelm, R., et al.: The Worst-Case Execution Time Problem: Overview of Methods and Survey of Tools. ACM Transactions on Embedded Computing Systems, 1–53 (April 2008)
13. Betti, E., Bovet, D.P., Cesati, M., Gioiosa, R.: Hard real-time performances in multiprocessor-embedded systems using asmp-linux. EURASIP Journal on Embedded System, 10:1–10:16 (April 2008)

14. Sacha, K.M.: Measuring the Real-Time Operating System Performance. In: Proceedings of Seventh Euromicro Workshop on Real-Time Systems, pp. 34–40 (1995)
15. Bovet, D.P., Cesati, M.: Understanding the Linux Kernel, 3rd edn. O'Reilly (November 2005) ISBN 0-596-00565-2
16. Klein, G., Elphinstone, K., Heiser, G., et al.: seL4: Formal verification of an OS kernel. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles (October 2009)
17. Sauermann, J., Thelen, M.: Realtime Operating Systems: Concepts and Implementation of Microkernels for Embedded Systems (1997)
18. Ramamritham, K., Stankovic, J.A.: Scheduling Algorithms and Operating Systems Support for Real-Time Systems. Proceedings of the IEEE 82(1), 55–67 (1994)
19. Franke, H., Nagar, S., Kravetz, M., Ravindran, R.: PMQS: scalable Linux Scheduling for high end servers. In: Proceedings of the 5th Annual Linux Showcase Conference (November 2001)
20. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation (May 2007)
21. AMD: AMD64 Architecture Programmer's Manual. AMD Corporation (September 2006)

Interrupt Modeling and Verification for Embedded Systems Based on Time Petri Nets

Gang Hou, Kuanjiu Zhou*, Junwang Chang, Rui Li, and Mingchu Li

School of Software, Dalian University of Technology (DUT), Dalian 116620, China
zhoukj@dlut.edu.cn

Abstract. The embedded systems are interrupt-driven systems, but the triggered methods of interrupts are with randomness and uncertainty. The behavior of interrupt can be quite difficult to fully understand, and many catastrophic system failures are caused by unexpected behaviors. Therefore, interrupt-driven systems need high quality tests, but there is lack of effective interrupt system detection method at present. In this paper, a modeling method of interrupt system is firstly proposed based on time Petri nets, which has ability of describing concurrency and time series. Then the time petri nets are transformed into timed automata for model checking. Consequentially, a symbolic encoding approach is investigated for formalized timed automata, through which the timed automata could be bounded model checked (BMC) with regard to invariant properties by using Satisfiability Modulo Theories (SMT) solving technique. Finally, Z3 is used in the experiments to evaluate the effectiveness of our approach.

Keywords: Interrupt Modeling, Bounded Model Checking, Timed automata, Satisfiability Modulo Theories, Time Petri Nets.

1 Introduction

In the process of embedded system development, interrupt technique is always be used to deal with the interaction between the host and peripherals. Interrupt can provide more efficient and convenient solution for synchronous processing, real-time processing and emergency handling, which makes interrupt technique indispensably in the development of embedded systems. But interrupt has a closely connection with the hardware, the environment and the application, which makes the design of interrupt software much more difficult than that of ordinary software. It will easily lead to interrupt nesting error, breakpoint protection error, and switch interrupt timing error, etc. Once a program error is caused by interrupt, it will lead to a series of serious software failures, which are hard to track, and the failures for the system are often fatal. Since the triggered method of interrupt is randomness and uncertainty, which makes the interrupt errors are difficult to check them out through dynamic testing or static code analysis, and bring huge difficulty to the interrupt tests.

* Corresponding author.

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 62–76, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Therefore, we consider using the formal modeling method in the software design phase to construct the interrupt behavior, and then strictly deduct through mathematics method. We believe that the method above will be an effective way to solve the interrupt detection problem.

At present, there are two formal modeling methods for embedded systems, which are Finite State Machine (FSM) and Petri nets. But the FSM cannot describe the concurrent behavior of interrupt, so it has certain limitations. In contrast, Petri net is a perfect formal modeling method with a precise formal definition, a rigorous derivation method, a preferable tool support, especially suitable to describe the system control flow, concurrency and asynchronous behavior. Recent years, there are lots of researches on applying Petri net to embedded systems modeling and verification. Rammig et al.[1] propose a modeling method of dynamically modifiable for embedded real-time systems. Gu et al.[2] provide an integrated approach to modeling and analysis of embedded real-time systems based on timed Petri nets. Costa et al.[3] give a method of Embedded Systems Co-design based on Petri net. Zhang et al.[4] propose a distributed real-time embedded systems scheduling model based on Petri nets. Basu et al.[5] introduces a rigorous system design flow based on the BIP (Behavior, Interaction, Priority) component framework, which is a new effective modeling method. But above methods do not consider the specific interrupt behavior modeling mechanism for embedded system.

In this paper, we propose an interrupt behavior detection method for embedded system based on time Petri net. This method firstly models the interrupt behavior as a formalized time Petri net, and then transforms it into an equivalent automaton model in order to facilitate the follow-up formal verification. Consequentially, we investigate a symbolic encoding approach for formalized timed automata, through which the timed automata could be bounded model checked with regard to invariant properties by using SMT solving technique. Finally, Z3 is used in our experiments to evaluate the effectiveness of our approach.

2 Time Petri Nets Model of Interrupt

2.1 Interrupt Behavior Analysis

An interrupt is a mechanism for pausing execution of whatever a processor is currently doing and executing a pre-defined code sequence called an interrupt service routine (ISR) or interrupt handler. Three kinds of events may trigger an interrupt. One is a hardware interrupt, where some external hardware changes the voltage level on an interrupt request line. In the case of a software interrupt, the program that is executing triggers the interrupt by executing a special instruction or by writing to a memory-mapped register. A third variant is called an exception, where the interrupt is triggered by internal hardware that detects a fault, such as a segmentation fault.

The trigger modes of the three type interrupts are different, but the interrupt response process is basically the same. A complete interrupt handling process should include: interrupt request, interrupt arbitration, interrupt response, interrupt service and interrupt return. All these parts above need to be handled separately in the Petri net modeling process.

Interrupt Request: When an interrupt request occurs, the interrupt source will send an interrupt request signal to CPU. There are two conditions need to be followed when a peripheral sends an interrupt request signal: 1) the work of a peripheral has come to an end; 2) the system allows the peripheral send an interrupt request. If the system does not allow the peripheral interrupt request, it can be blocked by interrupt control register. Therefore, the status of the interrupt control register needs to be considered when we use Petri net to construct an interrupt model.

Interrupt Arbitration: The interrupt request is random, and sometimes multiple interrupt sources may make requests at the same time. But the CPU can only respond to one interrupt request every time, so we must arrange a priority order in advance according to the degree of importance of every interrupt source. When more than one interrupt requests take place, CPU will response one of them according to the priority order. After finishing a high priority interrupt, CPU will response to the low-priority interrupt request. Therefore, we need to introduce priority into the timed Petri net model and achieve the priority response for concurrent state.

Interrupt Response: When an interrupt request is detected, CPU will abort the current program, save the program breakpoints, and execute the interrupt handler automatically. Interrupt response is a process of discovering and receiving interrupt, which is done automatically by hardware. Therefore, it does not require any additional control signal involved in the modeling process.

Interrupt Service: Interrupt service is performed by the ISR (or interrupt handler), which is core content of interrupt event. ISR generally executed in the order, so the order time Petri nets model can be used to model it. But when the ISR is running, there may be a new interrupt request with a higher priority than the current interrupt occurring. CPU needs to stop the current low-priority interrupt handling, turn to respond to the high-priority interrupt, and after the high-priority interrupt processing is complete, it will continue to process the low-priority interrupt. This situation is called "Interrupt Nesting". Therefore, In the Petri net modeling process, we need to introduce the representation of the hierarchical structure to express the interrupt nesting.

Interrupt Return: For hardware interrupt and software interrupt, once the ISR completes, the program that was interrupted resumes where it left off. In the case of an exception, once the ISR has completed, the program that triggered the exception is not normally resumed. Instead, the program counter is set to some fixed location where, for example, the operating system may terminate the offending program. Therefore, we need to design different interrupt handler return status for the three interrupts in Petri net model.

2.2 Time Petri Nets with Priority

The basic concepts and definitions of time Petri nets can be found in [6]. We only introduce a few concepts related to this paper in the following section.

Definition 1: A time Petri net (TPN) is a tuple $<P,T,Pre,Post,M_0,efd,lfd>$ in which:

—$P=\{p_1,p_2,...,p_m\}$ is a finite non-null set of places;

—$T=\{t_1,t_2,...,t_n\}$ is a finite non-null set of transitions;

—$Pre \subseteq P \times T$ is a finite non-null set of input arc, which defines the flow relationship between places and transitions;

—$Post \subseteq T \times P$ is a finite non-null set of input arc, which defines the flow relationship between transitions and places;

—M_0 is the initial identification of Petri net;

—*efd* is the earliest permission time of transitions fired; *lfd* is the latest permission time of transitions fired;

The priorities need to be introduced in time Petri nets for interrupt modeling, forming time Petri nets with priorities (PrTPN). PrTPN extend TPN with a priority relation on transitions, and the definition is as followed.

Definition 2: A time Petri net with priorities (PrTPN) is a tuple $<P,T,Pre,Post,M_0,efd,lfd,Pr,\Sigma,L>$ in which:

—$< P,T,Pre,Post,M_0,efd,lfd >$ is an original time Petri net;

—$Pr \subseteq T \times T$ is the Priority relation, irreflexive, asymmetric and transitive;

—Σ is a finite set of Action, or Label, not containing the silent action ;

—$L:T \rightarrow \Sigma$ is a function call the Labeling function.

For f, g: $P \rightarrow N^+$, $f \geqq g$ means $(\forall p\ P)(f(p) \geqq g(p))$ and $f\{+|-\}g$ maps $f(p)\{+|-\}g(p)$ with every p. A marking is a function m: $P \rightarrow N^+$, $t\ T$ is enabled at m iff $m \geqq efd(t)$, $\varepsilon_N(m)$ denotes the set of transitions enabled at m in net N. $(t_1,t_2)\ Pr$ is written $t_1 \succ t_2$ or $t_2 \prec t_1$ (t_1 has priority over t_2).

Definition 3: An identifier M is a set of distribution for token. For $p\ P$, the identifier is $M(p)$, which means multiple collection. For a specific identifier M, a place p is identified iff $M(p) \neq \varphi$.

Definition 4: The definition for pre-collection of a transition $t\ T$ is $\bullet t=\{p\ P|(p,\ t)\ Pre\}$, which is the input place set of t. The definition for post-collection of t is $t\bullet=\{\ p\ P\ |(t,p)\ Post\}$,which is the output place set of t. Similarly, the definition for pre-collection of a place $p\ P$ is $\bullet p=\{t\ T|(t,p)\ Post\}$, and the definition for post-collection of a place p is $p\bullet=\{\ p\ P\ |(p,t)\ Pre\}$.

Definition 5: For each transition $t\ T$, there is a transition function f. That is $\forall t\ T$, $\exists f$: $\tau(p_1) \times \tau(p_2) \times ... \times \tau(p_a) \rightarrow \tau(q)$, where $\tau(p)$ represent the set of possible type value of p, $\bullet t=\{p_1,p_2,...p_a\}$, $q\ t\bullet$.

Definition 6: For each transition $t\ T$, there is a minimum transition delay *efd* and a maximum transition delay *lfd*, both of them are nonnegative real numbers and meet $efd \leq lfd$. They represent the lower and upper bounds of execution time of the transition function separately.

Definition 7: A pocket of a transition t (where $\bullet t=\{p_1,p_2, ...p_a\}$) is a ordered set of token $b=\{k_1,k_2, ...k_a\}$, where $\forall p_i \in \bullet t$ and $k_i \in M(p_i)$. If transition t T is enabled for one pocket $b=\{k_1,k_2,...,k_a\}$, $et=\max\{r_1,r_2,\ldots,r_a\}$ is a period of time in which t is enable. If transition t T is enabled for one pocket $b=\{k_1,k_2, ...k_a\}$, the earliest fire time $tt-=et+efd$ is the lower bound and the latest fire time $tt+=et+lfd$ is the upper bound. An enabled transition t T can only fire between tt- and tt+, otherwise, it will not become enabled.

Definition 8: For $b=\{k_1,k_2, ...k_a\}$, when an enabled transition t T is fired, it will transform the identifier M into a new identifier M'. The transfer result is as followed: 1) The token in pre-collection $\bullet t$ is removed, that is $\forall p_i \in \bullet t$, $M'(p_i) = M(p_i)-\{k_i\}$; 2) A new token k join into post-collection $t\bullet$, that is $\forall p$ $t\bullet$, $M'(p) = M(p)+\{k\}$. The new value of token k is calculated by transition function. The time delay of token k is $tt= r^*$, where $tt^* \in [tt-,tt+]$; 3) The identifier in the place which has nothing to do with the pre-collection and post-collection remain the same, that is $\forall p$ $P\backslash\bullet t\backslash t\bullet$, $M'(p)=M(p)$.

2.3 Interrupt Model

In view of interrupt behavioral characteristics for embedded systems, we use three types of time Petri nets scenarios to model it.

Case 1: Interrupt Order Execution

At some moment, only one interrupt request occurs, and no high-priority interrupt request break the interrupt response service. In this case, the interrupt handler executes orderly, therefore, can be modeling by ordered time Petri nets (Fig. 1). IRQ is the status of interrupt request; IRP is the status of interrupt response; IS is the status of interrupt service; IRT is the status of interrupt return; the rest places are registers, global variables or mutual resources.

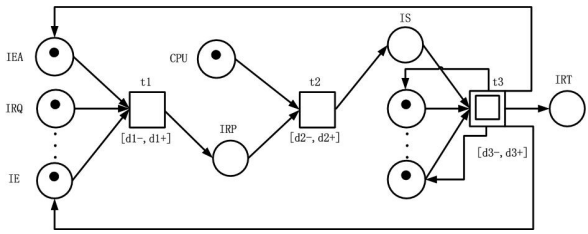


Fig. 1. Ordered time Petri nets for interrupt order execution. This figure shows the modeling method of interrupt order execution based on time Petri nets.

Case 2: Interrupt Nesting Execution

Only one interrupt request occurs at some point, then a higher priority interrupts request coming in the process of the current interrupt response. In this case, current executing interrupt is interrupted, and CPU turns to the execution of the high-priority interrupt request. When the high-priority interrupt execution is complete, CPU returns to the original low-priority interrupt to continue the remaining execution. Therefore, the hierarchical time Petri nets can be used to model interrupt nesting execution (Fig. 2).

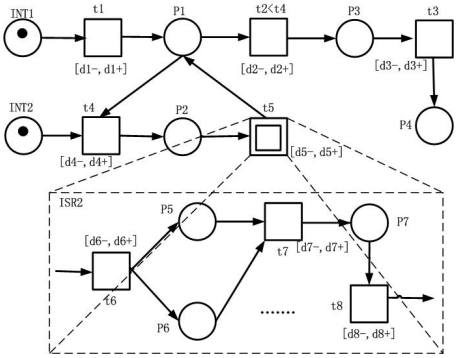


Fig. 2. Hierarchical time Petri nets for interrupt nesting execution. This figure shows modeling method of interrupt nesting execution based on time Petri nets.

Case 3: Interrupt Concurrent Requests

Multiple interrupt requests occur at one moment simultaneously. In this situation, CPU will arbitrate interrupt request based on interrupt priority, respond to high-priority interrupt, and shield low-priority interrupt (or add the low-priority interrupt to the request queue, process it after the high-priority interrupt is finished). Therefore, concurrent time Petri nets model can be used to model interrupt concurrent requests (Fig. 3).

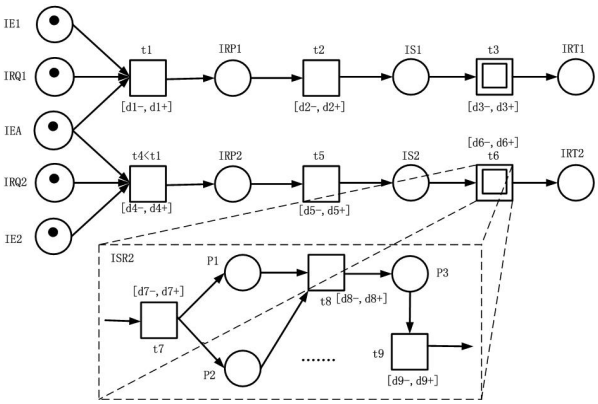


Fig. 3. Concurrent time Petri nets interrupt concurrent requests. This figure shows modeling method of interrupt concurrent request based on time Petri nets.

3 From Time Petri Nets to Timed Automata

There are some methods to solve the transformation from time Petri nets to timed automata. A structural transformation from TPN to NTA is defined in [7]. This transformation builds a timed automaton per transition of the TPN. Based on the time state space exploration, Lime et al. [8] propose a new approach to calculate the so-called

state class timed automata (SCTA) based on extended state class diagram (SCG). In order to analyze and verify the properties of a specific time Petri net model, a transformation method from time Petri nets to timed automata is given in [9]. However, the above methods are lack of priority description and transformation. Based on the above methods, we extend time Petri nets with a priority and propose a new method to transform it into timed automata.

Definition 9: A Timed Automaton (TA) [10] is a tuple $<S,S_0,\Sigma,X,I,T>$ in which:

—S is a finite set of locations;

—S_0 S is the initial location;

—X is a finite set of clocks;

—Σ is a finite set of characters;

—$I:S\rightarrow C(X)$ is the mapping from location to location invariant. $C(X)$ represents the variables constraints of the position.

—$T \subseteq S\times\delta\times C(X)\times 2^X\times S$ is a finite set of transitions or edges.

Priorities: We only consider static priorities here. The definition of TA is extended with a partial irreflexive, asymmetric and transitive priority relation among transitions. The semantics is updated accordingly: a transition can only be taken when no transition with higher priority can be taken at the same time.

In the process of transformation, construct an automaton and a clock for each transition of the time Petri nets. The transformation process includes the following steps:

Step 3.1: Define a label set Σ, which is the set of all the TPN transitions.

Step 3.2: Define global clock $c(c\ \ X)$ according to the symbol timestamp of TPN, which can be used to evaluate system performance by recording the statistics information of time consumption during the simulation.

Step 3.3: Defined local clock $c_i(c_i\ \ X)$,which can be used to count the time of local transition to ensure the transition fire in its earliest and latest time interval.

Step 3.4: For each transition of the time Petri Nets t_i, construct an timed automaton $\overline{t_i}$ in which there are r+1 states including $s_1,s_2,\ldots,s_r$, where r is the number of Pre-collection input token transitions. The Pre-collection is denoted by $pr(t_i)=\bigcup_{p\in\bullet t_i}\cdot p$.

(1) When the Pre-collection of t_i is empty, construct an automaton which contains only two states s_1 and nd.

(2) When t_i does not conflict with other transitions, let r=| $pr(t_i)$ | and construct r edges (s_1,s_2), r edges (s_2,s_3), ……, r edges (s_r,nd). Then assign the corresponding synchronization labels to the corresponding transitions of $pr(t_i)$ for each group of the r edges and assign synchronous label t_i for (nd,s_1) .

(3) When t_i conflicts with the transition t_j, let $C=pr(t_i)-pr(t_j)$, $l=|C|$, $m=|D|$, $n=|pr(t_i)|$. Divide each state of $s_2,\ldots s_n$ into $s_{2,c},\ldots,s_{n,c}$ and $s_{2,d},\ldots,s_{n,d}$. Firstly, define m edges $(s_{2,c},s_{3,c})$, m edges $(s_{3,c},s_{4,c})$,…, m edges $(s_{n,c},nd)$, m edges $(s_1,s_{2,d})$,…, m edges $(s_{n-1,d},s_{n,d})$, where each group of edges carriy with the synchronization labels corresponding to the transitions in D. Secondly, define l edges $(s_1,s_{2,c})$, l edges $(s_{2,d},s_{3,c})$,…, l edges $(s_{n,d},nd)$,

where each group of edges carry with the synchronization labels corresponding to the transitions in C. Thirdly, define 1 edge $(s_{2,c},s_1)$, 1 edge $(s_{3,c},s_{2,d})$,…, 1 edge $(nd,s_{n,d})$, where each edge carries with the synchronization label t_j. Finally, define 1 edge (nd,s_1) carrying with the synchronization label t_i.

Step 3.5: Define a clock $c_i(c_i \ X)$ for each transition t_i of time Petri nets. Define a variable $v_k(v_k \ V)$ for each place p_k of time Petri nets, where v_k is the token value when p_k is marked. The clock c_i is used to ensure that the transitions fire in its earliest and latest time interval.

Step 3.6: Defined $R(e_v)=\{c_i\}$ for each edge of the timed automata $\overline{t_i}$, where define $R(e_v)=\phi$ for other edges of $\overline{t_i}$. Define the invariant $c_i \leq lfd_i$ for state nd so that it can fire before the latest fire time (at the latest fire time).

Step 3.7: Assign clock constraints $efd_i \leq c_i \leq lfd_i$ for the edge $e=\{nd,s_1\}$(with the he synchronization label t_i) of timed automata $\overline{t_i}$. $\forall p_k \in t_i \cdot$, assign $v_k := f_i$ for e.

Step 3.8: Assign variable constraints G_i for the edge $e=\{nd,s_1\}$ of timed automata $\overline{t_i}$ if t_i has the guard G_i. Then construct an edge (nd,nd) without labels, on which add $\overline{G_i}$ ($\overline{G_i}$ is the complement G_i) and $R((nd,nd))=\{c_i\}$.

Step 3.9: If t_i, with guard g_i, has higher priority than t_j, with guard g_j (denoted by $t_i \succ t_j$ or $t_j \prec t_i$ in the original TPN), then replace g_j with $g_j \wedge \neg g_i$ in the transformation process.

Step 3.10: If the transitions in the original TPN is enabled at the initial marking time, let nd be the initial state of the timed automata $\overline{t_i}$. If there are l places initially marked in $\bullet t$ and $0 \leqslant l \leqslant |\bullet t|$, let s_{l+1} be the initial state of $\overline{t_i}$.

4 Model Checking for Timed Automata

There are many encoding methods in model checking area. The initial idea for converting a transition system into a quantifier-free formula and using SAT solving technique [11] to bounded check the formula has been proposed in [12]. A more recent work [13] uses a similar approach to analyze model programs (representing transition systems) but based on SMT. To avoid encoding variants in the model into Boolean type in the process of bounded model checking(BMC) and preprocessing clocks for timed automata (TA), a model checking method of BMC for timed automata based on SMT tools is proposed in [14]. Weiqiang et al.[15] investigate a symbolic encoding approach for STM(State Transition Matrix) design, through which the design could be bounded model checked wrt. invariant properties by using Satisfiability Modulo Theories (SMT) solving technique. We present a new encoding approach for TA with priorities based on the above work.

4.1 Encoding of Timed Automata Model *M*

We demonstrate our encoding approach by considering a TA design M consists of n TAs $A_0,A_1,\ldots, A_n$ and the given bound is denoted by K. In addition, we verify the

security properties and liveness properties of the real-time interrupt systems through the reachability analysis algorithm based on SMT. The invariant properties of the TA model design are represented by LTL(Linear Temporal Logic) formula(eg. *f=EFp|AGp*). We converting a transition system TA and verified properties into a quantifier-free conjunction formula (denoted by *BMC*(*M*,*f*,*K*)), through which the design could be bounded model checked by using Satisfiability Modulo Theories (SMT) solving technique. Next we will give the implementation methods of the encoding and in section 5 the state-of-the-art SMT solver–Z3 is used in our experiments to evaluate the effectiveness of our approach.

Formula for the initial state(s_0,v_0), denoted by step *Init*, representing the initial global state, is written as:

$$Init = Init_0 \wedge Init_1 \wedge ... \wedge Init_{n-1},$$

$Init_j$ denotes the initial position of the *j*-th time automatic, and all initial variable values of this time automatic clock should be set to 0. The concrete formula is defined as follows:

$$Init_j = \bigvee_{s \in S_0^j} (s_0 = s) \wedge \bigvee_{t \in X^j} (t_0 = 0)$$

We define $Const_k(j)$ to denote the variable constraints of the *j*-th timed automata at the *k*-th state. t_k denotes the value of the clock *t* at the *k*-th state. $t_k \geqq 0$ restricts the clock variable non-negative. s_k denotes the position of the system at the *k*-th state. We also should ensure that every TA can only be in one state and the state holds invariant $I^k(s_\mu)$. So $Const_k(j)$ is defined as follows:

$$Const_k(j) = \bigvee_{t \in X^j} (t_k \geq 0) \wedge \bigvee_{s_\mu \in S^j} \{(s_k = s_\mu) \rightarrow \bigvee_{s_v \in S^j \wedge s_v \neq s_\mu} ((s_k \neq s_v) \wedge I^k(s_\mu))\}$$

Based on the encoding of $Const_k(j)$, we could define the global invariant of (s_k, v_k) as follows:

$$Const_k = Const_k(0) \wedge ... \wedge Const_k(n-1), k \in [0, K]$$

When TA is running, there will be two types of transitions occurring to the system. One is the time delay transition, because time is constantly changing all the time. Once the clock variable changes, the whole system TA model will transfer from one state to another. So we define the time delay transition as follows:

$$T_\mu^j(i) = \neg \alpha_\mu^k \wedge (s_k = s_\mu) \wedge (s_{k+1} = s_\mu) \wedge \bigwedge_{t \in X^j} (t_{k+1} = t_k + d_k)$$

TA models the whole real-time interrupt system including all of the dynamic behaviors. When an TA executes some actions, the TA states will make changes. So the other type is the action transition, defined as follows:

$$T_\mu^j = \alpha_\mu^k \wedge (s_k = s_\mu) \wedge (s_{k+1} = s_\mu') \wedge \bigwedge_{t \in \lambda_u^j} (t_{k+1} = 0) \wedge \bigwedge_{t \notin \lambda_u^j} (t_{k+1} = t_k + d_k)$$

As the TA design M consists of n TAs, there must be mutual exclusion in the process of various TAs internal transition. In addition, when a TA is running, each TA can only move one step one time. So we define constrains for each $T_v =< s_v, \alpha_v, \sigma_v, \lambda_v, s_v' >$ to ensures the correctness of the TA design.

$$Exl_{\mu}^{j}(k) = \bigwedge_{T_v \in T \wedge T_\mu \neq T_v} \neg \alpha_v^k$$

In our encoding approach, we use BMC techniques to fight state space explosion. We define the bound as K. $Step_k(j)$ denotes the k-th transition of the j-th TA, where d_k is the clock delay. The complete transition process is defined as follows:

$$Step_k(j) = \bigvee_{T_\mu \in T^k} (Exl_{\mu}^{j}(k) \wedge T_{\mu}^{j}(k) \wedge T_{\mu}^{j} \wedge d_k > 0)$$

The k-steps path of TA is essentially a finite sequence of states. we can use an intuitive and clear expression to describe the whole process of the TA transition as follows:

$$(s_0, v_0) \xrightarrow{d_0} (s_1, v_1) \xrightarrow{d_1} \ldots \xrightarrow{d_{K-1}} (s_K, v_K)$$

Where (s_0, v_0) is the initial state of the path. (s_k, v_k) denotes the reachable k-th state after $d_0 + d_1 + \ldots + d_k$. $Step_k$ describes the k-th transition of the system, including the conjunction of the k-th transition $Step_k(j)$ of each timed automata. Each transition is denoted by a five tuple $T_\mu =< s_\mu, \alpha_\mu, \sigma_\mu, \lambda_\mu, s_\mu' >$. In the k-steps transitions, $Step_k$ $(s_k, v_k) \xrightarrow{d_k} (s_{k+1}, v_{k+1})$ is defined as follows:

$$Step_k = Step_k(j) \wedge \ldots \wedge Step_k(n-1), k \in [0, K-1]$$

Based on $Step_k$, we can get the k-steps path of the timed automata as follows:

$$Path_k = \wedge \bigwedge_{k=0}^{K} Step_k \cdot$$

Finally we can convert the whole timed automata model into logical formulas:

$$[[M]]_k = Init \wedge \bigwedge_{k=0}^{K} Const_k \wedge Path_k$$

4.2 Encoding of Properties *f*

The set of all reachable global states of M is denoted by R_M. An invariant property of M is a state predicate p which holds in all reachable global states of M. First, we use the LTL semantics to define the reachability as f=EFp. The f denotes that during the operation of TA, there is a path, on which there exists certain state that holds p. Based on that we can get $[[f]]_k = \bigvee_{k \in [0,K]} [[p]]_k$, where $[[p]]_k$ can be converted according to the common logical operators.

The security property is in contrast to the reachability. *f*=*AGp* denotes that p holds true in all the states of all paths in the state transition process of timed automata. In this condition, we can negate the *f* to get the corresponding reachability property and verify $f' = EF\neg p$. If f' is satisfied, we can say that there is a path, on which there exists certain state that holds $\neg p$, which means that *f* is false. On the contrary, if the property f' is unsatisfied, *f* is true until the timed automata run *k*-steps.

When we complete the encoding of *M* and *f*, the final logical formula is defined as follows:

$$BMC(M, f, K) = (\bigwedge_{k=0}^{K}[[M]]_k) \wedge (\bigwedge_{k=0}^{K}[[\neg f]]_k)$$

5 Implementation and Experiment

SMT [16] tools and SAT tools could be used to testify whether a problem is satisfied with certain propositional logic. But there are differences, SAT [17] tool is only able to solve logical proposition which contains only Boolean variables. But according to certain theory and logic, SMT could solve a wider range of propositional logic problems. These problems contain integer or real number type variable. This advantage provides a convenience for bounded model checking: there is no need to code model variables into Boolean variables, what we should do is to apply logical propositions to SMT tool which contain integer or real number variables. In this paper, we choose Z3 as the tool in the model checking process.

The timed automata model *M*, the character that is about to be verified *f*, the SMT tool and the setting expanded maximum step *k* are the input of the verification procedures. For each *k*<*K*, we can get the corresponding logical formulas *BMC*(*M*,*f*,*K*), and apply it to Z3.

ARM CortexTM-M3 is a 32-bit microcontroller, used in industrial automation and other application fields. It carries with a system timer called SysTick. The timer can be used to trigger the ISR, and be executed once every millisecond. Fig. 4 is a design instance based on the processor platform (given only the key parts). The main

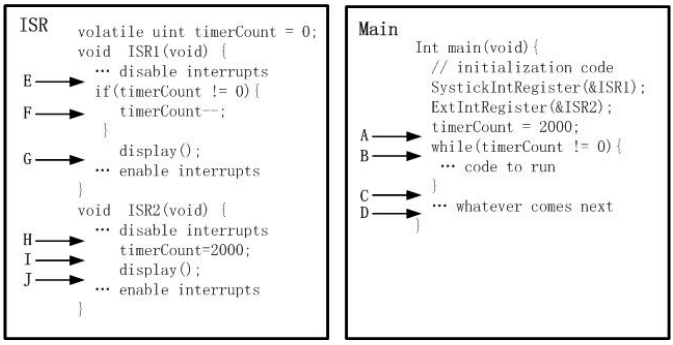


Fig. 4. Design instance of the interrupt. This figure shows the tasks of ISRs and Main function.

function is used to complete the main task of the system. ISR1 completes the task that counts down per millisecond from the initial value until the counter reaches 0 and then stops the countdown. ISR2 is an external key interrupt service routine which can complete the counter reset. External key interrupt has a higher priority than the interrupt timer interrupt, so it can interrupt the timer interrupt so as to complete the interrupt nesting.

Modeling the above instance using the time Petri nets based on our proposed interrupt modeling method (detailed in section 2), the modeling results are shown in Fig. 5.

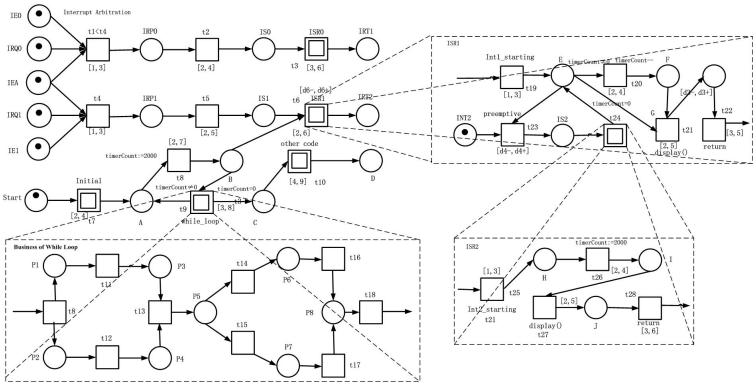


Fig. 5. Time Petri nets model of the design instance. This figure shows the interrupt behavioral characteristics using the time Petri nets.

We describe the process of transformation from time Petri nets to timed automata in section 3. Due to space limitations, we only give several specific transformations of the timed automata model, as Fig. 6.

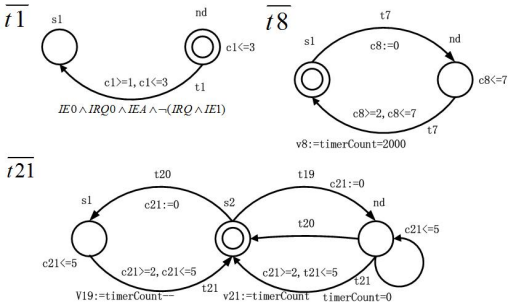


Fig. 6. time automata model transformation of the design instance. This figure shows several specific transformation results of timed automata.

We consider three kinds of invariant properties for the design instance. These properties are commonly desired by industrial practitioners for a TA design.

Reachability: We want to know that whether there is a finite path from s_a to $s_{a'}$ for certain TA, e.g.:

$$EPath_A(s_a^A, v_a^A) \rightarrow .... \rightarrow (s_{a'}^A, v_{a'}^A)$$

Two sample reachability properties of the above instance are declared as follows:

$$reachability1 = A \rightarrow ... \rightarrow C, reachability2 = E \rightarrow ... \rightarrow G$$

Static Constraints: Static constraints demand certain correlation between status of different TAs. Such a correlation can be in the form of, e.g., TA *B* is in status s_b if TA *A* is in status s_a, e.g.:

$$\forall (s,v) \in R_M, (s_a^A, v_a^A) \Rightarrow (s_b^B, v_B^B)$$

When a design *M* involves multiple (or large amount of) TAs, it is not quite easy for designers to maintain an overall image of the design and guarantee the satisfaction of it to static constraints (and dynamic constraints as well, to be introduced next).

Two sample static constraint properties of the above instance are declared as follows:

$$static1 = F \Rightarrow B, stactic2 = I \Rightarrow B$$

Dynamic Constraints: Dynamic constraints demand certain correlation between status-change in one TA with a specific status of another TA. Such a correlation can be in the form of, e.g., when the status of TA *A* has just changed from s_a to $s_{a'}$, the status of STM *B* should be s_b, i.e.:

$$\forall (s,v) \in R_M, \{(s_a^A, v_a^A) \xrightarrow{d} (s_{a'}^A, v_{a'}^A))\} \Rightarrow (s_b^B, v_b^B)$$

We show a dynamic constraint property of the instance as follows:

$$dynamic = (E \xrightarrow{d} F) \Rightarrow B$$

Next, we encode the instance model based on our proposed encoding approach, and transform the generated BMC logic formulas into SMT-LIB which is the input language of Z3 solver. Then we use Z3 4.3.0 (running on Windows 7, 2.8GHz, 4GB RAM) to check these properties. We intendedly introduced errors to this system for demonstration purpose. Table 1 is the results of the one with errors, and Table 2 is the results of the correct.

Table 1. Results of checking the original design instance. This table shows the design instance with errors.

Property	Step	Verdict	Time(s)
reachability1	23	unsat	2.14
reachability1	24	sat	2.21
reachability2	21	unsat	2.01
reachability2	22	sat	2.12
static1	20	unsat	2.45
static1	21	sat	2.53
static2	26	unsat	2.75
static2	27	sat	2.81
dynamic	25	unsat	2.44
dynamic	26	sat	2.51

Table 2. Results of checking the revised design instance. This table shows the correct design instance.

Property	Step	Verdict	Time(s)
reachability1	50	unsat	4.61
reachability2	50	unsat	4.58
static1	50	unsat	4.76
static2	50	unsat	4.5
dynamic	50	unsat	4.85

If the result is "***unsat***", then increase the value of k until the result is sat or k reaches the maximum value K. If the result is "***sat***", for reachability, this property is established. If the result is still "***unsat***" while k is not less than K, this shows that in K steps, the property is not satisfied or security is satisfied.

6 Discussions and Conclusions

In this paper, we propose an embedded system interrupt behavior modeling method based on time Petri nets. This method can effectively depict the nested interrupt and concurrent behavior. In addition, in order to verify the interrupt model, with the help of the mature technology in automaton model validation, we put forward a method of transforming the time Petri nets to timed automata, and then use SMT model checking method with BMC strategy for validation. According to the different modeling granularity, the verification of the properties is also different.

There are some specific details that need to be improved. Such as the BMC conversion formula is not optimized in the section of model encoding and validation. In addition, we didn't give the proofs for the transfer method from time Petri nets to timed automata. We will further improve the research above in the future.

Acknowledgments. This work is supported by the National Nature Science Foundation of China under Gant No. 61272174.

References

1. Rammig, F., Rust, C.: Modeling of dynamically modifiable embedded real-time systems. In: Proc. of the Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, p. 28 (2003)
2. Gu, Z., Shin, K.G.: An Integrated Approach to Modeling and Analysis of Embedded Real-time Systems Based on Timed Petri Nets. In: Proc. of the 23rd International Conference on Distributed Computing Systems, pp. 350–359 (2003)
3. Costa, A., Gomes, L.: Petri net Splitting Operation within Embedded Systems Co-design. In: Proc. of 5th IEEE International Conference on Industrial Informatics, pp. 503–508 (2007)
4. Zhang, H., Ai, Y.: Schedule Modeling Based on Petri Nets for Distributed Real-time Embedded Systems. Computer Engineering 32, 6–8 (2006)

5. Basu, A., Bensalem, S., Bozga, M., et al.: Rigorous system design: the BIP approach. Mathematical and Engineering Methods in Computer Science, pp. 1–19. Springer, Heidelberg (2012)
6. Merlin, P.M.: A study of the recoverability of computing systems. Ph.D. dissertation, University of California, Irvine (1974)
7. Cassez, F., Roux, O.H.: Structural translation from time Petri nets to timed automata. Journal of Systems and Software 79, 1456–1468 (2006)
8. Lime, D., Roux, O.H.: Model checking of time Petri nets using the state Class timed automaton. Journal of Discrete Event Dynamic Systems—Theory and Applications (DEDS) 16, 179–205 (2006)
9. Chuanliang, X.: A Translation Method from Time Petri nets to Timed Automata. Journal of System Simulation 20, 6–8 (2009)
10. Alur, R.: Timed automata. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 8–22. Springer, Heidelberg (1999)
11. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Handbook of Satisfiability. IOS Press, Fairfax (2009)
12. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
13. Veanes, M., Bjørner, N., Raschke, A.: An SMT Approach to Bounded Reachability Analysis of Model Programs. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 53–68. Springer, Heidelberg (2008)
14. Xiaoliang, W.: Bounded model checking of timed automata based on Yices. Computer Engineering and Design 31, 126–129 (2010)
15. Weiqiang, K., Shiraishi, T., Katahira, N., Watanabe, M., Katayama, T., Fukuda, A.: An SMT-based Approach to Bounded Model Checking of Designs in State Transition Matrix. IEICE Transactions on Information and Systems 94, 946–957 (2011)
16. Barrett, C., Moura, L., Stump, A.: Design and results of the first satisfiability modulo theories competition. Journal of Automated Reasoning 35, 373–390 (2005)
17. Le Berre, D., Simon, L.: The essentials of the SAT 2003 competition. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 452–467. Springer, Heidelberg (2004)

Pruning False Positives of Static Data-Race Detection via Thread Specialization

Chen Chen[1,2], Kai Lu[1,2], Xiaoping Wang[1,2], Xu Zhou[1,2], and Li Fang[3]

[1] National Laboratory for Parallel and Distributed Processing,
National University of Defense Technology, Changsha, P.R. China
[2] College of Computer, National University of Defense Technology, Changsha, P.R. China
[3] Information Center, National University of Defense Technology, Changsha, P.R. China
{chenchen2011,kailu,xiaopingwang,zhouxu,lifang}@nudt.edu.cn

Abstract. Static data-race detection is a powerful tool by providing clues for dynamic approaches to only instrument certain memory accesses. However, static data-race analysis suffers from high false positive rate. A key reason is that static analysis overestimates the set of shared objects a thread can access. We propose thread specialization to distinguish threads statically. By fixing the number of threads as well as the ID assigned to each thread, a program can be transformed to a simplified version. Static data-race analysis on this simplified program can infer the range of addresses accessed by each thread more accurately. Our approach prunes false positives by an average of 89.2% and reduces dynamic instrumentation by an average of 63.4% in seven benchmarks.

Keywords: Static analysis, false positive, data-race detection, specialization.

1 Introduction

Multithreaded programs have weak reliability due to concurrent errors, which are mostly caused by data races. A remarkable design goal of multithreaded programs is data-race-freedom. To this end, many data-race detection approaches are proposed, including static and dynamic ones. These approaches try to locate data races accurately and assist developers in fixing these errors.

Recently, an alternative way to deal with data races is dynamically tolerating them. For example, deterministic multithreading systems always produce the same output for the same input, even when the program contains data races. CoreDet [1] is such a deterministic software framework, which instruments shared-memory accesses to enforce a deterministic shared-memory accessing order. CoreDet instruments all shared-memory accesses except those are known to be thread-local. However, it is sufficient to only instrument data-race-involved memory accesses, which can be detected by a sound static data-race detector.

Unfortunately, static analysis tool is inherently conservative, hence a large fraction of warnings produced by the static data-race detector are false positives. Tracking these potential data races can still be costly. In order to reduce overhead, it is critical

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 77–90, 2013.
© Springer-Verlag Berlin Heidelberg 2013

to reduce false positive rates of static data-race detector. However, pruning false positives without compromising soundness remains an open problem.

Chimera [2] uses static analysis to detect potential data races and instruments them with *weak locks* for deterministic replay. It employs a sound hybrid program analysis to increase the granularity of weak locks, and reduces the cost from 53x to 1.39x. However, Chimera does not address false alarms. Besides, although efficient in deterministic replay systems, weak locks may cause frequent global synchronizations in deterministic multithreading systems. In fact, false alarm pruning is a more general approach to reduce the overhead of deterministic systems.

Schedule specialization [3] reduces false alarms of static analyses by transforming a program toward a schedule. After transformation, the control flow and data flow of the program are simplified. Hence, the precision of static analyses on the transformed program is improved. However, in order to specialize a program, developers have to record a set of schedules at first and then enforce these schedules at runtime for soundness.

In this paper we present a novel approach called *thread specialization* to prune false positives produced by static data-race detection. Our insight is that in most multithreaded programs the control flow mainly depends on the number of worker threads. For instance, the divide-and-conquer applications divide work among threads according to the number of worker threads. Each thread accesses data according to the thread ID assigned. If we can statically fix the number of worker threads and the ID each thread assigned, the control flow and data flow would be simplified. Based on this insight, our approach specializes the program toward a fixed worker thread count. By analyzing the specialized program, we vastly improved the precision of static data-race detection. The result of analysis is sound as long as we run the program with the analyzed thread counts.

One of the limitations of our approach is that we have to analyze the program for every thread count. We consider that the set of thread count need to be analyzed is small because the number of cores on commodity hardware is generally less than sixty four. Besides, all programs we evaluated achieve peak performance when the number of worker threads is less than or identical to the number of cores.

Based on thread specialization, we further propose two additional analyses to prune remaining false positives. First, we present *Code Region Analysis* (CRA), which categorizes all code regions into *single-threaded regions*, *non-parallel regions*, and *parallel regions*. Single-threaded regions, for instance, the initialization of some programs, can only be executed in single-threaded mode. Thus, accesses in single-threaded code regions do not race with any other accesses. Non-parallel regions cannot be executed simultaneously by two or more active threads. Hence, accesses in non-parallel code regions do not race with any other accesses in the same region. Any other code regions are conservatively considered as parallel regions, which can be executed by multiple threads simultaneously.

Second, we propose *Phase Analysis* (PHA). It is common that the computation of multithreaded programs consists of multiple phases. At the boundary of each phase, barrier synchronization is taken to make sure that every thread runs in lockstep. Hence, any data race between two phases is a false positive, as any two phases cannot

be executed concurrently. We soundly identify multiple phases in parallel regions. Using the results of phase analysis, we prune any data race between two phases.

The rest of this paper is organized as follows. Section 2 analyzes the false positives produced by static data-race detector. Section 3 introduces the thread specialization approach in detail. In Section 4 we evaluate our experimental setup and results. Section 5 surveys related work. Finally, Section 6 concludes this paper.

2 Static Data-Race Detection

RELAY [4] is a static data-race detector, which scales to millions of lines of code. However, a lot of warnings produced by RELAY are false positives. In this section, we first give an overview of RELAY detection algorithm. Then we analyze the false positives produced by RELAY.

2.1 RELAY

RELAY is a lockset-based data-race detector. A lockset is a set of locks held by program at every program point. A lockset analysis statically computes the lockset of each program point. If two accesses from different threads to the same shared object (1) may happen concurrently, (2) at least one access is a write, and (3) the intersection of the locksets at each point is empty, then a data-race warning is generated.

RELAY improves the lockset algorithm by computing relative lockset. Relative lockset is the set of locks being held at each point relative to the function entry point. As relative lockset is independent to the calling context, the relative lockset for different functions can be computed in isolation. RELAY runs a bottom-up relative lockset analysis over the call graph. Once the lockset analysis for a function is done, the access summary of that function is computed immediately. The access summary contains the information of all accesses performed by a function, including the shared object being accessed, the relative lockset at the point where access happen and the kind of access (either a read or a write). During the bottom-up analysis, once a function call is processed, RELAY composes the summaries of callee and caller functions.

After the bottom-up analysis, RELAY iterates the access summaries of every thread entry point. RELAY assumes that every thread entry point, except the *main* function, can start multiple active threads. All these threads may run concurrently. RELAY searches for any potential racy access pair between different threads. Such racy pair may access the same object, at least one access is a write, and the must-hold locksets do not overlap.

2.2 Data-Race Warnings

We evaluate the effectiveness of RELAY by using seven benchmarks which are listed in Table 1. These benchmarks cover different application fields, including network, desktop, and scientific applications. Further, we instrument the reported data races and evaluate the performance overhead by counting the dynamic instrumentation number. The evaluation inputs are listed in the last column of Table 1.

Table 1. Benchmarks and inputs used for evaluating RELAY. The number of LOC is measured for the CIL representation.

Benchmark		LOC	evaluation inputs
network	aget	1.2K	4 workers, download a 5.3MB file
desktop	pfscan	2.1K	4 workers, scan "exe" from klee [5] source code tree
scientific	lu	1.5K	4 workers, 512 * 512 matrix, 16 * 16 element blocks
	fft	1.4K	4 workers, 2^{10} data points, no inverse FFT check
	radix	1.3K	4 workers, 262144 keys, no sanity check
	water-n	2.5K	4 workers, 512 molecules, 3 steps
	ocean	5.3K	4 workers, 258*258 grid, 1e-07 error tolerance

Table 2. The number of warnings generated by RELAY, the number of racy instructions involved and the proportion of instrumentation to total memory accesses

Benchmark	warnings	racy instructions	% of instrumentation
aget	23	42	20.6
pfscan	19	51	0.3
lu	16	84	75.6
fft	24	138	99.1
radix	29	89	49.4
water-n	115	434	99.8
ocean	77	1824	99.8

The number of warnings generated by RELAY is shown in Table 2. A warning consists of multiple data races which access the same shared object and have the same access pattern (either read-write or write-write). Therefore, a warning may contain multiple racy instructions. Table 2 also lists the number of racy instructions. The last column in Table 2 gives the proportion of the dynamic instrumentation to all shared-memory accesses. From Table 2 we can see that the percentage of dynamic instrumentation is large, especially in scientific applications. The most important reason is that most memory accesses in the hot-spot are identified as potential data races. However, a large fraction of them are false positives. Pruning such false positives can drastically reduce the instrumentation overhead.

2.3 Analysis of False Positive

Unsurprisingly, manual inspection on these warnings reveals that the vast majority of them are false positives. Most of them can fell into a few categories described below.

Single-Threaded Section: The execution of multi-threaded programs usually contains some single-threaded sections. During such section only one thread is running. The most common pattern is the initialization and finalization of applications. In initialization and finalization sections a program often accesses the global objects without acquiring locks. RELAY conservatively assumes all threads start from the beginning of program execution, and therefore may mistakenly treat accesses in single-thread sections as data races.

Array: RELAY uses Steensgaard [6] and Andersen [7] flow-insensitive context-insensitive points-to analyses, which are efficient and can scale to large programs. However, these analyses are very conservative. For example, they do not distinguish array elements by index. Accessing an array through any index, no matter constant or variable, are treated as accessing the same objet. Such conservative treatment may lead to a lot of false positives, especially in scientific applications.

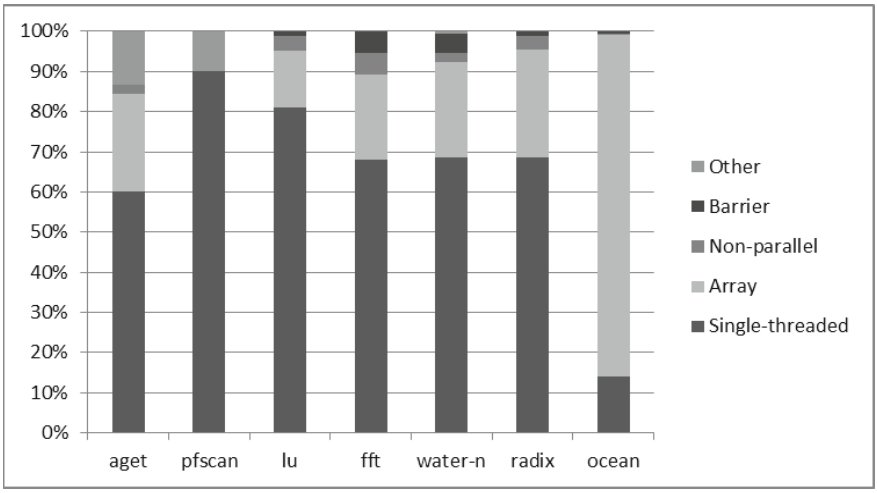


Fig. 1. Distribution of different kinds of false positives according to the number of racy instructions.

Happens-Before Relation: Sometimes it is possible to statically determine the happens-before relationship between two accesses. For example, accesses which dominate a barrier synchronization are happens-before accesses post-dominating that barrier. RELAY does not account for happens-before relationships due to fork/join, barriers, conditional variables, etc. Actually, some data races reported by RELAY can never be executed concurrently.

Non-parallel Region: Some paths would not be executed concurrently because the conditional statements ensure that only one active thread can execute them at the same time.

Unlikely Aliasing: Due to the inaccuracy of alias analysis, RELAY may report false data races which actually access two different objects.

The distribution of different categories is shown in Figure 1. As illustrated in Figure 1, single-threaded false positives are common among all benchmarks. It is very common that in the single-threaded section an application initializes global states without holding locks. For scientific applications, array elements result in a large number of false positives. Every benchmark we evaluated has a certain number of non-parallel false positives. Besides, *barrier* is the most common kind of the *happens-before relation* false positives in scientific applications. They are caused by ignoring of barrier synchronization.

In contrast to the number of racy instructions, we prefer to prioritize false positives according to the number of dynamic instrumentation. Figure 2 shows the distribution of different kinds of false positives according to the number of dynamic instrumentation. Like Figure 1, the distribution varies significantly among benchmarks. However, the proportion of each category is quite different. For scientific applications, the array false positives account for the largest proportion. It is mainly because accesses to array are often enclosed by loops.

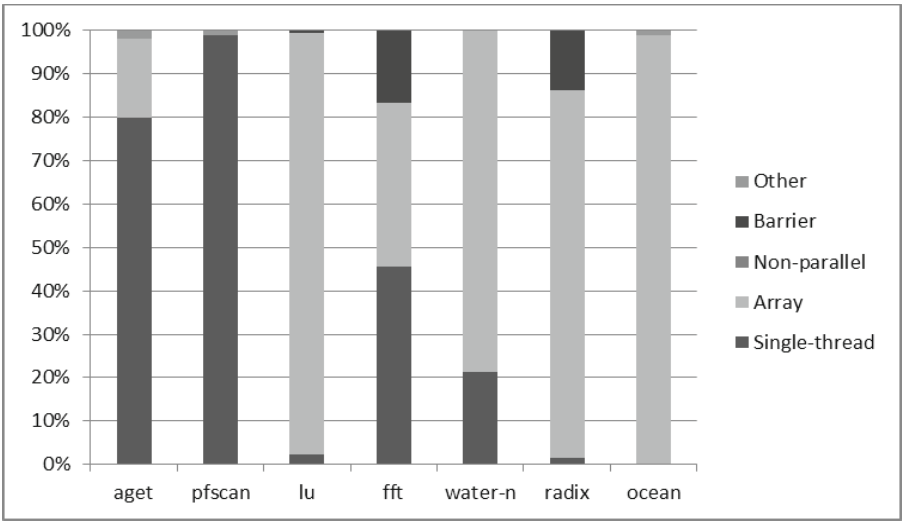


Fig. 2. Distribution of different kinds of false positives according to the number of dynamic instrumentation

3 Thread Specialization

This section proposes thread specialization which specializes a program for improving precision of static data-race detection.

3.1 Overview

Figure 3 shows an illustration example. Using RELAY to analyze this program would produce many false positives. For instance, RELAY reports data-races on *results* in line 20. However, the program divides the array *results* into multiple portions and assigns different portions to each worker thread. Hence, different threads would not access the same entry in the array concurrently. RELAY treats arrays as aggregates, so accesses in line 20 are considered as stores to the same object. Even if an alias analysis can distinguish the accesses to *results* by index, it may still fail to compute that these accesses are disjoint. The reason is that the portion each worker thread accesses depends on *my_id*, which cannot be determined statically.

Intuitively, if we can determine the number of worker threads statically, distinguishing them becomes easy. In addition, if we can statically fix the *my_id* of each thread, we can probably determine the range each thread access.

```
1 : int p, n, global_id, range;
2 : double results[MAX][MAX];
3 : int main(int argc, char *argv[]) {
4 :     int i;
5 :     p = atoi(argv[1]);
6 :     n = atoi(argv[2]);
7 :     range = n / p;
8 :     global_id = 0;
9 :     for (i = 0; i < p; ++i)
10 :         pthread_create(&child[i], 0, worker, 0);
11:     for (i = 0; i < p; ++i)
12:         pthread_join(child[i], 0);
13:     return 0;
14: }
15: void *worker(void *arg) {
16:     pthread_mutex_lock(&global_id_lock);
17:     int my_id = global_id++;
18:     pthread_mutex_unlock(&global_id_lock);
19:     for (int i = 0; i < range; ++i)
20:         results[my_id][i] = compute(i);
21:     return 0;
22: }
```

Fig. 3. A multithreaded program which partitions work among *p* threads. Variables *p* and *n* are two inputs. Variable *n* represents the size of the global array *results,* while *p* specifies the number of worker threads.

To take advantage of these observations, we propose thread specialization, which specializes a program toward a fixed thread count. It does so in two steps. The first step is called *thread count specialization*. It specializes the control flow of the program by cloning functions for every thread so that each statement can be executed by only one thread. Hence, it is easy to distinguish threads statically. Specifically, it first identifies the variable which represents the number of worker threads, and replaces it with the constant value specified. Then, it unrolls the loop when it can, especially the loop which forks or joins worker threads. After that, it clones thread start function for each worker thread, and uses a top-down recursive approach to descend into each thread start function. All functions on the call graph of thread start function are cloned and made thread-local. Figure 4 shows the result after thread count specialization. Loops which contain *pthread_create* or *pthread_join* are unrolled. Thread functions and the *compute* function are cloned, which makes it easy for an analysis to distinguish threads.

The second step is called *thread ID specialization*. In this step, our framework specializes the data flow by fixing the ID assigned to each thread and replacing the variable with constant ID. Figure 4 also shows the result of thread ID specialization.

The variable *my_id* of every function is replaced with constant value. In contrast to identifying the variable which represents thread count, it is not trivial to locate the variable which represents thread ID. Our framework currently relies on developers to specify the value of *my_id*, and then compare the specified value with the actual one at runtime.

```
1 : int n, global_id, range;
2 : double results[MAX][MAX];
3 : int main(int argc, char *argv[]) {
4 :     int i;
6 :     n = atoi(argv[2]);
7 :     range = n / 2;
8 :     global_id = 0;
9 :     pthread_create(&child[0], 0, worker_CLONE0, 0);
10 :    pthread_create(&child[1], 0, worker_CLONE1, 0);
11:     pthread_join(child[0], 0);
12:     pthread_join(child[1], 0);
13:     return 0;
14: }
17: void *worker_CLONE0(void *arg) {
18:     pthread_mutex_lock(&global_id_lock);
19:     global_id++;
20:     pthread_mutex_unlock(&global_id_lock);
21:     for (int i = 0; i < range; ++i)
22:         results[0][i] = compute_CLONE0(i);
23:     return 0;
24: }
25: void *worker_CLONE1(void *arg) {
26:     pthread_mutex_lock(&global_id_lock);
27:     global_id++;
28:     pthread_mutex_unlock(&global_id_lock);
29:     for (int i = 0; i < range; ++i)
30:         results[1][i] = compute_CLONE1(i);
31:     return 0;
32: }
```

Fig. 4. The resultant program after thread specialization, which is done in two steps. (1)Thread count specialization. The loops at lines 9–10 and lines 11–12 in Figure 3 are unrolled because we can determine the loop bounds statically after fixing the thread count, while the other loops are not. Thread function and compute function are cloned twice respectively, making it easy for an analysis to distinguish two worker threads. (2) Thread ID specialization. Variable *my_id* is replaced with constants.

Now, line 20 in the Figure 3 is transformed into two distinct statements in Figure 4 (line 22 and line 30). Each statement is mapped to only one active thread, so it is impossible for the statement to race with itself. Besides, since the indices of the first dimension in each statement are different, we can determine that the ranges each statement access do not overlap, so two accesses to *results* do not race with each other. As a result, the false positives on *results* are pruned.

Unfortunately, sometimes it is still impossible to compute precise loop bound even after thread specialization. Figure 5 shows an example. The loop bound do not directly depend on *my_id*. Although *my_id* can be replaced with constant value, the value of *i* can still have a large range. Fortunately, in such case, we can infer loop bound using inter-procedure range analysis [8]. We leave the implementing of range analysis for future work.

```
for (int i = ranges[my_id].first; i < ranges[my_id].last; ++i)
    results[i] = compute(i);
```

Fig. 5. A loop whose bounds cannot be precisely computed by thread specialization

3.2 Code Region Analysis

The basic thread specialization described above mainly deals with false positives caused by false sharing on arrays. We present two extra analyses based on thread specialization to prune other kinds of false positives. This subsection introduces *Code Region Analysis* (CRA), which classifies code regions into the following three categories.

Single-threaded regions cannot be executed concurrently with other code regions, and at any time only one active thread can execute them. For example, the initialization code region may be executed by the main thread before any worker thread is forked.

Parallel regions can be executed by multiple threads simultaneously. For example, thread start functions are parallel regions.

Non-parallel regions cannot be executed by multiple threads simultaneously, but they can be executed concurrently with other code regions. For instance, *aget* forks a help thread for signals handling. Since there is only one thread to execute the signal handler, the code of the signal handler could not race with itself.

Apparently, accesses in single-threaded region would not race with any other access, and accesses in non-parallel region would not race with each other. Based on this observation, we extend basic thread specialization with CRA to prune false positives further. CRA classifies every statement into one of the three region sets described above. If any access of a reported data race is contained in single-threaded regions, CRA determines it as a false positive. If both accesses of a data-race are contained in the same non-parallel region, CRA also determines it as a false positive.

Code region analysis is conducted simultaneously with thread count specialization. During specialization, functions are cloned if they are reachable from thread fork instructions. All functions which are cloned multiple times are identified as parallel regions. If a function is cloned for only one time, which means it can be reached by only one thread, it belongs to non-parallel region set. In all the remaining instructions, if an instruction is reachable from (1) the start of the main function or (2) the post-dominator of all thread join/cancel instructions, CRA identifies it as single-threaded region. In the end, CRA conservatively classify the remaining instructions into parallel region set.

We modify RELAY to read region information as inputs. Before warning generation, RELAY consults region information and filters out false positives which are relative to single-threaded regions and non-parallel regions.

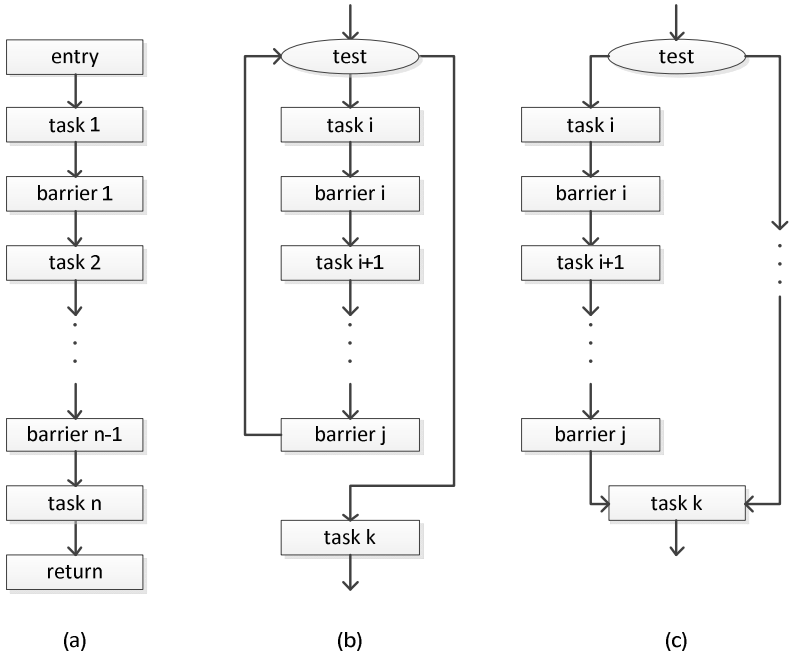


Fig. 6. Structure of parallel computation which is comprised of multiple phases. (a) Straight line (b) Loop (c) Branch.

3.3 Phase Analysis

It is common that a parallel region is comprised of multiple phases. At each phase boundary, every thread calls a barrier synchronization, waiting for other threads to finish the same phase. Figure 6(a) shows an example of such pattern. The computation of worker thread is divided into *n* tasks. Since all threads arrive at barrier before any can exit that point, we can statically infer happens-before relations between tasks. Hence, accesses from different tasks would not race with each other. Based on this insight, we propose the *Phase Analysis* (PHA) for parallel regions. The goal of PHA is constructing an access set for each phase. Suppose a reported data race contains two accesses a_1 and a_2, if there is no access set that contains two accesses both, PHA identifies the data race as a false positive.

Before constructing the access set for each phase, PHA identifies phases at first. It looks into the thread start function and identifies barrier points in those functions. If each of the thread start functions contains exactly the same number of barrier synchronization points (s_1, s_2, …, s_n), and s_{i-1} dominates s_i, then PHA has identified multiple phases. Note that it is possible that some barrier points are enclosed in a loop.

Figure 6(b) shows the example. In this case, the number of barrier points depends on the loop bound, so it is difficult to statically determine if all worker threads contain the same number of barrier points. Fortunately, in all benchmarks we evaluated, the bounds of barrier-involved loops are thread-independent. They depend on global variables which would not be written after initialization section. Hence, PHA can determine that each active thread executes the loop the same times, although it cannot determine the exact number of times. It is similar when the number of barrier points depends on the conditional variables. Figure 6(c) shows the example. If the loop bound or conditional variable depends on the global variables which are not written in parallel regions or non-parallel regions, PHA can still identify multiple phases.

After phase identification, a depth-first traversal of accesses is performed in each phase to construct access sets. These sets are also read by RELAY as inputs. RELAY filters out any false data race that both accesses of it belong to the same set.

4 Evaluation

We implemented thread specialization in OCaml, using CIL [9] as front end. Programs are transformed and analyzed by our framework according to thread counts. Based on the result of the above transformation and analyses, RELAY performs static analysis on the modified source code.

Table 3. Effect of proposed false positive pruning techniques on evaluated programs with respect to the number of racy instructions. Our approaches can reduce reported racy instructions from 80.2% (*fft* and *water-n*) to 99.0% (*ocean*).

Benchmark	Basic	CRA	PHA	Remain
aget	26.7	60.0	0	13.3
pfscan	0	90.2	0	9.8
lu	9.5	82.1	1.2	8.3
fft	6.9	67.9	5.3	19.8
water-n	6.5	68.7	5.0	19.8
radix	25.8	68.5	1.1	4.5
ocean	85.4	14.0	0.6	0
Average	**23.0**	**64.5**	**1.9**	**10.8**

We evaluated the effectiveness of our approach from two perspectives: (1) percentage reduction in the number of static racy instructions and (2) percentage reduction in dynamic instrumentation.

Table 3 shows the reduction in static racy instructions toward four worker threads. Note that the race warnings reported by RELAY is based on specialized programs. For comparison, we map the reported race warnings back to original programs. All warnings on the same group of statements which cloned from the same statement are

combined together. As we can see in Table 3, the basic specialization is effective to reduce race warnings for all evaluated programs except for program *pfscan*. It can reduce racy instruction count from 6.5% (*wat*er-n) to 85.4% (*ocean*). CRA works well on all benchmarks. It reduces the number of racy instructions from 14.0% (*ocean*) to 90.2% (*pfscan*). For scientific applications with barrier synchronizations, for instance, *fft* and *water-n*, PHA is effective.

Table 4. Reduction in the number of instrumentation with various analyses. The evaluation inputs are listed in the last column of Table 1.

Benchmark	Basic	CRA	PHA	Remain
aget	18.2	80.0	0	1.8
pfscan	0	99.0	0	1.0
Lu	0	2.3	0.5	97.1
Fft	0	43.3	15.9	35.8
water-n	2.6	21.4	0	76.0
radix	43.4	1.5	13.8	41.3
ocean	97.2	0	0	2.8
Average	**23.1**	**35.4**	**4.3**	**36.6**

Table 4 shows the reduction in the number of dynamic instrumentation with various analyses. Inputs are listed in Table 1. In general, results in Table 4 for different analyses are consistent with instruction reduction in Table 3. The basic specialization did not work well in *lu*, *fft* and *water-n*. It is mainly because that loop bounds in these applications do not directly depend on thread ID. On average, the combination of three analyses reduces 63.4% instrumentation.

5 Related Work

Static Data-Race Detection for C. There is a large body of work that applies static data-race detection to C programs [4, 10-13]. One main design goal of these works is exploring the tradeoff between precision and scalability. RACERX [10] can scale to large programs. However, it makes several compromises, which lead to missing of real races. RELAY [4] also scales to millions lines of code. Although soundness is guaranteed, a large fraction of warnings generated by REALY are false positives. LOCKSMITH [12] is a more precise approach. However, the performing of heavily constraint solving makes it difficult to scales to large programs. Our approach improves the precision of RELAY without compromising the scalability of RELAY as no costly constraint solving is introduced.

Pruning False Positives of Static Analysis. Z-ranking [14] uses statistic approach to rank warnings of static analysis. Joshi, etc. [15] employ sequential version of a concurrency programs to prune false alarms due to underspecified precondition. RELAY

provides several filters to prune false positives. However, all these filters are simple and unsound in that they can remove real races. We employ thread specialization to prune false positives. Our approach is sound as long as the specified thread counts are enforced at runtime.

Program Specialization. Program specialization specializes a program according to some constant values which are known to be common in the program [16-20]. Schedule specialization [3] specializes a program toward a small set of schedules, thus the results of static analyses on the specialized programs can be more precise. Unlike previous work, our approach specializes a program w.r.t fixed thread counts, thus static analyses can accurately distinguish threads.

6 Conclusion

In this paper, we propose thread specialization for pruning false positives of static data-race detection. Our approach specializes a program according to a set of fixed thread counts in order to make threads distinguishable statically. Hence, the set of shared objects accessed by each thread can be computed more precisely. Moreover, we present *Code Region Analysis* (CRA) and *Phase Analysis* (PHA) based on the basic specialization framework to reduce false positive rates further. The combination of three analyses prunes false positives by an average of 89.2% for seven benchmarks we evaluated. The pruning of false positives can be utilized for reducing instrumentation of shared memory operation. Suppose all reported data races are instrumented, our approach can reduce 63.4% instrumentation on average.

Acknowledgments. This work is partially supported by National High-tech R&D Program of China (863 Program) under Grants 2012AA01A301 and 2012AA010901, by program for New Century Excellent Talents in University and by National Science Foundation (NSF) China 61272142, 61103082, 61003075, 61170261 and 61103193.

References

1. Tom, B., et al.: CoreDet: a compiler and runtime system for deterministic multithreaded execution. In: Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, Pittsburgh, Pennsylvania, USA, pp. 53–64. ACM (2010)
2. Lee, D., et al.: Chimera: Hybrid Program Analysis for Determinism. In: PLDI 2012, Beijing, China (2012)
3. Wu, J., et al.: Sound and precise analysis of parallel programs through schedule specialization. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM (2012)
4. Voung, J.W., Jhala, R., Lerner, S.: RELAY: static race detection on millions of lines of code. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ACM (2007)

5. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (2008)
6. Steensgaard, B.: Points-to analysis in almost linear time. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM (1996)
7. Andersen, L.O.: Program analysis and specialization for the C programming language, University of Cophenhagen (1994)
8. Rugina, R., Rinard, M.: Symbolic bounds analysis of pointers, array indices, and accessed memory regions. ACM SIGPLAN Notices (2000)
9. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
10. Englerand, D., Ashcraft, K.: RacerX: Effective, static detection of race conditions and deadlocks. In: The 19th ACM Symposium on Operating Systems Principles (SOSP) (October 2003)
11. Kahlon, V., et al.: Static data race detection for concurrent programs with asynchronous calls. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ACM (2009)
12. Pratikakis, P., Foster, J.S., Hicks, M.: Locksmith: Practical static race detection for c. ACM Transactions on Programming Languages and Systems (TOPLAS) 33(1), 3 (2011)
13. Sterling, N.: Warlock: A static data race analysis tool. In: USENIX Winter Technical Conference (1993)
14. Kremenek, T., Engler, D.: Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 295–315. Springer, Heidelberg (2003)
15. Joshi, S., Lahiri, S.K., Lal, A.: Underspecified harnesses and interleaved bugs. ACM SIGPLAN Notices (2012)
16. Consel, C., Danvy, O.: Tutorial notes on partial evaluation. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM (1993)
17. Glück, R., Jørgensen, J.: Efficient multi-level generating extensions for program specialization. In: Swierstra, S.D. (ed.) PLILP 1995. LNCS, vol. 982, pp. 259–278. Springer, Heidelberg (1995)
18. Jørgensen, J.: Generating a compiler for a lazy language by partial evaluation. In: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM (1992)
19. Nirkhe, V., Pugh, W.: Partial evaluation of high-level imperative programming languages with applications in hard real-time systems. In: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM (1992)
20. Reps, T., Turnidge, T.: Program specialization via program slicing. In: Danvy, O., Thiemann, P., Glück, R. (eds.) Dagstuhl Seminar 1996. LNCS, vol. 1110, pp. 409–429. Springer, Heidelberg (1996)

G-Paradex: GPU-Based Parallel Indexing for Fast Data Deduplication

Bin Lin[1], Xiangke Liao[1], Shanshan Li[1], Yufeng Wang[2], He Huang[1], and Ling Wen[1]

[1] National University of Defense Technology, Changsha, China
{binlin,xkliao,shanshanli,hehuang,lingwen}@nudt.edu.cn
[2] Zhengzhou Municipal Supervisory Bureau for Quality and Technology, China
yufeng.wang@gmail.com

Abstract. Deduplication technology has been increasingly used to reduce the storage cost. In practice, the duplicate detection upon large on-disk index incurs unavoidable and significant overheads in write operations. Most existing deduplication methods perform single-pass processing, while pay little attention to develop highly parallel methods for the emerging parallel processors. In this paper, we present the design of G-Paradex, a novel deduplication framework that can significantly reduce the duplicate detecting time. Utilizing a prefix tree to organize the chunk fingerprints, G-Paradex is able to do fast deduplicating by using GPU to search the target tree in parallel. Leveraging the inherent chunk locality in writing data stream, we group consecutive chunks and extract the handprints into the prefix tree, aiming at shrinking the index size and reducing the on-disk accesses. Our experimental evaluation based on real-world datasets demonstrate that, compared with the traditional single-pass method, G-aparadex achieves a speedup of 2-4X for duplicate detecting.

Keywords: Parallel data deduplication, GPU, Prefix tree.

1 Introduction

Explosive data growth over the recent years has brought much pressure on the infrastructure and storage management. Faced with ever-growing storage needs and the limited budgets, lots of corporations are exploring deduplication technologies[1]. Data deduplication is a storage optimization technology that reduces the data footprint by looking up the whole storage to eliminate multiple copies of redundancy. Industries such as financial services, pharmaceuticals, and telecommunications have already adopted this technology in their daily work[2].

The majority of deduplication solutions aim to avoid the fingerprint-lookup disk bottleneck and enable more efficient duplicate detection in storage systems. To facilitate duplicate lookup, a single index containing the chunk fingerprints of all the data must be maintained. However, as the data grows, the index overflows the amount of RAM available and must be paged to disk. Faced with severe

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 91–103, 2013.
© Springer-Verlag Berlin Heidelberg 2013

fingerprint-lookup disk bottleneck, existing work mostly use similarity detection techniques to quickly find the similar contents, and exploit the inherent chunk locality in data stream to reduce accesses to on-disk index[3][4][5][6][7]. Although a number of successive efforts have been made to improve the efficiency, state-of-the-art approaches mostly perform single-pass processing over large amounts of data, still incurring many random disk accesses and severely limiting deduplication performance.

Growth in dataset size significantly outpaces the growth of CPU speed and disk throughput. As a result, the efficiency of existing data deduplication techniques is greatly challenged. One general trend for the need of accelerated performance is to develop highly parallel methods on the emerging parallel processors, such as multi-core processors, cell processor, and the general-purpose graphics processing units (GPU)[8]. A common method to speed up the access to data is to partition the data and to perform operations on it in parallel. To this end, we explore a GPU-based framework for parallel deduplication. Though it can quickly locate the duplicate contents, it is non-trivial to deploy modern GPU in deduplication process. GPU has shown to produce performance improvements for computation intensive applications, while it remains challenging for data intensive applications where it has to execute large data transfers before GPU processing. For example, the largest amount of memory available on NVIDIA's GPU is currently 6GB, which is much too small to hold all indexes from a data set of terabytes. Furthermore, the data transfer from host (CPU) to GPU is expensive that the utility of GPU tasks is highly constrained.

In order to address the above challenge, we propose a GPU accelerated deduplication system, G-Paradex, which aims at fully leveraging the parallel processing capability of GPU to reduce the duplicate detecting time. To fully utilize the promising parallelism GPU offered, we adopt a prefix tree index structure to organize the chunk fingerprints for efficiently mapping deduplication process to GPU processors. Meanwhile, we propose technique aiming at optimizing the index structure to shrink the index size and reduce the on-disk accesses. The main contributions are as follows.

First, we adopt the prefix tree structure to index the chunk fingerprints, which efficiently exploring the parallelism of deduplication process on the GPU environment.

Second, leveraging the inherent chunk locality, we group consecutive blocks into super chunk and extract the handprint into the prefix tree. Based on the super chunk, the deduplication process is divided into two phases of fast similarity detecting and further small chunk based deduplicating. With this, it largely shrinks the space usage of prefix tree and significantly reduces disk accesses.

Third, we implement the deduplication in real system of CPU and GPU environment. Our experimental evaluations of G-Paradex, based on several real-world datasets, show that it achieves a speedup of 1.5-3X for duplicate detecting compared with the traditional single-pass method.

The rest of the paper is organized as follows. Sec. 2 gives an overview of related work. Sec. 3 presents the preliminaries of this research. Sec. 4 presents

the baisc architecture and the challenge of the G-Paradex deduplciation system. Sec. 5 describes optimization technique to address the above challenge. Sec. 6 gives our experimental evaluation of G-Paradex based on real-world datasets. And Sec. 7 draws conclusions and outlines our future work.

2 Related Work

2.1 Data Deduplication

The scalability of deduplication systems has become an increasingly important issue, which faces potentially severe fingerprint-lookup disk bottleneck with growing data volumes[4][5][6][3][7]. DDFS is one of the earliest studies aiming at exploiting the inherent backup stream locality to reduce accesses to on-disk index[3]. Sparse indexing is a fast sampling method that chunks the data stream into multiple megabyte segments, and then samples a few segments to do duplicate detection[7]. Extreme Binning is a well-known similarity-based approach that exploits the file similarity to achieve a single on-disk index access for chunk lookup per file[4].

Recently, parallel data deduplication has been gaining increasing attention due to its ability to perform fast duplicate detection. Shredder is a GPU-based accelerator which seeks to overcome the CPU bottlenecks of content-based chunking in a cost-effective manner[9]. P-Dedupe is a fast and scalable deduplication system which composes pipelined and parallel computations to improve the chunking and fingerprinting performance[10]. MD-Approach parallelizes the deduplication process on multi-core processors using the MapReduce programming model[11]. Another work adopts the multiple bloom filters to deduplicate in parallel, and further present a queuing algorithm theoretic analysis to optimize their parallel algorithm on multi-core architectures[12]. In this paper, we focus on the parallelism during the process of duplicate detection after chunking and fingerprinting. Different from the previous work, we adopt an efficient hash index structure to fully explore the parallelism. Moreover, we make the full use of the mechanism of GPU and the inherent characteristic in deduplication process to optimize the use of the hash index and accelerate the write performance.

2.2 Hash Index

In-memory indexing is a well investigated topic since years. Early research mainly focused on reducing the memory footprint of traditional data structures and accelerating search operations. For example, the T-tree reduces the number of pointers of traditional AVL-trees while the CSB+-tree is an almost pointer-free and thus cache-conscious variant of the traditional B+-tree[13][14][15]. These structures usually offer a good read performance.

The prefix tree takes a different approach than the B+-tree[16]. Prefix tree indexing is that the performance of search queries only depends on the depth of the tree, i.e., the maximal length of the indexed strings, and not on the

total number of indexed strings, thus it has been widely used to do similarity search in large-scale dataset. Instead of processing single operations directly, the FAST approach optimized data layout for modern CPU and GPU to achieve fast reads[17]. Peter et al. use prefix tree structure to efficiently explore parallel database operations on GPU[18]. Our G-Paradex is based on the prefix tree and improves it in terms of reducing memory accesses and memory consumption.

3 Preliminaries

As a basis for our new method, we use a prefix tree on the GPU processor. In this section, we will give a detailed description of our hardware model and introduce the concept of adopting prefix tree to do parallel processing.

3.1 General-Purpose Computing on GPU

The Graphical Processing Units (GPU) has emerged as a source of great computing power. The high computational power is stem from the specialized design of GPU, where much more transistors are contributed to simple data processing units (ALUs) rather than used to integrate sophisticated pre-fetchers, control flows and data caches. Figure 1 illustrates a simplified architecture of a GPU. At a high level, GPU has massive Streaming Multiprocessors (SMs), each of which consists of a set of scalar processor cores (SPs). For example, NVIDIA GTX 580 has 512 processors in total. A SM works as SIMT (Single Instruction, Multiple Threads), where the SPs of a multiprocessor execute the same instruction simultaneously but on different data elements. GPU has a different memory hierarchy compared with the CPU, which is organized as multiple hierarchical spaces for threads in execution. The GPU has a high-bandwidth global memory with high latency shared by SMs. Each SM also contains a very fast, low latency on-chip shared memory to be used among its SPs.

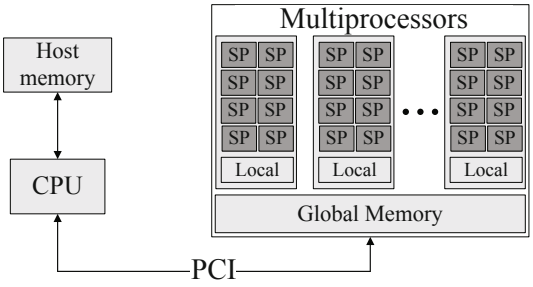


Fig. 1. GPU architecture

The CUDA programming model is the most popular programming models to extract parallelism and scale applications on GPU[19]. It directly exposes the

GPU's execution architecture and memory layout. In this programming model, a host program runs on the CPU and launches a kernel program to be carried out on the GPU device in parallel. More specifically, a GPU task is composed of five stages: data initializing on the host and device; host-to-device data transfer; GPU kernel parallel processing upon the data in the device global memory; device-to-host results transfer and post-processing. CUDA are extensions to the C programming language that makes it conveniently to write native C code and execute it on the GPU. The methods introduced in this paper are implemented using the CUDA programming model exclusively.

3.2 Introduction to Prefix Tree

Figure 2 shows an example of a prefix tree[18]. The basic is that an element consists of a key k_i and a value v_i. The key is inserted into the tree and the value is added to the leaf nodes of the tree. Differ from a binary search tree, the node in the tree does not store the key; instead, the position in the tree carries the information of keys with which it is associated. The path within the tree corresponds to a key k_i which is defined by the absolute value of the key instead of being defined by the relation of the key to the others. A key k_i is split into N even-size sub-keys k_i^n consisting of $b = |k_i|/N$ bits where $|k_i|$ is the number of bits in the key. Thus, each node contains an array of 2^b child node pointers. Starting from the root, the path through the nodes is then defined by following the pointers that are selected by the sub-keys, i.e., the ith sub-key chooses the pointer within the node on level i.

To parallel the search operation, the prefix tree is first partitioned into multiple sub-trees with given a level of m. Each sub-tree starts at a specific tree level k with a root node and covers m levels. At each leaf of a sub-tree, the next sub-tree starts, until the leaf nodes of the whole tree have been reached. Each sub-tree partition is assigned to one thread for lookup execution. Therefore, each level may consist of one or multiple sub-tree partitions, and each thread is responsible for traversing its assigned tree for a given sub-key k_i according to its starting level. Each sub-tree search provides one of three different types of results. The result may be a pointer to its child sub-tree partition. This is the case that the sub-tree is in the middle of the whole prefix tree and it contains a path for substring searched. The result can also be NULL, meaning that the sub-tree does not contain any matching path. The last possible result is a pointer to a value where the sub-tree partition reaches leaf nodes of the original prefix tree. Putting these altogether, we search the fingerprint index in parallel and obtain the final query result by combining the outputs of all threads.

Leveraging the prefix tree, it can efficiently parallel the duplicate detection by mapping it to the high number of cores on a GPU. Prefix tree uses multiple bits per node and, therefore, results in shorter trees and a reduction of the total number of query steps required.

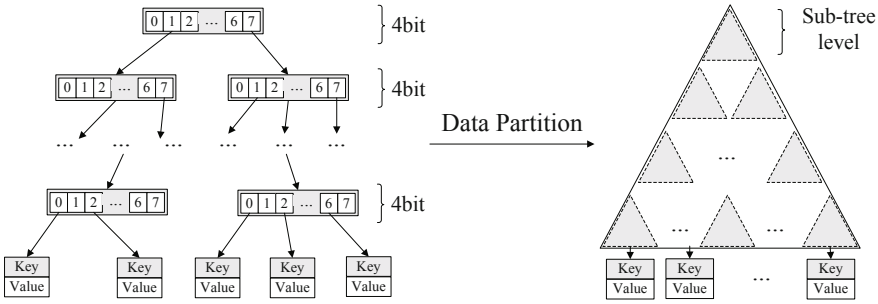


Fig. 2. Parallel Prefix Tree

4 System Overview and Challenges

In this section, we give an overview of the parallel deduplication system by employing prefix tree on GPU hardware. Next, we explain the main challenges in scaling up our basic design.

4.1 System Overview

Figure 3 depicts the workflow of the basic design for G-Paradex deduplicating service. In this initial design, a main thread running in user mode on the host (CPU) drives the GPU-based duplicate detecting operation. The framework is composed of four major modules. Fist, the Prefix Tree Partitioning thread on the host gets tree partitions from the large prefix tree index stored in underlying storage, and fetches them into the memory of the host. After that, the Data Transfer thread allocates global memory on the GPU and uses the DMA controller to transfer input data from the host memory to the allocated GPU memory. Once the data transfer from the CPU to the GPU is complete, the host launches the Deduplicate kernel for parallel detecting computations on the GPU. Once the Deduplicate kernel scans all the keys for the input data, it transfers the intermediate results of each sub-tree partition from the device memory to the host memory. As the all partitions can not be put into memory at one time, the host repeats the above steps until all the prefix tree partitions have been traversed, and builds the final result by scanning all the intermediate results.

The Deduplicate kernel is responsible for performing parallel traversing of the sub-tree partitions presented in the global memory of the GPU. Accesses to the data are performed by multiple threads that are created on the GPU. The data in the GPU memory is composed of many sub-tree partitions, as many as the number of threads. Each thread is responsible for handling one of these sub-tree partitions. For each sub-tree partition, a thread scans the tree in breadth-first way. In particular, each thread searches the corresponding sub-key from its assigned sub-tree partitions. Subsequently, all partitions are traversed in parallel, and the intermediate results are transferred back to a global results collector.

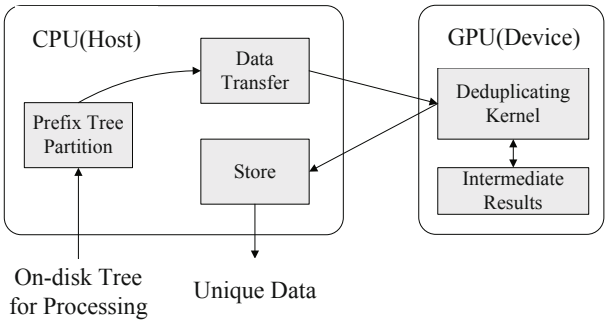


Fig. 3. Basic architecture of G-Paradex

4.2 Challenge of Parallel Deduplication

On-Disk Tree Index Bottleneck. In large-scale storage system, the size of the index would limit system size and increase system cost. Consider a chunk size of 4 KB and a fingerprint size of 32 bytes. Supporting 4 TB worth of unique chunks, would require 32 GB just to store the fingerprints. Despite of massive compression techniques, a prefix tree structure can still be too large to fit into the memory. Therefore, most parts of the tree have to be stored on disk which will incur many disk accesses. Another problem is the hash order in the prefix tree which is determined by the content but not the write order. In this way, the inherent chunk locality in the data stream can hardly be preserved and utilized to optimize the deduplication performance. For example, two chunks A and B, in a write stream appear in approximately the same order throughout multiple write streams with a high probability. DDFS[3] makes full use of this locality characteristic by storing the chunk fingerprints in the order of the backup stream on the disk and preserving the locality in the RAM to accelerate the duplicate detecting. However, the prefix tree structure breaks the locality of writing chunk fingerprints, where the locations of A and B inside tree would be far away from each other according to their hash content value. Thus, it can hardly use the chunk locality to improve the duplication performance.

Device Memory Bottleneck. The fact that the host and device are synchronized, where data has to be loaded into the GPU memory before being processed by the GPU kernel, which represents a serial dependency: sub-tree traversing only starts to execute after the corresponding transfer concludes. This serialized execution may not suit the needs of data intensive applications, where the cost of the data transfer step becomes a more significant fraction of the overall computation time due to the increasing data volumes. More specifically, the on-disk index is usually much more larger than device global memory that it has to execute data transfer from disk to host memory, and then to the GPU memory several times.

5 Super Chunk Organization

The basic design for G-Paradex that we presented in the previous section corresponds to the traditional way of adopting prefix tree index structure on the GPU. The above two challenges is induced by the growing data volumes. In order to avoid the on-disk tree index bottleneck, we propose to organize consecutive small chunks into the super chunk and extract handprint for each super chunk. The prefix tree only stores the fingerprints of super chunks, thus the size of the tree is largely shrank, allowing for holding the whole tree in the memory. In previous example, it only needs 1 GB memory to store all the fingerprints under a super chunk size of 128 KB. The chunks in the same super chunk are consecutive that the inherent chunk locality is likely to be preserved and can be utilized to optimize the deduplication process. Therefore, when deduplicating, we perform parallel similar detection of super chunks firstly, and then utilize the chunk locality in super chunk to accelerate the duplicate finding among similar super chunks like Extreme Bining[4]. To enable this, we use representative fingerprint to describe the super chunk. This is governed by Broder's theorem[20]:

Theorem 1: Consider two sets $S1$ and $S2$, with $H(S1)$ and $H(S2)$ being the corresponding sets of the hashes of the elements of $S1$ and $S2$ respectively, where H is chosen uniformly and at random from a min-wise independent family of permutations. Let $min(S)$ denote the smallest element of the set of integers S. Then:

$$Pr[min(H(S1)) = min(H(S1))] = \frac{S1 \cap S2}{S1 \cup S2} \tag{1}$$

Broder's theorem proves that the probability that the two sets $S1$ and $S2$ have the same minimum hash element is the same as their Jaccard similarity coefficient. So that, if $S1$ and $S2$ are highly similar then the minimum element of $H(S1)$ and $H(S2)$ is the same with high probability. In other word, if two super chunks are highly similar, they share many chunks and hence their representative fingerprint is the same with high probability.

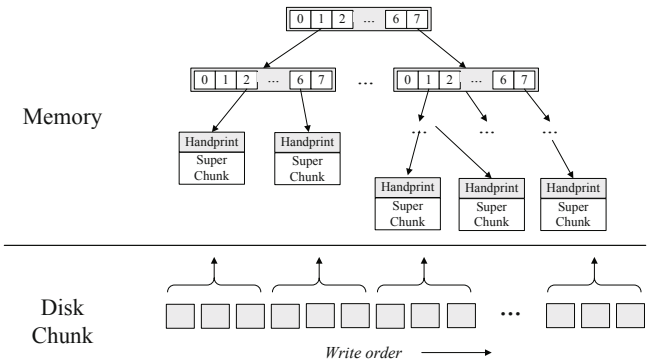


Fig. 4. Super chunk organization

By using the representative technique, the chunk index is split into two tiers between the memory and the disk. Figure 4 shows the structure of the two tiers hash index. One tier is the whole chunk fingerprints stored on disk where chunk fingerprints of the same super chunk is consecutive in write order, and the other is the super chunk fingerprints organized in the form of prefix tree in memory. In-memory tree operations avoids costly accesses to disk, while the fingerprints in a single super chunk are stored in write order as to increase the similarity between super chunks and preserve the locality and further be utilized to accelerate deduplicating.

6 Evaluation

In order to evaluate G-Paradex, we have implemented a prototype of the G-Paradex system that allows us to examine the performance impact and sensitivity of several important design parameters to provide useful insights into the design of deduplication-based storage system. The evaluation is driven by several real-world datasets that represent different workload characteristics in deduplication systems.

6.1 Experiment Setup

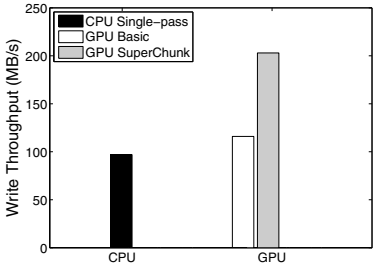
Datasets. We test our design using datasets from real systems. The datasets include web-vm (11.46 GB writes), mail (482.10 GB writes) and homes (148.86 GB writes), are from Florida International University and traces local researchers storage[21]. For these traces, we extract write I/Os and their MD5 hashes to form the write streams. The MD-5 hash is computed per 4096 bytes for web-vm and mail, per 512 bytes for homes. Therefore we divide the traces of web-vm and mail in unit of 4K, while divide the trace of homes in unit of 512 byte.

System Configuration. The experiments are carried out on a PC equipped with a NVIDIA GTX 580 GPU and an Intel Core i7 CPU 920. The global memory of GPU is 1.5 GB and the CUDA version is 3.1.

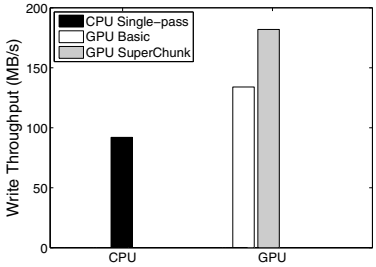
6.2 Impact of Parallel Deduplication

We first compare the parallel deduplication techniques with single-pass deduplication across a range of datasets. The parallel deduplication techniques include the basic design and the optimized versions, with the prefix tree of sub-key length of 4 (k'=4), number of sub-tree levels of 4 within a partition and super chunk size of 64 (number of basic chunks). The single-pass deduplication on the host (CPU) is implemented as the container-based deduplication with a locality preserved cache[3].

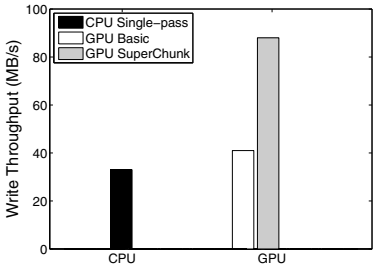
Figure 5(a-c) shows the deduplication write speed of these methods. Our results show that a basic GPU implementation can lead to some improvements over a host-only single-pass implementation for all three workloads, while the improvements are not significant. The observations clearly highlight the potential



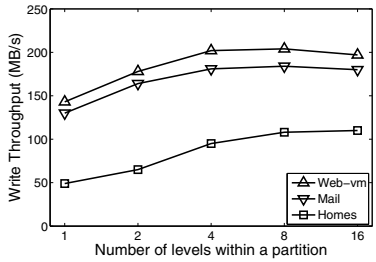
(a) Throughput comparison of deduplicating performance of *web-vm*



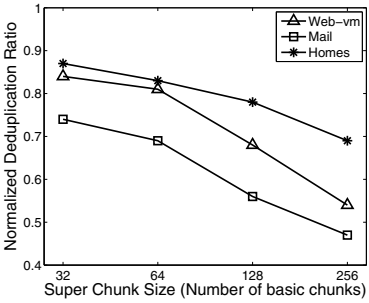
(b) Throughput comparison of deduplicating performance of *mail*



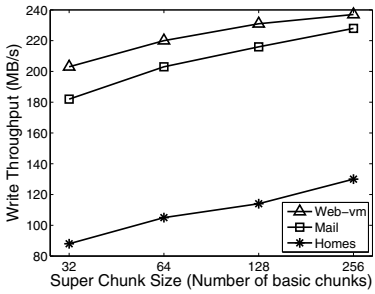
(c) Throughput comparison of deduplicating performance of *homes*



(d) Effectiveness of different levels within a partition



(e) Deduplication ratio as a function of chunk size



(f) Throughput as a function of chunk size

Fig. 5. Experiment results

of basic G-Paradex to alleviate parallel deduplication bottlenecks. Incorporating the optimizations leads G-Paradex outperforms the traditional single-pass implementation by a factor of over 1.5x for web-vm and mail workloads, and almost 3x for homes workload.

The performance of our approach depends on the number of levels within a partition. Figure 5(d) illustrates the performance of our approach with different

datasets. When the number equals 1, the performance of all three datasets is the lowest. This is because that the number of threads is too large to execute in parallel. And as the number of levels increases, the performance improves. But, at the point of the number of levels equals 8, the performance decreases for web-vm and mail. The reason behind is that large levels within a partition makes the processing of a single thread more complex and increases the search time.

6.3 Impact of Index Optimization

This subsection presents evaluation results on the impact of the super chunk organization. Using the super chunk organization, the index size will shrink. The larger the super chunk size, the smaller of the index tree size, while it may fail to find some duplicates. Figure 5(e) shows the deduplication ratio produced by applying our index optimization, normalized to that of the perfect deduplication (every duplicate chunk was identified), as a function of the super chunk size. As can be seen, the deduplication ratio falls off as the super chunk size increases, and the "knee" point at the size of 64 single chunks is a potential best tradeoff to balance deduplication effectiveness and index size shrank. The existence of chunk locality enhances the similarity between super chunks, and will be maintained in small size of super chunks. But, as more chunks being grouped, the degree of difference between similar super chunks increases, thus the locality is not that obvious and the deduplication ratio decreases a lot.

Figure 5(f) shows the write performance of G-Paradex with index optimization as a function of different super chunk size on three datasets. The larger the super chunk size, the faster the write speed our G-Paradex has, because the amount of partitions has to be examined decreases. Note that the deduplication ratio is inversely proportional to the super chunk size. By leveraging the chunk locality, use appropriate super chunk size not only largely improving the performance, but also eliminating massive duplicates in storage system.

7 Conclusion and Future Work

With an increasing amount of data and demands for eliminating redundancy, the optimization of duplicate detection continues to be a challenging task. In this paper, we propose G-Paradex, an fast and efficient GPU-based deduplication system that exploits parallelism in the deduplication process. In order to speed up index search, G-Paradex adopts prefix tree to organize the chunk fingerprints and traverse the tree in the GPU kernel. By effectively exploiting the ability of GPU and the inherent chunk locality, G-Paradex also proposes two techniques to further optimize the index structure and perform speculative pruning for accelerating the deduplication process.

There are several interesting avenues for future work. First, our index optimization technique is well suitable for workloads with high chunk locality, and we would like to develop new index structures for those without locality. Second,

our proposed techniques need to continuously adapt to changes in the technologies that are used by GPUs. Finally, with the ever increasing data that even an optimized index tree may not be fully loaded into main memory, thus we will explore on-disk tree layout and data prefetching techniques to cope with this problem.

Acknowledgments. This paper is supported by NSFC No.61272483 and Fund No.JC13-06-03.

References

1. Srinivasan, K., Bisson, T., Goodson, G., Voruganti, K.: idedup: latency-aware, inline data deduplication for primary storage. In: Proceedings of the 10th USENIX Conference on File and Storage Technologies, FAST 2012, p. 24. USENIX Association, Berkeley (2012)
2. Geer, D.: Reducing the storage burden via data deduplication. Computer 41(12), 15–17 (2008)
3. Zhu, B., Li, K., Patterson, H.: Avoiding the disk bottleneck in the data domain deduplication file system. In: Proceedings of the 6th USENIX Conference on File and Storage Technologies. FAST 2008, pp. 18:1–18:14. USENIX Association, Berkeley (2008)
4. Bhagwat, D., Eshghi, K., Long, D.D., Lillibridge, M.: Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In: IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems, MASCOTS 2009, pp. 1–9. IEEE (2009)
5. Fu, Y., Jiang, H., Xiao, N.: A scalable inline cluster deduplication framework for big data protection. In: Narasimhan, P., Triantafillou, P. (eds.) Middleware 2012. LNCS, vol. 7662, pp. 354–373. Springer, Heidelberg (2012)
6. Xia, W., Jiang, H., Feng, D., Hua, Y.: Silo: a similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In: Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference, USENIXATC 2011, pp. 26–28. USENIX Association, Berkeley (2011)
7. Lillibridge, M., Eshghi, K., Bhagwat, D., Deolalikar, V., Trezise, G., Camble, P.: Sparse indexing: large scale, inline deduplication using sampling and locality. In: Proccedings of the 7th Conference on File and Storage Technologies, FAST 2009, pp. 111–123. USENIX Association, Berkeley (2009)
8. 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., et al.: The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley (2006)
9. Bhatotia, P., Rodrigues, R., Verma, A.: Shredder: Gpu-accelerated incremental storage and computation. In: Proceedings of the 10th USENIX Conference on File and Storage Technologies, FAST 2012, p. 14. USENIX Association, Berkeley (2012)
10. Xia, W., Jiang, H., Feng, D., Tian, L., Fu, M., Wang, Z.: P-dedupe: Exploiting parallelism in data deduplication system. In: Proceedings of the 2012 IEEE Seventh International Conference on Networking, Architecture, and Storage, NAS 2012, pp. 338–347. IEEE Computer Society, Washington, DC (2012)

11. Dal Bianco, G., Galante, R., Heuser, C.A.: A fast approach for parallel deduplication on multicore processors. In: Proceedings of the 2011 ACM Symposium on Applied Computing, SAC 2011, pp. 1027–1032. ACM, New York (2011)
12. Bhattacherjee, S., Narang, A., Garg, V.K.: High throughput data redundancy removal algorithm with scalable performance. In: Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, HiPEAC 2011, pp. 87–96. ACM, New York (2011)
13. Rao, J., Ross, K.A.: Making b+- trees cache conscious in main memory. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, SIGMOD 2000, pp. 475–486. ACM, New York (2000)
14. Bayer, R., McCreight, E.: Organization and maintenance of large ordered indexes, pp. 245–262. Springer-Verlag New York, Inc., New York (2002)
15. Lehman, T.J., Carey, M.J.: A study of index structures for main memory database management systems. In: Proceedings of the 12th International Conference on Very Large Data Bases, VLDB 1986, pp. 294–303. Morgan Kaufmann Publishers Inc., San Francisco (1986)
16. Boehm, M., Schlegel, B., Volk, P.B., Fischer, U., Habich, D., Lehner, W.: Efficient in-memory indexing with generalized prefix trees. In: BTW (2011)
17. Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A.D., Kaldewey, T., Lee, V.W., Brandt, S.A., Dubey, P.: Fast: fast architecture sensitive tree search on modern cpus and gpus. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, pp. 339–350. ACM, New York (2010)
18. Volk, P.B., Habich, D., Lehner, W.: Gpu-based speculative query processing for database operations. In: Proceedings of the 1st International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (2010)
19. Nvidia cuda, `http://developer.nvidia.com/cuda-downloads`
20. Broder, A.: On the resemblance and containment of documents. In: Proceedings of the Compression and Complexity of Sequences, SEQUENCES 1997, pp. 21–29. IEEE Computer Society, Los Alamitos (1997)
21. Koller, R., Rangaswami, R.: I/o deduplication: utilizing content similarity to improve i/o performance. In: Proceedings of the 8th USENIX Conference on File and Storage Technologies, FAST 2010, p. 16. USENIX Association, Berkeley (2010)

Research on SPH Parallel Acceleration Strategies for Multi-GPU Platform

Lei Hu, Xukun Shen, and Xiang Long

Beihang University, Beijing 100191, PRC
hu_lei_1989@163.com,
{xkshen,long}@buaa.edu.cn

Abstract. This paper proposes an acceleration strategy for SPH on single-node multi-GPU platform. First the acceleration strategy for SPH on single-GPU is studied in conjunction with the characteristics of architecture. Then the changing pattern of SPH's computation time has been discussed. Based on the fact that the changing pattern is rather slow, using a simple dynamic load balancing algorithm an acceptable load balance is obtained on multi-GPU. Finally, an almost linear speedup is achieved on multi-GPU by further optimizing dynamic load balancing algorithm and communication strategy among multiple GPUs

Keywords: SPH, multi-GPU, dynamic load balancing, communication optimization.

1 Introduction

Smoothed Particle Hydrodynamics(SPH) is a mesh-free Lagrangian method based on particles. Due to its self-adaptivity, SPH can handle the problem of large deformation in simulation in a natural way and is free of several flaws that a grid-based method usually has, when they are used independently to deal with the same problems. In recent years, SPH has made much headway; its accuracy, stability and adaptability have already met the requirements of a great variety of engineering applications. However, the huge computation cost is a bottleneck to its use in many real-time situations. Therefore, nowadays great efforts have been made to explore its acceleration strategies.

In this paper, an appropriate neighbour list algorithm for SPH is selected according to the architectural characteristics of GPU. Then two optimization methods are proposed to solve code divergence problem and reduce potential neighbour particles. After observing the behavior of SPH computing process, a simple yet acceptable dynamic load balancing algorithm is given. Finally, optimization strategies of inter-GPU communication are illustrated.

2 SPH Method

SPH, developed by Monaghan[1] and Lucy[2] initially for astrophysical problems, has been studied and extended extensively; it has been applied to solving the

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 104–118, 2013.
© Springer-Verlag Berlin Heidelberg 2013

problem of dynamic response of material strength and hydrodynamics with large deformations. The objects to be simulated are divided into a set of discrete elements called particles, which can move freely in space. The physical quantities of particles are updated according to their neighbours in each time step. Two particles are called neighbours only if the distance between them is less than a special value, which is called smooth radius. The physical quantity A of a particle is given by

$$A_s(r_i) = \sum_j m_j \frac{A_j}{\rho_j} W(r_i - r_j, h) \tag{1}$$

where m_j is the mass of neighbour particle j; A_j is the physical quantity of neighbour particle j; ρ_j is the density of neighbour particle j; W is a kernel function governing the contribution of neighbour particle j according to the distance between particles i and j, and the smooth length h. As particles move freely in a scenario without spatial relationship, we need to search for their neighbours in each time step.

3 Related Work

Currently, most researches on the acceleration for SPH focus on two aspects: how to speed up neighbour search and how to utilize the computation power of parallel architecture.

SPH implements the interaction between particles, and so how to create the neighbour list is one of the key points to improve SPH's performance. To speed up neighbour search, the simulation space is divided into cubical cells, called uniform grid. The size of a cell is usually equal to smooth radius. Before neighbour search, each particle is assigned to only one cell according to its center point to create a particle list of each cell. To find the neighbours of a certain particle, $27(3*3*3)$ cells need to be searched(including the cell itself resident in and 26 adjacent cells). Denote by c the number of particles residing in each cell, then the complexity of this method is $O(cN)$ where N is the total number of particles in the simulation. Dominguez et al.[3] have compared the time cost and memory consumption between 4 gridding algorithms to create the particle list of each cell. Taking time cost and memory consumption both into consideration, the algorithm in which particles are sorted according to the cells performs the best. In this method, each cell's particle list only needs to store the start position of sorted particles in the particle array. By comparing the start position of continuous cells, we can find all particles in each cell. Dominguez et al. also compare time cost and memory consumptions between VL(Verlet List) algorithm and CLL(Cell Linked List) algorithm which are both used for the creation of each particle's neighbour list. The main difference between these two algorithm lies in the fact that VL keeps neighbour list and reuses it in several time steps while CLL does not.

Fleissner et al.[4] use CPU cluster to accelerate SPH. They select ORB (orthogonal recursive bisection) domain decomposition algorithm to split simulation

space from the perspective of optimizing communication between nodes. Each CPU is assigned a subspace. Dynamic load balance is achieved by moving the boundary of subspace with a PI controller. The main advantage of this method is the decreased amount of communication between CPUs, but it has a drawback that the communication pattern gets more sophisticated.

With the rapid development of GPU technology in recent years, the performance and programmability of GPU have been promoted significantly. GPU, which is used in computer graphics traditionally, has been extended to high performance computing field. GPU has higher floating-point performance and better power-efficiency than CPU. Many computation-intensive applications have migrated from CPU to GPU and a speedup of two orders of magnitudes has been achieved. Because of its natural parallelism, SPH is remarkably suitable for parallel architectures such as GPU. In fact, even before the advent of specific languages like CUDA and OpenCL, researchers had started to use GPU to speed up SPH with graphic API. Amada et al.[5] implement SPH method partially on GPU. They create neighbour list of each particle on CPU, and then transport it to GPU to calculate force between particles. Harada et al.[6] implement SPH on GPU entirely. After CUDA, Herault et al.[7] implement SPH with CUDA for the first time. The neighbour search algorithm they use is same with that of Simon Green's[8].

As it is hard to meet the speed requirement of real-time simulation in a scale of millions of particles on a single GPU, it is extremely necessary to utilize multi-GPU in single computing node or even GPU cluster. Rustico et al.[10] and Dominguez et al.[11] have proposed multi-GPU SPH implementation independently. They both use the one-dimensional decomposition to divide simulation space into subspaces, whose number is equal to that of GPUs. The boundary of subspace is aligned at the boundary of cell. Dynamic load balance is achieved by passing the outmost cell slices of a subspace, whose corresponding GPU is overloaded, to others. The difference between Rustico et al.'s and Dominguez et al.'s dynamic load balancing algorithm is only in implementing details. As for communication strategy, both implementations divide subspace further into two boundary areas and an inner area, and cover the overhead of data exchange cost by exchanging boundary particles' data in parallel with the computation of inner area. Dominguez et al. further point out that the main factor influencing the performance of GPU is synchronization between GPUs.

4 Accelerating SPH On Single-GPU

First an appropriate neighbour list algorithm for GPU is chosen from existing ones in terms of time cost and memory consumptions. Then SPH's code divergence problem in traditional CUDA implementation is analyzed in conjunction with characteristics of SIMT architecture of CUDA-enabled GPU. Finally, Smaller Cell optimization is presented from the viewpoint of reducing the quantity of potential neighbour particles in the neighbour search.

4.1 Choosing Appropriate Neighbour List Algorithm

For SPH in which all particles have the same smooth radius, there exists two popular methods to create neighbour list, named CLL(Cell Linked List) and VL(Verlet List), respectively. Both algorithms use the same method mentioned in Section 3 to create a particle list of each cell. The difference between the two algorithms is whether to keep the neighbour list or not. Over CLL algorithm, the main advantage of VL algorithm is that it keeps neighbour list in several time steps and reduces the time cost in neighbour search. In contrast, VL algorithm requires much more memory because of storing the neighbour list. According to Dominguez et al., under the condition of searching for neighbours only once in each time step, VL algorithm is 6% faster than CLL algorithm while its memory consumption is 30 times larger. As GPU has a relatively small size of memory(GTX480 has about 1.5 GB), VL algorithm brings minor performance promotion along with excessive memory consumption, thus limiting the simulation scale fatally. Rustico et al. keep neighbour list in several time steps in their multi-GPU SPH implementation and find that each particle needs nearly 1KB memory, and that a scale of only 1.8 million particles can be simulated on a GTX480 graphic card. Considering storage and performance comprehensively, CLL algorithm is more suitable for GPU.

4.2 Code Divergence Optimization

In SIMT architecture of CUDA-enabled GPU, threads are scheduled and executed in warp. A warp is composed by 32 threads which execute same instruction at the same time in parallel. Due to conditional control flow instructions, sometimes the threads in a same warp may need to execute instructions in different code paths. In that case, the threads of a warp will execute instructions in each path serially, thereby bringing huge negative influence on performance. In traditional implementation, a thread only computes one particle's force. Each thread traverses all 27 cells with a triple loop and traverses all particles in each cell in the innermost loop(shown in figure 1).

After traversing all particles in cell, as all threads in a same warp have to execute the same instruction, those threads must be synchronized implicitly before traversing the next cell. As the relationship between CUDA threads and particles is one-to-one and the number of particles in most cells is not equal to the number of threads in a warp, different threads in a same warp may process particles in different cells. So each thread in a warp is likely to traverse cells with different particle quantities at the same time, causing imbalance workload between two synchronization points. Obviously, reducing the frequency of synchronization can lower down the negative influence of imbalance workload on performance. The frequency of synchronization is equal to the number of cells to be searched. In CLL algorithm, particles are reordered according to the cells and cells following the order of Z, Y and X axis. The particles in continuous cells in Z axis will be continuous in particle array. As each cell only stores the range(start index and end index in our implementation) of particles contained in

```
for( x=-1; x<2; x++ )
{
    for( y=-1; y<2; y++ )
    {
        for( z=-1; z<2; z++ )
        {
            read neighbour cell k's info which reside in direction (x,y,z)
            for( each particle m in neighbour cell k )
            {
                read position of m and check if m is a neighbour
                if( m is a neighbour )
                {
                    force calculation
                }
            }
            (implicit synchronization exists in CUDA)
        }
    }
}
```

Fig. 1. Code snippets of neighbour search in traditional SPH implementation

itself in particle array, the range of 3 continuous cells in Z axis can be replaced by a larger cuboid with the same range. As a result, to find the neighbours of each particle, only 9 nearby cuboids should be searched and branch instructions reduce in number, consequently greatly improving the negative influence of code divergence, and meanwhile reducing global memory access of cell information. This optimization is called Cell Merging.

4.3 Reduce Potential Neighbours

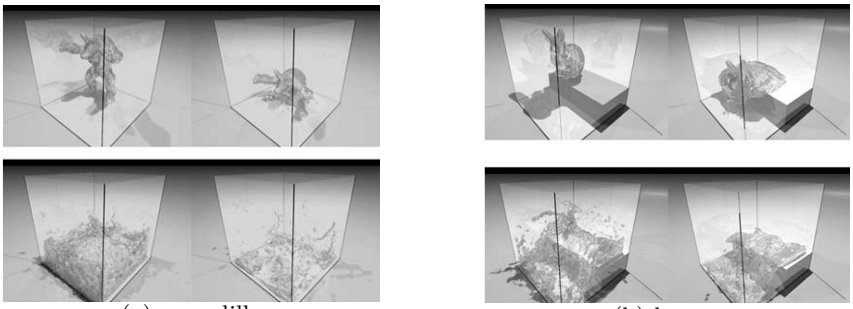
The size of cell determines the volume of the neighbour search space. When the cell size is equal to smooth radius h, each particle has to traverse 27 cells with a volume of $27h^3$. However, for each given particle i, it is only affected by particles in a sphere of radius h whose centre is i, and volume is $4\pi * h^3/3$. Smaller cell size can make the volume of neighbour search space of each particle get much closer to the volume of sphere to reduce potential neighbour particles. Supposing cell size is h/n, in order to ensure that all neighbor particles can be found for a given particle, the volume being searched should completely cover the volume of the sphere with radius h. So we have to search n cells with cell size h/n on both sides of the cell where the particle stays in each dimension of neighbour search space. Thus the side length of the cubic searched is $2*n*(1/n)h+1*(1/n)h = (2+1/n)h$ and the volume of neighbour search space is reduced to $(2 + 1/n)^3h^3$. However two drawbacks occur when n is too large. First, the total memory consumption of cells increases rapidly with the decreasing cell size:

$$mem_n = n^3 * mem_1 \tag{2}$$

where mem_n represents the total memory consumption of cells when the cell size is reduced to h/n. Second, larger n makes neighbour search more sophisticated and increases code divergence and global memory access of cell information. As mentioned in section 4.2, more neighbour cells which need to search to find neighbour particles lead to more implicit synchronization points. When n becomes larger, the amount of neighbour cells need to be searched increases rapidly, thus leading to the increasing of implicit synchronization points and code divergence. Besides, more neighbour cells means more global memory access, as the cell information is resident in the GPU's global memory. Taking all this into consideration synthetically, n=2 is the best choice in our implementation and the volume of the neighbour search space shrinks to $15.625h^3$. Smaller Cell is the name we give to this optimization.

4.4 Speedup on Single-GPU

For a better understanding of the experiment results shown in this paper, a brief description of our experimental environment is given first. The hardware platform is a dual quad-core Intel Xeon processor E5520 (2.27GHz, 8MB cache) Server with 4 GTX480 GPU cards, and each GTX480 has 480 CUDA cores with 1.5GB global memory. The operating system is Ubuntu 11.04 x86_64, CUDA runtime 4.1. Each experiment includes 1000 computing time steps for a certain scenario. Two scenarios, one of which has different scales, are used as testing cases in this paper, named armadillo and bunny respectively(shown in figure 2).



(a) armadillo (b) bunny

Fig. 2. Snapshot of scenario armadillo(a) and bunny(b) in 1st(upper-left), 300th(upper-right), 500th(lower-left), 1000th(lower-right) time step

Figure 3 shows the performance improvements with the two optimizations mentioned above. Only the time cost of updating particles' physical quantity is compared. As Smaller Cell will significantly increase code divergence(When n=2 and only Smaller Cell is applied, the time cost of updating particles' physical quantity increases by about 80%), it should be applied with Cell Merging optimization to get the best performance. In that case, each cuboid covers 5 continuous cells in Z axis.

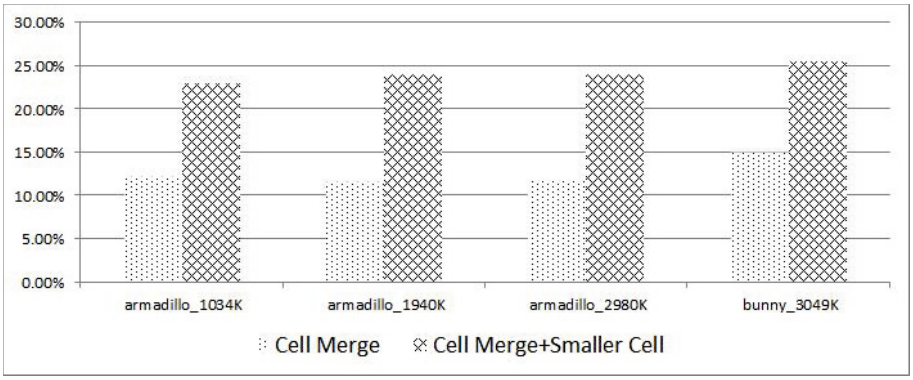


Fig. 3. Speedup on single-GPU with two optimization methods

5 Multi-GPU

In this section, we extend SPH implementation from single-GPU to multi-GPU. In multi-GPU implementation, the Cell Merging optimization is applied to accelerate the execution. Different from prior work on multi-GPU SPH, the feature of SPH which can be utilized to design a simple yet acceptable dynamic load balancing algorithm is mainly discussed instead of proposing a dynamic load balancing algorithm directly. Communication optimizations focus on additional steps introduced in multi-GPU SPH. For a better understanding of the problems faced in multi-GPU SPH, a brief description of the basic frame of multi-GPU SPH based on CLL algorithm is given first.

5.1 Basic Design

In our design, the same domain decomposition algorithm described in [9,11] is used to divide particles among multiple GPUs. Simulation space is divided into subspaces along X-axis whose number is same as the number of GPUs. The interface of subspaces is aligned at cells' boundary. The smallest unit of division is n cell slices, as each cell has a cell size of h/m in Y-Z plane. For convenience, we assume that the size of cell is equal to smooth radius below (the method is similar when cell size is different). For the reason that particle's physical quantity is influenced by all its neighbour particles, each GPU contains not only the particles in corresponding subspace, but also those near the interface within a distance of smooth radius but on the neighbour GPU's side. These particles are called ghost particles below. The ghost particles exist in both neighbour GPUs simultaneously but are updated by only one of them. After any physical quantity of particles is updated, each GPU needs to exchange the new physical quantity of ghost particles.

Multi-GPU algorithm should include 4 main steps as follows:

1. Create Neighbour List
2. Dynamic Load Balancing

```
if (load imbalance)
{
    divide the simulation space anew
    each GPU exchanges boundary particles
}
```

3. Update Particles' Physical Quantity

```
3.1 force calculation
3.2 integration in time
```

4. Particle Migration

```
4.1 gather particles needed by other GPU, called migrating
    particles.
4.2 exchange migrating particles
```

Step 1 and step 3 are the basic steps of single-GPU SPH. The reason why we put Dynamic Load Balancing step after Create Neighbour List is that CLL algorithm can make particles in same cell slice continuous in particle array, thereby simplifying the data exchange after the subspaces are repartitioned.

5.2 Dynamic Load Balancing

The key idea to obtain dynamic load balance between multiple GPUs is moving boundaries of each subspace to change the number of particles on each GPU. The key point of a good dynamic load balancing algorithm is how and when to move the boundaries. Before a new dynamic load balancing algorithm is proposed, the feature of SPH which can be utilized to design a simple yet acceptable dynamic load balancing algorithm is discussed first.

To ensure the accuracy of simulation in SPH, the moving length of any particle is usually set not longer than the smooth radius h in each time step, and consequently the spatial distribution of particles will not change a lot in two contiguous time steps. The relative stability of distribution keeps the two main factors that influence performance—the quantity of potential neighbours and neighbour particles—all stay stable. So in two continuous time steps, computation time changes slightly. We have used the scenario of armadillo with a scale of 2980K particles and scenario of bunny with 3049K to test changing range of each time step's time cost of updating particles' physical quantity in comparison with the previous one in the first 1000 time steps. Figure 4 shows the result.

From 300 to 480 time steps, the fluid in armadillo is compressed because of the initial collapse and then it becomes sparser gradually. So not only the number of potential neighbours but also the number of neighbours change severely; as a

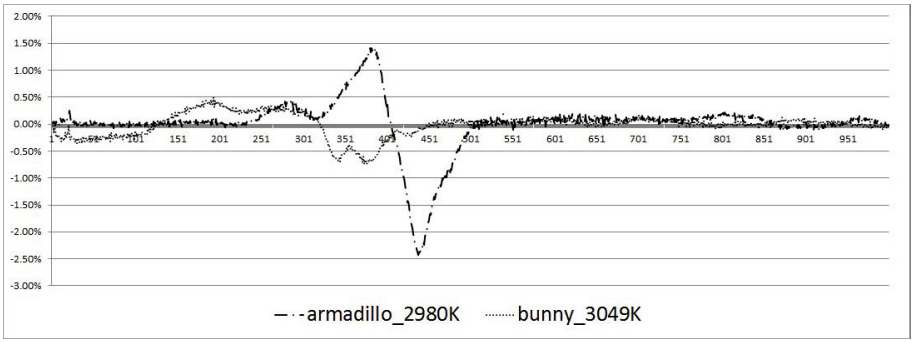


Fig. 4. Comparison of each time steps' time cost(the time cost of updating particles' physical quantity) with previous one tested with scenario armadillo and bunny

consequence, the time cost of updating particles' physical quantity also changes substantially. However, even in that case, the changing range remains within 3%. For bunny scenario that is affected less by collapse, its changing range stays within 1% all along. Because of the slow changing pattern of SPH's computation time, simple dynamic load balancing algorithm described below can get satisfying results:

```
for( each pair of neighbour GPUs A and B )
{
      if( time_A > time_B )
      {
          A gives one cell slice to B
      }else{
          B gives one cell slice to A
      }
}
```

Figure 5 shows a comparison of real wall time(longest time cost of updating particles' physical quantity among multiple GPUs) with an ideal one(average time cost of all GPUs), demonstrating that simple dynamic load balancing algorithm has acceptable effect. Figure 6 shows the speedup of multi-GPU in the same situation. As particle scale goes up, the effect of the simple dynamic load balancing algorithm improves.

The main disadvantage of simple dynamic load balancing algorithm is that it is hyper-sensitive to load imbalance among GPUs. As the whole boundary cell slice is the smallest unit of data exchanging, the alternation between the overload and underload on a GPU in continuous time steps is unnecessarily frequent, which leads to an unnecessary communication overhead in the step of Dynamic Load Balancing. There are two ways to reduce the frequency. One is to balance workload every k time steps. The other is to give a threshold to dynamic load balancing algorithm. Simulation space is repartitioned if and

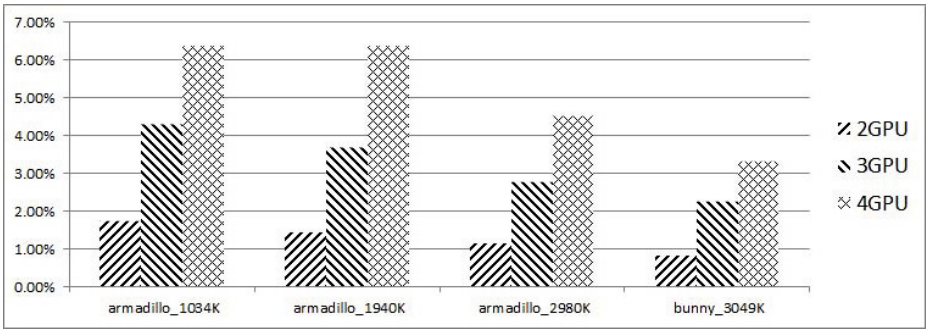


Fig. 5. Comparison of real wall time with ideal one when simple dynamic load balancing algorithm is applied

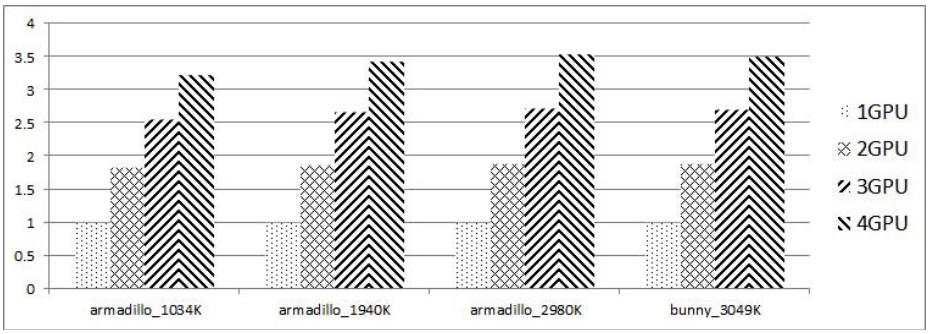


Fig. 6. Speedup with simple dynamic load balancing algorithm

only if the difference of computation overhead between two neighbour GPUs is bigger than this threshold. Dynamic load balancing algorithm should discover the imbalance among GPUs timely and gives response. So the second way is our choice.

$$new_time_diff_{left} = fabs(t_1 * nc_{left1} - t_2 * nc_{left2}) \tag{3}$$

$$new_time_diff_{right} = fabs(t_1 * nc_{right1} - t_2 * nc_{right2}) \tag{4}$$

$$nc = n_{new}/n_{original} \tag{5}$$

$$threshold = min\{new_time_diff_{left}, new_time_diff_{right}\} \tag{6}$$

where t_1 and t_2 represent the time GPU1 and GPU2(GPU1 and GPU2 are a certain pair of neighbour GPUs, GPU1 is in the left of GPU2) used to update particles' physical quantity respectively; nc stands for the changing rate of the number of particles updated by GPU after boundary moves in left or right direction; nc_{left} indicates the boundary moves to the left and nc_{right} on the opposite side; n_{new} stands for the number of particles in GPU's subspace after

boundary moves; $n_{original}$ stands for the number of particles in GPU's subspace before boundary moves. Figure 7 shows the speedup after adding the threshold. Figure 8 exhibits a comparison of real wall time with ideal one.

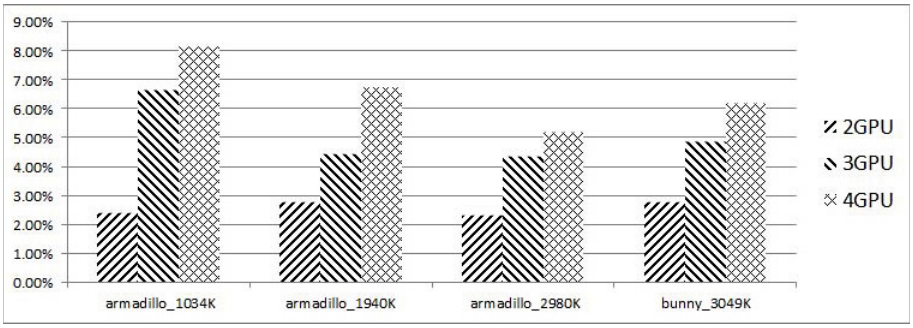


Fig. 7. Speedup of optimized dynamic load balancing algorithm compared to the simple one

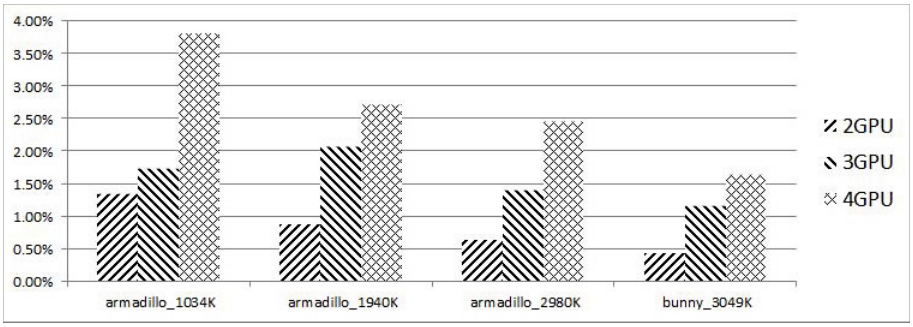


Fig. 8. Comparison of real wall time with ideal one. Optimized dynamic load balancing algorithm is applied.

Figure 9 demonstrates the ideal speedup(the performance of multi-GPU is only affected by dynamic load balancing algorithm without any other negative influences such as communication overhead) and the real speedup. Ideal speedup is calculated as (on homogeneous multi-GPU platform)

$$speedup_{ideal_n} = n * walltime_{ideal_n}/walltime_{real_n} \tag{7}$$

where $speedup_{ideal_n}$ represents ideal speedup when the number of GPU is equal to n, n represents the number of GPUs used in simulation, and walltime represents the longest time cost of updating particles' physical quantity in each time step. The real speedup is achieved without any optimization in the Dynamic Load Balancing and Particle Migration steps. Both steps are relative with communication among multiple GPUs.

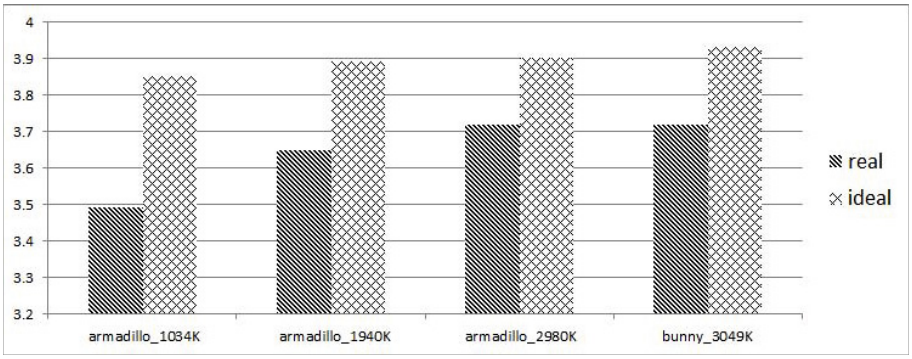


Fig. 9. Real speedup and ideal speedup of 4GPUs. Tested with: scenario armadillo with 1034K, 1940K, 2980K particles and bunny with 3049K particles.

5.3 Communication Optimization

An often used method in CUDA applications to hide communication overhead is to parallelize computation and communication. Communication among GPUs goes in the following steps:

1. Particle Migration

After sub-step integration in time in the step Updating Particles' Physical Quantity, particles may migrate from one subspace to another subspace. At the end of each time step, GPU needs to identify and exchange those particles(called migrating particle below), so there exists data transfer overhead. As the distribution of migrating particles in memory space is irregular, GPU needs extra computation to gather those particles into continuous memory space before sending them to neighbour GPU.

2. Updating Particles' Physical Quantity

As described before, it is necessary to exchange the information of boundary particles among GPUs.

3. Dynamic Load Balancing

Each GPU needs to wait for new space division and exchanges boundary cell slices according to new division of simulation space. As a result, there exist synchronization overhead and data transfer overhead.

The same method described in Rustico for covering the overhead is used in the Updating Particles' Physical Quantity step. We focus on the optimizations in Particle Migration and Dynamic Load Balancing steps. Here we give a brief description of how to hide communication overhead in those two steps.

The first sub-step of Particle Migration is gathering migrating particles in irregular distribution into continuous memory space. One way is to use a compress function to gather migrating particles, but non-negligible computation cost has to be added. For SPH, the time of updating particles' physical quantity is more than 10 times longer than the time of data exchange of boundary particles in general cases. The other way is to send migrating particles to neighbour

GPU(s) without gathering them. Migrating particles reside in two cell slices in the vicinity of GPU's boundary because the migration distance is shorter than smooth radius (the size of cell is smooth radius). Instead of gathering migrating particles, we send all particles in the two cell slices(called potential migrating particles below) to neighbour GPU(s), but it will unfortunately increase the cost of data exchange. If we can hide the communication overhead, the second way is clearly a better choice. To hide the time cost of exchanging potential migrating particles, the subspace of each GPU is divided into two boundary areas(consist of two cell slices) and an inner area. First, the kernel which is used to update boundary particles' physical quantity is launched. Then, the update of inner area works in parallel with the exchange of potential migrating particles. In this way, the time cost of exchange of potential migrating particles is hidden. In the next time step, when neighbour GPU(s) creates neighbour list with CLL algorithm, it is feasible to extract those particles that are not needed from others at a cost of a slight increase in the overhead only by giving them a sufficiently large cell value.

The space repartition in the Load Balancing step can be delayed to the time GPU starts to update particles' physical quantity. In this way, it can be parallelized with Updating Particles' Physical Quantity to hide the synchronization overhead. The data exchange of boundary cell slices after space repartition can be done together with particle migration at the end of each time step. As a result, all overheads in Dynamic Load Balancing step can be hidden through parallelization with Updating Particle's Physical Quantity.

6 Final Result

Figure 10 shows the final speedup of multi-GPU SPH implementation via communication optimization using optimized dynamic load balancing algorithm. Corresponding to the speedup presented in figure 6, the performance of multi-GPU increases by about 10%. Multi-GPU's speedups all exhibit the trend of linear acceleration when different numbers of GPUs simulate millions scale scenario.

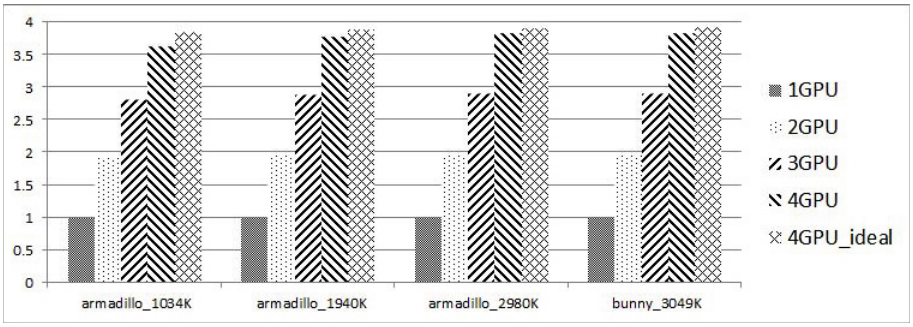


Fig. 10. The final speed up

7 Conclusions

An acceleration strategy for SPH method on single-node multi-GPU platform has been proposed in this paper. For single-GPU, we first choose an appropriate neighbour search algorithm CLL combined with architectural characteristics. Subsequently, two optimizations are made. To solve code divergence problem we merge continuous cells into a huge cell to reduce synchronization point in traditional implementation. By decreasing the cell size, less potential particles are searched in neighbour search. For multi-GPU, we focus on the changing patterns of SPH's computational time. Simple dynamic load balancing algorithm works well because the computational time of each time step changes slowly compared to previous time step. By further optimizing dynamic load balancing algorithm and the communication strategy among GPUs, a nearly linear speedup is achieved in different scenarios with a scale of millions particles.

8 Future Work

We will study the specific acceleration strategy for SPH on GPU cluster. The main difference between GPU cluster and single-node multi-GPU is that the bandwidth among nodes is far narrower than PCI-E 2.0(infiniband has not been taken into consideration yet), which may have significant effect on SPH's performance on GPU cluster. Therefore, its communication strategy may be different with what we addressed in this paper. This deserves further research.

References

1. Gingold, R.A., Monaghan, J.J.: Smoothed particle hydrodynamics: theory and application to non-spherical stars. Mon. Not. R. Astron. Soc. 181, 375–389 (1977)
2. Lucy, L.B.: A numerical approach to the testing of the fission hypothesis. Astron. J. 82, 1013–1024 (1977)
3. Dominguez, J.M., Crespo, A.J.C., et al.: Neighbour lists in smoothed particle hydrodynamics. International Journal for Numerical Methods in Fluids 67(12), 2026–2042 (2011)
4. Fleissner, F., Eberhard, P.: Parallel load-balanced simulation for short-range interaction particle methods with hierarchical particle grouping based on orthogonal recursive bisection. International Journal for Numerical Methods in Engineering 74(4), 531–553 (2011)
5. Amada, T., Imura, M., et al.: Partilce-based fluid simulation on GPU. In: ACM Workshop on General-Purpose Computing on Graphics Processors and SIGGRAPH (2004)
6. Harada, T., Koshizuka, S., et al.: Smoothed particle hydrodynamics on GPUs. In: Proceedings of Computer Graphics International (2007)
7. Herault, A., Bilotta, G., et al.: SPH on GPU with CUDA. Journal of Hydraulic Research 48(1, suppl. 1) (2010)
8. Simon Green: Particle Simulation using CUDA, `http://www.dps.uibk.ac.at/~cosenza/teaching/gpu/nv_particles.pdf`

9. Rustico, E., Bilotta, G., et al.: Smoothed particle hydrodynamics simulations on multi-GPU systems. In: 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2012, February 15-17 (2012)
10. Rustico, E., Bilotta, G., et al.: A journey from single-GPU to optimized multi-GPU SPH with CUDA. In: 7th SPHERIC Workshop (2012)
11. Dominguez, J.M., Crespo, A.J.C., et al.: New multi-GPU implementation for smoothed particle hydrodynamics on heterogeneous clusters. Computer Physics Communications (2013)

Binarization-Based Human Detection for Compact FPGA Implementation

Shuai Xie, Yibin Li*, Zhiping Jia, and Lei Ju

School of Computer Science and Technology, Shandong University, Jinan, China, 250101
xieshuai1210@mail.sdu.edu.cn,
{liyibing,jzp,leiju}@sdu.edu.cn

Abstract. The implementation of human detection in the embedded domain can be a challenging issue. In this paper, a real-time, low-power human detection method with high detection accuracy is implemented on a low-cost field-programmable gate array (FPGA) platform. For the histogram of oriented gradients feature and linear support vector machine classifier, the binarization process is employed instead of normalization, as the original algorithm is unsuitable for compact implementation. Furthermore, pipeline architecture is introduced to accelerate the processing rate. The initial experimental results demonstrate that the proposed implementation achieved 293 fps by using a low-end Xilinx Spartan-3e FPGA. The detection accuracy attained a miss rate of 1.97% and false positive rate of 1%. For further demonstration, a prototype is developed using an OV7670 camera device. With the speed of the camera device, 30 fps can be achieved, which satisfies most real-time applications. Considering the energy restriction of the battery-based system at a speed of 30 fps, the implementation can work with a power consumption of less than 353mW.

Keywords: HOG+SVM, Binarization Process, FPGA Implementation, Low Power Consumption.

1 Introduction

Real-time image-based human detection is an important implementation for vision systems, particularly for embedded environments. Apart from the vision domain, this implementation also has a wide range application prospects in areas such as entertainment, surveillance, robotics, and security. For embedded human-detection applications, real time, detection accuracy, hardware resource requirement, and power consumption are four primary considerations. In many applications, external memory is usually needed. Moreover, a tradeoff must exist between performance and power consumption is trade-off by owing to the limited resources of a field-programmable gate array (FPGA).

* Corresponding author.

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 119–131, 2013.
© Springer-Verlag Berlin Heidelberg 2013

The human detection process primarily contains two significant steps: feature description and classification. During feature description, important information is extracted from the image. The classifier algorithm is used to determine whether a person is present in an image. Various methods for feature description have been proposed, such as Haar wavelets [1], Haar-like features [2], Gabor filers [3], and SHIF descriptors [4]. Likewise, many classifier algorithms are available, such as the support vector machine (SVM) [5] and Adaboost [6]. Nevertheless, these algorithms cannot satisfy the requirements for detection accuracy.

In 2005, the famous histogram of oriented gradients (HOG) feature [7] was proposed, which subsequently became the most widely used algorithm for object detection. This algorithm significantly enhanced the detection accuracy of human detection. However, its high computational complexity has made the HOG algorithm impossible to run on a desktop computer in real time. Numerous hardware implementations of human detection based on HOG algorithm that could work in real time have recently been made. Nevertheless, such methods have always had lower detection accuracy and poor power consumption or required a high-end FPGA for its implementation.

To achieve a good balance of the four considerations mentioned in the first paragraph and to address the concern of having limited resources in embedded implementations such as wireless sensor networks (WSNs), we proposed a simplified human detection algorithm based on the HOG feature and linear support vector machine (SVM) targeting low-end FPGA devices. Binarization is adopted and optimized to replace the normalization process. Additionally, pipeline architecture is introduced to increase the detection speed. Furthermore, few other simplifications and optimizations are introduced during hardware implementation. Finally, our implementation can be mapped on a low-end Xilinx Spartan-3e FPGA and can work in real time, with slightly less detection accuracy and low power consumption.

The remainder of this paper is organized as follows: Section2 reviews related studies on human detection; Section3 provides the architecture of the proposed human detection process; Section4 explains the FPGA implementation details; Section5 recounts the implementation results and evaluation; and Section 6 presents a summary of the work.

2 Related Work

With the extensive literature on human detection, this section mentions only a few relevant papers on the acceleration or hardware implementation of human detection. Our algorithm is primarily based on the original HOG feature algorithm proposed by Dalal et al [7]. However, the original HOG algorithm has a very slow detection rate. In 2006, Zhu et al. [8] proposed a modified human detection algorithm based on a multi-scale HOG feature and a boosted cascade of the Adaboost classifier, which was first proposed in [9]. In this study, the researchers achieved nearly the same detection accuracy as Dalal's implementation that worked in real time, although this algorithm was unsuitable for the hardware used. In 2007, Kerhetet al. [10] proposed a human

detection implementation that had minimal power consumption through the FPGA development board. Although this work was not based on the HOG algorithm, it was implemented well on FPGA with good detection speed and low power consumption. In 2009, Kadotaet al. [11] introduced a hardware implementation of HOG feature extraction. The researchers proposed some ideals of simplification or modification for FPGA implementation and achieved a process speed of 30fps. To reduce the HOG feature size, [12] proposed an effective binarization scheme for the HOG feature. In 2011, Negi et al. [13] employed a deep pipelined architecture for the hardware implementation of human detection. With this architecture, external memory was no longer necessary, and less hardware resources were used. In 2012, Komorkiewicz et al. [14] implemented the original HOG algorithm by using single precision, 32-bit floating point values. Their implementation achieved high detection accuracy, although the use of a high-end Virtex6 FPGA resulted in very high resource utilization.

In the present study, we modified the algorithm used by Negi et al. and further optimized it for hardware implementation, thus achieving significantly improved performance.

3 Human Detection Algorithm

The HOG feature uses the local histograms of oriented gradients of each pixel to characterize the image. This feature expresses the contour of humans and avoids the interference of light and action to a certain extent. The detection process is achieved through the linear SVM classifier. In this study, some modifications were made on these algorithms to suit hardware implementation on FPGA.

The original HOG and SVM algorithms have four steps:

1) Gradient and direction calculation;
2) Histogram generation;
3) Normalization;
4) Classification.

This algorithm is unsuitable for hardware implementation as the dense of square, square root, multiplication, anti-trigonometric, and division operations are calculated during the detection process, and an external memory is always required for the storage of intermediate data. Thus, the binarization process was adopted and optimized to replace the normalization process, resulting in a series of modifications that will be discussed in detail in Sections 3.2 and 3.3. Likewise, other optimizations and simplifications are discussed below.

3.1 Gradient and Direction Calculation

The parameters of the cell and block used for the HOG extraction are 8×8 and 16×16 pixels, respectively. Based on Dalal's work, sample mask (-1,0,1) showed the best performance. Using this mask, the following equation was obtained:

$$\begin{cases} f_x(x,y) = f(x+1,y) - f(x-1,y) \\ f_y(x,y) = f(x,y+1) - f(x,y-1) \end{cases} \tag{1}$$

where *f(x,y)* represents the luminance value at coordinate *(x,y)*. For enhanced performance, the square root of each channel was obtained, such that each value for f_x or f_y ranges from 0 to 15. The magnitude *m* and direction θ are calculated by Eqs.(2) and (3). For the color image, the gradients of each color channel were calculated separately, and the largest *m(x,y)* was considered, as this value is the gradient vector of the pixel.

$$\mathrm{m(x,y)} = \sqrt{f_x(x,y)^2 + f_y(x,y)^2} \tag{2}$$

$$\theta(\mathrm{x,y}) = \tan^{-1}\frac{f_x(x,y)}{f_y(x,y)} \tag{3}$$

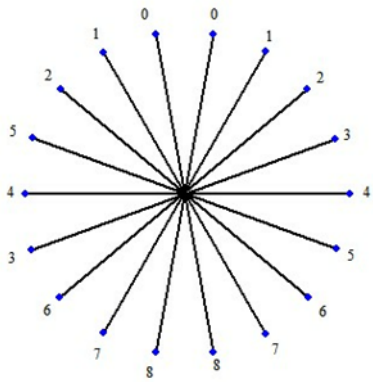


Fig. 1. Quantized gradient directions θ

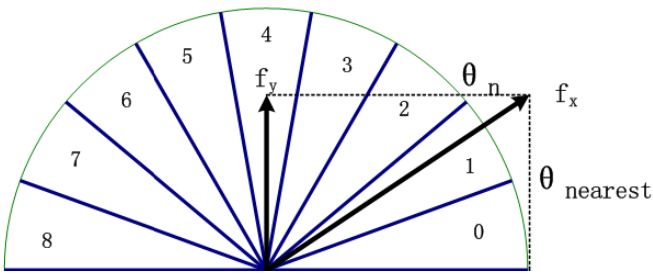


Fig. 2. Nine bins

To compute for the gradient of the orientation histogram, the orientation is divided into nine bins, as shown in Fig.1. For each pixel in a cell, two weighted votes are calculated for the nearest bin and the bin to which the pixel belongs (Fig. 2). The vote is based on the gradient magnitude *m*, whereas the weight is calculated according to the direction θ. This process is calculated using the following equations:

$$m_n = (1 - \mathrm{a})\mathrm{m}(\mathrm{x}, \mathrm{y}) \tag{4}$$

$$\mathrm{m}_{\mathrm{nearest}} = \mathrm{am}(\mathrm{x}, \mathrm{y}) \tag{5}$$

$$\mathrm{a} = \frac{b\theta(\mathrm{x,y})}{\pi} - (\mathrm{n} + 0.5) \tag{6}$$

In the proposed implementation, a standard RGB-565 image was used. Only the highest four bits from every channel were used for the calculation, which means that only 512 different values are available for gradient m and direction θ. Thus, the weight vote of each pixel can be calculated in advance and then pre-stored in a look-up-table with 1 kb BRAM.

Subsequently, the votes of a cell are grouped and summed according to their direction. Finally, the histogram is generated for each cell.

In this implementation, each block contains four histograms, which is a nine-dimension vector.

3.2 Histogram Normalization

Using steps 1, we obtained serve histograms, each of which is a nine-dimension vector in the hardware, given by:

$$F_{i,j}=[f^0_{i,j}, f^1_{i,j}, f^2_{i,j}, f^3_{i,j}, f^4_{i,j}, f^5_{i,j}, f^6_{i,j}, f^7_{i,j}, f^8_{i,j}]$$

Each element f_n of $F_{i,j}$ represents the value of bin n in each histogram. This element is called a feature vector. For each cell, we obtained a feature vector, whereas for each block, we obtained a large feature vector consisting of all the feature vectors from the cells that belong to the block. For example, the cell is 8×8 pixels, whereas each block consists of 2×2 cells. Thus, the feature vector of the block is a 36-dimension vector formed by the nine-dimension vectors of the four cells.

To weaken the effect of light and the slight movement of the human body on the feature vector, the feature vector should be normalized. As stated in Dalal's paper, the L2-norm has the best performance, which is given by

$$\mathrm{v} = \frac{V_k}{\sqrt[2]{\|V_k\|^2 + \varepsilon}} \tag{7}$$

where V_k is the feature vector of block k, ε is a constant to avoid division by zero, and v is the final feature vector.

Although this step also has a square root operation, it cannot be realized using a look-up table. The square root operation can be performed using a Cordic IP CORE with a delay of 20 clocks. However, such action would make hardware realization impossible, and a large memory would be necessary to store the feature vector.

In [12], the researchers proposed a binarization process, a method used by [13] with a constant threshold. Although this process degrades performance because of the loss of accuracy of the HOG features, the memory cost is considerably reduced. After normalization, the HOG features of a block become specific values. With this process, the HOG features of each block would have the same weight on the classifier training and detection processes, although the rate of each HOG feature would not be changed

as before. With a constant threshold, the effective features of a block can be highlighted. Moreover, with a threshold that represents the average value of all the features in a block, the same result can be obtained, along with other benefits.

As shown in Fig.3, the red line represents the average value of the 36 HOG features in one block. The final HOG feature is set to 1 if it is greater than the average value and is set to 0 otherwise. This process has two advantages. First, the rate of the HOG features for each block are unchanged with or without the normalization process because this process is no longer required given the selection of an average value as the threshold. Second, the features obtained after the binarization process take the value of either 1 or 0, which will further optimize the detection process with a SVM classifier. This optimization will be discussed in Section 3.3. Additionally, with an average value as the threshold, the same benefit can be obtained as with having a constant threshold.

During the HOG feature generation step, the normalization process costs the most calculation resources given the need to calculate the dense of square, square root, division, and multiplication operations for each block. Assigning average values as the threshold will reduce the resource cost of both the hardware and software.

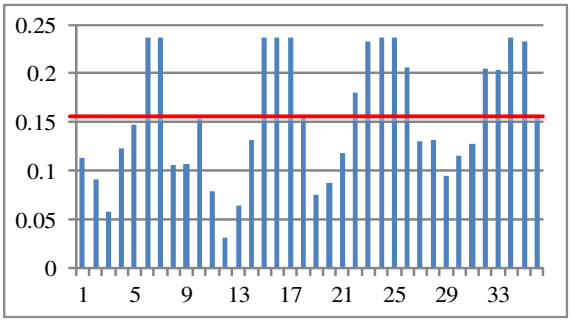


Fig. 3. Binarization process

After the two steps, the HOG feature of the image was obtained.

3.3 SVM Classification

SVM is a machine learning method used for classification and regression analysis. Given a set of training examples, each example is classified under two categories, and an SVM training algorithm builds a model that assigns new examples for each corresponding category.

For the proposed realization, the training data comprised the feature vector of each image, which is a 3780-dimension vector. To simplify the calculation, linear SVM classifier was employed. The SVM classifier was trained offline, and the final SVM classifier was a 3781-dimension vector.

The detection process using a linear SVM classifier involved multiplying the SVM vector by its corresponding HOG features. After the binarization process, the HOG features used to train the SVM classifier took the value of either 1 or 0, and the

multiplication operation was replaced by addition. Statistically, 40% of the HOG features take the value of 1. Finally, in each detection process, 1512addition operations would be calculated, instead of 3780 multiplication and 3780 addition operations. This modification saves hardware resources.

4 FPGA Implementation

An OV7670CMOS video camera was used as the input device. By changing the initial parameters, the input image was fixed at 320×240 pixels, and the frame rate was set as 30fps. Finally, the detection parameters used are shown in Table 1.

Table 1. Parameters

Input image	320x240 pixel
Detection windows	64x128 pixel
Cell	8x8 pixel
Block	2x2 cell
Step stride	8x8 pixel
Number of bins	9

4.1 Gradient Computation

To accelerate the classification process, the pipeline architecture was adopted. As shown in Fig. 4, three lines of three-stage shift registers were used to store four adjacent data during gradient and direction calculation. Two-line BRAM was used to store the other 317 values. As previously mentioned, the calculation of m_n and $m_{nearest}$ was performed using a look-up-table.

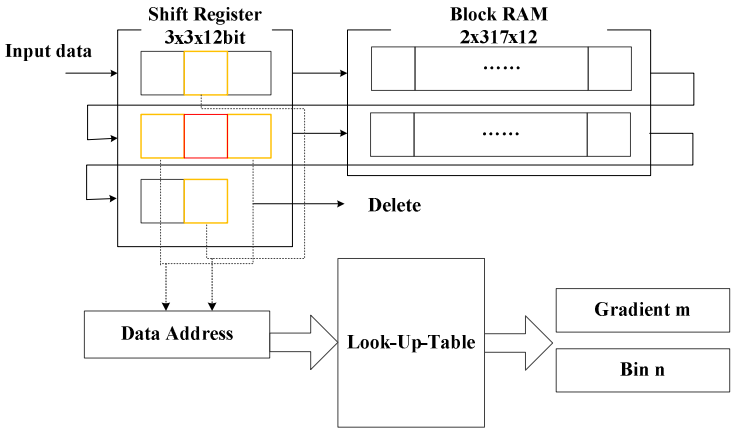


Fig. 4. Hardware structure for the calculation of gradient and bin

4.2 Histogram Generation

For the histogram generation process, pipeline architecture was also used. After the previous process, the weighted votes of each pixel were obtained. The histogram of each cell was generated by summing up the votes of one cell. As illustrated in Fig. 5, a partial histogram was calculated for every eight pixels and then stored in a temporary register. Subsequently, the stream of partial histograms was loaded into the BRAM, such that the partial histograms for eight lines are added up. Thus, the histogram for each cell was generated.

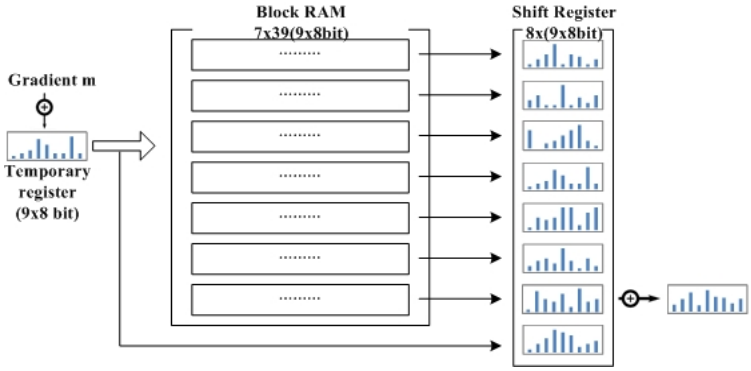


Fig. 5. Hardware structure for the histogram generation

4.3 Histogram Normalization

In our implementation, we adopted optimized binarization instead of the normalization process. The hardware structure and data stream are shown in Fig. 4. This process requires the adjacent feature vectors of four cells. Therefore, two lines of two-stage shift registers and one-line of BRAM buffers were used to store the feature vectors. The average value of the feature vector of each cell was calculated and cached in the temporary register, along with its feature vector. Every time a new feature vector was input, the average value of each block was calculated. Each feature value was then coded in binarization mode.

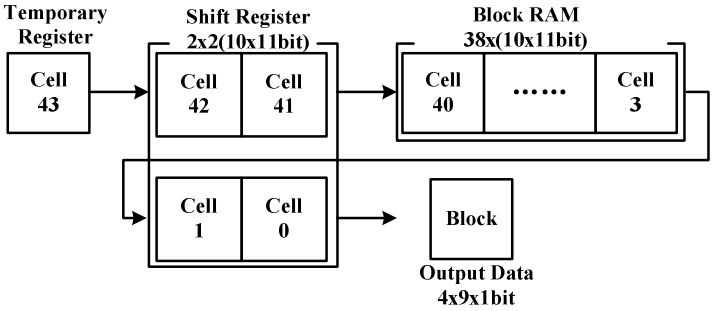


Fig. 6. Hardware structure for binarization

4.4 SVM Classification

With the binarization process, the classification process could be performed by adding the elements of the classifier to a corresponding HOG feature of 1. To accelerate the prediction process, a 3780-dimension filter was built, as shown in Fig.7. The HOG features were stored in the 15×7×(4×9) bit shift registers and the 14×32×(4×9) block ram. The whole HOG feature of a detection window was stored in the shift register and then loaded into the filter. Consequently, the detection results were calculated by adding the SVM elements, which have a corresponding HOG feature of 1. The hardware architecture of this process is shown in Fig.8.

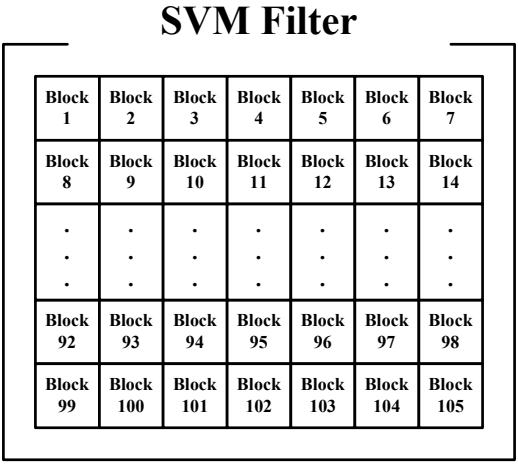


Fig. 7. SVM Filter

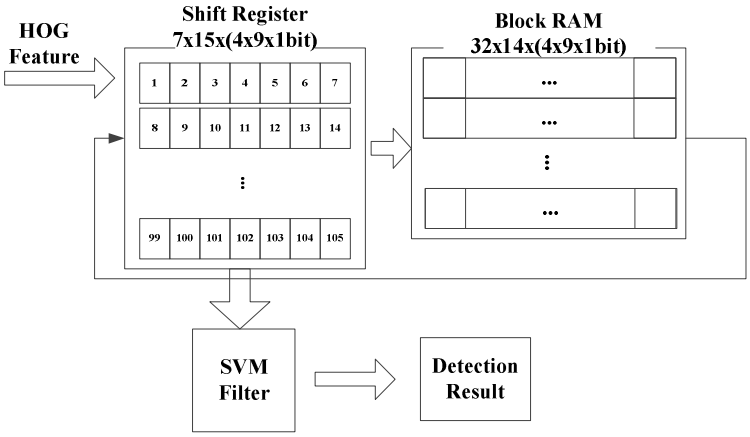


Fig. 8. Hardware structure for the human detection process

5 Implementation Result and Evaluation

The human detection project was implemented in a wireless video development broad made by the authors with a Spartan-3e XC3S500E FPGA (Fig. 9). Anstm32 microcontroller was placed in the board to control the camera and transmit the image data to the FPGA at the correct time. The detection result was transmitted by a 2.4GHz RF24L01B.

Fig. 9. Wireless video development board

Table 2 lists the implementation results of the current study and of Negi et al. The present work showed a reduced resource usage compared with that by Negi et al. Therefore, the present realization could be mapped on this low-end Spartan-3e FPGA with very limited programmable resources.

Table 2. Results oftheFPGA implementation

	Our work on Spartan-3e	Negi's work on Virtex-5	Komorkiewicz's work on Virtex-6
SLICE	2041	2,181	32,428
LUT	3,379	17,383	113,359
FF	2,602	2,070	75,071
BRAM	6	36	119

To capture a stable image data according to the camera device, the present implementation works on 24MHz. At this clock rate, the authors detected 30 input images per second, although this value is far from the limitation of the implementation. In spite of the restricted rate of the camera, our implementation can detect images at293 fps, with maximum frequency of 67.75MHz. The throughput of the FPGA implementation was compared to a software implementation generated by opencv3.1 using a PC with 2.33GHz Intel Core2 E6550 CPU and 4GB DDR2, operated by Windows7. The software implementation achieved about 2.1fps, and

Negi's implementation achieved112fps at a maximum frequency. Therefore, the current implementation is 139.5 times faster than the software implementation, and 2.6 times faster than Negi et al.'s implementation.

Subsequently, the authors re-implemented Negi et al.'s work on software. In this work, eight features are treated as one 8-bit-wide feature during the classifier training process. In contrast, in the current implementation, each HOG feature is used independently. The test results are summarized as a detection error trade-off (DET) curve in Fig. 10.Negi et al.'s work attained a 3.4% miss rate and a 20.7% FFPW. Alternatively, the current implementation attained a1.97% miss rate and 1% FFPW. In comparison with Negi et al.'s work, the detection accuracy of the current implementation was also evaluated by treating every 4 and 2 HOG features as one feature during the classifier training and detecting processes, as shown in Fig.10.Theresults show that this simplification method harmed the detection accuracy. Nevertheless, although the performance of the current implementation is worse than that of the original algorithm, it is still much better than that of Negi et al.

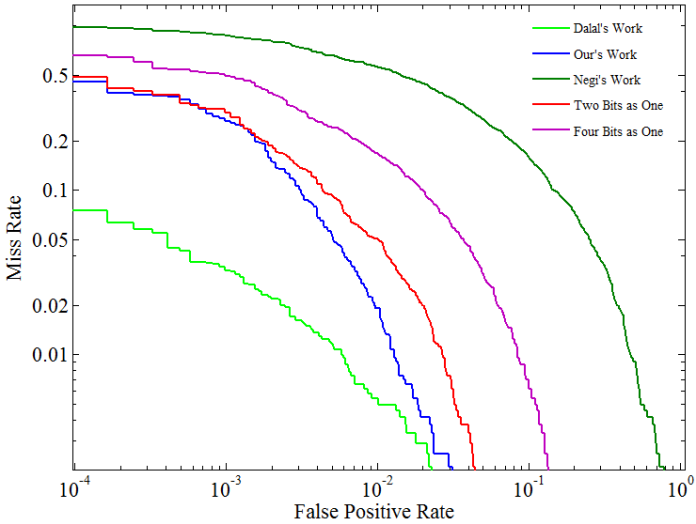


Fig. 10. DET curve for detection accuracy

Table 3. The power evaluation results

Realization	Quiescent Power	Dynamic Power	Total
Ours on Spartan-3e	83mW	270mW	353mW
Ours on Virtex5	444mW	120mW	564mW
Negi's on Virtex5	450mW	438mW	888mW

Finally, we compared the energy consumption of the current implementation and that of Negi et al. The results are summarized using the Xilinx XPower Estimator (Table 3). The current implementation that works on a Spartan-3e has a power

consumption of 353mW, which is extremely low and can satisfy the extreme limitation on a WSN node. Furthermore, using a Virtex5 FPGA, the current implementation achieved 564mW power consumption, whereas that of Negi et al.'s implementation reached 888mW. The quiescent power was almost the same, whereas the dynamic power is nearly a quarter of that of Negi et al.

6 Conclusion

In this paper, a real-time, low power consumption implementation of human detection using the HOG feature and linear SVM was presented. After an experimental implementation on FPGA and an evaluation of the algorithm's detection accuracy through software implementation, the current work achieved a detection rate of 30 fps, with relatively less hardware resources and lower power consumption. Although some simplifications have been made, the detection accuracy is acceptable and relatively higher than that of other implementations. With a high-speed camera, the maximum frequency of 293 fps can be achieved. The current implementation is suitable for the extreme limitation of an embedded platform, such as a WSN node.

References

1. Oren, M., Papageorgiou, C., Sinha, P., Osuna, E., Poggio, T.: Pedestrian detection using wavelet templates. In: IEEE Conference on Computer Vision and Pattern Recognition, pp. 193–199 (1997)
2. Viola, P., Jones, M.J., Snow, D.: Detecting pedestrians using patterns of motion and appearance. International Journal of Computer Vision 63(2), 153–161 (2005)
3. Cheng, H., Zheng, N., Qin, J.: Pedestrian detection using sparse gabor filter and support vector machine. In: IEEE Intelligent Vehicles Symposium, pp. 583–587 (2005)
4. Lowe, D.G.: Distinctive image features from scale-invariant keypoints. International Journal of Computer Vision 60(2), 91–110 (2004)
5. Chang, C.-C., Lin, C.-J.: LIBSVM: a library for support vector machines. ACM Transactions on Intelligent Systems and Technology (TIST) 2(3), 27 (2011)
6. Freund, Y., Schapire, R.E.: A desicion-theoretic generalization of on-line learning and an application to boosting. In: The 2nd European Conference on Computational Learning Theory, London, UK, pp. 23–37 (1995)
7. Dalal, N., Triggs, B.: Histograms of oriented gradients for human detection. In: The 2005 International Conference on Computer Vision and Pattern Recognition, Washington, DC, USA, vol. 2, pp. 886–893 (2005)
8. Zhu, Q., Yeh, M.-C., Cheng, K.-T., Avidan, S.: Fast human detection using a cascade of histograms of oriented gradients. In: Computer Vision and Pattern Recognition (CVPR), pp. 1491–1498 (2006)
9. Viola, P., Jones, M.: Rapid object detection using a boosted cascade of simple features. In: Computer Vision and Pattern Recognition (CVPR), pp. 511–518 (2001)
10. Kerhet, A., Leonardi, F., Boni, A., Lombardo, P., Magno, M., Benini, L.: Distributed video surveillance using hardware-friendly sparse large margin classifiers. In: Advanced Video and Signal Based Surveillance (AVSS), pp. 87–92 (2007)

11. Kadota, R., Sugano, H., Hiromoto, M., Ochi, H., Miyamoto, R., Nakamura, Y.: Hardware architecture for HOG feature extraction. In: Intelligent Information Hiding and Multimedia Signal Processing, pp. 1330–1333 (2009)
12. Sun, W., Kise, K.: Speeding up the detection of line drawings using a hash table. In: Pattern Recognition (CCPR), pp. 1–5 (2009)
13. Negi, K., Dohi, K., Shibata, Y., Oguri, K.: Deep pipelined one-chip FPGA implementation of a real-time image-based human detection algorithm. In: Field-Programmable Technology (FPT), pp. 1–8 (2011)
14. Komorkiewicz, M., Kluczewski, M., Gorgon, M.: Floating point HOG implementation for real-time multiple object detection. In: Field Programmable Logic and Applications (FPL), pp. 711–714 (2012)

HPACS: A High Privacy and Availability Cloud Storage Platform with Matrix Encryption

Yanzhang He, Xiaohong Jiang*, Kejiang Ye, Ran Ma, and Xiang Li

College of Computer Science, Zhejiang University,
Hangzhou 310027, China
{heyanzhang,jiangxh,yekejiang,maran,lixiang}@zju.edu.cn

Abstract. As the continuous development of cloud computing and big data, data storage as a service in the cloud is becoming increasingly popular. More and more individuals and organizations begin to store their data in cloud rather than building their own data centers. Cloud storage holds the advantages of high reliability, simple management and cost-effective. However, the privacy and availability of the data stored in cloud is still a challenge. In this paper, we design and implement a High Privacy and Availability Cloud Storage (HPACS) platform built on Apache Hadoop to improve the data privacy and availability. A matrix encryption and decryption module is integrated in HDFS, through which the data can be encoded and reconstructed to/from different storage servers transparently. Experimental results show that HPACS can achieve high privacy and availability but with reasonable write/read performance and storage capacity overhead as compared with the original HDFS.

Keywords: Cloud storage, Data privacy, Availability, Matrix encryption.

1 Introduction

Clouds [1] are a large shared resource pool of easily usable and accessible computing and storage infrastructures, which can be elastically configured to accommodate variable kinds of workloads for achieving optimal Quality of Service (QoS) and resource utilization. These large-scale shared resources are typically exploited by a pay-per-use model. According to the service type, clouds can be classified by three fundamental models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS).

Cloud storage provides internet-based storage capacity to individuals and organizations. Users can access their data anytime and anywhere by various devices which are able to connect to the Internet. Cloud Storage Providers (CSP) should ensure their products with high reliability, simple management and cost-effective. A recent survey [2] among over 60 thousand cloud users shows that, for most of users, the primary reason for tuning to cloud is to save the cost in infrastructure construction and maintenance.

* Corresponding author.

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 132–145, 2013.
© Springer-Verlag Berlin Heidelberg 2013

The promising benefits of the cloud service attract various companies, including both large enterprises and startup companies, to provide cloud storage service with different Service Level Agreements (SLAs). For example, Amazon's Simple Storage Service (S3), Apache's Hadoop Distributed File System (HDFS), Microsoft's SkyDrive [3], etc. S3 [4] promises an average error rate less than 0.1, which is computed in a fairly straightforward manner. They first divide the monthly billing cycle into five-minute intervals, and then compute the error rate for each interval, finally compute the average error rate over all intervals. Apache Hadoop [5] is an open-source software framework that supports data-intensive distributed applications. All the input data are stored on HDFS as blocks and every block has several replicas in different DataNodes. JobTracker could gain the block locations of input data and assign tasks to the nodes which store the input data for parallel processing.

Although the cloud storage service holds numerous advantages, it has two main challenges. (i) **Privacy problem**: Since the data stores outside user's infrastructure, it is perceived to be normal that the companies may loss control of data. Multiple classes of personnel may access the physical storage devices to read or write other users' data. Data encryption is a frequently-used technology to solve this problem, which can prevent cloud administrators and attackers from accessing the original data. (ii) **Availability problem**: It means the cloud providers might be out of service, and users cannot access the cloud storage platform at that interval. Report [6] shows that most CSPs have run into failures from time to time, causing services to stop for hours or even days. In the previous works, redundant technique is often applied to solve the availability problem, including data replication and erasure code strategies.

Although there are some works on privacy problem and availability problem respectively, few works solve the problems simultaneously. In this paper, we combine these two factors and provide a high privacy and availability cloud storage platform (HPACS). Our contributions can be summarized as follows:

- First, we propose a matrix encryption algorithm and a matrix decryption algorithm to achieve the high privacy and availability in cloud storage;
- Then we integrate the algorithms to the HPACS, which is designed to automatically and transparently encode original data into multiple partitions or reconstruct original data from multiple partitions. The partitions are stored on different servers;
- At last, we perform extensive evaluation to verify the effectiveness of HPACS as well as its overheads. Experimental results show that HPACS can achieve high privacy and availability but with reasonable write/read performance overhead and storage capacity overhead as compared with the original Hadoop system.

The rest of this paper is structured as follows. In section 2, we first describe the architecture of HDFS and HPACS platform. Then we explain the write/read flows of HPACS and the theory of matrix encryption/decryption algorithms. Finally, we present the implementation of HPACS. In section 3, we conduct a series of experiments, comparing the write/read performance between HDFS and HPACS platform. We also consider the data availability and usage of storage capacity. In section 4, we give the introduction of related works. In the last section, we conclude our work and present the future work.

2 Designing and Implementation of HPACS Platform

In this section, we first describe the architecture of HDFS and HPACS platform. Then we explain the write/read flows of HPACS and the theory of matrix encryption/ decryption algorithms. Finally, we present the implementation of HPACS.

2.1 HDFS and HPACS Platform Architecture

The Hadoop distributed file system [7] contains three parts: (i) NameNode: It holds the namespace of distributed file system, which is a hierarchy of files and directories. File or directory is represented by inode which records the attributes like permission, modification, access time and disk space quota, etc. The NameNode also maintains the mapping information of file to blocks (each block has three replicas in default) and block to DataNodes. When clients want to read data, they first contact the NameNode for the locations of data blocks comprising the file and then read block contents from the DataNode closest to the Client. When clients want to write data, they first request the NameNode to appoint a suite of DataNodes to host the block replicas and then write block contents to the DataNodes in a pipeline form. The current design of Hadoop platform holds only one NameNode for each cluster and results in the problem of single point of failure. (ii) DataNode: It is responsible for the storage of file content itself. When the file size exceeds the appointed block size, it will be split into large blocks (typically 64 megabytes) which are independently replicated on multiple DataNodes. The cluster can have thousands of DataNodes and tens of thousands of clients, and each DataNode may execute multiple application tasks concurrently to improve the whole throughput. DataNode sends heartbeat to the NameNode to inform that the DataNode is running and the block replicas it hosts are available. The default heartbeat interval is three seconds. (iii) Client: User applications access the Hadoop distributed file system using the HDFS client. Similar to conventional file system, HDFS supports operations to read, write and delete files, and operations to create and delete directories.

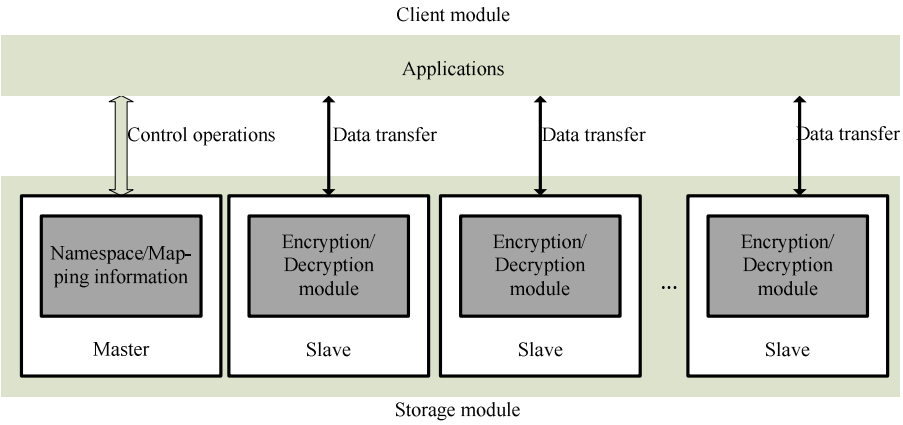


Fig. 1. HPACS platform architecture (Client module, Encryption/Decryption module and Storage module)

Figure 1 illustrates the system architecture of HPACS platform. It consists of three main modules: Client module, Encryption/Decryption module and Storage module. The Encryption/Decryption module is newly designed in this paper which is a supplement to the current HDFS to enhance its privacy and availability. It includes the functions of encoding original data to partition data and reconstructing partition data to original data. We design a matrix redundant algorithm in the Encryption/Decryption module which is implemented on slaves of storage module. The other two modules are similar to the original HDFS, we use the terms Master/Slave instead of NameNode/DataNode in the storage module. The master should also maintain the secret key (Vandermonde matrix in this paper) except the mapping information of file to blocks and block to DataNodes.

2.2 Write and Read Flows of HPACS

Figure 2 shows the file write flow of HPACS platform. It includes 8 steps:

(1) The client creates the file by calling *create()* on *DistributedFileSystem* object;
(2) *DistributedFileSystem* makes an RPC call to the master to create a new file in the namespace. If the *create* operation is executed successfully, it returns a success signal, else it returns an error signal;
(3) As the client writes data, the file is split into blocks (typically 64 megabytes), *FSDataOutputStream* writes the data to an internal queue by packets. The packet queue is consumed by the *DataStreamer*, whose another responsibility is to ask the master to allocate a list of suitable slaves (always 3) to store the partition data encoded from the original data;
(4) The block packet is written to the first slave and sent to *FileEncode* function in the matrix encryption and decryption module to encode independently partition packets;
(5) The first slave stores one partition packet on its local file system, and other two partition packets are sent to the second slave and the third slave;
(6) *FSDataOutputStream* also maintains an internal queue of packets that are waiting to be acknowledged by slaves, called the *ack* queue. The Vandermonde matrix is brought by the *ack* signal to the client;
(7) When the client finishes writing data, it calls *close()*;
(8) Finally, the client contacts the master to send secret key and to mark that the file is written completely.

Figure 3 shows the file read flow of HPACS platform. It includes 7 steps:

(1) The client opens the file it wishes to read by calling *open()* on *Distributed-FileSystem* object;
(2) *DistributedFileSystem* calls the master, using RPC, to obtain the locations of the partitions for the first few blocks in the file and the corresponding secret keys;
(3) The client calls *read()* on the *FSDataInputStream*;
(4) The *FSDataInputStream* transfers the secret key to the first slave;
(5) Partitions are streamed from other slaves to the first slave;
(6) *FileReconstruction* function in the matrix encryption and decryption module reconstructs the original data. Data is streamed from the first slave back to the

client. When the end of the block is reached, *FSDataInputStream* will close the connection to the slave, and then find another three slaves for the next block. This happens transparently to the client, which from its point of view is just reading a continuous stream;

(7) When the client has finished reading, it calls *close()* on the *FSDataInputStream*.

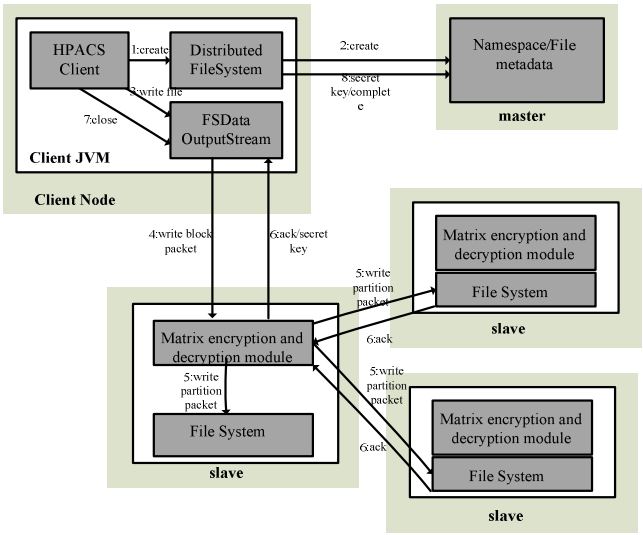


Fig. 2. File write flow of HPACS platform

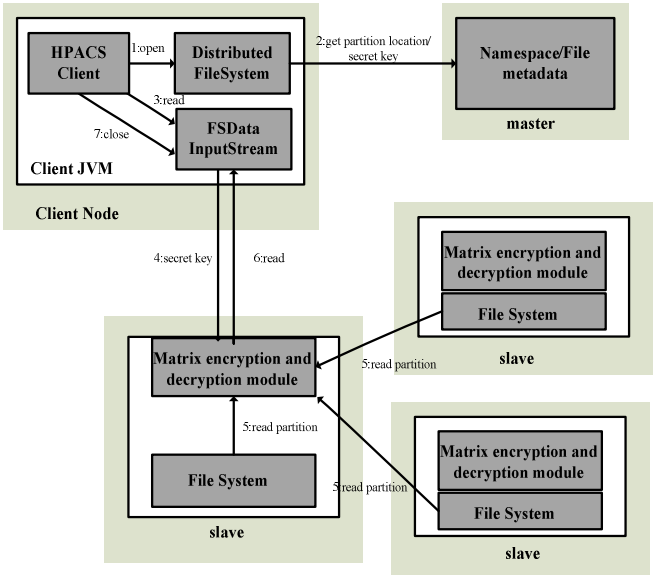


Fig. 3. File read flow of HPACS platform

2.3 Matrix Encryption and Decryption Algorithm

In this section, we describe the details of matrix encryption and decryption algorithms [8], which are implemented in the Encryption/Decryption module as shown in Figure 1.

From the fundamental theorem of algebra, we know that every polynomial equation of k-order has k roots. We use this fact to encode original data into k partitions such that each of the partition can be stored on different servers. The partition data in themselves do not reveal any information, only when all the partitions and the corresponding secret key are brought together, the data can be revealed through reconstruct algorithm. The secret key in this paper is the Vandermonde matrix.

Consider a polynomial equation of k-order,

$$x^k + a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \ldots + a_1 x + a_0 = 0 \tag{1}$$

Equation 1 has k roots denoted by $\{r_1, r_2, \ldots, r_k\} \subseteq \{\text{set of complex numbers}\}$ and can be rewritten as equation 2.

$$(x - r_1)(x - r_2)\ldots(x - r_k) = 0 \tag{2}$$

We replace a_0 in equation 1 with the data d which we want to encode, then we can obtain equation 3, and call the roots as partitions.

$$d = \prod_{i=1}^{k} r_k \tag{3}$$

In the above situation, the partitions are stored on different slaves in storage module. When any one of the slaves on which the partitions are stored becomes unavailable, users couldn't be able to reconstruct the original data from the available partitions. In order to solve the low reliability and availability mentioned early, we introduce the redundancy technology through matrix transformation.

We use a linearly independent matrix to extend the k partitions to n partitions. From all the n partitions, we only need any k partitions to reconstruct the original data. We define $r = n/k$ as the redundancy rate, referring that we bring the overhead of r on the storage costs. Through this redundancy we can achieve high availability and resist against possible failures of particular servers. For example, we construct n linearly independent equations such as equation 4.

$$\begin{cases} a_{11}r_1 + a_{12}r_2 + \ldots + a_{1k}r_k = p_1 \\ a_{21}r_1 + a_{22}r_2 + \ldots + a_{2k}r_k = p_2 \\ \ldots \\ a_{n1}r_1 + a_{n2}r_2 + \ldots + a_{nk}r_k = p_n \end{cases} \tag{4}$$

The above linear equations can be redefined as matrix operation as equation 5, and we call it as (k, n) data encode algorithm.

$$\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1k} \\ a_{21} & a_{22} & \ldots & a_{2k} \\ \vdots & & & \\ a_{n1} & a_{n2} & \ldots & a_{nk} \end{pmatrix} \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_k \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{pmatrix} \; (A \cdot \vec{R} = \vec{P}) \tag{5}$$

The reconstruct algorithm is the inverse process of encode algorithm. Because equation 4 is linearly independent, the value of $k \times k$ determinant consists any k rows of matrix A doesn't equal to zero. We can choose any k partitions of $\{p_1, p_2, \ldots, p_n\}$ to reconstruct the k roots, using equation 6. Finally, we can use equation 3 to compute the original data d.

$$\begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_k \end{pmatrix} = \begin{pmatrix} a_{h_1 1} & a_{h_1 2} & \ldots & a_{h_1 k} \\ a_{h_2 1} & a_{h_2 2} & \ldots & a_{h_2 k} \\ \vdots & & & \\ a_{h_k 1} & a_{h_k 2} & \ldots & a_{h_k k} \end{pmatrix}_{k*k}^{-1} \begin{pmatrix} p_{h_1} \\ p_{h_2} \\ \vdots \\ p_{h_k} \end{pmatrix}_{k*1} \quad (1 \le h_1 < h_2 < \ldots < h_k \le n) \tag{6}$$

When computing the polynomial equation roots, we use the finite field $\mathbb{Z}_p$ where p ($0 \le d \le p-1$) is a large prime for simplicity. We can obtain k roots $\{r_1, r_2, \ldots, r_k\}$of equation 1 in finite field $\mathbb{Z}_p$. For example, if the original data (d = 189) needs to be encoded into three partitions then a 3-order polynomial equation is chosen and the roots computed. We set the prime p to 257 which is bigger than the value of one byte. Through the Algorithm 1 in Appendix I, we can achieve three roots (r_1 = 57, r_2 = 60, r_3 = 113) which correspond to the partitions. The partitions in themselves do not reveal any information. When all the partitions are brought together, we can reconstruct the original data use equation 7.

$$d = \sum_{i=1}^{3} r_i \bmod p = (57 \times 60 \times 113) \bmod 257 = 189 \tag{7}$$

2.4 Implementation of HPACS

We implement HPACS platform based on Apache Hadoop-1.0.1, and add *FileEncode* function and *FileReconstruction* function in HDFS.

- FileEncode Function

In the HDFS, we use the *–put* command line API to write file to the distributed file system, while we use *-newput* command line API instead in our new platform. The Algorithm 1 in Appendix I shows the pseudo-code of *FileEncode* function.

- FileReconstruction Function

Similar to the write process, we use *-newget* command line API instead in HPACS platform. The Algorithm 2 in Appendix I shows the pseudo-code of *FileReconstruction* function.

- Difference between HDFS and HPACS

There are four main differences between HDFS and HPACS. (i) When users want to write file, the pipeline replication mechanism in HDFS is canceled. The block packet is sent to *FileEncode* function in the matrix encryption and decryption module to create independently partition packets. The first slave stores one partition packet on its local file system and other partition packets are transferred to the other slaves. (ii) When users want to read file, all partitions are streamed to the first slave. Then the

FileReconstruction function in the matrix encryption and decryption module reconstructs the original file data. (iii) We ignore the block replica number in HDFS, and conduct the transmission control by the value of (k, n). Every block is encoded into n partitions with the redundancy rate $r = n/k$. (iv) HPACS has the *–newput* and *–newget* command line API. In addition, the master maintains the secret key except the mapping information of file to blocks and block to DataNodes.

3 Experimental Results and Analysis

In this section, we study both the write/read performance and the data availability. In the write/read performance analysis, we first study the performance of HDFS with different block replica number, and then investigate the performance difference between HDFS and HPACS platform. In addition, we analyzed the data availability of HPACS platform.

3.1 Experimental Configuration

Due to the limited quantity of physical machine, all the experiments are performed within virtual machines on four Dell PowerEdgeR720 servers, with 2 Quad-core 64-bit Xeon processors E5-2620 at 2.00GHz and 64GB DRAM. We use Ubuntu12.04 in Domain 0, and Xen 3.3.1 as the virtualization hypervisor. Each virtual machine is installed with Ubuntu12.04 as the guest OS with the configuration of 1VCPU and 2048MB vMemory.

3.2 Write/Read Performance of HDFS and HPACS

We first create a 16-node Hadoop virtual cluster (1 NameNode and 15 DataNodes) running on four physical machines to study the write and read performance of HDFS. Figure 4 shows the write performance when the file size ranges from 1MB to 200MB with different block replica number. The experimental data is chosen from TOEFL (The Test of English as a Foreign Language) English materials. From this figure, we can find the write time is short when the file size is relatively small, and increases obviously as the file size scales. We set the file size to 200MB, observing that the write time with more block replicas is higher. Furthermore, the write time is not a multiple of block replica number, which means the time cost by the NameNode to nominate DataNodes to host the block replicas cannot be ignored.

Figure 5 represents the read performance of HDFS with different block replica number. From this figure, we can find that the read time increases as the file size scales. Since the client reads block contents from the closest DataNode, the read performance difference with different block replica number is negligible. When the block replica number equals to three, we can observe the read time of 10MB file is 22.362s and 200MB file is 25.266s. The reason is that the time spending on data transmission is relatively short and most of the time is cost by other network communication.

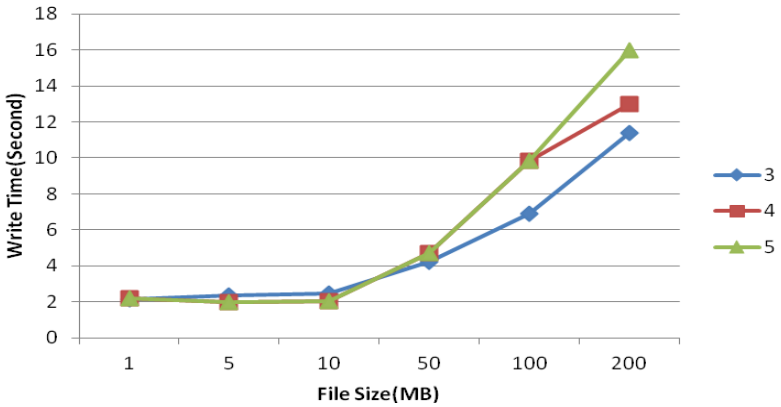


Fig. 4. Write performance of HDFS with different block replica number

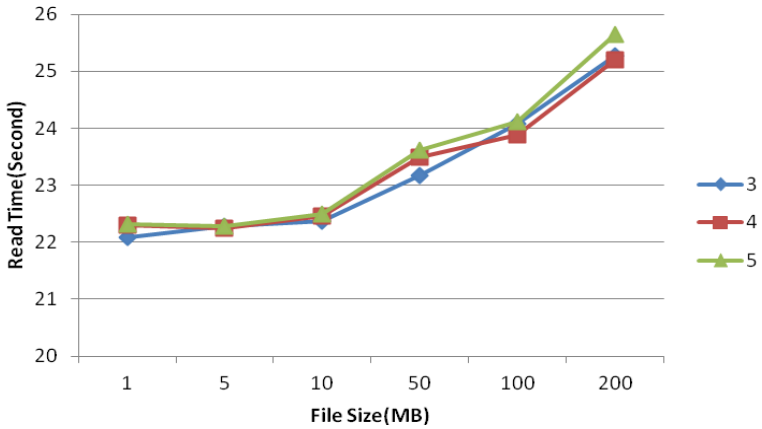


Fig. 5. Read performance of HDFS with different block replica number

Figure 6 shows the write performance of HPACS platform with different (k, n) combination. We consider three situations: (k, n) = (3, 4), (3, 5), (3, 6), and make a comparison with the HDFS of three block replicas. From this figure, we can observe that the write time of HPACS is much higher than HDFS, which means the overhead of matrix encryption algorithm occupies the vast majority of the write time. For instance, the write time of HPACS platform exceeds 4 minutes while it is less than 20s in the HDFS when the file size is 200MB. The file size has little impact on the write operation efficiency of HDFS (see the "put" performance in Figure 6). When the file size scales, the write time shows a slight elevation and doesn't change apparently. On the contrary, the file size impacts the write operation efficiency of HPACS obviously. When the file size exceeds a certain value, the write time will have a nearly linear growth.

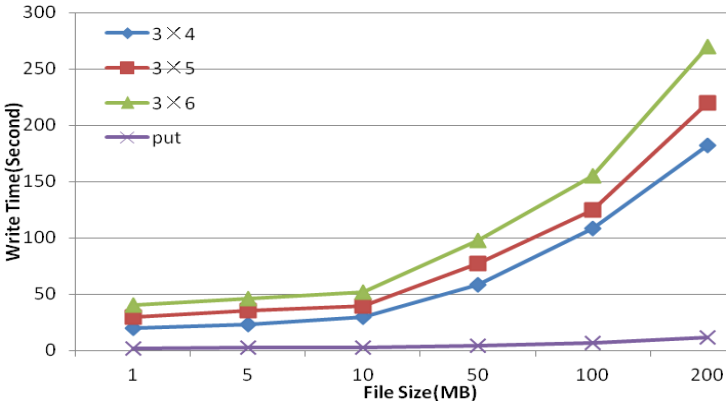


Fig. 6. Write performance of HPACS with different (k, n)

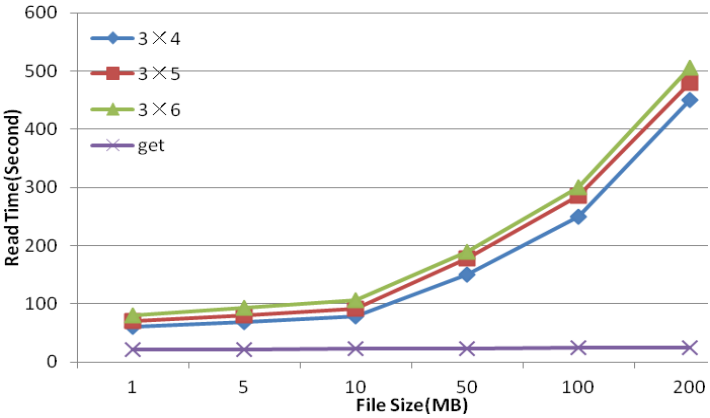


Fig. 7. Read performance of HPACS with different (k, n)

Figure 7 illustrates the read performance of HPACS platform with different (k, n). We also consider three situations: (k, n) = (3, 4), (3, 5), (3, 6), and make a comparison with the HDFS of three block replicas. From this figure, we can observe that the read time of HPACS is much higher than HDFS, which means the overhead of matrix decryption algorithm occupies the vast majority of the read time. Similar to the write performance, the file size has little impact on the read operation efficiency of HDFS (see the "get" performance in Figure 7) and has obvious impact on HPACS.

3.3 Availability Analysis

We make an analysis of the availability problem. From figure 6 and figure 7, we can observe the overhead of write/read performance is relatively low to improve the availability by increasing the redundancy rate. For example, the read time of (k, n) = (3, 4) and (3, 5) is 11.397s and 12.977s respectively. In order to reconstruct the

original data, only one partition is allowed to be destructive in the situation of (k, n) = (3, 4), while two partitions can be destroyed in the situation of (k, n) = (3, 5). We also can find that the difference of the read time is smaller than the write time among distinct (k, n), because only k partitions are needed to reconstruct the original data when we execute read operation.

4 Related Work

There are a few works on the data privacy and security in the cloud. In [9], the author discusses the criticality of the privacy issues in cloud computing, and points out that obtaining information from cloud storage provider is much easier than from the creator himself. Zhang et al. give a study on the protection method of data privacy based on cloud storage [10]. Many encryption approaches have been proposed for hiding the data from the cloud storage provider and hence preserving data privacy [11-12]. DepSky [13] presents several protocols to improve the confidentiality of data stored in the diverse clouds. It uses encryption and secret sharing to promise data safety. SCMCS [14] proposes a secured cost-effective multi-cloud storage model, which divides the user's data into pieces and distributes them among the available CSPs for providing better privacy. BLAST [15] presents a secure storage architecture enhanced with a stream cipher rather than a block cipher with a novel block accessible encryption mechanism based on streaming ciphers. Focusing on the privacy protection of the on-disk state, BIFS [16] reorders data in user files at the bit level, and stores bit slices at distributed locations in the storage system. While providing strong privacy protection, BIFS still retains part of the regularity in user data, and thus enables the CSP to perform a certain level of capacity space optimization.

RACS [17] applies erasure coding to improve availability and tolerate provider price hikes, thus reducing cost of data migrations and vendor lock-in. HAIL [18] is another system trying to build an independent integration layer through erasure coding to achieve high availability. Due to the missing consideration of global access experience, paper [19] proposes the μLibCloud system. It works as a library at client side, transparently spreading and collecting data to/from different cloud providers through erasure code, aiming to improve the availability and global access experience of clouds, and to tolerate provider failures and outages. However, all the above works don't refer to the Hadoop system directly.

Currently, Hadoop has no specific availability support yet, and it is not trivial to enhance its privacy and availability. The work [20] proposes a metadata replication based solution to enable high availability by removing single point of failure. Intel [21] proposes a fast low-overhead encryption method for Apache Hadoop, which enables real-time analytics on massive data sets with enterprise-class data protection. zNcrypt [22] outlines some of the challenges associated with securing big data and offers tips for protecting your most important business asset. However, they haven't taken the privacy and availability problem into account simultaneously.

Through the analysis of previous works, we find few works focus on the combinative measurements to solve the privacy and availability problems. The matrix encryption hasn't applied in cloud storage system. Most of the studies improve them at the level of different cloud storage providers. The HPACS platform built on Hadoop can improve privacy and availability simultaneously and is also suitable for the private cloud.

5 Conclusions and Future Work

After the cloud computing paradigm proposed as a pay-per-use service business model, cloud storage service is becoming increasingly popular. In this paper, we design and implement a HPACS platform built on Apache Hadoop in order to improve the data privacy and availability in the cloud. In the HPACS platform, we add a matrix encryption/decryption module in the slave nodes. In experiments, we first compare the write and read performance of HDFS with different block replica number. The result shows that the write and read time increases as the file size scales. When the block replica number scales, the write time increases simultaneously, while the read time has no obvious changes. Then we conduct a series of experiments to contrast the write/read performance between HDFS and HPACS platform. We find the performance of HDFS is higher than HPACS. The file size has little impact on the write/read operation efficiency of HDFS and has obvious impact on HPACS. Finally, we observe the overhead of performance is low and acceptable to improve the availability by increasing the redundancy rate.

Future work will include using real-time encryption technology in the matrix encryption and decryption algorithm to improve the encryption/decryption efficiency, and integrating the algorithms to other open-source cloud computing system to prove the effectiveness.

Acknowledgments. This work is supported by National High Technology Research 863 Major Program of China (No.2011AA01A207) and National Natural Science Foundation of China (No.61272128).

References

1. Vaquero, L.M., Rodero-Merino, L., Caceres, J., Lindner, M.: A break in the clouds: towards a cloud definition. ACM SIGCOMM Computer Communication Review 39, 50–55 (2009)
2. CSC cloud usage index, `http://www.csc.com/`
3. Microsoft SkyDrive cloud storage platform, `https://skydrive.live.com`
4. Amazon S3 service level agreement, `http://aws.amazon.com/s3-sla/`
5. Bialecki, A., Cafarella, M., Cutting, D.: O'MALLEY: Hadoop: a framework for running applications on large clusters built of commodity hardware (2005), Wiki at `http://lucene.apache.org/hadoop`
6. Serious cloud failures and disasters of 2011 (2011), `http://www.cloudways.com/blog/cloud-failures-disasters-of-2011/`

7. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10. IEEE, Florida (2010)
8. Parakh, A., Kak, S.: Online data storage using implicit security. Information Sciences 179(19), 3323–3331 (2009)
9. Cavoukian, A.: Privacy in the clouds. Identity in the Information Society 1(1), 89–108 (2008)
10. Zhang, S., Li, X., Wang, B.: Study on the Protection Method of Data Privacy Based on Cloud Storage. International Journal of Information and Computer Science 1(2) (2012)
11. Shin, S.H., Kobara, K.: Towards secure cloud storage. Demo for CloudCom2010 (2010)
12. Wang, C., Wang, Q., Ren, K., et al.: Privacy-preserving public auditing for data storage security in cloud computing. In: 2010 IEEE INFOCOM, pp. 1–9. IEEE, Florida (2010)
13. Bessani, A., Correia, M., Quaresma, B., et al.: DepSky: dependable and secure storage in a cloud-of-clouds. In: The 6th ACM Conference of Computer Systems (EuroSys 2011), pp. 31–46. ACM, Washington (2011)
14. Singh, Y., Kandah, F., Zhang, W.: A Secured cost-effective multi-cloud storage in cloud computing. In: 2011 IEEE Computer Communications Workshops, pp. 619–624. IEEE, Florida (2011)
15. Park, K.W., Kim, C., Park, K.H.: Blast: Applying streaming ciphers into outsourced cloud storage. In: 2010 IEEE Parallel and Distributed Systems (PADS), pp. 431–437. IEEE, Florida (2010)
16. Sheng, Z., Ma, Z., Gu, L., et al.: A privacy-protecting file system on public cloud storage. In: 2011 IEEE Cloud and Service Computing (CSC), pp. 141–149. IEEE, Florida (2011)
17. Abu-Libdeh, H., Princehouse, L., Weatherspoon, H.: RACS: a case for cloud storage diversity. In: The 1st ACM Symposium on Cloud Computing (SOCC), pp. 229–240. ACM, Washington (2003)
18. Bowers, K.D., Juels, A., Oprea, A.: HAIL: a high availability and integrity layer for cloud storage. In: The 16th ACM Conference on Computer and Communications Security (CCS), pp. 187–198. ACM, Washington (2009)
19. Mu, S., Chen, K., Gao, P., Ye, F., Wu, Y.W., Zheng, W.M.: μLibCloud: providing high available and uniform accessing to multiple cloud storages. In: 2012 13th ACM/IEEE International Conference on Grid Computing (GRID), pp. 201–208 (2012)
20. Wang, F., Qiu, J., Yang, J., et al.: Hadoop high availability through metadata replication. In: The First ACM International workshop on Cloud Data Management, pp. 37–44. ACM, Washington (2009)
21. Kadim, I.: Fast, low-overhead encryption for Apache Hadoop, `https://hadoop.intel.com/pdfs/IntelEncryptionforHadoopSolutionBrief.pdf`
22. Securing big data – what every organization needs to know. Gazzang's whitepaper, `http://www.gazzang.com/products/zncrypt/apache-hadoop`

Appendix I

Algorithm 1: (k, n) data encode

```
Input: original data
   Begin:
     int i, j, k, prime, m, n, element[n];
     int root[k], partition[n], vandermonde[n][k];
```

```
for(i = 0; i < n; i++){
    element[i] = (int)Math.rint(Math.random() * 99 + 1);
    for(j = 0; j < k; j++)
       vandermonde[i][j] = (int)Math.pow(element[i], j);
}
int data=reader.read(); //read data from the block packet
while(data != -1){
   m = 1;
   for(i = 0; i < k-1; i++){
     root[i] = (int)Math.rint(Math.random()*(prime-2)+1);
     m *= root[i];
   }
   for(j = 1; j < prime; j++){
       if((m * j) % prime == data)
       break;
   }
   root[k-1] = j;       //obtain the k roots
   matrix R = new matrix(root, k);
   matrix P = vandermonde.times(R);
   //obtain n partitions
   partition = P.getColumnPackedCopy();
   data = reader.read();  //until the last byte of block
}
 end.
```

Output: *n* partitions

Algorithm 2: (*k*, *n*) data reconstruct

Input: *k* partitions of $\{p_1, p_2, \ldots, p_n\}$

```
 Begin:
int i, j, k, prime, res;
int root[k], element[k], vandermonde[n][k];
for(i = 0; i < k; i++){//obtain the transformation matrix
          element[i] = reader.readInt();
   for(j = 0; j < k; j++)
      vandermonde[i][j] = (int)Math.pow(element[i], k);
}
matrix A = new Matrix(vandermonde);
matrix B = A.inverse();
for(i = 0; i < k; i++){//obtain k partitions
   partition = raf.readInt();
   element[i] = partition;
}
matrix C = new Matrix(element, k);
matrix R = B.time(C);
root = R.getColumnPackedCopy(); //calculate the k roots
res = 1;
for(i = 0; i < k; i++)
   res *= res[i];
res += prime;
res %= prime;
data = ((int)Math.round(res)) % prime;
 end.
```

Output: original data

ECAM: An Efficient Cache Management Strategy for Address Mappings in Flash Translation Layer

Xuchao Xie, Qiong Li, Dengping Wei, Zhenlong Song, and Liquan Xiao

School of Computer, National University of Defense Technology,
Changsha, China, 410073
xiexuchao@foxmail.com, qiong_joan_li@yahoo.com.cn, dpwei@nudt.edu.cn,
songzhl@sina.com, marshell.xiao@gmail.com

Abstract. Solid State Drives (SSDs) have been widely adopted in both enterprise and embedded storage systems with the great improvement in NAND flash memory technology. With the growing size of NAND flash memory, how to keep the most active address mappings be cached in limited on-flash SRAM is crucial to a Flash Translation Layer (FTL) scheme, that plays an important role in managing NAND flash. In this paper, we propose an efficient cache management strategy, called ECAM, to enhance the capability of caching page-level address mappings in demand-based Flash Translation Layer. In ECAM, we optimize the structure of Cached Mapping Table (CMT) to record multiple address mappings with consecutive logical page numbers and physical page numbers in just one mapping entry, and propose another two tables, Cached Split Table (CST) and Cached Translation Table (CTT). CST can cache the split mapping entries caused by the partial updates in CMT and CTT is used to reduce the overhead of address translation for large number of sequential requests. By the cooperation of CMT, CST and CTT, ECAM implements an efficient two-tier selective caching strategy to jointly exploit the temporal and spatial localities of workloads. The simulation on various realistic workloads shows that ECAM can improve the cache hit ratio and reduce the number of expensive extra read/write operations between SRAM and flash efficiently.

Keywords: SSD, FTL, NAND Flash, Cache Management Strategy, Cached Split Table, Cached Translation Table.

1 Introduction

In the past decade, a number of excellent research results and significant advances have been obtained in flash memory technologies, especially NAND flash memory. NAND flash is a kind of erase-before-write non-volatile memory and has a limited number of erase cycles. Due to its advantages such as low access latency, low power consumption etc., NAND flash memory has become an important substitute of the traditional mechanical Hard Disk Drives (HDDs) in the

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 146–159, 2013.
© Springer-Verlag Berlin Heidelberg 2013

form of Solid State Drives (SSDs) and received strong interests in both academia and industry[1][3].

In NAND flash memory, the granularity of an erase operation is a block that usually contains 32-128 pages, while that of a write operation is a page. Because of its erase-before-write feature, NAND flash memory does not support in-place-update, which would require an erase-per-update and degrade the performance[3]. In order to avoid unnecessary erase operations, NAND flash-based SSDs usually exploit an out-place-update pattern, which updates one page in an already erased page, remaps the address mapping table to reflect this change and invalidates the old page. To reclaim the invalid pages caused by the out-place-update pattern, a garbage collection mechanism is introduced. Moreover, wear-leveling technique that can reduce the number of an erase operation and balance the erase count of each block is also an important issue because of the limited lifespan of NAND flash memory.

To realize these functions, a firmware layer named flash translation layer (FTL) is implemented in the controller of SSDs, which is crucial to the overall performance of NAND flash memory based storage system. FTL is the core engine of SSDs to translate logical addresses to physical addresses on flash, which is named address mapping. Various FTL schemes have been proposed such as page-level FTL, block-level FTL[2][6] and hybrid FTL[10][13][14][15] schemes. The page mapping table of page-level FTL usually cannot be kept in limited on-flash SRAM with the increasing size of NAND flash. Besides, both block-level FTL and hybrid FTL schemes exhibit poor performance for enterprise-scale workloads with significant random write patterns. In this situation, demand-based Flash Translation Layer (DFTL)[7] is proposed which selectively caches page-level address mappings to overcome the shortcomings of the current FTL schemes above.

DFTL shows great advantages in response time and garbage collection overhead. However, the heavy I/O overhead between SRAM and flash when the needed address mapping missed in SRAM may be a burden as the size of NAND flash memory increases. And DFTL takes only temporal locality into consideration, in which the spatial locality of enterprise-scale workloads is not considered.

In this paper, we mainly focus on improving the ability of address translation without consuming more resources based on DFTL. We propose an efficient cache management strategy called ECAM to make best use of the limited space of on-flash SRAM. In ECAM, we optimize the structure of Cached Mapping Table (CMT) and propose another two tables, Cached Split Table (CST) and Cached Translation Table (CTT). With the help of CMT, CST and CTT, ECAM achieves an efficient two-tier selective caching strategy that can jointly exploit the temporal and spatial localities of workloads.

The rest of this paper is organized as follows. Section 2 provides a brief overview of related work. In Section 3, the details of ECAM are presented. Experimental results are presented in Section 4. Finally, our conclusions and future work are described in Section 5.

2 Related Work

FTL is a firmware implemented in SSD controller and plays an important role in providing address mapping, wear-leveling and garbage collection. Because of the out-place-update pattern of NAND flash memory, address mapping table is required to keep the address translation between logical address and physical flash address. Current FTL schemes can be classified into page-level, block-level and hybrid FTLs depending on the granularity of address mapping[5].

Page-level FTL scheme is the best grained and the most flexible scheme translating a logical page number (LPN) to a physical page number (PPN) in NAND flash memory. Page-level FTL can get a very fast address translation and write a data in any place of flash memory since it maintains an address translation table containing the whole LPNs and their corresponding PPNs. Garbage collection can also benefit from page-level mapping and obtain high efficiency. However, this solution is very expensive since page mapping table usually cannot be kept in SRAM because of its large size.

In order to reduce the size of address mapping table, another scheme, i.e., block-level FTL scheme is proposed, which translate only logical block number (LBN) to physical block number (PBN) and can significantly lower memory requirement. In block-level FTL scheme, a logical address includes a LBN of the block and the offset in the block. The target page address can be found according to the PBN translated from LBN and the logical page offset. However, since block-level FTL only stores one logical page in the fixed position of blocks, it may lead to large number of internal fragments and expensive garbage collection overhead when the coming requests only overwrite part of a block constantly.

In order to get a tradeoff between performance and consumption of limited system resource, various hybrid mapping schemes between page-level FTL and block-level FTL have been proposed in literatures[10][13][14][15], in which blocks are divided into data blocks and log blocks logically. Updates on the data blocks are always written to log blocks. Fully Associative Sector Translation (FAST)[14] allows log blocks to be shared by all data blocks and reserves a sequential log block to perform sequential updates. However, the high correlation of a log block among large number of data blocks makes the reclaim more expensive and gets a higher possibility of the full merge operations caused by its fully associative policy. Locality Aware Sector Translation (LAST)[15] deals with sequential writes and random writes separately according to the numbers of requested pages. FASTer[16] focuses on online transaction processing (OLTP) systems based on the FAST FTL scheme, which has the best performance when the workloads are generally random and small I/O requests just like OLTP systems.

However, in most enterprise-scale servers, the access patterns of workloads are composed of sequential and random writes in different ratios. A recent proposal, WAFTL[20] makes FTL workload adaptive by sending the data cached in buffer zone to either page-level mapping blocks or block-level mapping blocks depending on their access frequencies. ADAPT[19] is a fully-associative hybrid mapping FTL and employs a novel way to manage log space for different

workloads. ADAPT can adjusts the partitions of log space dynamically to satisfy the sequential and random write requests at runtime.

DFTL[7] is another FTL scheme which selectively caches partial page-level address mappings rather than the whole address translation table in SRAM. Cached mapping table (CMT), global translation directory (GTD) in SRAM, and the entire page-level address translation table on NAND flash memory together realized the translation from a logical page number to a physical page number. CMT keeps active address mapping entries on demand in SRAM to minimize the overhead of address translation by using the temporal locality of workloads and GTD can track all physically dispersed translation pages over the entire NAND flash memory. Though DFTL improves overall performance significantly, the heavy I/O overhead between SRAM and flash during the address translation may be a burden with the size of NAND flash memory increasing.

A recent proposal, S-FTL[9], exploits the spatial locality in the workloads to reduce the page mapping table size and extra I/O overhead during address translation. S-FTL exploits continuity in the write requests to create a concise representation of in-cache translation pages by maintaining the bitmap for translation pages. CAST[21] uses a compact packing methodology for the page-level address mappings in SRAM. However, the caching mechanism in CAST may cause several extra eviction operations when a small write request reaches. Recently, content-aware Flash Translation Layer (CAFTL)[4] and CA-SSD[8] are proposed to enhance the endurance of SSDs by removing unnecessary duplicate writes to flash memory.

3 The Design of ECAM

3.1 Overall Architecture

The overall architecture of ECAM is shown in Figure 1. In ECAM, we design four tables in SRAM logically, cached mapping table (CMT), cached split table (CST), cached translation table (CTT) and global translation directory (GTD). CMT is used to cache the address translation entries of on-demand active data page, and GTD keeps track of all translation pages on flash, just as DFTL. Unlike DFTL, we change the structure of CMT and propose another two tables, CST and CTT, which are the most important parts to jointly exploit the temporal and spatial localities of workloads. In ECAM, CST can cache the split mapping entries caused by the partial updates in CMT and CTT is used to reduce the overhead of address translation for large number of sequential requests. The segmented LRU array cache management algorithm[11] for replacement is used in CMT and CST. Flash memory is divided into two partitions in ECAM, the translation block and the data block. One Translation block is composed of several translation pages that are special pages used to store address mappings on flash, while one data block is composed of several data pages that store real data accessed during read/write operations.

CMT and CST in ECAM have three parts, i.e., LPN, PPN and SIZE. LPN indicates the logical page number, PPN indicates the physical page number

corresponding to LPN, and SIZE indicates the number of consecutive pages whose starting address is the LPN in this mapping entry. SIZE makes one cached mapping entry record more than one address mapping information, which is beneficial to requests with consecutive LPNs and consecutive PPNs.

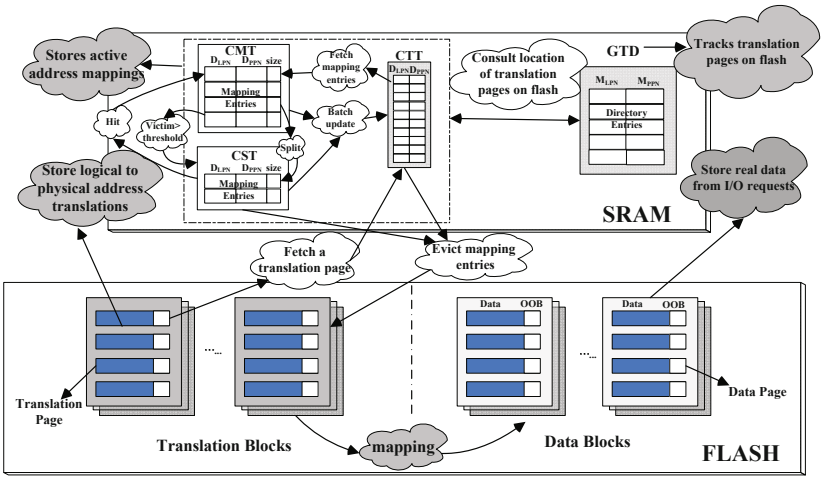


Fig. 1. The Architecture of ECAM

With the optimization of the CMT structure, a mapping entry with a big size needs to be updated when a small write request write partial pages. The updated mapping entry will split into several mapping entries because of the out-place-update characteristic of NAND flash. We call it split operation. In this case, ECAM updates on mapping entry, called hit entry, in CMT, and caches other mapping entries, called split entries, into CST rather than just retain them in CMT or write back to the translation page in flash directly.

When a requested address mapping entry missed in SRAM, ECAM needs to fetch the needed entry from translation page. Considering that the mapping information with the consecutive LPNs may be needed by the successive requests, ECAM always fetches the whole translation page containing the needed mapping entry in CTT. Besides, CTT implements a lazy batch update scheme of address mappings from SRAM to translation pages when the victims from CMT or CST are cached in CTT. ECAM caches the victims to CTT and searches the mapping entries in replace segment of the segment LRU array[11]. The entries that contained in the same translation page with the victim entry will be evicted together and update to a free translation page on flash.

3.2 The Two-Tier Selective Caching Strategy

In ECAM, we propose an efficient two-tier selective caching mechanism to cache the most active address mapping entries in SRAM by the cooperation of the optimized CMT, CST and CTT.

ECAM uses different strategies to handle read requests and write requests because of the out-place-update pattern of NAND flash memory. Since no split operation happens to the read requests, ECAM searches the needed address mapping entries from CMT, CST and CTT in sequence for all the request pages and implements the page-by-page address mapping for read request. If the needed mapping entries are not presented in SRAM, they will be fetched into CMT from flash. When a mapping entry needs to be stored in CMT that is already full, one address mapping in CMT needs to be evicted and stored in CST. In this case, CST can only be used to cache the victims evicted from CMT with the sizes bigger than threshold. For a write request, there are three situations that may happen, hit, miss and partial hit.

HIT. The hit situation happens when the needed mapping information of addresses LPN to LPN+SIZE-1 can be found in CMT or CST. A split operation may happen in this situation, ECAM just caches one or two split mapping entries in CST if needed. Figure 2(a) and 2(b) describes a hit situation. The mapping entry (11, 220, 9) in CMT can satisfy the request (D_{LPN}=15, SIZE=2), in this case, ECAM just updates the mapping information of the hit entry to (15, 420, 2) and moves it to the head of CMT. 420 is the PPN of a free page allocated to LPN=15 in Current Data Block. Besides, ECAM caches the split entries (11, 220, 4) and (17, 226, 3) in CST.

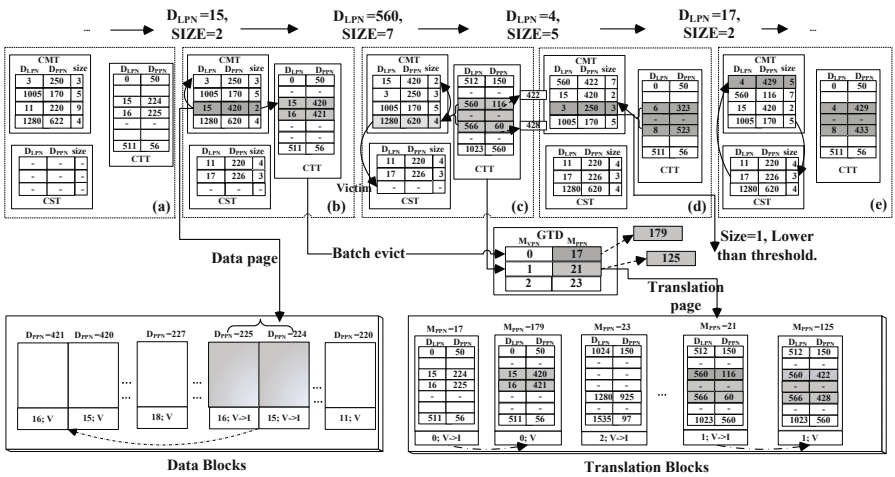


Fig. 2. An Example of Address Translation Process of ECAM

In ECAM, the split mapping entries caused by a write request are always cached in CST rather than retained in CMT or write back to the translation page in flash directly. The reason for this consideration is mainly to better jointly exploit spatial locality and temporal locality of workloads. Generally speaking,

different workloads show different access patterns, temporal locality intensive, spatial locality intensive or random access intensive. If the split entries are always designed to cache in CMT and CMT is full, some entries needed to be evicted from CMT to make space for caching the split entries. For the temporal locality intensive workloads, the evicted entries may be reused soon which are much more important than the split entries. For the spatial locality intensive workloads, numerous split operations may be triggered in a short time and CMT may be filled with large number of split entries, which will destroy the structure of segment LRU array[11] that used to organize CMT. Besides, the numerous split entries occupy most space of CMT, which go against the purpose of CMT. Therefore, we just selectively cache the split mapping entries in elsewhere, i.e., CST in ECAM, to exploit the spatial locality of workloads.

We use a selective caching strategy to better exploit the limited space of SRAM by caching the split entries in CST. In order to implement this selective caching strategy, we set a threshold to help ECAM to determine whether the split entry should be cached in CST or not. ECAM always selectively caches the split entries with SIZE lager than threshold, since the entry with larger size represents more address mapping information and will be used soon with more probability. Therefore, the victims with large size from CMT will be cached in CST, ECAM can get them from CST directly if the mappings hit soon, rather than read from translation pages in flash. The mapping entries whose sizes are smaller than the threshold will be evicted to flash instead. We set the threshold of size to 2 in ECAM. In the example shown in Figure 2(b), the two split entries (11, 220, 4) and (17, 226, 3) are both cached in CST because the sizes of the two split entries are bigger than the threshold.

MISS. If the mapping information of the requested LPN cannot be found in SRAM (CMT, CST or CTT), the miss situation happens. We have to read the translation page containing the needed mapping information from flash, and cache the corresponding mapping entry in CMT. For example, the LPN of request (D_{LPN}=560, SIZE=7) is 560 shown in Figure 2(c), and this address is mismatching in CMT, CST and CTT. The translation page (M_{PPN}=21) in flash contains the mapping information of this request address, and ECAM just reads it to CTT. Since CTT contains the whole address mappings of translation page (M_{VPN}=0, M_{PPN}=17) and there are two mapping entries (D_{LPN}=15 and D_{LPN}=16) updated in last request, ECAM has to batch update the (M_{VPN}=0) mapping to a free translation page (M_{PPN}=179). After that, ECAM just read the translation page (M_{PPN}=21) into CTT, which will batch satisfy the request (D_{LPN}=560, SIZE=7) in one time. While CMT is full, a victim (1280, 620, 4) has to be evicted to make space for the needed address mappings of request (D_{LPN}=560, SIZE=7). In the end, an address mapping entry (560, 422, 7) is added in the head of CMT.

ECAM always selective cache the victims evicted from CMT depends on the SIZE, just as we explained above. The victims with large size from CMT will be cached in CST while the one with small size will be evicted to flash instead. Therefore, the victim (1280, 620, 4) is cached into CST, since the size of this entry is 4.

PARTIAL HIT. The partial hit situation happens if the needed mapping information can be partially found in SRAM. In this case, the request can be divided into two sub-requests, (LPN, SIZE1) and (LPN+SIZE1, SIZE-SIZE1). One request (LPN, SIZE1) can be satisfied in SRAM, just as the hit situation we mentioned, the other (LPN+SIZE1, SIZE-SIZE1) is treated just as a new write request, whose mapping information is then searched in SRAM or Flash from scratch. Just as shown in Figure 2(d), the request (D_{LPN}=4, SIZE=5) is partial hit in SRAM, since the request (D_{LPN}=4, SIZE=2) can be satisfied in address entry (3, 250, 3), while the request (D_{LPN}=6, SIZE=3) is mismatched in SRAM and the mapping information has to be fetched from flash before the request (D_{LPN}=6, SIZE=3) is handled, the partial hit entry (3, 250, 3) is updated to (4, 429, 5), which is moved to the head of CMT. The split entry (3, 250, 1) cannot be stored in CST and need to be written back to flash, since its size is smaller than the threshold 2.

The mapping entries in SRAM will be written to translation pages when CST is full and needs to make space for new mapping entries.

4 Performance Evaluation

4.1 Evaluation Setup

We performed a trace-driven simulation to evaluate ECAM for managing a 32GB NAND flash memory with configurations shown in Table 1. We choose DFTL for comparison, since DFTL has been shown that it has better performance than both block-level and hybrid-level mapping schemes[7]. To better evaluate the impact of CST and CTT on the performance of reducing the extra read/write operations, we implement another two FTL schemes, ECAM without CST and ECAM without CTT. We implement this evaluation with four configurations of cache size for address mappings, which is set to 64KB, 128KB, 256KB and 512KB.

Table 1. Parameters of Simulated SSD

Parameters		Values
SSD Capacity		32GB
Page Size, Block Size		4KB, 256KB
Time for Page Read, Page Write, Block Erase		25us, 200us, 1.5ms
Program/Erase Cycles		100K
Cache Size	DFTL	CMT=X KB, (X=64/128/256/512)
	ECAM without CST	CST=0KB, CTT=4KB, CMT=(X-4)KB
	ECAM without CTT	CST=4KB, CTT=0KB, CMT=(X-4)KB
	ECAM	CST=4KB, CTT=4KB, CMT=(X-8)KB

In our evaluation, various types of traces from realistic workloads we selected are shown in Table 2. Financial traces and WebSearch trace[12] are available from Storage Performance Council (SPC). Financial1 trace is obtained from OLTP

applications running at a financial institution as a temporal write intensive trace. Financial2 trace is also made from OLTP applications and used as a type of random performance measurements. WebSearch2 trace is obtained from popular search engines and shows a read intensive access pattern with larger numbers of sequential read requests. MSR trace is the mds_0 trace[17] that collected on media servers by Microsoft Research Cambridge. The Windisk trace is a trace collected from a laptop during the installation of a large application by Diskmon[18]. MSR trace and Windisk trace are write-intensive traces with large number of sequential write requests. In our simulation, a request is treated as sequential request if the size exceeds 16 sectors.

Table 2. Trace Characteristics

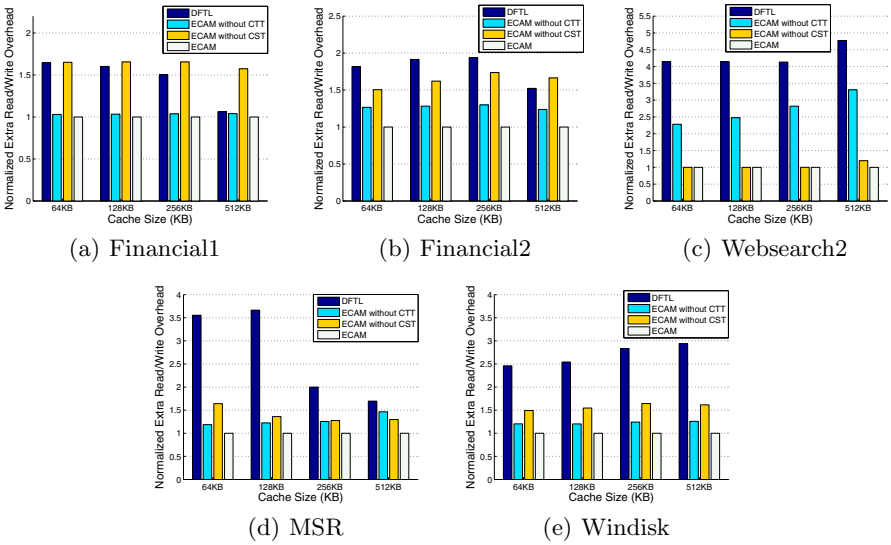
Trace	Total requests	Write(%)	Seq. Write(%)	Seq. read(%)
Financial1	5334987	76.84	4.69	1.22
Financial2	3699195	17.65	1.22	2.91
WebSearch2	4579809	0.02	0	99.97
MSR	1211034	88.11	13.44	5.38
Windisk	244863	72.25	22.52	15.47

4.2 Extra Read/Write Overhead between SRAM and Flash

Because of the special characteristic of NAND flash memory, the high expensive extra read/write operations between SRAM and flash heavily impact the storage system performance. In this experiment, we analyze the effectiveness of ECAM in reducing the number of extra read/write operations. As shown in Figure 3, ECAM has less extra read/write operations between SRAM and translation pages in flash than other schemes.

As shown in Figure 3, both CST and CTT play an important role in reducing the number of extra read/write operations. For write intensive and random access workloads such as Financial1 and Financial2 (shown in Figure 3(a) and Figure 3(b)), we can find that CST is much more important than CTT, which shows the importance of the split entries of address mapping information in demand-based FTL schemes. Just as we described above, this is mainly because that CST in ECAM can cache both the split mapping entries and the victims with a big enough size evicted from CMT. By the evaluations in Figure 3(a) and 3(b), we can conclude that the split entries caused by partial update in CMT have great importance in demand-based address mapping schemes. That is the reason why we cache the split entries in CST separately rather than cache them elsewhere or just evict them from SRAM.

For the sequential read dominated workloads such as WebSearch2, CTT is much more efficient. Just as we described, CTT can cache the mapping entries of one latest used translation page for spatial locality intensive workloads. As shown in Figure 3(c), ECAM can achieve an average 14.8% reduction on extra read/write overhead between SRAM and flash when the cache size is set to



(a) Financial1 (b) Financial2 (c) Websearch2

(d) MSR (e) Windisk

Fig. 3. Extra Read/Write Overhead

64KB. For the workloads with large number of sequential write requests such as MSR trace and Windisk trace, ECAM can get a great improvement on different cache sizes. For example, when the cache size for address mapping is set to 64KB, ECAM can get an improvement up to 71.86% and 59.34%. We can find from the evaluation results that the effects of CST and CTT are similar in this kind of workload. This is mainly because the access characteristics of sequential write requests, i.e., the sequential write intensive workloads have large number of write requests with strong spatial locality, which can be solved by CTT and CST efficiently.

4.3 Cache Hit Ratio

Cache hit ratio is one of the most important factors in the performance of NAND flash memory based storage system.

Figure 4 shows the cache hit ratio of DFTL and ECAM on each workload with different cache sizes with configurations shown in Table 1. As the figure shown, the increment of hit ratio is obvious with the cache size growing. This is mainly because both DFTL and ECAM can cache more temporal request in CMT by using the segment LRU array to record mapping entries in SRAM. The cache hit ratios of ECAM are always higher than DFTL on the same workload. That indicates the effectiveness of the cooperation of CMT, CST and CTT even the size of CMT is smaller than that in DFTL.

For both write intensive and random access workloads such as Financial1 and Financial2 traces, ECAM can get about 9.3%, 8.5%, 7.2%, 3.8% and 10.2%, 7.4%, 5.1%, 2.6% improvement than DFTL on the cache hit ratio with 64KB,

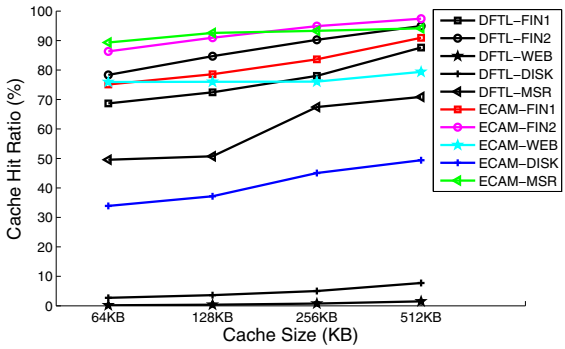


Fig. 4. Cache Hit Ratios

128KB, 256KB and 512KB cache respectively. This is mainly because the CST in ECAM can cache both the split mapping entries and the victims with a big enough size evicted from CMT, which may be reused soon, by using our two-tier selective caching mechanism. ECAM can efficiently take advantage of CST with the growing size of cache.

For the workloads with large number of sequential write requests such as the mds_0 trace, ECAM can get an improvement up to 80.3%, 82.5%, 38.35% and 32.8% when the SRAM cache is set to 64KB, 128KB, 256KB and 512KB. This is because CST in ECAM can cache the split mapping entries efficiently, and CTT can cache the mapping entries of one latest used translation page for workloads dominated by sequential request that is spatial locality intensive and can be satisfied timely with the mapping entries in SRAM without searching the entries in translation pages. Besides, for a sequential write request involving $n(n \geqslant 2)$ consecutive pages, DFTL has to maintain n address mapping entries while ECAM needs to maintain only one address mapping entry with a size of n. The improving of total numbers of address mapping information represented in ECAM will benefit cache hit ratio evidently.

For the sequential read dominated workload such as websearch2 trace, CST may be helpless to cache the split entries since there is little split operations in read dominated workloads. In this case, CST can only be used to cache the victims evicted from CMT with size bigger than threshold. Since CTT helps ECAM to use a translation page as the caching unit, ECAM can fully exploit the spatial locality of the sequential read requests in websearch2 to satisfy the needed address mappings. For example, for a sequential read request involving $n(n \geqslant 2)$ consecutive pages whose needed address mapping entries have not cached in SRAM, DFTL has to read the needed mapping entry of each page one by one, which will cause n times of cache misses. Unlike DFTL, ECAM always use a translation page as the caching unit by introducing CTT. Therefore, ECAM can satisfy them just by reading the related translation page in one time, which will just cause 1 time of cache miss and n-1 times of cache hits. That is the reason why ECAM can improve the cache hit ratio and reduce the extra read/write

operations between SRAM and flash simultaneously for the WebSearch trace. We can conclude that CTT always benefit the successive requests with consecutive LPNs significantly, especially in the enterprise-scale environments.

4.4 Impact of CST Size on Reducing Extra Read/Write Overhead

As we have shown the performance of ECAM with a 4KB CST size in various workloads, we would like to see the impact of CST size on reducing the extra read/write overhead. Figure 5 shows the extra read/write ratio of each workload with CST size varying in a large range.

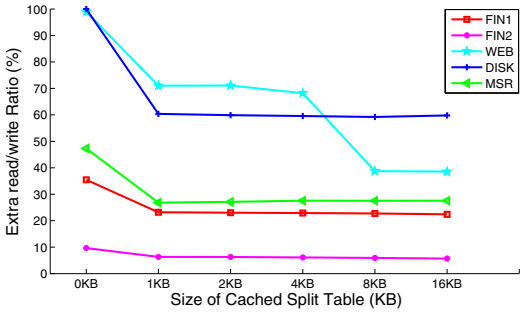


Fig. 5. Impact of CST size in ECAM

Just as shown in Figure 5, when CST size is set to 1KB, i.e., 128 mapping entries in CST, the impact of CST is obvious for each workload. The extra read/write ratio decreased rapidly and become stable with the increasing of CST size. Every workload decreases with different speed, Windisk decreases fastest and Finaccial2 decreases much slower than others. As the statistic results shown in Table 2, Financial2 has the minimum sequential write requests and Windisk has the maximum sequential writes. CST is used to cache the split entries and victims with large sizes from CMT in ECAM. Sequential requests are the origin of split operations in CMT. Besides, we found that the split entries cached in CST are very important to the subsequent requests. CST is the key point to decrease the extra read/write overhead in ECAM. The more sequential writes requests, the more reduction of extra read/write overhead in ECAM.

When CST size increased to 4KB, the ratios are stable except for workload Websearch2. By analyzing the characteristic of Websearch2 trace, we can find that the reason why CST is always efficient for Websearch2 with CST size increasing. Websearch2 trace has large number of sequential read requests with big sizes. The two-tier selective caching mechanism in ECAM always uses CST to cache the victims with large sizes from CMT since the size part in CMT represents the range of the address mapping in one entry. The bigger the size is, the more probability the entry will be used soon, especially for the temporal and spatial localities intensive enterprise-scale workloads.

5 Conclusions and Future Work

In this paper, we proposed ECAM, a novel strategy to enhance the caching capacity of address mappings for page-level flash translation layer. In ECAM, we optimized the structure of CMT and propose another two tables CST and CTT to better use the spatial locality of sequential requests, and an efficient two-tier selective caching strategy is designed in ECAM to cache the most active address mapping entries in SRAM. ECAM exploits the spatial locality of the sequential requests to satisfy the needed address mappings and update the overwritten mappings to flash in batches rather than one by one. Our experiments show that ECAM can improve the cache hit ratio and reduce the number of extra read/write operations significantly on various realistic workloads by the cooperation of CMT, CST and CTT, especially for the workloads with large number of sequential write requests.

Our future work will focus on improving our strategy so that it can explore the characteristics of workloads dynamically and then making the strategy adaptive to various workloads to achieve better performance. Besides, it is an interesting work to enhance the endurance of SSDs by removing the unnecessary duplicate writes to flash memory.

Acknowledgement. This research is partially supported by the National Natural Science Foundation of China (NSFC) NO.61202118 and the National High Technology Research and Development Program of China (863 Program) No. 2012AA01A301.

References

1. Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J.D., Manasse, M., Panigrahy, R.: Design tradeoffs for ssd performance. In: USENIX 2008 Annual Technical Conference on Annual Technical Conference, pp. 57–70 (2008)
2. Ban, A.: Flash file system optimized for page-mode flash technologies (August 10, 1999), uS Patent 5,937,425
3. Chen, F., Koufaty, D.A., Zhang, X.: Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In: Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, pp. 181–192. ACM (2009)
4. Chen, F., Luo, T., Zhang, X.: Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In: Proceedings of the 9th USENIX Conference on File and Stroage Technologies, p. 6. USENIX Association (2011)
5. Chung, T.S., Park, D.J., Park, S., Lee, D.H., Lee, S.W., Song, H.J.: A survey of flash translation layer. Journal of Systems Architecture 55(5), 332–343 (2009)
6. Estakhri, P., Iman, B.: Moving sequential sectors within a block of information in a flash memory mass storage architecture (July 27, 1999), uS Patent 5,930,815
7. Gupta, A., Kim, Y., Urgaonkar, B.: Dftl: A flash translation layer employing demand-based selective of page-level address mapping. In: Proc. of ASPLOS, vol. 9, pp. 7–11 (2009)

8. Gupta, A., Pisolkar, R., Urgaonkar, B., Sivasubramaniam, A.: Leveraging value locality in optimizing nand flash-based ssds. In: Proceedings of the 9th USENIX Conference on File and Stroage Technologies, p. 7. USENIX Association (2011)
9. Jiang, S., Zhang, L., Yuan, X., Hu, H., Chen, Y.: S-ftl: An efficient address translation for flash memory by exploiting spatial locality. In: 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–12. IEEE (2011)
10. Jung, D., Kang, J.U., Jo, H., Kim, J.S., Lee, J.: Superblock ftl: A superblock-based flash translation layer with a hybrid address translation scheme. ACM Transactions on Embedded Computing Systems (TECS) 9(4), 40 (2010)
11. Karedla, R., Love, J.S., Wherry, B.G.: Caching strategies to improve disk system performance. Computer 27(3), 38–46 (1994)
12. Bates, K., McNutt, B.: Websearch trace from umass trace traces. The Storage Performance Council (2002), `http://traces.cs.umass.edu/index.php/Storage/Storage`
13. Kim, J., Kim, J.M., Noh, S.H., Min, S.L., Cho, Y.: A space-efficient flash translation layer for compactflash systems. IEEE Transactions on Consumer Electronics 48(2), 366–375 (2002)
14. Lee, S.W., Park, D.J., Chung, T.S., Lee, D.H., Park, S., Song, H.J.: A log buffer-based flash translation layer using fully-associative sector translation. ACM Transactions on Embedded Computing Systems (TECS) 6(3), 18 (2007)
15. Lee, S., Shin, D., Kim, Y.J., Kim, J.: Last: locality-aware sector translation for nand flash memory-based storage systems. ACM SIGOPS Operating Systems Review 42(6), 36–42 (2008)
16. Lim, S.P., Lee, S.W., Moon, B.: Faster ftl for enterprise-class flash memory ssds. In: 2010 International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI), pp. 3–12. IEEE (2010)
17. Narayanan, D., Donnelly, A., Rowstron, A.: Write off-loading: Practical power management for enterprise storage. ACM Transactions on Storage (TOS) 4(3), 10 (2008)
18. Russinovich, M.: Diskmon for windows v2.01 (2006), `http://technet.microsoft.com/en-us/Sysinternals/Bb896646.aspx`
19. Wang, C., Wong, W.F.: Adapt: Efficient workload-sensitive flash management based on adaptation, prediction and aggregation. In: 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–12. IEEE (2012)
20. Wei, Q., Gong, B., Pathak, S., Veeravalli, B., Zeng, L., Okada, K.: Waftl: A workload adaptive flash translation layer with data partition. In: 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–12. IEEE (2011)
21. Xu, Z., Li, R., Xu, C.Z.: Cast: A page-level ftl with compact address mapping and parallel data blocks. In: 2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC), pp. 142–151. IEEE (2012)

A Performance Study of Software Prefetching for Tracing Garbage Collectors

Hao Wu, Zhenzhou Ji, Suxia Zhu, and Zhigang Chen

Harbin Institute of Technology, Harbin, China
{wuhaoster,jizhenzhou,zhusuxia,chenzhg}@hit.edu.cn

Abstract. In automatic memory management programming language, the garbage collector is an important feature which reclaims memory that is no longer referenced by the program. The tracing collector performs a transitive closure across the object reference graph. And this process may result in too many cache misses, because the traverse exhibits poor locality. Previous studies have already proposed using software prefetching to improve memory performance of garbage collector. In this work, we studied the characteristic of prefetch instructions on x86 platforms in detail, and proposed a method to control the prefetch ratios by address thresholds. The experimental results show that a further improvement and optimization is obtained by tuning the prefetch ratio. We use these results to devise some guidelines for optimizing the memory performance as well as minimizing the collection pause times.

Keywords: Software Prefetching, Memory Performance, Garbage Collector, Cache Architecture.

1 Introduction

In many object-oriented programming languages such as Java and C#, the memory is automatically managed by the runtime component, e.g., java virtual machine. The programmers do not have to explicitly manage the memory that previously allocated. The memory reclaim work is performed by garbage collector, the garbage collector identifies the dead objects that will never be referenced in the heap and release the memory before the system exhausted the whole heap. However, although the garbage collector improves the security and reliability of the programs, it also impacts the performance of introducing additional overhead. When performing the memory reclamation, most of the garbage collector will stop the execution of the application, and that is so called 'stop-the-world' garbage collector. The pause time leads to a performance bottleneck that limits the response performance of the application.

The garbage collector that traverse the reference graph of the objects to identify the live and dead objects is called tracing collector. It will visit all objects reachable from the root set, and the objects which were not visited during the trace are identified as dead objects. The tracing process is a time consuming operation, because the objects may scatter throughout the whole heap, and the object graph may arbitrary reference

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 160–169, 2013.
© Springer-Verlag Berlin Heidelberg 2013

from one object to another. Previous studies [5, 13] showed that the memory performance during the tracing garbage collectors is poor, and too many cache misses are caused.

The data prefetching technology is designed to reduce or hide the latency of the main memory access which is performed by a non-blocking load instruction. There are two kinds of prefetching, hardware and software. Hardware prefetching may speculate the memory access behavior and then prefetch automatically the data before the application accessing it. While software prefetching is supported by some form of fetch instructions, and the instructions are manually added in the appropriate place of the application by the programmers. Using software prefetching to optimize the memory performance needs special care. Sometimes, it may lead to performance degradation due to the inappropriate prefetching time or place. Prefetching too early or too late will not obtain the performance optimization.

In this work, we make a study of the software prefetching to optimize the tracing garbage collector on x86 platform. We studied the 'prefetch' instruction on IA32 architecture, and we concentrate on minimizing the pause time of the tracing collector. We proposed a strategy to control prefetch ratios in order to minimize the cache pollution, and the prefetching time is dynamically adjusted. By tuning the prefetching ratios, a further performance improvement is obtained. We found that it is difficult to measure the low level instructions (e.g. prefetch) on a practical hardware platform rather than simulator. So the results can be devised as some guidelines for optimizing the memory performance as well as minimizing the collection pause times.

The rest of this paper is organized as follows; in section 2 we provided a brief overview of tracing garbage collector. Software prefetching method is studied and proposed in section 3. Experimental results are showed in section 4 and related work is discussed in section 5. Finally, we conclude in section 6.

2 The Tracing Garbage Collector

The tracing garbage collector performs the memory reclaim work when an allocation request cannot be satisfied, namely, the heap memory is going to be exhausted. The collector stops the application and takes a transitive closure to identify the dead objects which will be no longer used, and then release that memory. The application is resumed after the collection process, and this is typically how a ‘stop-the-world’ collector works.

There are two typical tracing garbage collectors, mark-sweep collector and semi-space collector, the difference is whether or not the collector performs objects moving during the reclaim phase.

- **Mark-Sweep (MS):** the mark-sweep collector is a non-moving collector, it traverses all reachable objects, recording each visited objects. After the traverse completes, all the objects that are not visited are considered as dead objects which will be swept up for later reuse.
- **Semi-Space (SS):** the semi-space collector splits the heap into two regions, one is called from-space, and the other is called to-space. All of the objects are allocated

in the from-space during program execution. When garbage collection is triggered, the collector "flips" the two spaces and copies all reachable objects from the from-space to the to-space, and all that remains in the from-space is considered as garbage. The SS collector is also based on a tracing process, but the reclaim strategy is different from that of MS collector.

Generally speaking, the SS collector needs more heap than MS collector. However, both of the collectors have their own advantages, decisions on adopting which collector depends on application's demand.

In this work, we only concrete on the traverse phase of the garbage collection, We use DaCapo [1] benchmark to measure the percentage of tracing time during the collection, all of experiments are performed on the Jikes RVM [2] platform. We recorded the total traversal and collection time, and the results are showed in table 1. The results show that the traverse work takes a large proportion between 73% and 98%, which means the tracing is a key factor affecting the memory performance of the garbage collector.

Table 1. Tracing time percentage of total GC

Benchmark program	Tracing Time		Heap Used (MB)	
	MS	SS	MS	SS
antrl	78%	86%	98	140
avrora	79%	85%	57	91
bloat	80%	88%	196	263
eclipse	83%	90%	490	525
fop	85%	87%	95	138
hsqldb	98%	98%	195	282
jython	85%	90%	535	623
luindex	77%	83%	50	88
lusearch	77%	84%	230	216
pmd	87%	92%	190	322
sunflow	73%	85%	210	245
xlan	79%	85%	325	454

3 Software Prefetching

3.1 The Challenge

There are several technologies to improve the memory performance, and one of these is using prefetching mechanism to tolerate cache miss latencies. Prefetching is a hardware or software technique to separate the tasks of data requesting and data using. Hardware prefetching means that the processors prefetch data by some prediction mechanism automatically without using any extra instructions. While the Software

prefetching explicitly inserts prefetch instructions to perform the data fetch. A prefetch instruction is based on some form of non-blocking load operation, and it moves the data closer to the processor in the memory hierarchy.

The challenge of an effective software prefetching is to determine when to issue the prefetch instruction before the data is accessed by the program. Prefetch too early or too late will not improve the performance. If prefetching too early the data may be evicted from the cache before use. On the other hand, if prefetching too late, the processor may stall before data arrives. Moreover, inappropriate software prefetching may lead to serious cache pollution. So all the above issues should be carefully considered before using the software prefetching technique.

3.2 Address Prefetching on Jikes RVM

We use Jikes RVM [2] and MMTk [3] as our experimental platform. Jikes RVM (Research Virtual Machine) is an open source JVM which is almost written in Java, but a small portion is written in C in order to access the underlying file system, and processor resource et al. MMTk is a memory management subsystem which provides different garbage collection schemes for Jikes RVM, and in this work we only concrete on MS and SS collector. The low-level primitives can be accessed through the Assembler subsystem as showed in Figure 1. The prefetch operation can be achieved by an unboxed abstract class called Address [4], which represents an address in memory. The compiler compiles the prefetch() method declared in Address and translates it to a single instruction on the host architecture. So we can use prefetch operations provided by Jikes RVM to optimize the garbage collector as showed in Figure 2.

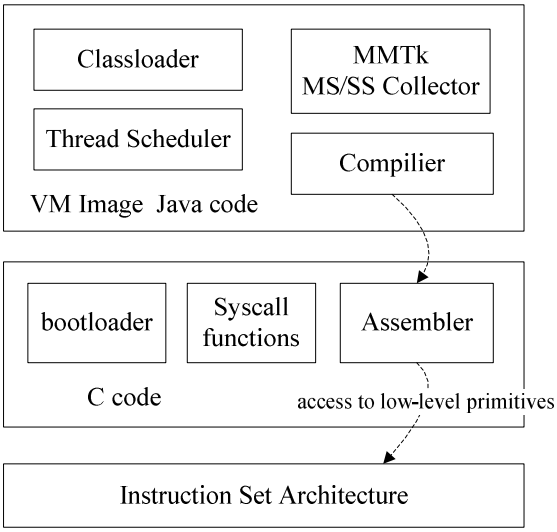


Fig. 1. Low-level primitives access on Jikes RVM platform

3.3 "PREFETCH" Instructions on IA32

The Intel 64 and IA-32 processors perform varying amounts of prefetching, and they also provide several instructions for managing cache [8]. Software prefetching operations can be performed by using the PREFETCHh (h indicates the cache level, more details see [8]) instructions which are streaming SIMD Extensions instructions on IA-32 architectures. The PREFETCHh instructions offer more granular control over caching, and reduce the long latency typically associated with reading data from memory. The instructions merely provide a hint to the hardware, and suggest the processor to prefetch the data from a specified address in memory into the cache hierarchy. It is noteworthy that the instructions will not perform the data prefetch in the case that the prefetch operation causes a TLB miss. Moreover, overuse of PREFETCHh instructions may result in a performance penalty due to the memory bandwidth consumption and cache pollution.

The architecture provides two kinds of prefetch instructions, temporal prefetch and non-temporal prefetch. Temporal means that the data exhibit good locality, while non-temporal means lacking temporal locality. So non-temporal prefetch notify the processor that the data will not be reused soon, and the operations don't follow the normal cache-coherency rules. Its original intention is to minimize the cache pollution.

3.4 Prefetching for Tracing Collector

The tracing is a process that started from a root set to traverse the object reference graph. Reference represents an address which points from one object to another. Software prefetching technique is adopted by a FIFO-based buffer which is called prefetch queue. All of the reference addresses are required to pass through the prefetch queue in their traverse order. The prefetch operation is executed when the reference enqueued. While when dequeued, the reference is handed over to the collector for further processing. That is the basic manner to apply software prefetching technique into tracing garbage collection, and the length of the queue represents the prefetch distance. In the above process, each of the reference address will be prefetched before the collector processing it. Besides the prefetch distance, there are two factors may influence the performance improvement. One is that the prefetch operation will consume memory bandwidth; the other is that the prefetch operation will cause cache pollution. Unfortunately, these two factors are almost impossible to be intuitively measured on a practical computer, and this problem may be a challenge topic.

```
//Enqueue the given ObjectReference ref
Address pfAddr = ref.toAddress();
pfAddr.prefetch();
pfQueue.push(ref);

//process pfQueue
while(!pfQueue.empty()){
    ObjectReference ref = pfQueue.pop();
    //Process ref by the collector
    ...  }
```

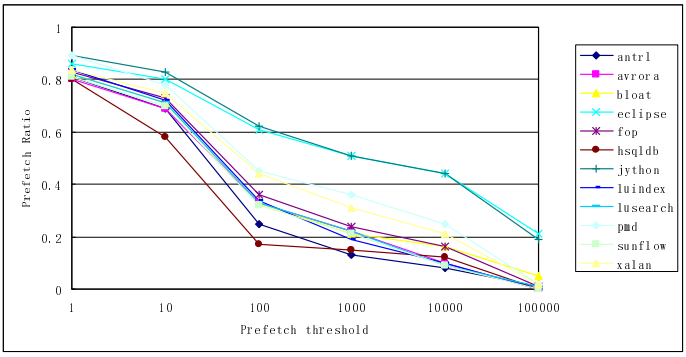
(a) Basic prefetching

```
//Enqueue the given ObjectReference ref
Address pfAddr = ref.toAddress();
//prefetch contraints with threshold
if(pfAddr.diff(lastPA).GT(Threshold)){
        pfAddr.prefetch();
 //update last prefetched address
        lastPA = pfAddr;
}
pfQueue.push(ref);
```

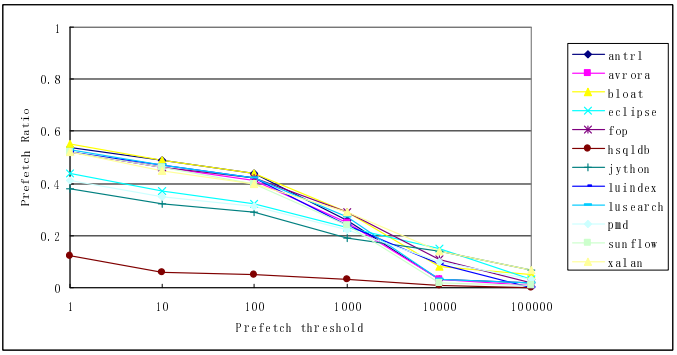
(b) Constrainted prefetching

Fig. 2. Prefetch queue operations

Although the measurement of the above factors temporarily cannot be put into practice, this does not prevent us from further optimizing the software prefetching. We added some constraints when performing the prefetch operation to control the prefetch ratio (the ratio of prefetched addresses to total addresses), and a threshold T was proposed to perform the tuning work. For a sequence of addresses that to be prefetched, ai, a_{i+1}, a_{i+2}, ..., a_{i+k}, if the sequence has better locality features, it is obvious that only one temporal prefetch with ai is enough. As showed in Figure 2(b), for a new enqueued address a_{i+k+1}, the prefetch with a_{i+k+1} is performed only if $|a_{i+k+1} - a_i| > T$, otherwise, no prefetch action. The goal of that is to reduce memory bandwidth consumption and cache pollution. Threshold T was selected as integral times of cache line size, and the prefetch ratios of DaCapo benchmarks with different Ts were present in Figure 3.



(a) Prefetch ratios for MS collector



(b) Prefetch ratios for SS collector

Fig. 3. Prefetch Ratios of DaCapo Benchmarks with different thresholds

4 Evaluation

All experiments are conducted on a 2.40GHz Pentium 4 platform with 8KB L1 data cache and 512KB L2 data cache, 64 byte line size, and 512MB memory. The operating system is Ubuntu 11.04, kernel 2.6.38. We use Jikes RVM as our experimental platform, and use DaCapo as the evaluation benchmark.

In order to evaluate the memory performance improvement obtained by software prefetching, we first extracted the address sequences of DaCapo benchmarks which are accessed by the collector during the tracing process. Then we simulated the tracing process with these address sequences, we change the prefetch threshold and by this way we can obtain different prefetch ratios. For MS collector, we evaluated 4 prefetch ratio intervals: 0.6-0.8, 0.4-0.6, 0.2-0.4, and 0.0-0.2, while 3 intervals for SS collector: 0.4-0.8, 0.2-0.4, 0.0-0.2. The tracing time with prefetch ratio 1.0 is considered as the baseline time. We picked 10 different thresholds in each prefetch ratio interval, and then compared the average tracing time to the baseline time. The evaluation results are showed in Figure 4 and Figure 5, we can see that most of the benchmark programs obtain a better tracing performance than the baseline. The best results for MS collector appeared in prefetch ratio interval (0.4-0.6) with 5% performance improvement in average, while for SS collector that is 5% in prefetch ratio interval (0.2-0.4).

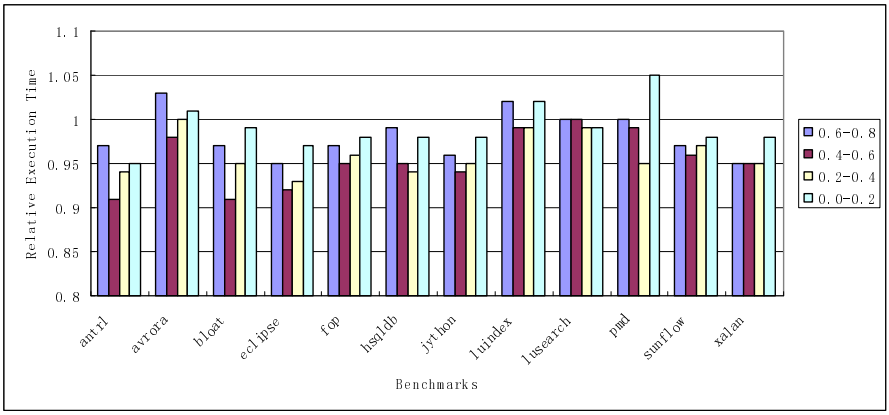


Fig. 4. Relative Tracing time of MS collector with different prefetch ratios

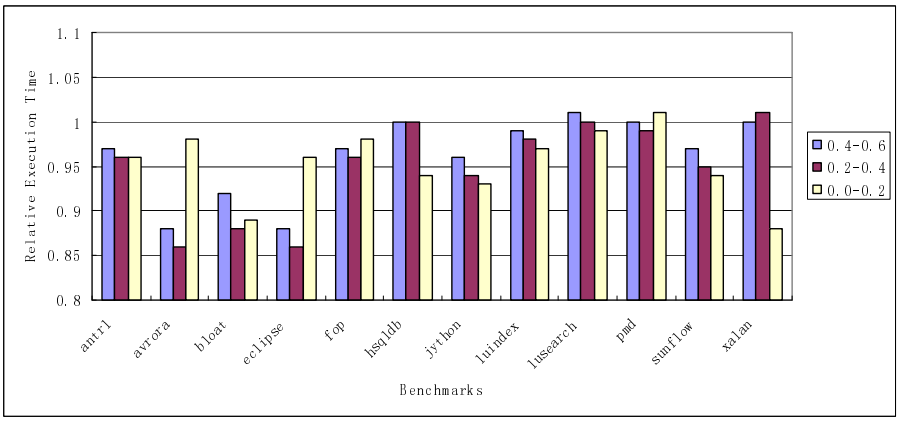


Fig. 5. Relative Tracing time of SS collector with different prefetch ratios

Based on the previous experimental results, we apply this software prefetching technique to Jikes RVM from which we can see how much performance can be improved in a practical Java virtual machine. The prefetch ratios we chosen are (0.4-0.6) for MS collector, and (0.2-0.4) for SS collector. We run the benchmarks with the prefetching optimized RVM, each garbage collection time are recorded and then compared with the un-optimized version of RVM. Figure 6 presents the improvements achieved by applying software prefetching. A positive percentage means that the collection performance is improved, the higher the better.

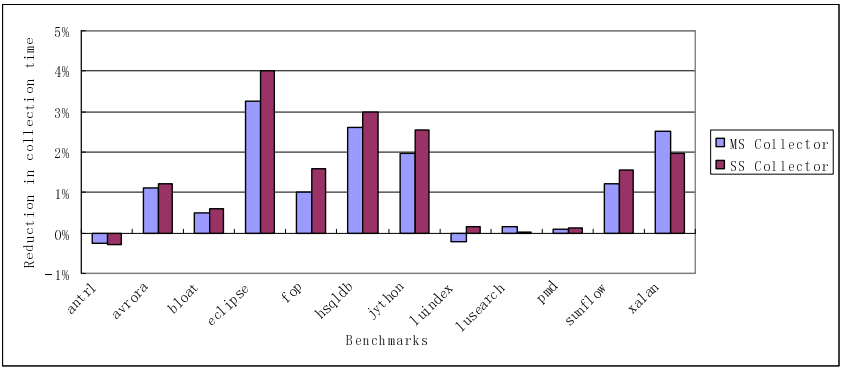


Fig. 6. Reduction in GC time obtained by software prefetching

5 Related Work

Both of the hardware and software data prefetching are studied in several research work [6, 7, 9], and benefits and limitations of software prefetching is studied in [10, 11]. Cher [13] and Garner [5] proposed buffered prefetch strategy to improve tracing performance of the mark sweep garbage collector, the strategy adopted a FIFO prefetch buffer, and the depth of the FIFO indicates the prefetch distance. Our work adopted the similar strategy with Cher and Garner's approaches. The difference is that our approach doesn't prefetch all the addresses in the prefetch queue, and prefetch ratio is controlled by prefetch threshold. Using prefetching technology to reduce stall times and improve the efficiency of reference counting collector was first studied by Paz et al. [12]. They investigated the memory access patterns of the reference counting collector on Jikes RVM, and report that the patterns are different from the tracing collectors.

There are several previous research focus on reducing or eliminating the pause time of the garbage collector, included the following three aspects: 1) Improving the collectors to meet real-time bounds under multiprocessor platform [14, 15, 20, 22]; 2) Investigating concurrent garbage collectors which no longer need to pause the program when performing the collection [16, 17, 18, 19], but the overhead on the program execution is not negligible; 3) Using hardware technologies to obtain better responsiveness [21].

6 Conclusions

In this work, we addressed the performance of software prefetching which used for tracing garbage collector. Investigating and evaluating the software prefetching on x86 platform, we found out that, the prefetch per tracing loop is not a good choice, and the performance improvement is very much limited. Based on the previous studies, we use threshold to tuning the prefetch ratios. The experimental results show that the proposed method can obtain a further memory performance improvement.

Acknowledgments. The authors would like to thank Jikes RVM Team for providing Jikes RVM as an open source research infrastructure. Without their initial work, our implementation would be much harder. This research was supported by the grant from the National Natural Science Foundation of China (No.61173024).

References

1. Blackburn, S., Garner, R., et al.: The DaCapo benchmarks: Java benchmarking development and analysis. In: Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications, pp. 169–190 (2006)
2. Alpern, B., Attanasio, C., Barton, J., et al.: The Jalapeno virtual machine. IBM Systems Journal 39(1), 211–221 (2000)
3. Blackburn, S.M., Cheng, P., McKinley, K.S.: Oil and water? High performance garbage collection in Java with MMTk. In: Proceedings of the 26th International Conference on Software Engineering, pp. 137–146. ACM, New York (2004)
4. Framption, D., Blackburn, S.M., Cheng, P., et al.: Demystifying magic: high-level low-level programming. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 81–90. ACM, New York (2009)
5. Garner, R., Blackburn, S.M., Frampton, D.: Effective prefetch for mark-sweep garbage collection. In: Proceedings of the 6th International Symposium on Memory Management, pp. 43–54. ACM, New York (2007)
6. Guo, Y., Chheda, S., Koren, I., Krishna, C.M., Moritz, C.A.: Energy-Aware Data Prefetching for General-Purpose Programs. In: Falsafi, B., VijayKumar, T.N. (eds.) PACS 2004. LNCS, vol. 3471, pp. 78–94. Springer, Heidelberg (2005)
7. Guo, Y., Naser, M.B., Moritz, C.A.: PARE: a power-aware hardware data prefetching engine. In: Proceedings of the 2005 International Symposium on Low Power Electronics and Design, pp. 339–344. ACM, New York (2005)
8. Intel 64 and IA-32 Architectures Optimization Reference Manual, Order Number 248966-026, `http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf`
9. Lee, J., Lakshminarayana, N.B., Kim, H., Vuduc, R.: Many-Thread Aware Prefetching Mechanisms for GPGPU Applications. In: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 213–224. IEEE, Los Alamitos (2010)
10. Lee, J., Kim, H., Vuduc, R.: When Prefetching Works, When It Doesn't, and Why. ACM Transactions on Architecture and Code Optimization 9(1), 2:1-2:29 (2012)
11. Marathe, J., Mueller, F.: PFetch: software prefetching exploiting temporal predictability of memory access streams. In: Proceedings of the 9th Workshop on Memory Performance: Dealing with Applications, Systems and Architecture, pp. 1–8. ACM, New York (2008)

12. Paz, H., Petrank, E.: Using Prefetching to Improve Reference-Counting Garbage Collectors. In: Adsul, B., Odersky, M. (eds.) CC 2007. LNCS, vol. 4420, pp. 48–63. Springer, Heidelberg (2007)
13. Cher, C., Hosking, A.L., Vijaykumar, T.N.: Software prefetching for mark-sweep garbage collection: hardware analysis and software redesign. In: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 199–210. ACM, New York (2004)
14. Auerbach, J., Bacon, D.F., Cheng, P., et al.: Tax-and-spend: democratic scheduling for real-time garbage collection. In: Proceedings of the 8th ACM International Conference on Embedded Software, pp. 245–254 (2008)
15. Pizlo, F., Frampton, D., Petrank, E., Steensgaard, B.: Stopless: A real-time garbage collector for modern platforms. In: Proceedings of the 6th International Symposium on Memory Management (2007)
16. McGachey, P., Adl-Tabatabai, A.-R., et al.: Concurrent GC leveraging transactional memory. In: 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 217–226 (2008)
17. Pizlo, F., Petrank, E., Steensgaard, B.: A study of concurrent real-time garbage collectors. In: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 33–44 (2008)
18. Kliot, G., Petrank, E., Steensgaard, B.: A lock-free, concurrent, and incremental stack scanning for garbage collectors. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 11–20 (2009)
19. Siebert, F.: Concurrent, Parallel, Real-Time Garbage-Collection. In: Proceedings of the 2010 International Symposium on Memory Management, pp. 11–20 (2010)
20. Click, C., Tene, G., Wolf, M.: The pauseless GC algorithm. In: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments, pp. 46–56 (2005)
21. Schoeberl, M., Puffitsch, W.: Non-blocking real-time garbage collection. ACM Transactions on Embedded Computing Systems (2010)
22. Auerbach, J., Bacon, D.F., et al.: Design and implementation of a comprehensive real-time Java virtual machine. In: Proceedings of the 7th ACM & IEEE International Conference on Embedded Software, pp. 249–258 (2007)

Adaptive Implementation Selection in the SkePU Skeleton Programming Library

Usman Dastgeer, Lu Li, and Christoph Kessler

IDA, Linköping University, 58183 Linköping, Sweden
{usman.dastgeer,lu.li,christoph.kessler}@liu.se

Abstract. In earlier work, we have developed the SkePU skeleton programming library for modern multicore systems equipped with one or more programmable GPUs. The library internally provides four types of implementations (*implementation variants*) for each skeleton: serial C++, OpenMP, CUDA and OpenCL targeting either CPU or GPU execution respectively. Deciding which implementation would run faster for a given skeleton call depends upon the computation, problem size(s), system architecture and data locality.

In this paper, we present our work on automatic selection between these implementation variants by an offline machine learning method which generates a compact decision tree with low training overhead. The proposed selection mechanism is flexible yet high-level allowing a skeleton programmer to control different training choices at a higher abstraction level. We have evaluated our optimization strategy with 9 applications/kernels ported to our skeleton library and achieve on average more than 94% (90%) accuracy with just 0.53% (0.58%) training space exploration on two systems. Moreover, we discuss one application scenario where local optimization considering a single skeleton call can prove sub-optimal, and propose a heuristic for bulk implementation selection considering more than one skeleton call to address such application scenarios.

Keywords: Skeleton programming, GPU programming, implementation selection, adaptive offline learning, automated performance tuning.

1 Introduction

The need for power efficient computing has lead to heterogeneity and parallelism in today's computing systems. Heterogeneous systems such as GPU-based systems with disjoint memory address space already became part of mainstream computing. There exist various programming models (CUDA, OpenCL, OpenMP etc.) to program different devices present in these systems and, with GPUs becoming more general purpose every day, more and more computations can be performed on either of the CPU or GPU devices present in these systems.

Known for their performance potential, these systems expose programming difficulty as the programmer often needs to program in different programming models to do the same computation on different devices present in the system which limits code-portability. Furthermore, sustaining performance when porting an application between

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 170–183, 2013.
© Springer-Verlag Berlin Heidelberg 2013

different GPU devices (*performance portability*) is a non-trivial task. The skeleton programming approach can provide a viable solution for computations that can be expressed in the form of skeletons, where *skeletons* [1, 2] are pre-defined generic components derived from higher-order functions that can be parameterized with sequential problem-specific code. A skeleton program looks like a sequential program where a skeleton computation can internally exploit parallelism and leverage other architectural features transparently by e.g. keeping different implementations for a single skeleton targeting different architectural features of the system. Map/Zip and Farm are examples of data and task-parallel skeletons respectively.

We have developed the SkePU skeleton programming library for GPU-based systems in our earlier work [3]. The library targets single-node GPU-based systems and provide code portability for skeleton programs by providing sequential C++, OpenMP, CUDA and OpenCL implementations for each of its skeleton. In this paper, we present an adaptive offline machine learning method to tune implementation selection in the SkePU library automatically. The proposed technique is implemented inside the SkePU library allowing automatic implementation selection on a given GPU-based system, for any skeleton program written using the library. To the best of our knowledge, this makes SkePU the first skeleton library for GPU-based systems that provides general-purpose, automatic implementation selection mechanism for calls to its skeleton.

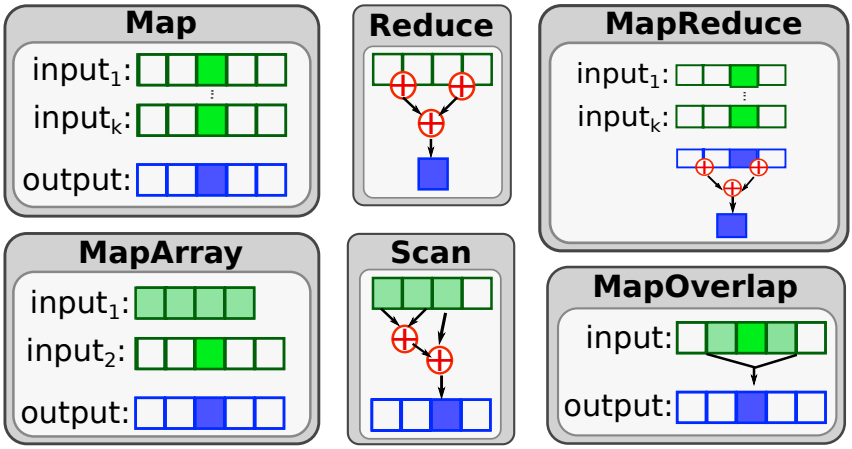


Fig. 1. Six data-parallel skeletons, here shown for vector operands: Map applies a user-defined function element-wise to input vectors. Reduce accumulates an associative binary user function over all input vector elements to produce a scalar output. MapReduce combines Map and Reduce in one step. MapArray is similar to map but all elements from the 1st operand are accessible. MapOverlap is similar to Map where elements within a (user-defined) neighbourhood are also accessible in the user function. Scan is a generic prefix-sums operation.

The paper is structured as follows: In Session 2 we briefly describe our SkePU skeleton library. The proposed adaptive tuning framework is explained in Section 3 followed by evaluation in Section 4. Related work is discussed in Section 5 while Section 6 concludes the work.

2 The SkePU Skeleton Library

The SkePU skeleton library [3] is designed in C++ and offers several data parallel skeletons including map, reduce, mapreduce, maparray, mapoverlap and scan. The operand data to skeleton calls is passed using 1D Vector and 2D Dense matrix containers. These containers internally keep track of data residing on different memory units (main memory, GPU device memory etc.) and can transparently optimize data transfers by copying data only when it is necessary. The memory management for skeleton calls' operand data is implicitly handled by the library. This can for example allow multiple skeleton operations (reads, writes) on the same data on a GPU and copies data back to main memory only when the program accesses the actual data (detected using the `[]` operator for vector elements).

Figure 1 shows a graphical description of different skeletons when used with the 1D vector container. The MapReduce skeleton is just a combination of Map and Reduce skeletons applied in a single step which is different from Google MapReduce. For a 2D matrix container operand, semantics are extended to, e.g., apply MapOverlap across all row vectors and/or across all column vectors.

```
1  // #include directives
2
3  // generates a user function 'mult_f' to be used in skeleton instantiation
4  BINARY_FUNC(mult_f, double, a, b,
5      return a*b;
6  )
7
8  int main()
9  {
10   skepu::Map<mult_f> vecMultiply(new mult_f);/* creates a map skel. object */
11
12   skepu::Vector<double> v0(50); /* 1st input vector, 50 elements */
13
14   skepu::Vector<double> v1(50); /* 2nd input vector, 50 elements */
15
16   skepu::Vector<double> res(50); /* output vector, 50 elements */
17
18   ...
19
20   vecMultiply(v0, v1, res); /* skeleton call on vectors */
21
22   std::cout<<"Result: " << res <<"\n"; /* output result vector */
23
24   ...
25 }
```

Listing 1. Multiplying two vectors element-wise using the Map skeleton

Listing 1 shows a simple operation of multiplying two vectors element-wise and writing the result into an output vector. As each skeleton in the library has multiple implementations (C++, OpenMP, CUDA, OpenCL) available, the skeleton call on Line

12 will internally be mapped to one of those implementations. Up to now, this decision was controlled by the user or was statically determined (e.g., to always use the CUDA implementation when a CUDA GPU is available). In the next section, we will explain the mechanism for tuning this implementation selection in a more intelligent manner automatically.

3 Adaptive Tuning Framework

Any skeleton call in our library enables implementation selection choice which can have performance implications. In the ideal case, we would like to select an implementation which would result in the shortest execution time. For this purpose, we devise an empirical prediction technique based on offline sample executions. In the following, we describe the technique and how it is implemented.

3.1 Idea

A simple way to do empirical offline tuning could be to exhaustively try out all *variants* for different *call context instances* to find out the *best variant* and use that for actual skeleton calls. In our case, variants are mainly the different skeleton implementations (CPU, OpenMP etc.), *call context instances* are characterized by the sizes of operand data and the *best variant* is one which results in shortest execution time. Trying out all possible context instances using exhaustive search is practically infeasible. Our offline tuning technique is rather an adaptive hierarchical search based upon a heuristic *convexity assumption* which basically means that if a certain implementation is performing best on all vertices of a D-dimensional context parameter subspace then we assume it is also the best choice for all points within the subspace. For example, considering a 1-dimensional space (i.e. only 1 input size parameter), if we find out that a certain implementation performs best on two distinct input sizes i and j we consider it best for all points between these points (i.e. for the whole range $[i, j]$). This concept is extended to D-dimensional space as described below.

3.2 Algorithm

The space $C = I_1 \times ... I_D$ of context instances for a skeleton with D possibly performance-relevant properties in the context instances is spanned by the D context property axes with considered (user-specified or default) finite intervals I_i of discrete values, for $i = 1, ..., D$. A continuous subinterval of an I_i is called a (context property value) *range*, and any cross product of such subintervals on the D axes is called a *subspace* of C. Hence, subspaces are "rectangular", i.e., subspace borders are orthogonal to the axes of C.

Our idea is to find sufficiently precise approximations by adaptively recursive splitting of subspaces by splitting the intervals I_i, $i = 1, ..., D$. Hence, subspaces are organized in a hierarchical way (following the subspace inclusion relation) and represented by a 2^D-ary tree T_C (cf. quadtrees/octrees etc.).

Our algorithm for off-line measurement starts from a trivial tree T_C that has just one node, the root (corresponding to the whole C), which is linked to its 2^D corner points

(here, the 2^D outer corners of C) that are stored in a separate table of recorded performance measurements. The implementation variants of the skeleton under examination are run with each of the corresponding 2^D context instances, possibly multiple times for averaging; a variant whose execution exceeds a timeout for a context instance is aborted and not considered further for that context instance. Now we know the winning implementation variant for each corner point and store it in the performance table, too, and T_C is properly initialized.

Consider any leaf node v in the current tree T_C representing a subspace $S_v = R_1^v \times \ldots \times R_D^v$ where $R_i^v \subset I_i$, $i = 1, \ldots, D$. If the same specific implementation variant runs fastest on all context instances corresponding to the 2^D corners of S_v, we stop further exploration of that subspace and will always select that implementation whenever a context instance at run-time falls within that subspace. Otherwise, the subspace S_v may be refined further. Accordingly, the tree is extended by creating new children below v which correspond to the newly created subspaces of S_v. By iteratively refining the subspaces in breadth-first order, we generate an adaptive tree structure to represent the performance data and selection choices, which we call *dispatch tree*.

The user can specify a *maximum depth* (training depth) for this iterative refinement of the dispatch tree, which implies an upper limit on the runtime lookup time, and also a maximum tree size (number of nodes) beyond which any further refinement is cut off. Third, the user may specify a timeout for overall training time, after which the dispatch tree is considered final.

At every skeleton invocation, a run-time lookup searches through the dispatch tree starting from the root and descending into subspace nodes according to the current run-time context instance. If the search ends at a *closed leaf*, i.e., a leaf node with equal winners on all corners of its subspace, the winning implementation variant can be looked up in the node. If the search ends in an *open leaf* with different winners on its borders (e.g., due to reaching the specified cut-off depth), we perform an approximation within that range by choosing the implementation that runs fastest on the subspace corner with the shortest Euclidean distance from the current run-time context instance.

The deeper the algorithm explores the tree, the better precision the dynamic composer can offer for the composition choice; however, it requires more off-line training time and more runtime lookup overhead as well. We give the option to let the user decide the trade-off between training time and precision by setting the cut-off depth, size and time in the component interface descriptor. Figure 2 shows an example for 1-dimensional space exploration. The algorithm can recursively split and refine subspaces until it finds common winners for all points for a subspace (i.e. the subspace becomes closed) or the user-specified maximum depth is reached.

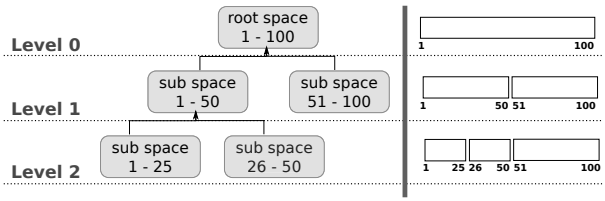


Fig. 2. Depiction of how a 1-dimensional space is recursively cut into subspaces (right) and the resulting dispatch tree (left)

3.3 Implementation Details

In order to transparently integrate the tuning mechanism in our existing skeleton library, we have designed it using C++ templates as an include header. As shown in Listing 2, a `Tuner` class is introduced which is parameterized by the skeleton type and user function(s). The user needs to supply a unique `ID` (string) for the *skeletonlet*[1] being tuned as well as lower and upper bounds for the size of each operand. The `ID` decouples the skeletonlet and tuner, and allows e.g. multiple tuning scenarios even for the same skeletonlet to co-exist. Internally the tuner applies certain optimizations (e.g. dimensionality reduction) and returns an execution plan which is later assigned to the skeleton object. An execution plan is a simple data structure that internally tracks the best implementation for each subspace and provides lookup facilities. After the execution plan is set, the expected best implementation for any skeleton in a given call context will be automatically selected.

The `Tuner` supports automatic persistence and loading of execution plans. If the execution plan with same configuration already exists, it loads and returns it from a repository without any tuning overhead; otherwise it invokes the tuning algorithm and constructs an execution plan. The generated tuning plan is stored for future usages to avoid re-tuning every time the skeleton program is executed. Furthermore, the tuning and actual execution can happen during the same program execution, as shown in Listing 2. When porting the same skeleton program to a new architecture, the tuner would automatically construct an exection plan for the new architecture without requiring any changes in the user program.

```
...

int main()
{
  skepu::Map<mult_f> vecMultiply(new mult_f);

/* specify where input and output operand data (need to) resides */

  int opInLoc[]  = {0, -1}; /* 1st/2nd input operand in GPU/main memory */
  int opOutFlag[] = {1};    /* copy result back to main memory or not */

/* specify lower and upper bounds for training range */

  int lowerBounds[] = {10, 10, 10};
  int upperBounds[] = {50000000, 50000000, 50000000};

/* invoke the tuner whichs returns the execution plan */

  skepu::ExecPlan plan = skepu::Tuner<mult_f, MAP>("vMult", 3, lowerBounds,
                                          upperBounds, opInLoc, opOutFlag)();

/* assign the execution plan to the skeleton object */

  vecMultiply.setExecPlan(execPlan);

  ...
}
```

Listing 2. Tuning the vector multiply skeleton call

[1] A pair of user-function(s), skeleton type.

Dimensionality Reduction. We apply several optimizations based on domain specific knowledge that each skeleton implementation exposes. For example, considering the fact that all operands (inputs, output) in a map skeleton should be of exactly the same size, we have considered it as 1-dimensional space instead of 3-dimensional space (with 2 input and 1 output operands). This significantly reduces the training cost and is transparently done by considering semantics of the skeleton being used. Similar optimizations are applied for MapOverlap, MapReduce and Scan skeletons.

Data Locality. Current GPU based systems internally have disjoint physical memory and both the Vector and Matrix containers in our skeleton library can track their payload data on different memory units. The operand data locality matters when measuring the execution time for both CPU and GPU execution for a skeleton implementation, as operand data may or may not exist in the right memory unit; in case it is not available in the right place, extra overhead for data copy needs to be encountered. Selection of the expected best implementation for a given problem size cannot be made without considering where the input data resides and where the output data needs to be copied back as the data copying overhead between different memory units could affect the selection of the best performing variant. One solution could be to assume that operand data is always located in a specific memory unit (e.g., main memory) and, depending upon where the skeleton implementation executes, a copy may or may not be required. This solution is simple but unflexible as even different operands of a single skeleton call may reside at different memory spaces depending upon their previous usage with other skeleton calls. On the other hand, delaying the decision about operand data locality to runtime is infeasible as we need to know the data transfer cost to determine, offline, the best variant for a given problem size.

We have devised a simple mechanism for the programmer to specify knowledge about operands' data locality. By default, we assume that operand data resides in main memory and cost for transferring output data back to main memory is not included in the skeleton execution. However, the programmer can easily override this behavior by specifying:

- An integer flag for each input operand specifying the memory unit where it is residing (default main memory = -1). In the example in Listing 2 (line 6), the tuner will determine the best implementation considering that the first operand resides in the GPU device memory while the second input operand resides in main memory.
- A binary flag for each output operand specifying whether it should be transferred back to main memory or not. In the example in Listing 2 (line 7), the best implementation is determined considering that the output operand needs to be copied back to main memory after skeleton execution.

4 Evaluation

For evaluation, we have implemented five applications (NBody simulation, **S**mooth **P**article **H**ydrodynamics, LU factorization, Mandelbrot, Taylor series) and four kernels (**M**ean **S**quared **E**rror, **P**eak **S**ignal-to-**N**oise **R**atio, **P**earson **P**roduct-**M**oment

Correlation **C**oefficient, dot product) with skeletons available in our skeleton library. We use two different systems to demonstrate effectivness of our tuning mechanism in doing implementation selection while adjusting to platform differences: **System A** with Xeon® E5520 CPUs running at 2.27GHz with 1 NVIDIA® C2050 GPU with L1/L2 cache support and **System B** with Xeon® X5550 CPUs running at 2.67GHz with a lower-end GPU (NVIDIA® C1060 GPU).

For each application/kernel, we call the tuner on a given training range (i.e., problem size ranges for each operand) and it internally explores some points in the training space and construct an execution plan using the algorithm described in Section 3.2. Afterwards, we do the actual execution by selecting a set of sample points (different from the training points) within that range and do the actual execution using the tuned version as well as using each implementation variant (CPU, OpenMP, CUDA) on those selected points[2]. The same problem size ranges are used for experiments on both systems and no modifications in the program source code are made when porting the applications between both systems. Furthermore, for all experiments, we set the maximum training depth to 10 and Euclidian distance is used to estimate the best variant if no best variant is found for a subspace until depth 10.

4.1 Tuning Efficiency

Figure 3 shows execution of eight applications/kernels on System A. On the horizontal axis, we list the problem sizes whereas the vertical axis represents the execution time. For each application/kernel, we list the percentage of training space that is explored by the tuner to construct the execution plan as well as average accuracy of execution with the tuned version[3]. As it is practically infeasible to try out all points in the training range, accuracy is measured by averaging over the ratio of execution time with the tuned configuration with execution time of the best from direct execution (CPU, OpenMP, CUDA) over all sample points. Due to small variations in execution times during actual measurements, accuracy could become more than 100% in some cases (e.g. an OpenMP implementation can take slightly different time even between successive executions [18]). Averaging over all eight applications/kernels, 94% accuracy has been achieved with just 0.2% training space exploration.

When porting the applications to System B, no changes in the applications' source code are required and the execution plan is automatically tuned before first execution on the new platform. As shown in Figure 4, the tuner is able to effectively adjust to platform differences without requiring any user intervention and we achieved on average 91% accuracy with just 0.3% training space exploration. For execution with the tuned version, the overhead of looking up the best implementation in the execution plan for a given call context is included in the measurement, which proved to be negligible.

[2] We did not consider the OpenCL implementations for experiments as they are similar to CUDA in performance on NVIDIA GPUs and are primarily written for execution on accelerator devices not supporting CUDA.

[3] In the tuned version, implementation selection is made based upon the execution plan returned by the tuner.

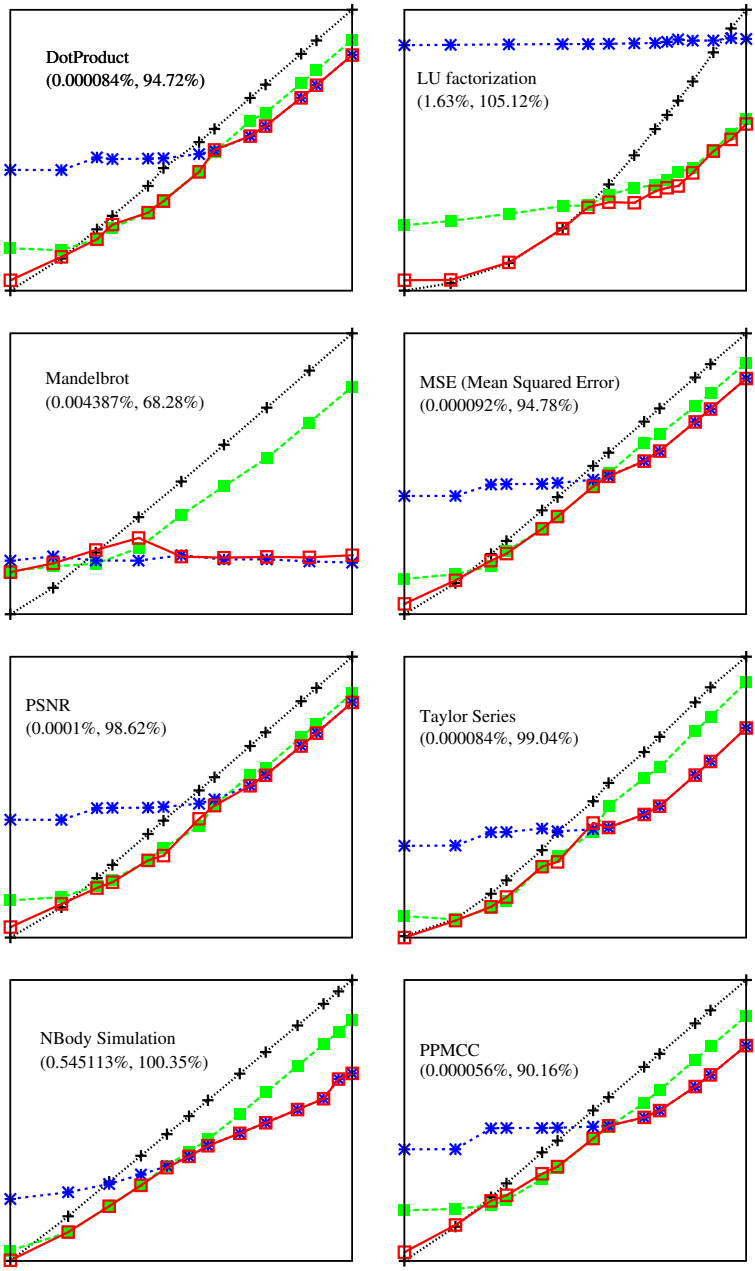


Fig. 3. Execution time of eight applications/kernels for different problem sizes on System A with respective *training space, accuracy* figures. On average, 94% accuracy has been achieved with just 0.2% training space exploration. [**Legend:** Black(CPU, +), Green(OpenMP ■), Blue(CUDA ∗), Red(Tuned □)] .

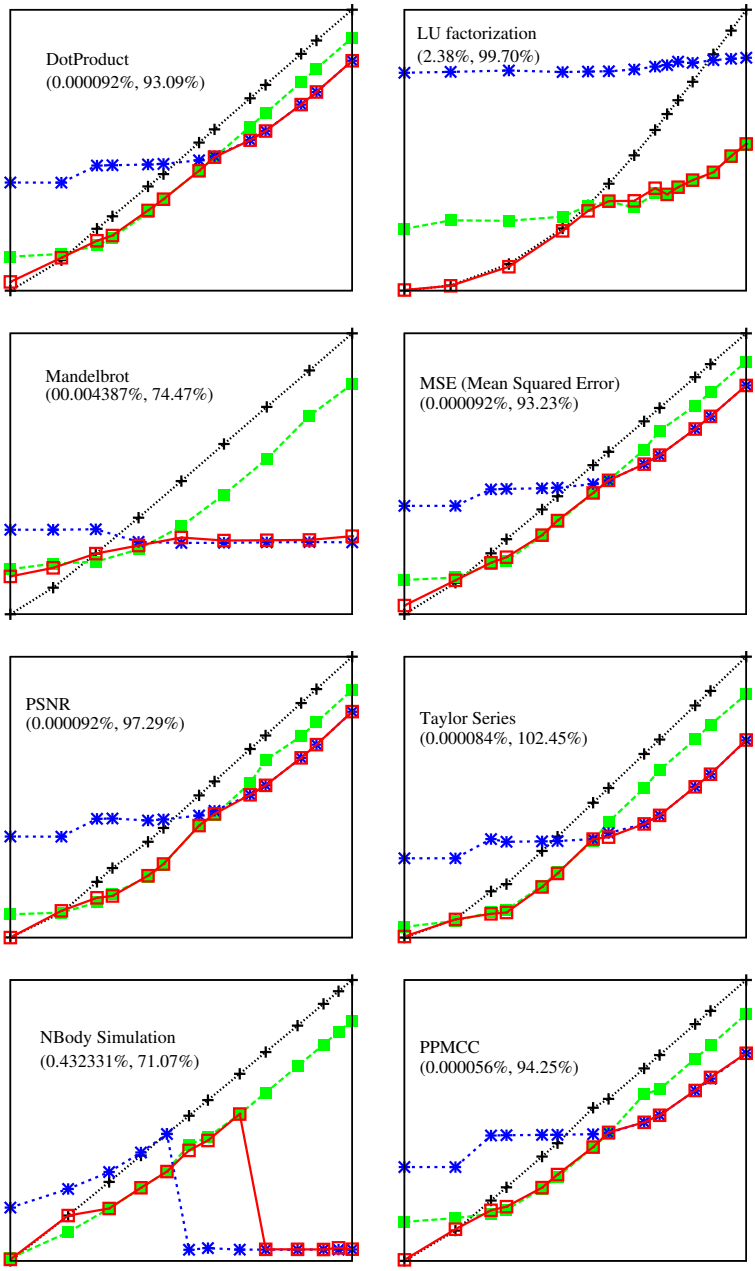


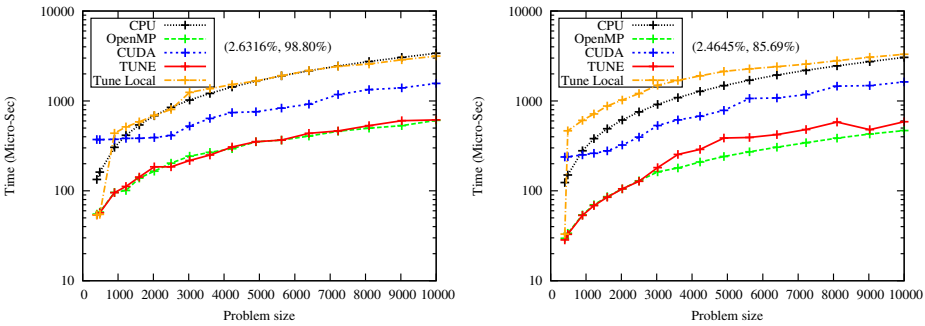
Fig. 4. Execution time of eight applications/kernels for different problem sizes on System B with respective *training space, accuracy* figures. On average, 91% accuracy has been achieved with just 0.3% training space exploration. [**Legend:** Black(CPU, +), Green(OpenMP ■), Blue(CUDA ∗), Red(Tuned □)].

4.2 Bulk Execution

The Tuner predicts the best implementation for each skeleton call individually based on operand data locality and execution time of each skeleton implementation available. As we have seen in the previous section, this works fine for both simple and complex applications/kernels with skeleton calls of one or more types. However, in some applications with multiple skeleton calls having different computational complexity and constrained in a data dependency chain, locally optimal decisions for each skeleton call may result in a globally sub-optimal decision. Listing 3 shows such an application scenario in the SPH (Smooth Particle Hydrodynamics) application. This application has three different types of skeleton calls with different computational complexity, operating on the same data inside a loop. For a given problem size, the tuner might determine OpenMP, CUDA and OpenMP execution as best for `skeleton_1`, `skeleton_2` and `skeleton_3` calls respectively. Although making the best decision for each skeleton call individually, this would result in lot of expensive data transfers (over PCIe bus between main and GPU device memory) as output produced by the `skeleton_1` call becomes an input to the `skeleton_2` call and so forth. Doing it inside a loop makes it even worse as these data transfers would need to be done in each loop iteration.

```
...
for(....)
{
    skeleton_1(v0, v1, v1);
    skeleton_2(v1, v0, v0);
    skeleton_3(v0, v0);
}
...
```

Listing 3. SPH pseudo-code



(a) System A (b) System B

Fig. 5. Execution of SPH on both systems with respective *training space, accuracy* figures

For such skeleton calls with different computational complexity and tight data dependency, we implement a simple bulk selection heuristic in our tuner. For a sequence of skeleton calls constrained in a data dependency chain, the skeleton programmer can

specify a `group id` as a last (optional) argument to each skeleton call. All skeleton calls with same group id are scheduled on compute units with the same memory address space which is determined as the expected best one for the first skeleton call in the group.

Figure 5 shows execution of SPH on both evaluation systems. As shown in the figure, predicting the best implementation for each skeleton call individually (the `Tune Local` version) yields poor performance in this case. The tuner version with the bulk heuristic performs better in this case by considering the interconnection between the skeleton calls and mapping them to the same backend.

5 Related Work

Besides SkePU, SkelCL [4] and Muesli [8] are currently the two main skeleton programming libraries for GPU-based systems. They all provide some common data parallel skeletons such as Map/Zip and also provide memory management for GPU-execution. Hybrid execution and automatic implementation selection are some important capabilities of SkePU that distinguish it from the other two libraries (see [3] for details).

MultiSkel [9] provides a CUDA code generation facility for skeleton programs written in C++. Bones [11] targets automatic transformation of C programs to CUDA for GPU execution by identifying occurences of pre-defined patterns/skeletons in the sequential code. Buono et al. [10] describe a set of low-level algorithmic constructs that can be composed in a hierarchical manner to match application-level patterns.

In our earlier work [6], we used a genetic algorithm to do offline implementation selection as well as selection of some tuning parameters for each implementation type (number of threads for OpenMP and thread block size for CUDA and OpenCL backend). However, the current approach using the convexity assumption requires much less traning space exploration while achieving better accuracy for a variety of applications.

Empirical exploration is employed by Collins et al. [12] in their FastFlow parallel skeleton framework. They use Monte Carlo search of a random subset of the space and use knowledge about variable dependencies to further reduce the search space. However, their tuning is about finding the suitable values for the tuning parameters rather than implementation selection; also the FastFlow library currently targets multicore homogeneous systems and does not support GPU-based systems.

There exist a large body of work in empirical tuning (e.g. [17, 13]) as well as usage of decision trees [15, 14] and C4.5 algorithm [16]. Our work differs from other empirical auto-tuning approaches in two ways: First, our focus is on implementation selection rather than tuning (machine- or algorithm- specific) parameters for an implementation. This enables us to use the convexity assumption to significantly reduce the training cost compared to random sampling employed by other parametric tuning frameworks. A similar approach using the convexity assumption is used in our earlier work [7] for PEPPHER components composition [5]. Secondly, we use an adaptive method to explore the sampling space selectively in an attempt to minimize the sampling and training cost while building the dispatch tree simultaneously. This is in contrast to classical approaches that do the sampling and learning separately; thereby considering many uninteresting but expensive sample points.

6 Conclusion

Having different implementations for a computation, possibly in different programming models, can give both performance and portability if some intelligent selection mechanism is in place. We proposed and implemented an efficient empirical auto-tuning method for doing implementation selection in a skeleton library for GPU-based systems. It uses an adaptive algorithm based on a heuristic convexity assumption to build up a decision tree by exploring parameter subspaces in a recursive manner. Evaluation with nine applications/kernels have demonstrated effectiveness of our approach in predicting the best implementation, with great accuracy (more than 90%), for a given execution context with just 0.5% training space exploration on two different systems. The selection and tuning mechanism is implemented inside the SkePU skeleton library, requiring no modifications in the user-code when porting the application to a new system.

References

[1] Cole, M.: Algorithmic Skeletons: Structured management of parallel computation. MIT Press, Cambdridge (1989)

[2] Kessler, C., Gorlatch, S., Enmyren, J., Dastgeer, U., Steuwer, M., Kegel, P.: Skeleton Programming for Portable Many-Core Computing. In: Pllana, S., Xhafa, F. (eds.) Programming Multi-Core and Many-Core Computing Systems, 20 pages. Wiley Interscience, New York (2013)

[3] Dastgeer, U.: Skeleton Programming for Heterogeneous GPU-based Systems. Licentiate thesis. Thesis No 1504. Dept. of Comp. and Inf. Sci., Linköping University (October 2011)

[4] Steuwer, M., Kegel, P., Gorlatch, S.: SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In: IEEE Int. Sym. on Par. and Dist. Proc. Workshop and Phd Forum (IPDPSW), Anchorage, USA (2011)

[5] Dastgeer, U., Li, L., Kessler, C.: The PEPPHER Composition Tool: Performance-Aware Dynamic Composition of Applications for GPU-based Systems. In: Proc. 2012 Int. Workshop on Multi-Core Computing Systems (MuCoCoS 2012), in conjunction with Supercomputing Conference (SC 2012), Salt Lake City, Utah, USA (2012)

[6] Dastgeer, U., Enmyren, J., Kessler, C.W.: Auto-tuning SkePU: A multi-backend skeleton programming framework for multi-GPU systems. In: Proc. of the 4th Int. Workshop on Multicore Soft. Eng (IWMSE 2011). ACM, NY (2011)

[7] Li, L., Dastgeer, U., Kessler, C.: Adaptive off-line tuning for optimized composition of components for heterogeneous many-core systems. In: Daydé, M., Marques, O., Nakajima, K. (eds.) VECPAR. LNCS, vol. 7851, pp. 329–345. Springer, Heidelberg (2013)

[8] Ernsting, S., Kuchen, H.: Algorithmic skeletons for multi-core, multi-GPU systems and clusters. Int. J. of High Perf. Comp. and Netw. 7(2), 129–138 (2012)

[9] Tung, L.D., Duc, N.H., Anh, P.T., Hoang, N.H., Thap, N.M.: An Intermediate Library for Multi-GPUs Computing Skeletons. In: IEEE RIVF International Conference on Computing and Communication Technologies, Research, Innovation, and Vision for the Future, RIVF (2012)

[10] Buono, D., Danelutto, M., Lametti, S., Torquati, M.: Parallel Patterns for General Purpose Many-Core. In: Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2013. IEEE Computer Society Press (2013)

[11] Nugteren, C., Corporaal, H.: Introducing 'Bones': A parallelizing source-to-source compiler based on algorithmic skeletons. In: Proc. 5th Annual Workshop on General Purpose Proc. with Graph. Proc. Units (GPGPU-5). ACM, NY (2012)
[12] Collins, A., Fensch, C., Leather, H.: Auto-tuning parallel skeletons. Parallel Processing Letters 22(02) (2012)
[13] Thomas, N., Tanase, G., Tkachyshyn, O., Perdue, J., Amato, N.M., Rauchwerger, L.: A framework for adaptive algorithm selection in STAPL. In: Proc. 10th Symposium on Principles and Practice of Parallel Programming (PPoPP 2005). ACM, New York (2005)
[14] Kohavi, R.: Scaling Up the Accuracy of Naive-Bayes Classifiers: A Decision-Tree Hybrid. In: Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining. AAAI Press (1996)
[15] Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code Generation for DSP Transforms. Proceedings of the IEEE, Special issue on *Program Generation, Optimization, and Adaptation* 93(2), 232–275 (2005)
[16] Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers Inc., San Francisco (1993)
[17] Vuduc, R., Demmel, J.W., Bilmes, J.A.: Statistical Models for Empirical Search-Based Performance Tuning. Int. J. High Perform. Comput. Appl. 18(1) (2004)
[18] Mazouz, A., Touati, S., Barthou, D.: Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of SPEC OMP applications on Intel architectures. In: International Conference on High Performance Computing and Simulation, HPCS (2011)

Automatic Skeleton-Based Compilation through Integration with an Algorithm Classification

Cedric Nugteren, Pieter Custers, and Henk Corporaal

Eindhoven University of Technology, The Netherlands
{c.nugteren,h.corporaal}@tue.nl, pieterjjmcusters@gmail.com

Abstract. This paper presents a technique to fully automatically generate efficient and readable code for parallel processors. We base our approach on skeleton-based compilation and '*algorithmic species*', an algorithm classification of program code. We use a tool to automatically annotate C code with species information where possible. The annotated program code is subsequently fed into the skeleton-based source-to-source compiler 'Bones', which generates OpenMP, OpenCL or CUDA code and optimises host-accelerator transfers. This results in a unique approach, integrating a skeleton-based compiler for the first time into an automated flow. We demonstrate the benefits of our approach on the PolyBench suite by showing average speed-ups of 1.4x and 1.6x for GPU code compared to ppcg and Par4All, two state-of-the-art compilers.

Keywords: Parallel Programming, Algorithm Classification, Algorithmic Skeletons, Source-to-Source Compilation, GPUs.

1 Introduction

The past decades of processor design have led to an increasingly heterogeneous computing environment, in which multi-core CPUs are used in conjunction with accelerators such as graphics processing units (GPUs). Both parallelism and heterogeneity have made programming a challenging task: programmers are faced with a variety of new parallel languages and are required to have detailed architectural knowledge to fully optimise their applications. Apart from programming, maintainability, portability in general, and performance portability in particular have become issues of major importance. Despite a significant amount of work on compilation, auto-parallelisation and auto-tuning (e.g. [2, 6, 7, 11, 13, 21, 22, 23]), many programmers are still struggling with these issues, having to deal with low-level languages such as OpenCL and CUDA.

Existing compilers for parallel targets fall short in at least one of the following areas: 1) they are not fully automatic and require code restructuring or annotations, 2), they directly produce binaries or generate human unreadable code, or 3), they do not generate efficient code. The first shortcoming mostly affects application programmers who are unfamiliar with parallel architectures and concurrent programming, while the second shortcoming mostly affects savvy

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 184–198, 2013.
© Springer-Verlag Berlin Heidelberg 2013

programmers who are leveraging compilers to perform the initial parallelisation and are willing to further optimise the resulting code. The third shortcoming affects all types of users. The goal of this work is to address these shortcomings. We take a unique approach: we use an algorithm classification to drive a source-to-source compiler based on *algorithmic skeletons*.

In this work we present a technique to automatically generate efficient and readable parallel code for parallel architectures (with a focus on GPUs). We base this technique on '*algorithmic species*' [16, 17], an algorithm classification of program code based on the polyhedral model [8]. Algorithmic species encapsulate information such as data re-use and memory access patterns. Algorithmic species form the backbone of our approach (see Fig. 1), which includes a tool to automatically extract species from static affine loops (ASET) and a source-to-source compiler based on skeletons (BONES). The contributions of this work are summarised as follows:

- We present a unique integration of a skeleton-based compiler (BONES) with an algorithm classification (algorithmic species) in Sect. 3. With this combination, skeleton-based compilers can be used in fully-automatic compilation flows, as they no longer require manual identification of skeletons.
- We optimise host-accelerator transfers (e.g. CPU-GPU) in ASET and extend BONES as presented in [15] with new targets, skeletons and optimisations, including register caching, thread coarsening, and zero-copy transfers (Sect. 4).
- We discuss and demonstrate the benefits of our unique approach (Sect. 5) by generating OpenMP, OpenCL and CUDA code for the PolyBench benchmark suite. We focus on the CUDA target, for which we show a speed-up compared to two state-of-the-art polyhedral compilers for individual kernels (1.4x and 1.6x on average) and for complete benchmarks (1.2x and 3.0x on average). Additionally, we demonstrate the importance of host-accelerator transfer optimisations (1.8x speed-up on average).

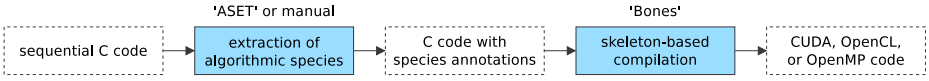


Fig. 1. Overview of the approach taken in this work

2 Related Work

There is a large body of work targeted at (partially) automating parallel programming (e.g. [1, 2, 6, 7, 11, 13, 15, 21, 22, 23]). However, existing work targeting code generation for OpenMP, OpenCL or CUDA often requires annotations in the form of directives [6, 11, 15, 23] or requires major code restructuring [7, 21]. Furthermore, they often produce human unreadable source code for further optimisations [6, 23] or no modifiable source code at all [7]. Exceptions are the polyhedral-based compilers PAR4ALL [1] and PPCG [22] that are able to

fully-automatically compile sequential code into readable parallel code. We compare our approach in terms of performance with these state-of-the-art compilers for CUDA in Sect. 5. PAR4ALL [1] is based on the theory of convex array regions and is implemented using the PIPS source-to-source compiler. PPCG [22] is an optimising polyhedral compiler based on PET and ISL. Both PAR4ALL and PPCG are able to perform host-accelerator (e.g. CPU-GPU) transfer optimisations.

Among the large amounts of existing algorithm classifications (e.g. [4, 7]), we identify only one classification, named *idioms* [18], that provides a tool to perform automatic extraction of classes from source code. However, only six basic classes are provided (stream, transpose, gather, scatter, reduction and stencil), resulting in a significantly lower amount of detail compared to algorithmic species and ASET's automatic extraction.

3 Combining Skeletons with Algorithmic Species

In our two-step approach (see Fig. 1), the first step is to extract relevant information from source code. This information is encapsulated in the form of *algorithmic species*[1] [16, 17]: an algorithm classification based on polyhedral analysis [8]. In this section, we first briefly provide background on algorithmic species and algorithmic skeletons. Following, we discuss the combination of skeleton-based compilation with algorithmic species. Finally, we illustrate the integration of skeleton-based compilation and species by discussing example skeletons.

3.1 Classifying Code using Algorithmic Species

We illustrate algorithmic species by showing an example: matrix-vector multiplication $\boldsymbol{r} = \mathbf{M} \cdot \boldsymbol{s}$ (see Listing 1). To produce a single *element* of the result $\boldsymbol{r}$, we need a complete row from matrix $\mathbf{M}$ and the entire vector $\boldsymbol{s}$. In terms of access patterns used to build algorithmic species, we name the row access from $\mathbf{M}$ a *chunk* access. The vector $\boldsymbol{s}$ is needed entirely to calculate a single element of the result $\boldsymbol{r}$, which we characterise with the *full* pattern. The final algorithmic species is shown in line 1 of Listing 1, including array names and access ranges.

Listing 1. Matrix-vector multiplication classified using algorithmic species

```
1 M[0:127,0:63]|chunk(-,0:63) ∧ s[0:63]|full → r[0:127]|element
2 for (i=0; i<128; i++) {
3   r[i] = 0;
4   for (j=0; j<64; j++) {
5     r[i] += M[i][j] * s[j];
6   }
7 }
```

The example covers three access patterns, forming a single species when combined. In total, five patterns (*element*, *neighbourhood*, *chunk*, *full*, and *shared*) can be combined into an unlimited amount of different species. Species and their

[1] Species is both the English plural and singular form.

patterns are defined formally based on the polyhedral model. Since the patterns are descriptive and intuitive, they may be derived from source code by hand in certain cases. However, to ease the work of a programmer, to avoid mistakes, and to be able to automatically generate parallel code, species are derived automatically using ASET, an algorithmic species extraction tool [5][2]. We make a note that this tool merely identifies species in source code, it does not perform any transformations to extract (more) parallelism from the code. Therefore, our two-step approach could be combined with existing parallelising compilers (e.g. [19]), which can be seen as additional pre-processing.

3.2 Compilation Based on Algorithmic Skeletons

Algorithmic skeletons [4] is a technique that uses parametrisable program code, known as *skeletons* or *skeleton implementations*. An individual skeleton can be seen as template code for a specific class of computations on a specific processor. Users of previous skeleton-based compilers were required to select a skeleton suitable for their algorithm and target processor by hand, and could subsequently invoke the skeleton to generate program code for the target processor. If no skeleton implementation was available for the specific class or processor, it could be added manually. Future algorithms of the same class could then benefit from re-use of the skeleton code. Benefits of skeleton-based compilation are among others the flexibility to extend to other targets, and the performance potential: optimisations can be performed in the native language within the skeletons. Examples of recent skeleton-based compilers are [3, 7, 15, 21].

3.3 From Species to Skeletons

In contrast to existing skeleton-based compilers, we use algorithmic species information to select a suitable skeleton. This does not only enable automation of the whole tool-chain, but also overcomes common critique for skeleton-based compilers illustrated by questions such as 'how difficult is it to select a suitable skeleton' or 'what if the user selects an incompatible skeleton'.

We modified the BONES skeleton-based source-to-source compiler [15] to be combined with algorithmic species. The compiler takes C code annotated with species information as an input. The algorithmic species (extracted by a pre-processor) are used directly to determine the skeleton to use. Additionally, they are used to enable/disable additional transformations and optimisations, although most optimisations can be made within the skeletons themselves.

Algorithmic species map one-to-one to skeletons: the compiler needs to supply a skeleton for every species it wants to support. These skeletons should be constructed in such a way that they are correct (and preferably optimised) for all algorithms belonging to a given species. For practical reasons however (to save work/code duplication), we provide a species-to-skeleton mapping, such that

[2] ASET can be replaced by the more recent A-DARWIN tool [16].

multiple species can map to the same skeleton. For example, for a GPU target, algorithmic species of the form '`neighbourhood` → `element`' and '`neighbourhood` ∧ `element` → `element`' both map to a single skeleton that enables explicit caching of the neighbourhood in the GPU's scratchpad memory.

We illustrate the working of BONES through Fig. 2. In this figure, we show an example input with two loop nests identified as two different species ('species X' and 'species Y' in the figure). The compiler first loads and invokes the corresponding skeletons for a given target. Then, the skeleton-specific transformations are performed, and finally, BONES combines the results to obtain target code. The optimisations as described in Sect. 4 are performed in the last stage.

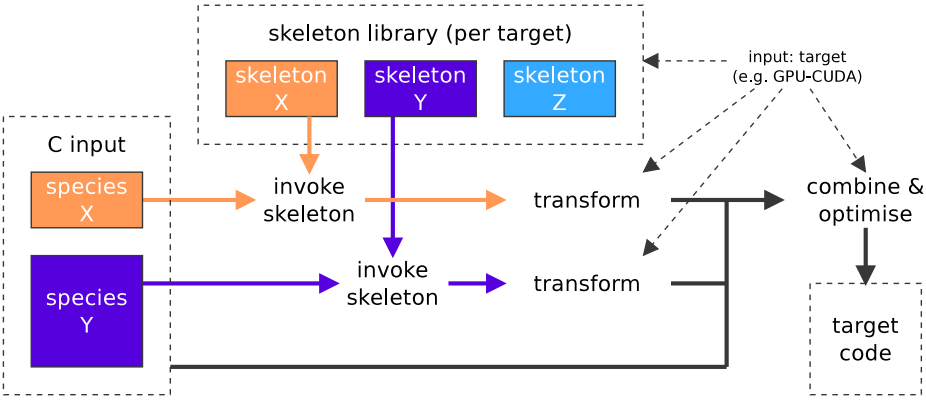


Fig. 2. Illustration of the structure of the BONES compiler for an example input with two different species

3.4 Example Skeletons

To illustrate the use of skeletons within BONES, we first show a simplified OpenMP skeleton in Listing 2 and its instantiation for the matrix-vector multiplication example (Listing 1) in Listing 3. The skeleton in Listing 2 shows highlighted keywords, which are filled in as follows: `parallelism` represents the parallelism found in the species, `ids` computes the identifier corresponding to the current iteration, and `code` fills in the transformed code. For illustration purposes, the example skeleton is heavily simplified, excluding comments, boundary and initialisation code, function calls and definitions, and makes several assumptions, such as the divisibility of the amount of parallelism by the thread count.

Additionally, we show an example of a skeleton for the CUDA target in Listing 4. This skeleton is specific to species of the form '`0:N,0:N|chunk(-,0:N)` → `0:N,0:N|element`', similar to the matrix-vector multiplication kernel shown in Listing 1. A naive mapping to CUDA will result in *uncoalesced* accesses to the *chunk* array (**M** in the example): subsequent accesses will be made by the same

Listing 2. A simplified skeleton for OpenMP. Details are left out for clarity.

```
1 int count;
2 count = omp_get_num_procs();
3 omp_set_num_threads(count);
4 #pragma omp parallel
5 {
6   int tid, i;
7   int work, start, end;
8   tid = omp_get_thread_num();
9   work = <parallelism>/count;
10  start = tid*work;
11  end = (tid+1)*work;
12
13  // Start the parallel work
14  for(i=start; i<end; i++) {
15    <ids>
16    <code>
17
18
19
20 }
```

Listing 3. Instantiated code for the Listing 1 example. Optimisations are omitted.

```
1 int count;
2 count = omp_get_num_procs();
3 omp_set_num_threads(count);
4 #pragma omp parallel
5 {
6   int tid, i;
7   int work, start, end;
8   tid = omp_get_thread_num();
9   work = 128/count;
10  start = tid*work;
11  end = (tid+1)*work;
12
13  // Start the parallel work
14  for(i=start; i<end; i++) {
15    int gid = i;
16    r[gid] = 0;
17    for (j=0; j<128; j++)
18      r[gid] += M[gid][j]*s[j];
19  }
20 }
```

thread. To re-enable coalescing in these cases, which is paramount for performance, a special skeleton with a pre-shuffling kernel is designed. The skeleton in Listing 4 shows a kernel for the actual work (lines 1-8) and a kernel to reorder the input array by re-arranging data in the on-chip memory of the GPU (lines 9-20). The use of this skeleton implies a transformation in the original code as well (e.g. from `M[i][j]` into `M[j][i]`), which is handled by the compiler. Again, this skeleton is heavily simplified for illustration purposes, e.g. not showing boundary checks nor the host code to launch the kernels.

Listing 4. A simplified skeleton for the CUDA target to enable coalesced accesses

```
1 // CUDA kernel for the actual work (simplified)
2 __global__ void kernel0(...) {
3   int gid = blockIdx.x*blockDim.x+threadIdx.x;
4   if (gid < <parallelism>) {
5     <ids>
6     <code>
7   }
8 }
9 // CUDA kernel for pre-shuffling (simplified)
10 __global__ void kernel1(...) {
11   int tx = threadIdx.x; int ty = threadIdx.y;
12   __shared__ <type> b[16][16];
13   int gid0 = blockIdx.x*blockDim.x + tx;
14   int gid1 = blockIdx.y*blockDim.y + ty;
15   int nid0 = blockIdx.y*blockDim.y + tx;
16   int nid1 = blockIdx.x*blockDim.x + ty;
17   b[ty][tx] = in[gid0 + gid1*<dims>/<params>];
18   __syncthreads();
19   out[nid0 + nid1*<params>] = b[tx][ty];
20 }
```

4 Compiler Optimisations

In this section we discuss the host-accelerator transfer optimisations made by ASET and BONES, and illustrate some of the optimisations possible within BONES. The discussed optimisations in this section are new to BONES: they are not present in earlier non-species based versions [15]. Both ASET, and BONES are open-source and are freely available[3]. The compiler BONES is programmed in Ruby and uses the C-to-AST module CAST[4]. BONES currently supplies a total of 15 skeletons for 5 different targets: OpenMP for CPUs, CUDA for NVIDIA GPUs, OpenCL for AMD GPUs, and OpenCL for CPUs (AMD and Intel SDK).

4.1 Host-Accelerator Transfer Optimisations

Many of today's parallel processors are designed as an *accelerator*: they require a *host* processor to dispatch tasks (or: kernels). Furthermore, they often have a separate memory (e.g. GPUs, Intel MIC), requiring host-accelerator transfers of input and output arrays. When executing multiple kernels, this gives opportunities to optimise these transfers in several ways [10, 12]: 1) transfers might be omitted (e.g. subsequent kernels use the same data), 2) transfers can run in parallel with host code (e.g. start the copy-in as soon as the data is ready), and 3) transfers unrelated to a specific kernel can run in parallel with kernel execution.

We perform such optimisations within ASET. After delimiting the identified algorithmic species by pragma's, ASET is instructed to: 1) mark inputs and outputs as copy-ins and copy-outs for the current kernel, and 2) add synchronisation barriers after the transfers. BONES then generates a second host thread, receiving transfer requests and performing synchronisations. An example of non-optimised output is shown in the form of pseudo code in Listing 5, in which each copy-in and copy-out implies that it is allowed to start the copy at that specific point, and must be finished before the given deadline (see also Fig. 3).

After producing the non-optimised form (e.g. Listing 5), the tool performs different types of optimisations in an iterative way, including: 1) copy-ins directly after copy-outs are removed (e.g. from Listing 5 to Listing 6), 2) copy-ins are moved to the front if the data is not written by the previous species, 3) deadlines of copy-outs are increased if the data is not written by the next species (e.g. from Listing 6 to Listing 7), 4) unused synchronisation barriers are omitted, and 5) transfers are moved to an outer loop if possible. The effects of the transfer optimisations are discussed in Sect. 5.

4.2 Optimisations within Bones

The compiler BONES performs various optimisations to the source code before it instantiates a skeleton. Most optimisations are conditionally applied based on the algorithmic species. For example, array indices and the corresponding array

[3] BONES and ASET can be found at: `http://parse.ele.tue.nl/species/`.
[4] CAST can be found at `http://cast.rubyforge.org/`.

Listing 5. Transfer example (original)

```
1  copy−in(A,1)
2  sync(1)
3  kernel: B ← A
4  copy−out(B,2)
5  sync(2)
6  copy−in(B,3)
7  sync(3)
8  kernel: C ← B
9  copy−out(C,4)
10 sync(4)
```

Listing 6. Transfer example (partly optimised)

```
1 copy−in(A,1)
2 sync(1)
3 kernel: B ← A
4 copy−out(B,2)
5 sync(2)
6 kernel: C ← B
7 copy−out(C,4)
8 sync(4)
```

Listing 7. Transfer example (fully optimised)

```
1 copy−in(A,1)
2 sync(1)
3 kernel: B ← A
4 copy−out(B,4)
5 kernel: C ← B
6 copy−out(C,4)
7 sync(4)
```

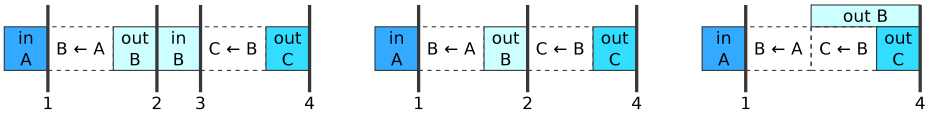


Fig. 3. Illustrating the optimisation steps: original (left, Listing 5), partly optimised (middle, Listing 6), fully optimised (right, Listing 7)

names for input arrays in *neighbourhood*-based skeletons for GPUs are replaced by local indices and local array names. This transformation is a simple matter of name-changing, the actual definition of the local indices and the pre-fetching into local memories is performed within the corresponding skeletons.

Bones also performs several performance-oriented transformations, including caching in the register file and *thread coarsening*. Register file caching replaces array accesses (mapped to off-chip memories) with scalar accesses (mapped to registers) in certain cases. For example, in Listing 1, the accesses to vector $\boldsymbol{r}$ in lines 3 and 5 can be replaced by scalar accesses under the condition that a final store to $\boldsymbol{r}$ is added after line 6. Thread coarsening (or merging) is a technique to increase the workload per thread. This comes at the cost of parallelism, but could increase data re-use through locality or factor out common instructions [14]. In Bones, coarsening is enabled if the species encapsulates re-use over a complete data structure. For example, in the case of '`0:M,0:N|chunk(-,0:N)` → `0:M,0:N|element`', a total of $N \cdot M$ elements are produced, while only M chunks are available as input, resulting in the re-use of the entire input data structure with a factor N. Coarsening is only enabled for kernels without divergent control flow and with sufficient data re-use and parallelism.

To further improve the performance of the generated code, Bones enables *zero-copy* for the OpenCL targets using an aligned memory allocation scheme. In OpenCL, data needs to be explicitly copied from *host* (typically a CPU) to *device* (the accelerator). In some cases, the host and device share the same memory, e.g. for CPU targets or for fused CPU/GPU architectures. In these cases, a memory copy can be saved by performing a pointer-only copy, i.e. a *zero-copy*. Bones enables zero-copying in OpenCL for Intel CPUs by fulfilling two requirements: 1) using specific OpenCL memory map and memory unmap functions, and 2), aligning all memory allocations to 128-byte boundaries.

To ensure aligned memory allocations in the original code, BONES provides a custom `malloc` implementation and furthermore replaces existing stack allocations with aligned dynamic allocations in a matter similar to [12].

Furthermore, BONES flattens data structures to a single dimension when generating OpenCL or CUDA code. Parallel loops are also flattened, decoupling the amount of loops from the thread or work-item structures provided by OpenCL and CUDA. In contrast, many existing approaches (e.g. [1, 7, 22]) map multi-dimensional loops to the multi-dimensional thread or work-item structures provided by OpenCL and CUDA. Although this might be a straightforward solution, it limits the applicability of these approaches to 2 or 3-dimensional loops and data structures. By flattening, BONES omits these limitations and is able to handle any degree of loop nesting and arrays of any dimension.

5 Evaluation and Experimental Results

To evaluate the applicability of algorithmic species and to validate ASET and BONES, we test against the PolyBench/C 3.2 benchmark suite[5], which is a popular choice for evaluating polyhedral-based compilers [1, 22]. We choose this suite because it allows us to compare against polyhedral-based compilers, and allows us to perform automatic extraction of algorithmic species. Nevertheless, BONES can still be used for code that does not fit the polyhedral model, albeit that the classification of code has to be performed manually (see [16] for examples).

The PolyBench suite contains 30 algorithms, in which we identify a total of 110 species[6] with ASET, or 60 if we exclude those found within other species (i.e. nested species). The classified code for these 60 species can now be fed into BONES. However, there is a large number of benchmarks with species in the form of inner-loops with little work, executing in a fraction of a millisecond, making start-up and measurement costs dominant. For our evaluation, we therefore exclude `adi`, `cholesky`, `dynprog`, `durbin`, `fdtd-2d-apml`, `gramschmidt`, `lu`, `ludcmp`, `reg_detect`, `symm`, `trmm` and `trisolv`. The exclusion of these benchmarks can be automated by integrating a basic roofline-like performance model. Additionally, we exclude `floyd-warshall` and `seidel-2d` because they contain no parallelism in their current form. All in all, we include 34 species (not necessarily unique) spread across 16 benchmarks. Within a benchmark, we number the found algorithmic species sequentially[7].

Although GPUs are currently the primary target of BONES, we also include an evaluation of our multi-core CPU targets: we perform a comparison of the 3 CPU targets against single-threaded code. Next, we compare our approach for the CUDA target against two state-of-the-art polyhedral-based compilers: PAR4ALL [1] and PPCG (based on the PLUTO algorithm) [22]. To the best of our knowledge, these are the only available fully-automatic compilers able to

[5] PolyBench website: `www.cse.ohio-state.edu/~pouchet/software/polybench/`.
[6] PolyBench annotated with species: `http://parse.ele.tue.nl/species/`.
[7] See `http://parse.ele.tue.nl/species/` for the corresponding species.

generate CUDA code directly from C (we exclude C-to-CUDA [2], as it is limited to kernel generation only). Finally, we discuss the benefits of our unique combination of a skeleton-based compiler and an algorithm classification.

5.1 OpenCL and OpenMP on a Multi-core CPU

We perform experiments on a 4-core CPU comparing the 3 CPU targets (OpenMP, Intel SDK OpenCL, AMD SDK OpenCL) against single-threaded C code. We use an Intel Core i7-3770 (4 cores, 8 threads) with support for AVX. We use the auto-vectorising GCC 4.6.3 (`-O3`) compiler for the single-threaded and OpenMP targets. Furthermore, we use AMD APP 2.7 and Intel OpenCL 2012 for the OpenCL targets. We disable the CPU's *Intel Turbo Boost* technology. We use the 'large dataset' as defined in PolyBench with single-precision floating point computations. We average over multiple runs and start each run with a warm-up dummy computation followed by a cache flush.

We show the results of the experiments in Fig. 4. We see geometric mean speed-ups of 2.1x (Intel SDK OpenCL), 2.4x (AMD SDK OpenCL), and 2.7x (OpenMP). Although the three targets use the same hardware, we still observe significant performance variation for the individual benchmarks. Differences are among others the lower thread creation cost for OpenMP, the different thread scheduling policies, and different auto-vectorisers. For a detailed comparison of OpenMP against OpenCL in general, we refer to other work, such as [20]. Furthermore, we expect to see improved speed-ups by applying a pre-processing parallelisation pass first (such as PLUTO or [19]), which we leave for future work.

5.2 Comparison of the CUDA Target against the State-of-the-Art

For our CUDA experiments, we use an NVIDIA GeForce GTX470 GPU (448 CUDA cores) and GCC 4.6.3 (`-O3`) and NVCC 5.0 (`-O3 -arch=sm_20`) for compilation. We test BONES version 1.2 against the latest versions of PAR4ALL (version 1.4.1) and PPCG (pre-release commit 94af357, Feb 2013)[8]. We test BONES with and without host-accelerator transfer optimisations. Furthermore, we include as a reference PolyBench/GPU [9], hand written CUDA code[9]. As before, we use the single-precision 'large dataset' and average over 10 runs, starting with a warm-up.

We perform two tests. For our first test, we evaluate the quality of the generated CUDA kernels only. We generate kernels for each of the found algorithmic species in isolation using BONES, PAR4ALL, and PPCG. As before, within a benchmark, we number the found algorithmic species incrementally, while averaging the execution time of kernels executed multiple times. The results of the kernel quality test can be found in Fig. 5 in terms of speed-up of BONES compared to PAR4ALL and PPCG. We make the following remarks:

[8] We test PPCG with default tiling options. Manual tuning of the tiling parameter for each benchmark is omitted in this work to keep the obtained results fully automatic.

[9] PolyBench/GPU provides non-optimised CUDA code. Unfortunately, fully optimised hand-written code is not available for these benchmarks.

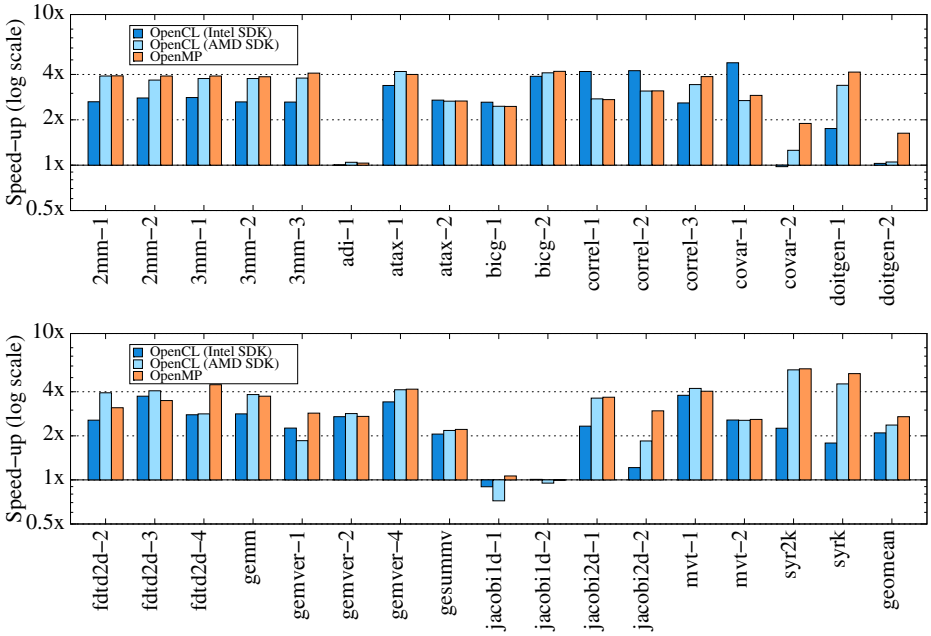


Fig. 4. Speed-ups of the OpenCL (■ for the Intel SDK and ■ for the AMD SDK) and OpenMP (■) targets compared to single-threaded code on a 4-core CPU

- Bones shows significantly better performance (2x or more) for 21 kernels (compared to Par4All) and 11 kernels (compared to ppcg).
- In a few cases (e.g. `correl-3`,`covar-2`,`jacobi2d-2`) ppcg and/or Par4All are slightly ahead due to better data locality enabled by loop tiling.
- A number of kernels (e.g. `atax-1`, `bicg-2`, `mvt-1`, `syr2k`, `syrk`) use a skeleton in the form of Listing 4 to ensure coalesced memory accesses, yielding a significant speed-up over Par4All and a moderate speed-up over ppcg.
- In the cases of `2mm` and `3mm` (both matrix multiplication), Bones is on-par with ppcg. In these cases, Bones relies on the hardware cache, while ppcg performs loop tiling and explicit caching through the GPU's local memories.
- In several cases (`2mm`, `3mm` and `gemm`), Bones and ppcg both perform *thread coarsening*, gaining performance over Par4All.
- Bones achieves a 1.6x (vs. Par4All) and 1.4x (vs. ppcg) average speed-up.

For the second test, we measure the full benchmark ('scop' in PolyBench terminology). We show the results in Fig. 6 in terms of speed-up compared to optimised Bones code. The (experimental) option to optimise CPU-GPU transfers for Par4All (`--com-optimization`) does not work for these benchmarks. For a fair comparison, Par4All should therefore be compared to Bones without transfer optimisations. We make the following remarks with respect to Fig. 6:

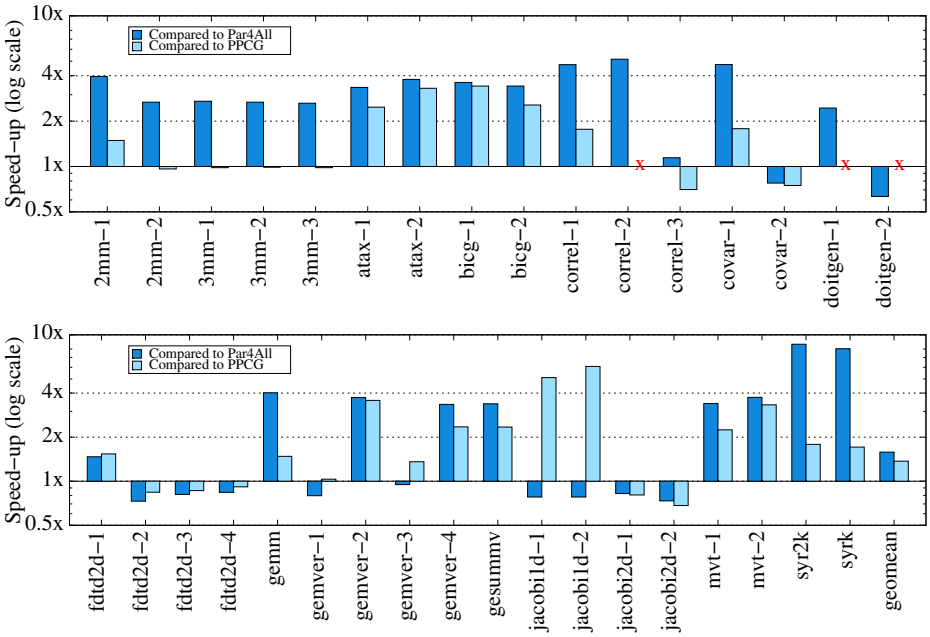


Fig. 5. Speed-up of CUDA kernel code generated by Bones compared to Par4All (■) and ppcg (■) for the PolyBench suite (higher is in favour of Bones). ppcg was unable to generate code for `correl-2`, `doitgen-1`, and `doitgen-2` (marked by an x).

- In almost all cases, Bones with transfer optimisations outperforms the other compilers. On average, Bones achieves a speed-up of 3.0x and 1.2x over Par4All and ppcg respectively (excluding the 100x or worse cases).
- Hand written (non-optimised) PolyBench/GPU code [9] is in many cases significantly slower compared to compiler generated code (on average 4.2x compared to Bones).
- A 1.8x average speed-up is achieved by performing CPU-GPU transfer optimisations with aset and Bones, although further optimisations are still possible. For example, in the case of `fdtd-2d`, ppcg is able to move additional CPU-GPU transfers to outer loops, resulting in a significant speed-up.

In general, we conclude that the supplied skeletons for Bones already outperform Par4All and ppcg on average for the CUDA kernels found in these benchmarks. For the 34 kernels shown in Fig. 5, Bones uses 5 different skeletons, each used at least twice. Furthermore, we note that Bones, in contrast to polyhedral compilers, can be used for code containing non-affine loop nests.

5.3 Benefits of Skeleton-Species Integration

By integrating algorithmic species with a skeleton-based compiler, we have created a unique approach to automatically generate code for parallel targets.

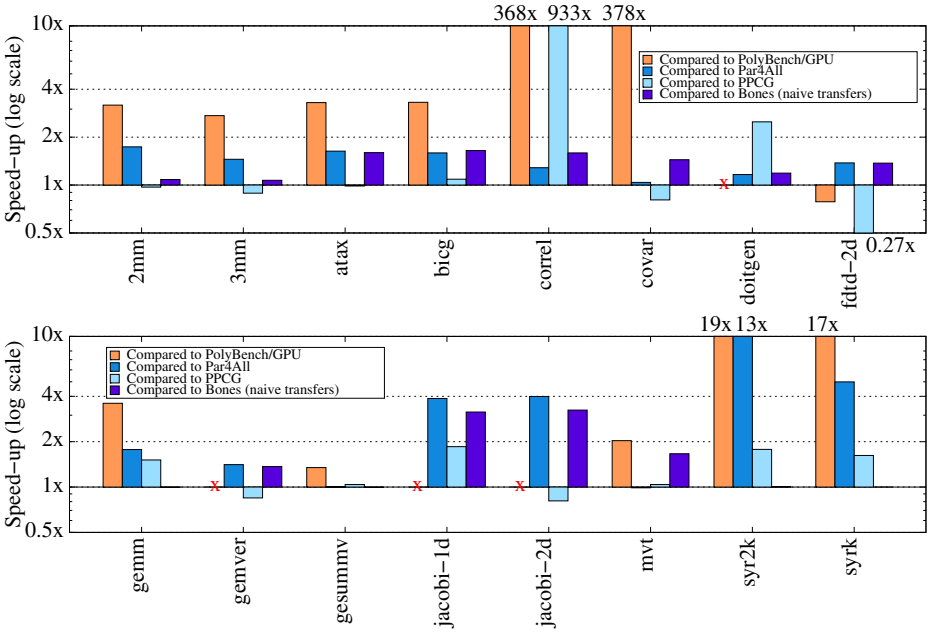


Fig. 6. Speed-up of Bones compared to PolyBench/GPU (■), Par4All (■), ppcg (■), and Bones without transfer optimisations (■) (higher is in favour of Bones). PolyBench/GPU is not available for `doitgen`, `gemver`, `jacobi-1d`, and `jacobi-2d` (x).

We see this novel combination as a way to profit from the benefits of skeleton-based compilation, without having its drawbacks.

Skeleton-based compilation has several benefits [4]. Firstly, compilation requires only basic transformations that can be performed at abstract syntax tree level, omitting the need for intermediate representations which often lose code structure and variable naming. This allows the compiler to generate readable code, enabling opportunities for further fine-tuning and manual optimisation. Furthermore, the skeletons themselves can be formatted to include structure and code comments, greatly benefiting readability. Secondly, skeleton-based compilation benefits from the flexibility of improving the compiler or extending to other targets: simply adjust or write the appropriate skeletons. Finally, several optimisations applied within skeletons in Bones cannot be applied as code transformations on the original code. For example, polyhedral compilers could generate the first kernel of the example skeleton in Listing 4 (lines 1-8), but will not be able to generate an additional pre-processing kernel (lines 10-23), as it is not a permutation of the original code.

Because of the integration of algorithmic species, Bones is the first skeleton-based compiler that can be used in a fully-automatic tool-flow. This removes the requirements of existing skeleton-based approaches such as SkePU [7] and SkeCL [21] to manually identify a skeleton and modify the code such that the

skeleton can be used. Furthermore, algorithmic species provides a clear, structured, and formally defined way of using skeletons, which can be beneficial in cases where manual classification is unavoidable.

6 Conclusions and Future Work

In this work we have demonstrated the successful integration of an algorithm classification (algorithmic species) with skeleton-based compilation. This results in a novel method to perform automatic generation of parallel code, transforming sequential C code into readable CUDA, OpenCL or OpenMP code. With ASET, we are able to automatically extract species from affine loop nests. The species are used within the skeleton-based compiler BONES to select a skeleton. BONES is able to produce readable code, provides flexibility for new targets and optimisations, and generates efficient code. Furthermore, the combination of ASET and BONES allows us to perform host-accelerator transfer optimisations.

Many existing skeleton-based compilers [3, 4, 7, 21] have failed to become popular, despite their flexibility and high performance potential. By combining skeletons with an algorithm classification, we overcome their drawbacks: we do not require users to select a skeleton, and we have a fixed and structured classification. Compared to polyhedral compilers on the other hand (e.g. [1, 22]), we have shown significant speed-ups: 1.6x and 1.4x for CUDA kernel code versus PAR4ALL and PPCG respectively. Additionally, BONES generates readable code, leaving the user with further possibilities for optimisation or fine-tuning of the algorithm. Furthermore, BONES is applicable outside the scope of affine loop nests, although species may have to be identified manually.

In this work, we have focused mainly on the CUDA GPU target. As part of future work, we plan to compare our OpenCL targets against the state-of-the-art as well. Furthermore, we plan to extend BONES by investigating the benefits of kernel fusion and fission.

References

[1] Amini, M., Creusillet, B., Even, S., Keryell, R., Goubier, O., Guelton, S., Mcmahon, J.O., Pasquier, F.-X., Péan, G., Villalon, P.: Par4All: From Convex Array Regions to Heterogeneous Computing. In: IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques (2012)

[2] Baskaran, M.M., Ramanujam, J., Sadayappan, P.: Automatic C-to-CUDA Code Generation for Affine Programs. In: Gupta, R. (ed.) CC 2010. LNCS, vol. 6011, pp. 244–263. Springer, Heidelberg (2010)

[3] Caarls, W., Jonker, P., Corporaal, H.: Algorithmic Skeletons for Stream Programming in Embedded Heterogeneous Parallel Image Processing Applications. In: IPDPS: Int. Parallel and Distributed Processing Symposium. IEEE (2006)

[4] Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press (1991)

[5] Custers, P.: Algorithmic Species: Classifying Program Code for Parallel Computing. Master's thesis, Eindhoven University of Technology (2012)

[6] Dolbeau, R., Bihan, S., Bodin, F.: HMPP: A Hybrid Multi-core Parallel Programming Environment. In: GPGPU-1: 1st Workshop on General Purpose Processing on Graphics Processing Units (2007)
[7] Enmyren, J., Kessler, C.W.: SkePU: A Multi-backend Skeleton Programming Library for Multi-GPU Systems. In: HLPP 2010: 4th International Workshop on High-level Parallel Programming and Applications. ACM (2010)
[8] Feautrier, P.: Dataflow Analysis of Array and Scalar References. Springer International Journal of Parallel Programming 20, 23–53 (1991)
[9] Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S., Cavazos, J.: Autotuning a High-Level Language Targeted to GPU Codes. In: Workshop on Innovative Parallel Computing (2012)
[10] Guelton, S., Amini, M., Creusillet, B.: Beyond Do Loops: Data Transfer Generation with Convex Array Regions. In: Kasahara, H., Kimura, K. (eds.) LCPC 2012. LNCS, vol. 7760, pp. 249–263. Springer, Heidelberg (2013)
[11] Han, T., Abdelrahman, T.: hiCUDA: High-Level GPGPU Programming. IEEE Transactions on Parallel and Distributed Systems 22, 78–90 (2011)
[12] Jablin, T., Jablin, J., Prabhu, P., Liu, F., August, D.: Dynamically Managed Data for CPU-GPU Architectures. In: CGO 2012: International Symposium on Code Generation and Optimization. ACM (2012)
[13] Khan, M., Basu, P., Rudy, G., Hall, M., Chen, C., Chame, J.: A Script-Based Autotuning Compiler System to Generate High-Performance CUDA Code. ACM Transactions on Architecture and Code Optimisations 9(4), Article 31 (January 2013)
[14] Lee, Y., Krashinsky, R., Grover, V., Keckler, S.W., Asanovic, K.: Convergence and Scalarization for Data-Parallel Architectures. In: CGO 2013: International Symposium on Code Generation and Optimization. IEEE (2013)
[15] Nugteren, C., Corporaal, H.: Introducing 'Bones': A Parallelizing Source-to-Source Compiler Based on Algorithmic Skeletons. In: GPGPU-5: 5th Workshop on General Purpose Processing on Graphics Processing Units. ACM (2012)
[16] Nugteren, C., Corvino, R., Corporaal, H.: Algorithmic Species Revisited: A Program Code Classification Based on Array References. In: MuCoCoS 2013: International Workshop on Multi-/Many-core Computing Systems (2013)
[17] Nugteren, C., Custers, P., Corporaal, H.: Algorithmic Species: An Algorithm Classification of Affine Loop Nests for Parallel Programming. ACM TACO: Transactions on Architecture and Code Optimisations 9(4), Article 40 (2013)
[18] Olschanowsky, C., Snavely, A., Meswani, M., Carrington, L.: PIR: PMaC's Idiom Recognizer. In: ICPPW 2010: 39th International Conference on Parallel Processing Workshops. IEEE (2010)
[19] Park, E., Pouchet, L.-N., Cavazos, J., Cohen, A., Sadayappan, P.: Predictive Modeling in a Polyhedral Optimization Space. In: CGO 2011: International Symposium on Code Generation and Optimization. IEEE (2011)
[20] Shen, J., Fang, J., Sips, H., Varbanescu, A.: Performance Gaps between OpenMP and OpenCL for Multi-core CPUs. In: ICPPW: International Conference on Parallel Processing Workshops. IEEE (2012)
[21] Steuwer, M., Kegel, P., Gorlatch, S.: SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In: IPDPSW 2011: International Symposium on Parallel and Distributed Processing Workshops and PhD Forum. IEEE (2011)
[22] Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C., Catthoor, F.: Polyhedral Parallel Code Generation for CUDA. ACM Transactions on Architecture and Code Optimisations 9(4), Article 54 (January 2013)
[23] Wolfe, M.: Implementing the PGI Accelerator Model. In: GPGPU-3: 3rd Workshop on General Purpose Processing on Graphics Processing Units. ACM (2010)

Optimizing Program Performance via Similarity, Using a Feature-Agnostic Approach

Rosario Cammarota, Laleh Aghababaie Beni,
Alexandru Nicolau, and Alexander V. Veidenbaum

Department of Computer Science, University of California Irvine, Irvine, USA
{rosario.c,laghabab,nicolau,alexv}@ics.uci.edu

Abstract. This work proposes a new technique for performance evaluation to predict performance of parallel programs across diverse and complex systems. In this work the term system is comprehensive of the hardware organization, the development and execution environment.

The proposed technique considers the collection of completion times for some pairs (*program*, *system*) and constructs an empirical model that learns to predict performance of unknown pairs (*program*, *system*). This approach is feature-agnostic because it does not involve previous knowledge of program and/or system characteristics (features) to predict performance.

Experimental results conducted with a large number of serial and parallel benchmark suites, including SPEC CPU2006, SPEC OMP2012, and systems show that the proposed technique is equally applicable to be employed in several compelling performance evaluation studies, including characterization, comparison and tuning of hardware configurations, compilers, run-time environments or any combination thereof.

Keywords: Program Characterization, Feature-agnostic, Cluster Analysis, Empirical Performance Modeling, Program Optimization.

1 Introduction

Over the past decade there has been an exponential growth in computer performance [1] that quickly led to more sophisticate and diverse software and computing platforms (e.g., heterogeneous multi-core platforms [2], parallel browsers [3]). The cost of software development and hardware design too increases and creates the need for evaluating performance of proposed software and system changes before the actual implementation and deployment begin.

However, given the increasing complexity of modern micro-architectures [2], software [4, 5], development and execution environments [6], performance of a program on new systems (specularly, performance that a system delivers to new programs) is difficult to predict. Constructing a comprehensive model that includes all the possible aspects featuring software and computing platform is practically limited by the cost of feature retrieval compared with the performance goal to reach. For example, while having negligible run-time overhead, collecting

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 199–213, 2013.
© Springer-Verlag Berlin Heidelberg 2013

a large number of hardware performance counters is not possible at once - it requires multiple runs of a software as only a limited number of hardware performance counters can be collected in one run, e.g., up to eight in modern Intel microprocessors [7]. Instrumentation based retrieval and simulation also incur in significant run-time overhead [8].

As a result, prior research (a) mainly focuses on the characterization and empirical performance modeling of serial programs and (b) limits its attention to a number of program/system features that are tied to a specific performance study (e.g., system procurement [9, 10], tuning of compiler heuristics [11, 12], task-to-core allocation tuning for heterogeneous and homogeneous multi-cores [13, 14], selection of the number of cores and parallel scheduling algorithm [15], design space exploration [16–18]) and do not generalize to other performance studies. These techniques can be categorized as feature-aware because the features chosen to characterize the specific performance study are extracted from either system and program properties, e.g., hardware performance counters, or program-inherent properties, e.g., instructions mix, working set size [19], or system design properties, e.g., number, type, size and organization of hardware components [16, 18, 20].

In this work we present a new feature-agnostic technique for performance modeling, prediction and, more important, we present the application of the proposed technique to several cases of serial and parallel program performance optimization. The type of characterization presented in this work differs from the characterizations introduced in previous research. In fact, in this work only the knowledge of completion times for some pairs ($program$, $system$) is leveraged to predict performance for unknown pairs ($program$, $system$). The characterization of programs and systems is embedded in series of known performance of a programs on certain systems and known performance that some systems give to some programs. Because of the above, the proposed technique is said to be feature-agnostic because it does not involve a knowledge of programs (e.g., the number of instructions of a certain type, the memory footprint) and systems (e.g., the memory hierarchy organization, the operating frequency) characteristics.

Using the information available, first a technique based on micro-array visualization [21] is proposed to highlight the presence of coherent patterns of similarity between programs and systems. Programs exhibiting similar series of performance on different systems are clustered together using hierarchical clustering [22]. Likewise, systems exhibiting similar series of performance on different programs are clustered together. A heat-map is constructed to provide a joint visualization of coherent patterns, i.e., areas on the map on which ($program$, $system$) performance are nearly-equal. The presence of such patterns highlights opportunities to predict program performance by similarity. For example, two (or more) programs that attain nearly-equal performance on a number of systems are deemed to be similar under a certain similarity metric. These programs are also likely to attain nearly-identical performance on a new system. Hence, the knowledge of performance for some of these programs on a new system is predictive of performance of a similar, unseen programs on this system.

Performance modeling and prediction happens as follows: two prediction models are constructed and their outcomes are combined to predict performance for unknown pairs (*program*, *system*). The first model aims to predict performance that a new system provides to a program, given the available performance that other systems provided to the program. The second model aims to predict performance of a program on a new system given the available performance of other programs on the system. Performance prediction for unknown pairs (*program*, *system*) is obtained as the weighted average of the outcomes of the two predictors above. The importance of combining the two predictions above is to improve prediction accuracy compared with the accuracy of the individual model. To construct prediction models, we use, evaluate and compare machine learning algorithms for regression, e.g., Linear Regression (`LR`) [23] and Support Vector Regression (`SVR`) [24].

Program optimization occurs in the form of selecting the combination of algorithmic optimization, compilation, execution environment and hardware design that maximizes performance, i.e., minimizes completion time, of a program of interest. A procedure based on k-fold cross-validation [25] is proposed to validate the proposed prediction technique.

To the best of our knowledge this is the first work that considers a feature-agnostic, practical technique for program and system characterization, performance modeling and prediction. In particular, this work makes the following contributions: (*i*) It provides a new, practical and generally applicable technique for cross-system performance modeling and prediction. Performance prediction relies on a fairly general and simple to retrieve program characterization, i.e., performance of some pairs (*program*, *system*) - which usually is available in industry and research settings; (*ii*) The application of the proposed technique to several practical performance studies is shown, e.g., the cases of system selection, compiler and run-time settings selection are illustrated; (*iii*) It is also shown that for the sake of several performance studies, the problem of program characterization and specifically that of feature selection can be moved to the problem of selecting appropriate simpler programs, e.g., programs from different application domains. Performance modeling and prediction happens by similarity between series of completion time of a program on different systems and of a system on different program; (*iv*) It is shown that the proposed characterization is not only applicable across different systems, but also holds for both serial and parallel programs.

The rest of the paper is organized as follows: Basic concepts and definitions that are adopted in this work are introduced in Section 2. Section 3 discusses previous research; The characterization and performance modeling technique is described in Section 4. Section 4 also discusses on the evaluation methodology developed to validate the proposed performance modeling technique; The experimental evaluation and results are presented and discussed Section 5. Key highlights and future developments are discussed in Section 6.

2 Basic Concepts and Definitions

In this work a property of a program, e.g., instruction mix, working set size, and/or of a system, e.g., memory hierarchy configuration, average number of cycles to execute each instruction in the instruction set architecture (`ISA`), is a *feature* of the program and/or the system. A set of features is a *signature*. This work focuses on signatures whose features assume real number values, techniques to analyze patterns (similarity) occurring between features, empirical performance modeling that associate performance to features.

A signature is a n-dimensional vector of real numbers $\mathbf{s} = [f_1, f_2, \cdots, f_n]$ whose elements can be homogeneous (they measure the same quantity, e.g., a series of energy consumptions or completion time) or heterogeneous (they measure different quantities, e.g., number of cache misses, cost of communication). A feature is agnostic of program and/or system properties if its value cannot be directly associated to any property of a program or of a system. Examples of such type of features are total completion time, total energy consumption etc. Otherwise a feature is aware of program and/or system characteristics. Techniques to model program and system performance for program optimization are *feature-agnostic* when the signature adopted by the specific technique is composed of features that only latently account for program and/or system properties. Otherwise the approach is said to be *feature-aware*.

Given a set of signatures, a similarity measure [26] provides a way of measuring the degree of similarity between two programs (or systems). The comparison of two signatures under a certain similarity measure makes program characterization (i.e., discovery of groups of signatures with similar properties) possible. More in general, the use of cluster analysis (as explained in the next Section) makes possible to partition a set of signatures into smaller groups of signatures with similar features [22]. Because the type of characterization proposed in this work is feature-agnostic and the elements of a signature are completion times, the similarity measure adopted in this work is the Euclidean distance.

3 Related Work

For a specific performance study, deciding whether to adopt a feature-aware or -agnostic characterization has pros and cons. Feature-aware characterizations appear in many common practices in program characterization and tuning [19, 27, 28], algorithmic optimization [29], various intelligent forms of profile-guided optimization [11, 12] and design space exploration [16, 18]. While feature-aware characterizations provide information to interpret the behavior of a program and/or system, feature extraction comes at the cost of iterating multiple program runs or program/compiler instrumentation and/or simulation. In addition to the above, the amount of features amongst which one can choose is enormous. As a result, only a small subset of features are usually considered at once and the selection of features is usually subjective and/or time/cost constrained. While statistical analysis techniques can be employed to select essential features from a group of features, the selection of the initial group of features remains subjective.

On the contrary, feature-agnostic characterizations have been used to analyze the coverage of programs within a benchmark suite [30], for hardware design rating [9, 31] and compiler aided program optimization [32]. However, in prior work feature-agnostic characterizations have been employed in conjunction with serial programs and only one of the following signatures was used at a time: program performance across systems (*program signature*); performance of multiple programs on a system (*system signature*). Differently from previous research, not only this work covers serial program, but also considers and focuses on parallel program optimization. Furthermore, this work combines the use of program and system signature, which enables (*i*) the construction of very accurate performance models and (*ii*) the application of the proposed technique to different program optimization scenarios, as illustrated in Section 5.

4 Learning to Optimize Programs by Similarity

This section presents a unified approach to program and system characterization and a technique to construct effective performance models based on such characterization. The performance characterization and modeling technique presented in this work assumes the knowledge of performance values (*completion times*) for some pairs (*program*, *system*). These values are organized in a $m \times n$ matrix $\mathcal{M}$ such that each element $m_{i,j}$ of $\mathcal{M}$ represents performance of the program i on the system j.

Rows in the matrix $\mathcal{M}$ can be interpreted as program signatures, i.e., $\pi_i = [m_{i,1}, m_{i,2}, \cdots, m_{i,n}]$. Likewise, columns in the matrix $\mathcal{M}$ can be interpreted as system signatures $\sigma_j = [m_{1,j}, m_{2,j}, \cdots, m_{m,j}]$. Therefore, a program signature is the series of performance values that a program attains on different system configurations, whereas a system signature is the series of performance values that a system configuration delivers to a set of programs.

The matrix $\mathcal{M}$ is in general sparse because performance for several entries corresponding to pairs (*program*, *system*) may be missing. Two cases are considered in this work: (*i*) The first case in that $\mathcal{M}$ is dense, i.e., all the entries are known. In this case we present a technique to visualize patterns of similarity between programs and systems; (*ii*) The second case in that the sparsity of $\mathcal{M}$ is limited to just one program of interest, whose performance is unknown on a number of systems. Hence, the program signature for this program is as follows: $\imath_i = [m_{i,1}, m_{i,2}, \cdots, ?, ?, ?, \cdots, ?]$, where the question marks indicate unknown elements in the signature.

The performance modeling and prediction technique presented in this work aims to predicts performance of the program of interest on the unknown systems by similarity, i.e., with respect to performance attained by similar programs on the unknown systems. In particular, in this work, predicting by similarity means that performance prediction is based on a combination of the following two aspects in the matrix $\mathcal{M}$: (*i*) Performance that the program of interest attained on known systems; (*ii*) and Performance that other programs attained on a new system.

4.1 Program and System Similarity

The Euclidean distance is adopted in this work as similarity measure [26]. The Euclidean distance is an appropriate similarity measure as the goal of this study is to discover similarity patterns where programs attain nearly-equal performance on different system configurations and/or system configurations deliver nearly-equal performance to different programs.

For either program or system signatures, similarity analysis on a set of signatures is assessed using average-linkage hierarchical clustering [22, 26]. Hierarchical clustering organizes rows (program signatures) of $\mathcal{M}$ into a tree using the following procedure: First, for any set of m signatures, an upper-diagonal (similarity) matrix is computed by using the Euclidean distance between the signatures; Second, the similarity matrix is visited to identify the highest similarity value, i.e., the lowest distance that connects the most similar pair of signatures. A node is created from joining these two signature, and a signature expression profile is computed for the node by averaging observation for the joined elements; Third, the similarity matrix is updated with this new node replacing the two joined elements. This process is repeated $m - 1$ times until only a single element remains. Likewise, the process above is repeated for the columns (system signatures) in $\mathcal{M}$.

Given the similarity trees, the rows and columns in $\mathcal{M}$ are re-organized. Rows are permuted such that similar program signatures are adjacent. Likewise, columns are permuted such that similar system configurations are adjacent. A clustergram visualization [21] associates a heat-map to the permuted version of $\mathcal{M}$. The heat-map is composed of rectangular tiles arranged in a matrix shape where the position of each tile corresponds to the position of an element in the permuted $\mathcal{M}$. Each tile is associated to a color corresponding to the value of the element in the permuted relatively to the average value of the elements $\mathcal{M}$. Values corresponding or close to the average value are colored in black or dark shades of either green or red. Values in the matrix above the average are colored with shades of red - the higher the value is above the average, the lighter red is associated to the corresponding tile. Values in the matrix below the average are colored with shades of green - the lower the value is below the average, the lighter green is associated to the corresponding tile. Finally, the similarity trees are appended to the margins of the heat-map to compose the clustergram - refer to Figure 1. The usefulness of such a representation is that coherent patterns are represented by patches of the same gradient of colors on the heat-map. The formation of such patches is induced by the similarity structure in the signatures and may indicate a functional relationship among system signatures and programs.

4.2 Performance Modeling

In this work performance modeling relies on known machine learning algorithms for regression to predict performance for unknown pairs ($program$, $system$).

Model Components - Let us assume for a moment that the signature of a program of interest contains only a single missing entry,

$$\pi_i = [m_{i,1}, m_{i,2}, \cdots, m_{i,j-1}, ?, m_{i,j+1}, \cdots, m_{i,n}]$$

The first step of the proposed modeling technique is to construct a model to predict performance of the system at column j for the program i from the knowledge of performance of the program i on the other systems. Hence, from $\mathcal{M}$, a regression model is constructed to learn the following map $\Sigma : \mathbf{f} \rightarrow p_i$, where $\mathbf{f}$ is the feature vector of performance of a certain program on the systems $1, 2, \cdots, j-1, j+1, \cdots, n$ and its outcome is performance that a program i attains on the system j, i.e., p_j.

The missing entry in π_i is also missing in the system signature of the system i, i.e.,

$$\sigma_j = [m_{1,j}, m_{2,j}, \cdots, m_{i-1,j}, ?, m_{i+1,j}, \cdots, m_{m,j}]$$

Hence, the second step of the proposed modeling technique is to construct a model able to predict performance of the program at the row i from the knowledge of performance of other programs in the column j. Hence, from $\mathcal{M}$, another regression model is constructed to learn the following map $\Pi : \mathbf{g} \rightarrow p_j$, where $\mathbf{g}$ is the vector feature of performance of that a certain system delivers to the programs $1, 2, \cdots, i-1, i+1, \cdots, m$ and its outcome is performance that the system j with deliver to a program i, i.e., p_i.

Combined Model - A machine learning algorithm $\mathcal{A}$ is trained in the two cases above to construct two models for the maps Σ and Π. These models are referred as $\hat{F}$ and $\hat{G}$. Prediction for the missing entry in position (i, j) in the dataset $\mathcal{M}$ is performed by combining the following quantities

$$\hat{p_j} = \hat{\Sigma}\left([m_{i,1}, m_{i,2}, \cdots, m_{i,j-1}, m_{i,j+1}, \cdots, m_{i,n}]\right)$$

and

$$\hat{p_i} = \hat{\Pi}\left([m_{1,j}, m_{2,j}, \cdots, m_{i-1,j}, m_{i+1,j}, \cdots, m_{m,j}]\right)$$

It is clear now that $\hat{p_j}$ and $\hat{p_i}$ attempt to predict the same quantity, but from two different perspectives. The combined prediction is obtained using a weighted average of the quantities above, i.e.,

$$\hat{m}_{i,j} = w_j \times \hat{p_j} + w_i \times \hat{p_i}$$

In this work, the Equal Weight Average (`EWA`), i.e., the arithmetic average of the two predictions is taken. Other averaging techniques are possible and a survey of such techniques is in [33].

In this work it is always assumed that the number of missing entries is much smaller than the number of entries in the dataset $\mathcal{M}$ - several entries are available from the history of previous tests. This assumption is likely to be satisfied in practice because testing for performance of a software for different system configurations is a daily routine in industry. Hence, in the case of missing entries, the procedure above is repeated for each missing entry.

Validation Metrics - For an incomplete program signature, i.e.,

$$\pi_i = [m_{i,1}, m_{i,2}, \cdots, ?, ?, ?, \cdots, ?]$$

the modeling technique illustrated in the previous section constructs the following signature $\hat{\pi}_i = [m_{i,1}, m_{i,2}, \cdots, \hat{m}_{i,n-k}, \hat{m}_{i,n-k+1}, \hat{m}_{i,n-k+2}, \cdots \hat{m}_{i,n}]$, where $\hat{m}_{i,j}$ represent performance predictions for the program of interest on the systems $n-k, n-k+1, \cdots, n$, and $k+1$ is the number of unknown performance values.

Given the true performance values that the program of interest attains on the systems $n-k, n-k+1, \cdots, n$, i.e., $m_{i,n-k}, m_{i,n-k+1}, m_{i,n-k+2}, \cdots m_{i,n}$, and the predicted values, i.e., $\hat{m}_{i,n-k}, \hat{m}_{i,n-k+1}, \hat{m}_{i,n-k+2}, \cdots \hat{m}_{i,n}$ the evaluation of the prediction accuracy and the quality of the predicted values is determined in terms of Minimum Absolute Error (`MAE`) - which is defined as the sum of the pairwise absolute differences of components of π_i and $\hat{\pi}_i$ divided by the number of systems n. This metric measures the accuracy in terms of error and error magnitude in the predictions.

In addition, the discrepancy ϵ between the performance delivered to the program of interest by the optimal system and the average performance delivered by the predicted optimal systems in R repetitions of predictions for k missing systems is evaluated. This metric is important to assess the goodness of the proposed modeling technique at selecting an arbitrary system configurations - ultimately, program optimization happens via the selection of the system configuration that minimizes completion time.

Model Validation Procedure - Given a dense dataset $\mathcal{M}$, the following procedure based on cross-validation [25] is proposed. For each program in the set of programs the dataset is split such as a number of entries $1 \leq k \leq n-1$ is chosen at random and taken out from $\mathcal{M}$. For each missing entry a model constructed as described in Section 4.2 is build and performance for the missing entry is predicted. The random selection of k entries is repeated for a large number of times R compared to the number of entries, such that, a good coverage of the prediction ability of the proposed technique for arbitrarily missing subset of k values is evaluate. Hence, for each k, the average `MAE` and ϵ are reported.

5 Experiments

This section evaluates the proposed technique and illustrates its application to program optimization. Program optimization is the selection of a system configuration that makes program performance satisfactory (or, ideally, minimizes completion time). Even though a variation in program performance is driven by program attribute changes, e.g., a change in compiler settings, in the context of the proposed technique attribute changes represent latent information that is hidden in $\mathcal{M}$ and summarized by series of performance values.

Table 1. Dataset descriptions

Dataset name	n. of programs x n. of systems
CINT2006	13 x 4308
CFP2006	18 x 4250
OMP2001	10 x 401
OMP2012	10 x15
NAS-ICC	10 x 40

5.1 Datasets

The datasets considered in this Section and their sizes - the number of pairs $(program, system)$ - are illustrated in Table 1. The first four datasets, i.e., `CINT2006` [34], `CFP2006` [34], `OMP2001` [35], `OMP2012` [36], are obtained from the past decade - from January 2001 to December 2012 - of SPEC benchmark results - that is publicly available from SPEC website (`http://www.spec.org`). Programs in the SPEC benchmarks synthesize real life applications from different application domains - i.e., that are developed with different goals and software development constraints. In particular, programs in `SPEC CPU2006` are intended to exercise system's processor, memory subsystem and compiler. Programs in `SPEC OMP2001` and `SPEC OMP2012` benchmarks are intended to measure performance of shared memory multi-processor systems and heavily exercise the memory sub-system using parallel programs that are compliant with the OpenMP v2.x and OpenMP v3.x (`http://www.openmp.org`) specifications respectively.

In terms of system configurations, the variety of micro-architectures (e.g., UltraSparcIII, Intel Itanium, R12000 processors), and compilers (e.g., Sun, HP), is richer in the case of shared-memory multiprocessors evaluation than it is for single processor (on which most of the previous modeling techniques focus), where most benchmark records refer to Intel architectures and compilers ($\approx 91\%$ of the total records). Vice versa, UNIX/Linux operating system (Linux $\approx 48\%$, Sun $\approx 24\%$) is the choice for multi-processors evaluation. Linux, Windows and Solaris appear in single processor evaluations.

The last dataset concerns performance of the OpenMP version of the Nasa Parallel Benchmarks [37] subject to changes in compiler settings - inlining settings and AVX vectorization [38] in the `Intel ICC` compiler - and number of threads. The target architecture is Intel Sandy Bridge.

Each dataset in Table 1 is log-transformed, i.e., the natural logarithm of each entry in the dataset is taken.

5.2 Similarity Analysis

This section illustrates clustergrams for some of the datasets introduced in the previous Section. While a detailed analysis of all the families (clusters) that are formed using the clustering procedure illustrated in Section 4.1 is out of the scope of this paper, a comparative analysis of small clusters (each cluster contains ≈ 16 systems) from the red and the green areas of the clustergrams is briefly conducted.

A first cluster is extracted from the red patch (that indicates completion times above the average) on the right hand side of Figure 1. System configurations in this cluster are Intel-based and belong to the families Intel Pentium, Core and Xeon M. A second cluster is extracted from the green patch (completion times below the average) toward the right hand side of Figure 1. System configurations in this cluster are Intel-based and belong to the families Xeon E and X. Clusters group together system configurations based on micro-architectures in the same entry level. In the case of `CFP2006` the clustergram can be roughly divided in six areas - refer to Figure 2. Similarly to the case of `CINT2006`, a comparative analysis of small clusters (each cluster contains ≈ 20 systems) from the red and the green areas is briefly conducted. A first cluster is extracted from the red patch toward the right hand side of Figure 2. System configurations in this cluster are Intel-based and belong to the families Xeon E and X. A second cluster is extracted from the green patch in the center of Figure 2. System configurations in this cluster belong to the families Core-i3E/EV2 and Pentium G. Even in this case clusters group together system configurations based on micro-architectures in the same entry level, however, the comparison of these clusters among `CINT2006` and `CFP2006` shows that system configurations based on architectures whose performance are above the average for `SPEC CINT`, correspond to system configurations whose performance is below the average for `SPEC CFP`. At a larger granularity - larger clusters - differences between clusters become noticeable according to compilers' type, version and settings as well as system library levels - e.g., included as a part of different Linux distributions.

The clustergram for `SPEC OMP2001` is roughly divided in three areas of system configurations. As in the case of `SPEC CPU`, two small clusters (each cluster composed of ≈ 20 systems) from the red and the green areas are briefly analyzed. System configurations in a red cluster are based on micro-architecture families powered by Intel Xeon X. System configurations in the green cluster belong to families powered by Intel Itanium 2, R12000 and Ultra Sparc III. Limited to the records retrieved from SPEC website, the information from the two clusters above indicates that system configurations based on Intel Itanium 2 and HP compilers deliver performance above the average.

5.3 Program Optimization via Similarity

The application of the proposed technique to program optimization concerns the utilization of performance predictions to predict system configurations that minimize the completion time of a given program of interest. System selection corresponds to hardware and compiler settings selection in the cases of `SPEC CPU2006`; it corresponds to hardware, compiler and run-time environment settings in the cases of `SPEC OMP2001`, `SPEC OMP2012` and `NPB`. We use and compare Linear Regression and Support Vector Regression [24] to construct the model components Π and Σ. The combined model is indicated as (Π, Σ). Predictions results are averaged over 500 repetitions of randomly picking k system configurations from a program signature, for each program. Experimental results compare

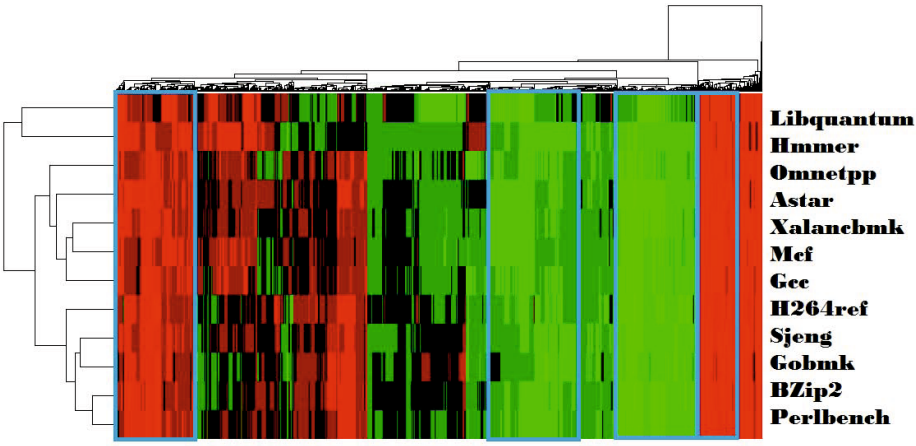


Fig. 1. Feature-agnostic Characterization of `CINT2006`

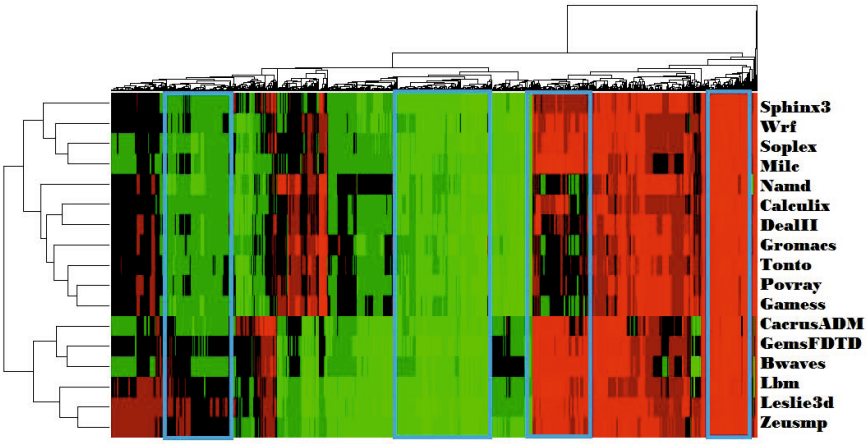


Fig. 2. Feature-agnostic Characterization of `CFP2006`

the proposed technique against a random selection of one out of the predicted system configurations.

Table 2 illustrates `MAE` and ϵ for `SPEC CINT2006`. Table 3 illustrates `MAE` and ϵ for `SPEC CFP2006`. In both cases the hybrid model reduces the discrepancy of one order of magnitude - this enforces the concept that both the information about programs and systems in a feature-agnostic characterization are important to construct effective predictors. System configuration predictions with the hybrid model $(\Pi,\ \Sigma)_{SVR}$ are almost perfect, i.e., performance prediction is less than 1% from optimal performance.

Table 2. MAE and ϵ for CINT2006

Missing Item (k)	MAE					ϵ			
	Π_{LM}	$(\Pi,\ \Sigma)_{LM}$	Π_{SVR}	$(\Pi,\ \Sigma)_{SVR}$	RND	Π_{LM}	$(\Pi,\ \Sigma)_{LM}$	Π_{SVR}	$(\Pi,\ \Sigma)_{SVR}$
1	1.00	0.43	0.36	0.20	2.52	0.0470	0.0086	0.0076	0.0008
2	1.06	0.53	0.34	0.20	1.82	0.0887	0.0179	0.0139	0.0015
3	1.05	0.54	0.35	0.20	2.05	0.1207	0.0307	0.0232	0.0020
4	1.05	0.57	0.35	0.20	2.19	0.1796	0.0411	0.0216	0.0027
5	0.98	0.53	0.35	0.20	2.19	0.1923	0.0538	0.0271	0.0030
6	1.13	0.54	0.35	0.20	2.04	0.2224	0.0640	0.0377	0.0036

Table 3. MAE and ϵ for CFP2006

Missing Item (k)	MAE					ϵ			
	Π_{LM}	$(\Pi,\ \Sigma)_{LM}$	Π_{SVR}	$(\Pi,\ \Sigma)_{SVR}$	RND	Π_{LM}	$(\Pi,\ \Sigma)_{LM}$	Π_{SVR}	$(\Pi,\ \Sigma)_{SVR}$
1	1.43	0.75	0.26	0.15	1.40	0.1330	0.0594	0.0463	0.0006
2	1.60	0.82	0.26	0.15	1.36	0.2211	0.1101	0.0521	0.0018
3	1.72	0.87	0.25	0.15	1.28	0.3099	0.1574	0.0076	0.0031
4	1.87	0.95	0.26	0.15	1.26	0.3341	0.2024	0.0541	0.0033
5	2.05	1.11	0.25	0.15	1.41	0.4061	0.2434	0.0604	0.0044
6	2.02	1.07	0.25	0.15	1.27	0.3446	0.1557	0.0171	0.0054

Table 4. MAE and ϵ for OMP2001

Missing Item (k)	MAE					ϵ			
	Π_{LM}	$(\Pi,\ \Sigma)_{LM}$	Π_{SVR}	$(\Pi,\ \Sigma)_{SVR}$	RND	Π_{LM}	$(\Pi,\ \Sigma)_{LM}$	Π_{SVR}	$(\Pi,\ \Sigma)_{SVR}$
1	1.38	0.76	0.37	0.25	2.19	0.0446	0.0012	0.0020	0.0006
2	1.39	0.76	0.37	0.25	2.19	0.0843	0.0027	0.0047	0.0009
3	1.41	0.74	0.37	0.25	2.19	0.1303	0.0040	0.0073	0.0017
4	1.42	0.75	0.36	0.25	2.19	0.1647	0.0070	0.0101	0.0033
5	1.42	0.77	0.36	0.25	2.21	0.2214	0.0077	0.0115	0.0026
6	1.43	0.77	0.36	0.25	2.20	0.2596	0.0102	0.0155	0.0028

Table 5. MAE and ϵ for OMP2012

Missing Item (k)	MAE					ϵ			
	Π_{LM}	$(\Pi,\ \Sigma)_{LM}$	Π_{SVR}	$(\Pi,\ \Sigma)_{SVR}$	RND	Π_{LM}	$(\Pi,\ \Sigma)_{LM}$	Π_{SVR}	$(\Pi,\ \Sigma)_{SVR}$
1	0.16	0.44	0.18	0.28	0.82	0.0000	0.0269	0.0000	0.0000
2	0.17	1.76	0.18	0.30	1.54	0.0000	0.1038	0.0000	0.0099
3	0.17	1.63	0.18	0.31	1.52	0.0000	0.1023	0.0000	0.0161
4	0.17	0.61	0.18	0.35	1.28	0.0000	0.1265	0.0000	0.0391
5	0.18	8.77	0.19	0.37	1.75	0.0002	0.1240	0.0000	0.0453
6	0.19	0.59	0.21	0.46	1.74	0.0015	0.0935	0.0000	0.0416

Table 4 illustrates MAE and ϵ for SPEC OMP2001. Table 5 illustrates MAE and ϵ for SPEC OMP2012. As in the case of serial benchmarks, hybrid models reduce the discrepancy of one order of magnitude. However, both hybrid and model based on programs signatures are suitable for the purposed of system selection. System configuration predictions are almost perfect, i.e., performance prediction is less than 1% from optimal performance.

Table 6 illustrates and compare average MAE and ϵ, where the average is taken across programs.

Experiments for the last dataset concern the selection of combinations of compiler and run-time settings that, in addition to the baseline optimization level O3, enable/disable vectorization, i.e., -xavx, assigns an inlining-level and

Table 6. MAE and ϵ for NAS-ICC

	MAE					ϵ			
Missing Item (k)	Π_{LM}	$(\Pi,\ \Sigma)_{LM}$	Π_{SVR}	$(\Pi,\ \Sigma)_{SVR}$	RND	Π_{LM}	$(\Pi,\ \Sigma)_{LM}$	Π_{SVR}	$(\Pi,\ \Sigma)_{SVR}$
1	4.68	3.34	0.79	0.40	3.75	0.12	0.09	0.05	0.04
2	3.84	1.87	0.79	0.40	4.30	0.16	0.12	0.06	0.04
3	4.66	1.77	1.21	0.62	3.42	0.17	0.12	0.06	0.04
4	6.57	1.78	0.81	0.41	3.38	0.16	0.13	0.06	0.04
5	2.83	1.59	0.79	0.41	3.32	0.16	0.13	0.06	0.05
6	3.33	1.36	0.96	0.49	3.41	0.15	0.12	0.06	0.05

number of parallel threads, i.e., to configure the OpenMP run-time environment. The compiler and the architecture under attention are the Intel `ICC v13.1` compiler and Intel Ivy Bridge `Core i7-3632QM` respectively. In this case, the hybrid model $(\Pi,\ \Sigma)_{SVR}$ provides superior performance in terms of predicting compiler and run-time settings with an average prediction error that is at most 5% from the optimal selection.

6 Conclusion

This work proposed a new feature-agnostinc technique for program and system characterization, performance modeling, prediction and its application to program optimization. The characterization approach to performance modeling considers the collection of completion times for some pairs (*program*, *system*). Thereby, the proposed technique is practical, because it does not require feature selection, neither does it incur in overheads due to feature retrieval. Because of the above, the proposed technique is suitable to be applied in industry settings to reduce engineering effort for optimizing software. Such an effort involves to find rapid solutions to performance studies, involving the discovery of complex systems settings, that the proposed technique can effectively address.

Experimental results show that the proposed modeling technique equally applicable to be employed in several compelling performance studies, including characterization, comparison and tuning of hardware configurations, compilers, run-time environments or any combination thereof and for both serial and parallel programs.

Acknowledgements. This work was partially supported by the NSF grant number CCF-1249449, the NSF Variability Expedition Grant number CCF-1029783 and a grant from Intel Corp.

References

1. Moore, G.E.: Cramming more components onto integrated circuits. In: Readings in Computer Architecture, pp. 56–59 (2000)
2. Borkar, S., Chien, A.A.: The future of microprocessors. Commun. ACM 54(5), 67–77 (2011)

3. Jones, C.G., Liu, R., Meyerovich, L., Asanovic, K., Bodik, R.: Parallelizing the web browser. In: Proceedings of the First USENIX Workshop on Hot Topics in Parallelism (2009)
4. Paulson, L.D.: Developers shift to dynamic programming languages. Computer 40(2), 12–15 (2007)
5. Ruparelia, N.B.: Software development lifecycle models. SIGSOFT Softw. Eng. Notes 35(3), 8–13 (2010)
6. Hall, M.W., Padua, D.A., Pingali, K.: Compiler research: the next 50 years. Commun. ACM 52(2), 60–67 (2009)
7. Levinthal, D.: Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors (2009)
8. Luk, C., 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. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (2005)
9. Piccart, B., Georges, A., Blockeel, H., Eeckhout, L.: Ranking commercial machines through data transposition. In: Proceedings of the 2011 IEEE International Symposium on Workload Characterization (2011)
10. Van Craeynest, K., Jaleel, A., Eeckhout, L., Narvaez, P., Emer, J.: Scheduling heterogeneous multi-cores through performance impact estimation (pie). In: Proceedings of the 39th Annual International Symposium on Computer Architecture (2012)
11. Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'Boyle, M.F.P., Temam, O.: Rapidly selecting good compiler optimizations using performance counters. In: Proceedings of the International Symposium on Code Generation and Optimization (2007)
12. Fursin, G., Temam, O.: Collective optimization: A practical collaborative approach. ACM Trans. Archit. Code Optim. 7(4) (December 2010)
13. Grewe, D., O'Boyle, M.F.P.: A static task partitioning approach for heterogeneous systems using opencl. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 286–305. Springer, Heidelberg (2011)
14. Moore, R.W., Childers, B.R.: Automatic generation of program affinity policies using machine learning. In: Jhala, R., De Bosschere, K. (eds.) CC 2013. LNCS, vol. 7791, pp. 184–203. Springer, Heidelberg (2013)
15. Zhang, Y., Voss, M.: Runtime empirical selection of loop schedulers on hyperthreaded smps. In: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (2005)
16. Lee, B.C., Brooks, D.M.: Accurate and efficient regression modeling for microarchitectural performance and power prediction. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (2006)
17. Dubach, C., Jones, T.M., O'Boyle, M.F.P.: Microarchitectural design space exploration using an architecture-centric approach. In: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (2007)
18. Dubach, C., Jones, T.M., Bonilla, E.V., O'Boyle, M.F.P.: A predictive model for dynamic microarchitectural adaptivity control. In: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (2010)
19. Hoste, K., Phansalkar, A., Eeckhout, L., Georges, A., John, L.K., Bosschere, K.D.: Performance prediction based on inherent program similarity. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (2006)

20. Meeuws, R., Ostadzadeh, S.A., Galuzzi, C., Sima, V.M., Nane, R., Bertels, K.: Quipu: A statistical model for predicting hardware resources. ACM Trans. Reconfigurable Technol. Syst. 6(1) (2013)
21. Eisen, M.B., Spellman, P.T., Brown, P.O., Botstein, D.: Cluster analysis and display of genome-wide expression patterns. Proc. Natl. Acad. Sci., PNAS (1998)
22. Sokal, R.R., Michener, C.D.: A statistical method for evaluating systematic relationships. University of Kansas Scientific Bulletin 28 (1958)
23. Rousseeuw, P.J., Leroy, A.M.: Robust regression and outlier detection. John Wiley & Sons, Inc. (1987)
24. Smola, A.J., Schölkopf, B.: A tutorial on support vector regression. Statistics and Computing 14(3) (August 2004)
25. Stone, M.: Cross-validatory choice and assessment of statistical predictions. Journal of the Royal Statistical Society B 36(1), 111–147 (1974)
26. Janowitz, M.F.: Ordinal and Relational Clustering. World Scientific (2010)
27. Phansalkar, A., Joshi, A., Eeckhout, L., John, L.K.: Measuring program similarity: Experiments with spec cpu benchmark suites (2005)
28. Reinders, J.: VTune Performance Analyzer Essentials: Measurement and Tuning Techniques for Software Developers. Engineer to Engineer Series. Intel Press (2005)
29. Jung, C., Rus, S., Railing, B.P., Clark, N., Pande, S.: Brainy: effective selection of data structures. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 86–97 (2011)
30. Dujmovic, J.J.: Universal benchmark suites. In: Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (1999)
31. Yi, J.J., Lilja, D.J., Hawkins, D.M.: A statistically rigorous approach for improving simulation methodology. In: Proceedings of the 9th International Symposium on High-Performance Computer Architecture (2003)
32. Park, E., Cavazos, J., Alvarez, M.A.: Using graph-based program characterization for predictive modeling. In: Proceedings of the Tenth International Symposium on Code Generation and Optimization, pp. 196–206 (2012)
33. Granger, C.W.J., Ramanathan, R.: Improved methods of combining forecasts. Journal of Forecasting 3 (1984)
34. Henning, J.L.: SPEC CPU2006 benchmark descriptions. SIGARCH Comput. Archit. News 34(4) (September 2006)
35. Aslot, V., Eigenmann, R.: Performance characteristics of the spec omp2001 benchmarks. SIGARCH Comput. Archit. News (2001)
36. Müller, M.S., et al.: SPEC OMP2012 — An Application Benchmark Suite for Parallel Systems Using OpenMP. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 223–236. Springer, Heidelberg (2012)
37. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The nas parallel benchmarks: summary and preliminary results. In: Proceedings of the Conference on Supercomputing (1991)
38. Firasta, N., Buxton, M., Jinbo, P., Nasri, K., Kuo, S.: Intel AVX: New frontiers in performance improvements and energy efficiency. Intel white paper (2008)

Scalable NIC Architecture to Support Offloading of Large Scale MPI Barrier

Shaogang Wang, Weixia Xu, Dan Wu, Zhengbin Pang, and Pingjing Lu

School of Computer, National University of Defense Technology, Changsha, China
wshaogang79@gmail.com,
{xuweixia,daisydanwu,zbpang,pingjinglu}@nudt.edu.cn

Abstract. MPI collective communication overhead dominates the communication cost for large scale parallel computers, scalability and operation latency for collective communication is critical for next generation computers. This paper proposes a fast and scalable barrier communication offload approach which supports millions of compute cores. Following our approach, the barrier operation sequence is packed by host MPI driver into the barrier "descriptor", which is pushed to the NIC (Network-Interfaces). The NIC can complete the barrier automatically following its algorithm descriptor. Our approach leverages an enhanced dissemination algorithm which is suitable for current large scale networks. We show that our approach achieves both barrier performance and scalability, especially for large scale computer system. This paper also proposes an extendable and easy-to-implement NIC architecture supporting barrier offload communication and also other communication pattern.

Keywords: barrier offload, dissemination algorithm, MPI collective communication.

1 Introduction

Collective communication (barrier, broadcast, reduce, all-to-all) is very important for scientific applications running on parallel computers, it has been shown that the collective communication overhead could take over 80% communication cost for large scale super computers[1]. The barrier operation is the most common used collective communication, its performance is critical for most MPI parallel applications. In this paper, we focus on the implementation of fast barrier for large scale parallel systems.

For next generation exascale computers, the system could have over 1 million cores, a good barrier implementation should achieve both low latency and scalability[2]. In order to achieve the overlapping of communication and computation, offload the collective communication to hardware could have obvious benefits for these systems. Present barrier offload technique, like Core-*Direct* [3], TIANHE-1A[4], uses a triggered point-to-point communication approach, Core-*Direct* software initiates multiple point-to-point communication requests to the

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 214–226, 2013.
© Springer-Verlag Berlin Heidelberg 2013

hardware and sets the request to be triggered by other messages, in this way, the whole collective communication can be handled by hardware without further software intervention.

We observe that present barrier offload method may suffer from long delay and poor scalability. The Core-*Direct* must push many working-queue-element for a single barrier operation in each node, e.g., for barrier group with 4096 nodes, each node needs to push 12 work requests to the hardware[5], we observe that this incurs long host-NIC (Network-Interface-Card) communication.

For next-generation computer networks, its point-to-point communication delay is usually high as its topology usually uses the torus mode. But each chip's network bandwidth is high due to the technology advances in serdes. In this paper, we propose a new barrier communication offload approach, which fits well for the next generation system networking.

Our approach offloads the MPI barrier operation through the following steps:

step1: each barrier node's MPI driver calculates the barrier communication sequence,i.e, the communication pattern performed by the barrier algorithm, and packs it into the barrier descriptor. All descriptor is packed following the enhanced dissemination algorithm, while the host does not perform any real communication during this step.

step2: the descriptor is sent to the NIC, our approach enables the NIC to complete the real barrier communication automatically without any further host intervention, the barrier communication is performed solely by NIC hardware.

step3: when the NIC completes the whole barrier communication, it informs the host through the host-NIC communication.

Compared with other approach, our approach only requires the host to push 1 communication descriptor to the NIC in each node, and the hardware can follow the descriptor to automatically finish the full barrier algorithm. We also give the NIC hardware architecture that smoothly supports the new barrier offload approach, due to the simple barrier engine architecture, the NIC can dedicate more hardware resources to collective communication. From simulation results, we show that our approach performs better than present barrier offload technique.

2 Barrier Offload Algorithm

The dissemination algorithm is a common used barrier method[6,7]. It supports the barrier group with arbitrary number of nodes. The basic dissemination barrier algorithm requires multiple rounds, each round sends and receives one barrier message from other node. The following round communication can be initiated only after the previous round has been finished. We observe that in large scale systems, the network system usually uses the torus topology, the network point-to-point communication delay is high as it may require many hops to reach the destination node. For these systems, the basic dissemination algorithm is not efficient. In most cases, every node takes long time waiting for the source barrier message in each round.

To efficiently hide the barrier message delay, our NIC hardware uses an enhanced K-way dissemination algorithm to offload the barrier communication. The modified algorithm is able to send and receive K messages parallel in each round. Our approach defines a new message type which is used for solely barrier communication. The new barrier message is very small, so even the NIC does not support multi-ports parallel message processing, the barrier messages can be sent and received very fast. The example 2-way dissemination algorithm is shown in 1.

$N \leftarrow$number of barrier nodes
$rank \leftarrow$ my local rank
$round \leftarrow -1$
repeat
  $round \leftarrow round + 1$
  $sendpeer1 \leftarrow rank + 3^{round}\ mod\ p$
  $sendpeer2 \leftarrow rank - (3^{round} + p)\ mod\ p$
  $recvpeer1 \leftarrow rank + 3^{round}\ mod\ p$
  $sendpeer1 \leftarrow rank - (3^{round} + p)\ mod\ p$
  send barrier msg to $sendpeer1$ with $round$ id
  send barrier msg to $sendpeer2$ with $round$ id
  receiv barrier msg from $recvpeer1$ with $round$ id
  receiv barrier msg from $recvpeer2$ with $round$ id
until $round \geq log(3, N) - 1$

Fig. 1. 2-way dissemination algorithm

We can prove that the 2-way dissemination algorithm requires total $log(3, N)$ rounds to complete the barrier for N nodes. The obvious benefit of the new algorithm is that it can greatly reduce the number of communication rounds, e.g., for the 2-way dissemination algorithm, the algorithm rounds can be reduced from $\log(2, N)$ to $\log(3, N)$, we observe that the whole barrier delay can benefit from less algorithm rounds.

3 Barrier Algorithm Descriptor

When offloading the collective communication to the NIC hardware, there is one approach that offloads the full collective algorithm to the hardware. For example, the collective optimization over Infiniband[8] uses an embedded processor to execute the algorithm. We observe that this approach will greatly complicate the

NIC design. The embedded processor is usually limited by its performance, it is far slow compared with NIC's bandwidth and the host processor's performance.

We propose a new approach that does not require the hardware to execute the full barrier algorithm, instead, the barrier's communication sequence is calculated by host's MPI driver, and the hardware simply follows the operation sequence to handle the real communication. We see that this will lead to simple hardware design, through which more hardware can be dedicated to the real collective communication.

For any node in the barrier group, we can see from the dissemination algorithm that even before real communication, each round's source and destination nodes can be statically determined. Our approach leverages each node's MPI driver to calculate the barrier sequences and pack it to the algorithm descriptor. After the descriptor is generated, it is pushed to the NIC through its host interface. The NIC hardware can follow the descriptor to automatically communicate with other nodes, after the sequence is completed, the whole barrier is completed. The host interface may be varied for different systems, for example, the command queue residents in the host memory, or the descriptor is directly written to NIC on-chip RAM through PCIE write command.

An example structure of barrier descriptor supporting 2-way dissemination algorithm is shown in figure 2. The *DType* field indicates descriptor type, along with the barrier descriptor, the system may support other collective or point-to-point communication type, the following descriptor field is interpreted according to its descriptor type. the *BID* is a system wide barrier ID, and it is predefined when the communication group is created, each node's barrier descriptor for the same barrier group uses the same barrier ID. The barrier message uses the *BID* to match the barrier descriptor for the target node. The *SendVec* field is a bit vector, its width equals the maximum barrier algorithm round, and each bit indicates whether the corresponding round should send out barrier messages to the target node. The *RC1, RC2,.. RC16* shows the number of barrier message in each algorithm round it should received. only after receiving all the source barrier messages and sending out all the barrier messages, the NIC barrier engine proceeds to the next algorithm round. *SendPeer* indicates the target node ID for each communication round.

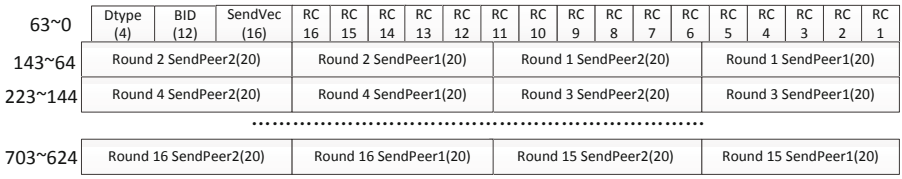


Fig. 2. Example descriptor structure for 2-way dissemination algorithm

The example descriptor is only 704 bits, but it can hold all the information to execute 16 rounds of the 2-way dissemination algorithm. In theory, the example descriptor can support the barrier group with maximum 43046721 (i.e, 3^{16}) nodes. We see that this should be easy to support the next-generation systems. The host-NIC communication cost is low as it only requires each node to push 1 descriptor to the node. The barrier descriptor is small compared with most standard point-to-point message descriptor, for example, the TIANHE-1A computer's MP (Message Passing) descriptor has 1024 bits[4].

Each node has its own barrier descriptor, and it should be generated completely through node's local information. The target and source rank id for each barrier round can be easy generated if local process rank and the barrier group size is known. But for the NIC hardware to perform the real communication, the NIC should know the target and source node id, our approach leverages the MPI driver to translate the process rank id to the physical node id. The translation process should be handled by local node without any communication, so our approach requires that the rank to node mapping information is saved in each node's memory when the MPI communicator group is created.

The barrier message takes the information on BID, RID, DestID to the destination node. Through these information, the destination node can easily determine which point it has reached for the algorithm. If the destination node has not reached the barrier, the barrier messages are saved in temporal buffer and wait for the destination node's own descriptor to be pushed to the NIC. When the NIC has completed the sequence defined in the descriptor, the group barrier communication is finished. It then informs the host that all other nodes have reached the barrier.

4 The Hardware Implementation

In this section, we propose an architecture of the NIC that easily enables barrier communication offload shown in the previous section. The NIC hardware follows a scalable structure that can be easily extended to support more communication type. Besides the NIC architecture, we also give the detail design of the barrier engine.

For our target NIC architecture, the host may generate multiple communication descriptor, we uses the virtual ring queue as the host-NIC communication interfaces[9]. The virtual ring queue is like a fifo resides in the host memory. Its write port is managed by the host or NIC, and the read port is managed by the other end. This is an efficient way for host-NIC interface which enables large data communication.

Our system incorporates various types of virtual ring queues, including the communication descriptor queue, network message receiving queue, the interrupt queue, the completion event queue. The main function for each type queue is shown in table 1.

Besides the barrier descriptor, our system supports standard communication descriptor, including the MP descriptor for short message point-to-point

Table 1. NIC virtual ring queue type and its function

communication descriptor queue	The host initiates NIC communication through this queue, each descriptor defines the information needed to perform the communication, for example, the target node, source memory address etc. Modern NIC usually supports the MP and RDMA descriptor, our system enables new collective barrier communication. The NIC fetches the descriptor from the queue through DMA.
network message queue	When NIC receives message from network, it saves the messages in this predefined queue, the host will fetches the message from it. This queue is usually used for point-to-point short message communication
interrupt queue	the NIC report detailed interrupt information through this queue, it stores various interrupt information, e.g, the interrupt type, interrupt source etc. When this queue is not empty, the NIC sends the hard interrupt command to host, when the host is on the interrupt processing task, it reads interrupt queue for further processing.
completion event queue	Our system can also uses the event to inform the host that NIC communication is completed, this queue is used when the host process is loop waiting for the completion of of the communication. NIC writes a completion packet to this queue when some communication task has finished, and the host can fetch from this queue to determine which communication request has finished.

communication, RDMA (Remote-Direct-Memory-Address) descriptor for point-to-point large data block copy. The descriptor can also be extended with other type of collective communication mode, we show that our NIC architecture can be easily extended with new communication mode.

4.1 NIC Architecture

The schematic NIC architecture is shown in figure 3, it mainly includes the following components.

DQM(Descriptor Queue Management): this module is responsible for fetching the communication descriptor from host memory. Depends on the descriptor type (MP, RDMA, Barrier), it dispatches the descriptor to the target processing engine. When host has pushed new descriptors to the queue, it writes a predefined NIC register to inform the NIC that new descriptor is valid in the queue. The NIC then fetches the descriptor.

BE(Barrier Engine): the module receives the barrier descriptor dispatched by DQM and automatically perform the barrier communication with other nodes. It follows the descriptor to perform the real barrier communication. From our experiences that the BE is very simple and requires small chip resources, and this permits more BEs to be instantiated in order to support multiple parallel barrier requests. BE can handle the unexpected messages that is received from

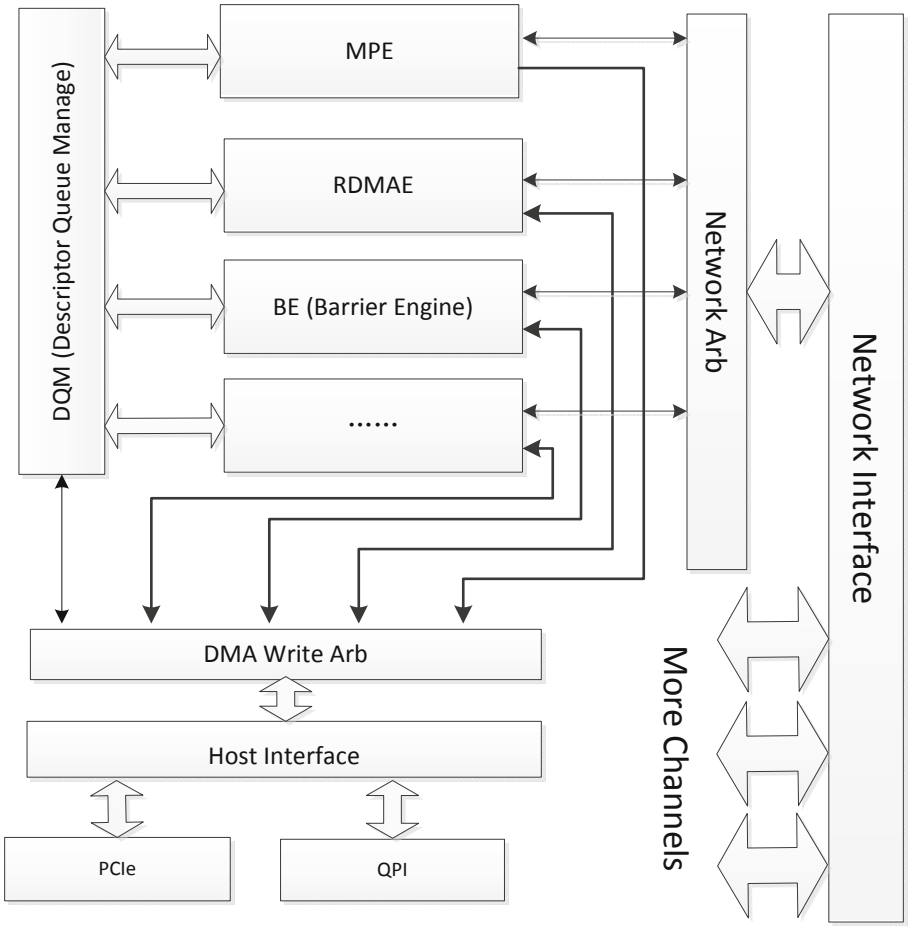


Fig. 3. The NIC architecture supporting collective communication

network, in this case, the host has not reached the barrier, so the source barrier message does not have the matching descriptor.

WA(DMA Write Arbiter): in NIC, there are many sources that want to DMA access to the host memory, for example, each communication engine may write the receiving data to host memory, the RDMA engine may read and write host memory. The WA module is responsible for arbitrating the DMA requests to host memory. Through this module, it hides the communication details of the host-NIC, which may be the PCIe, QPI depending on implementations. To support virtual address, this module is also responsible for virtual-to-physical address translating.

The NIC also incorporates other communication engines, for example, the MP, RDMA engine, which is responsible for processing MP and RDMA descriptor.

The network interfaces are the implementation of serdes. Each NIC can incorporate more channels, and each channel is independently managed by host driver. This paper focuses on the barrier communication, we will not give further details on these modules.

4.2 Barrier Engine Architecture

In this section, we show a simple BE architecture. Our system put the complex computation to the host, when the barrier communication sequence is generated, the hardware can dedicate its logic to pure communication, in this way, we keep the BE's design simple.

For the barrier communication, a complicated case is to deal with different processing arriving patterns. The timing difference between collective communication group nodes can have a significant impact on the performance of the operation, it requires the hardware to be carefully designed to avoid performance degression.

To handle this problem, BE leverages the DAMQ (Dynamically-Allocated Multi Queue)[10] to hold incoming barrier messages. The packets stored in this queue can be processed out of order. If the barrier reaches the target node who has not reached the barrier, BE saves the packets in the DAMQ buffer.

We use a simple barrier message handshake protocol to barrier between two nodes, shown in figure 4. In the barrier descriptor, if the *recvpeer* is valid, this indicates that local node should wait the source node to reach barrier; if the

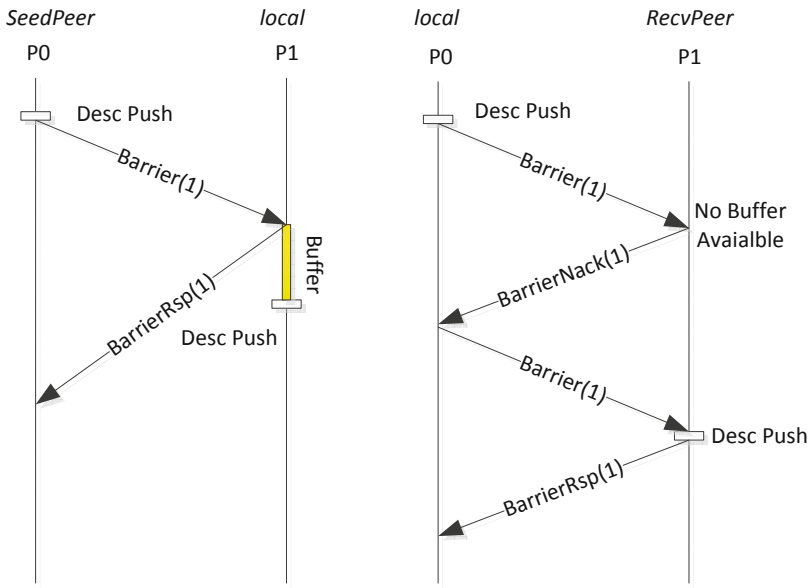


Fig. 4. Barrier handshake message flow

sendpeer is valid, this indicates that local node should tell the destination node that it reaches the barrier. If the barrier messages from *sendpeer* reaches the target node, but the target node has not reached the barrier, the barrier messages are saved in a DAMQ buffer, then the target barrier engine sends back the **BarrierRsp** message. On receiving the **BarrierRsp** message, the source node knows that the target node is sure to get the message. When DAMQ buffer is full, the target node nacks the source node with **BarrierNack** message, on receiving this message, the source node will resend the barrier message after a predefined delay.

When the barrier descriptor reaches the NIC, the barrier engine will first check its local DAMQ buffer to see if there are any previously reached barrier messages. If there are any, BE processes these messages immediately.

Note that each barrier message takes the information on the barrioud id *BID*, through which it is matched with destination node's descriptor. If the message's *BID* equals the descriptor *BID*, the source node and target node are from the same barrier group. The *BID* could be derived from the MPI communicator group id, it is required that all the nodes on the same barrier group agree on the BID. This BE engine supports multi barrier run in parallel, with each barrier uses different ID.

The structure of the barrier engine is shown in figure 5. The logic is separated by the barrier message sending (TE) and receiving module (SE). The SDQ and HDQ are descriptor queue. SDQ is resident in host memory, and HDQ is resident in on-chip RAM. The HDQ is mainly used for fast communication and fully controlled by host MPI driver. The OF is the fetching module which reads from descriptor queue and dispatches the descriptor to the barrier engine.

The SE module is responsible for receiving network *barrier* message that comes from *sendpeer*. The barrier engine saves the messages in the DAMQ buffer; To reduce hardware requirement, all barrier group messages are saved in one buffer, and the DAMQ buffer can be handled out of order. When the receiving DAMQ receives one message, it directly sends back the rsp reply to the sender. For local node, it does not need to known the barrier message is from which node, so the barrier descriptor only holds the number of messages it should

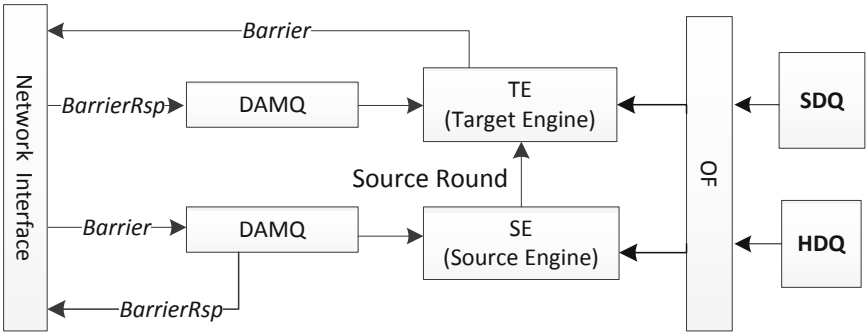


Fig. 5. Barrier engine architecture

received. The SE module uses a booking table to holds the number of source barrier messages for each round. Because each node may reach the barrier in the arbitrary sequence, the receiving messages may reaches local node out of order, so current receiving round id is *RID*, where from round 1 to *RID-1*, all barrier messages have received.

The TE module is responsible for sending barrier messages to target nodes following the sequence defined in the descriptor. For algorithm round i, barrier messages to node *sendpeer* are sent if: the SE module has received all barrier messages before round i, and the *sendpeer* is valid for round i. The barrier messages for current round are sent in pipeline before their responsive messages are received. If target node replies with nack message, the barrier message is resent after a pre-configured delay.

TE and SE are running in parallel and independently. Because the TE module needs to know current receiving round, SE module directly gives this information through module ports.

From our experience that the barrier engine is very simple, this permits more engines can be instantiated in one NIC. This enables more flexible usage for the MPI driver, for example, the MPI3.0 supports non-blocking collective communications[11], so one communication group can use more than one barrier engine to handle parallel barrier requests.

5 Experiments

We implemented the barrier engine using the SystemVerilog language, and integrated the barrier engine module into TIANHE-1 NIC's RTL model. TIANHE-1's point-to-point communication engine uses the descriptor for MP (Message-Passing) and RDMA (Remote-Direct-Memory-Access)[4], and we add the new barrier descriptor type. From our experiences that the barrier engine is easy to design, we model the barrier engine with less than 6000 SystemVerilog code lines.

The new NIC model is simulated by synopsys VCS simulator. We test the barrier latency for different sized barrier. To simulate large scale barrier groups, we designed a simplified NIC model using SystemVerilog language, the simplified model requires less simulation resources and runs more fast, yet its processing delay is similar with the real RTL model.

To simulate the network, we use a general model which route point-to-point message to the target node, the point-to-point delay is calculated based on the number of hops for the 2D torus network.

5.1 Barrier Latency

We compare the barrier communication delay between our approach and the Core-*Direct* approach, the simulation result is shown in figure 6.

We see from the simulation that when the point-to-point delay are high compared with message startup time, the barrier engine takes long time waiting for

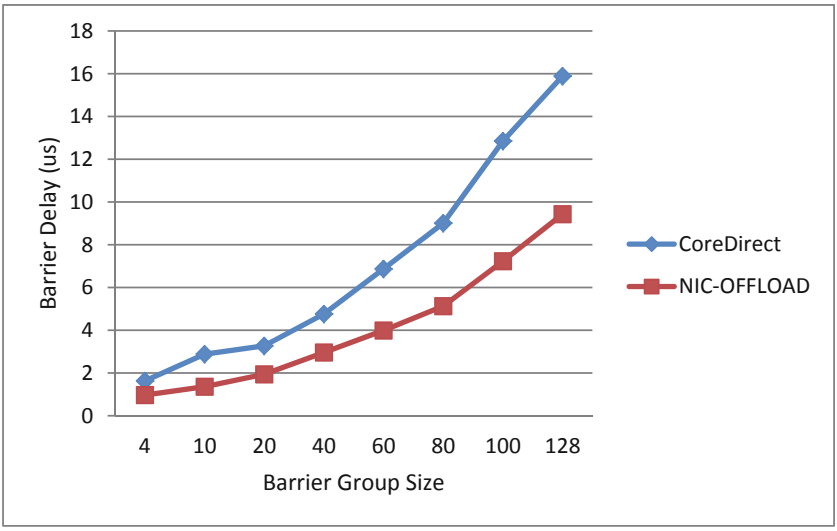


Fig. 6. Barrier communication delay

the source barrier message, this situation gets worse when point-to-point delay increases. Increasing the K can benefit from more message overlapping. suppose the point-to-point delay is l, the barrier message sending time is t_s, we see that the total barrier delay should be continually reduced if the following condition holds.

$$K * t_s < 2 * l \tag{1}$$

We also see that the reduced barrier rounds have obvious impact on total barrier delay. Our approach also shows better scalability, the speedup over Core-*Direct* approach increase as the barrier group size grows. For the Core-*Direct* approach, although the dissemination algorithm can be executed, yet the host-NIC communication cost is very high. The Core-*Direct* approach uses basic MP message to carry the barrier info, so the message startup time is higher than our approach. These limitations can greatly hurdle the barrier performance.

5.2 Barrier Delay Compared with Software Only Approach

In this section, we test the average barrier speedup over the software only approach. The software approach is simulated by modifying the timing parameters collected from the real hardware. The software only barrier delay is compared with our approach, shown in figure 7.

Our barrier offload approach gets obvious delay reduction compared with the software only approach, and shows much better performance scalability.

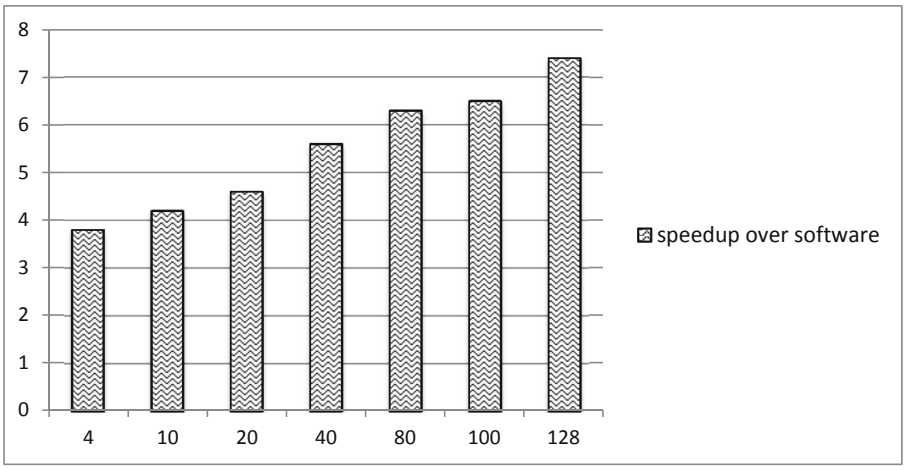


Fig. 7. NIC barrier offload speedup over software only approach

For the barrier group with 128 nodes, our NIC offload approach is 7.2x faster than the software only approach. The performance benefits come from the following reasons:

1. The software approach uses standard MP message for barrier communication, we see that the MP packet is too large for barrier communication, it incurs long NIC processing delay.
2. The host-NIC communication cost is high, it is even worse than the Core-*Direct* approach which uses the triggered point-to-point operations. For each point-to-point message, the host needs to push the MP descriptor to the NIC, and waits for the NIC's completion events.
3. The offload approach permits more communication and computation overlapping, the performance benefits may depends on the applications.

6 Conclusion

We propose a new barrier offload approach, with the new hardware-software interfaces, the barrier engine is . The NIC hardware follows the descriptor to executes the complex K-way dissemination algorithm. Simulation results show that our approach reduces barrier delay efficiently and achieves good computation and communication overlap. From our experiences, the barrier engine is easy to implement and requires less chip resources, so the NIC can dedicate more logic for real communication, this is important for next-generation super computer, where each NIC must support more processor threads.

Acknowledgement. This research is sponsored by Natural Science Foundation of China(61202124?61103014).

References

1. 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)
2. Miyazaki, H., Kusano, Y., Shinjou, N., Shoji, F., Yokokawa, M., Watanabe, T.: Overview of the K computer System. FUJITSU Sci. Tech. J. 48(3), 255–265 (2012)
3. Venkata, M.G., Graham, R.L., Ladd, J., Shamis, P.: Exploring the All-to-All Collective Optimization Space with ConnectX CORE-Direct. In: 2012 41st International Conference on Parallel Processing (ICPP), pp. 289–298. IEEE (2012)
4. Xie, M., Lu, Y., Liu, L., Cao, H., Yang, X.: Implementation and Evaluation of Network Interface and Message Passing Services for TianHe-1A Supercomputer. In: 2011 IEEE 19th Annual Symposium on High-Performance Interconnects (HOTI), pp. 78–86. IEEE (2011)
5. Hemmert, K.S., Barrett, B., Underwood, K.D.: Using triggered operations to offload collective communication operations. In: Keller, R., Gabriel, E., Resch, M., Dongarra, J. (eds.) EuroMPI 2010. LNCS, vol. 6305, pp. 249–256. Springer, Heidelberg (2010)
6. Hoefler, T., Mehlan, T., Mietke, F., Rehm, W.: Fast barrier synchronization for InfiniBand^{TM}. In: IPDPS 2006: Proceedings of the 20th International Conference on Parallel and Distributed Processing. IEEE Computer Society (April 2006)
7. Hensgen, D., Finkel, R., Manber, U.: Two algorithms for barrier synchronization. International Journal of Parallel Programming 17(1) (February 1988)
8. Mamidala, A.R.: Scalable and High Performance Collective Communication for Next Generation Multicore Infiniband Clusters. Phd Thesis (2008)
9. Sonja, F.: Hardware Support for Efficient Packet Processing. Phd Thesis, pp. 1–207 (March 2012)
10. Tamir, Y., Frazier, G.L.: Dynamically-allocated multi-queue buffers for VLSI communication switches. IEEE Transactions on Computers 41(6), 725–737 (1992)
11. Tipparaju, V., Gropp, W., Ritzdorf, H., Thakur, R., Traff, J.L.: Investigating High Performance RMA Interfaces for the MPI-3 Standard. In: 2009 International Conference on Parallel Processing (ICPP), pp. 293–300. IEEE (2009)

Hierarchical Clustering Routing Protocol Based on Optimal Load Balancing in Wireless Sensor Networks

Tianshu Wang, Gongxuan Zhang, Xichen Yang, and Ahmadreza Vajdi

Department of Computer Science, Nanjing University of Science and Technology, Nanjing 210094, China
{214591473,814169722}@qq.com, gongxuan@njust.edu.cn, ahmadreza.vajdi.cs@gmail.com

Abstract. Nowadays, Wireless Sensor Networks (WSNs) are widely used in different aspects of human life such as agriculture, health care systems, traffic engineering and so on. By improving WSN's design and applicability, still due to energy limitation in the sensors, the main concern is about the lifetime of sensors. Beside hardware aspect, establishing some efficient techniques for data sensing, processing and transition in the sensors can increase WSN's lifetime. In this paper, a new optimized routing protocolcalled OLBHC (Optimal Load Balancing Hierarchical Clustering) is designed. In contrast to some traditional methods such as LEACH, OLBHC could decrease the energy consumption in the sensors by utilizing the equalization method. In OLBHC, a Flag matrix which stores cluster head nodes' connection status is used and then an optimization algorithm is applied for selecting the best clusters in the network based on the energy consumption. The simulation results prove this proposed approach's efficiency because by applying OLBHC the average lifetime of nodes is about 160% longer than LEACH and almost all of the nodes are dead in LEACH when the first node begins to die in OLBHC.

Keywords: wireless sensor networks, equalization, clustering, hierarchical routing, matrix.

1 Introduction

Wireless sensor networks (WSNs) are defined as wireless networks consist of randomly distributed tiny nodes through self-organization [1]. These tiny nodes are integrated with sensor unit, data processing unit and communication unit. Currently wireless sensor networks have been actually used in many areas such as smart home, military reconnaissance and environmental monitoringand so on. Their development and application will give human far-reaching implications in various fields of life and production[2].

Owing tolimitations of hardware and software in the wireless sensor networks, the sensor nodes can't providea sufficient amount of energy for computing, storage and

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 227–240, 2013.
© Springer-Verlag Berlin Heidelberg 2013

communication functions,permanently [3]. Therefore, it's necessary to in-depth research about routing protocols in wireless sensor networks to optimize the distribution of nodes, increase the lifetime of a single node and eventuallyachieve the goal to extend the life cycle of the entire wireless sensor network[4].

The main function of the routing protocol is to find the optimal path of data transmission between nodes [5]. Routing protocols can be divided intoplanar routing protocolsand hierarchical routing protocols according to the network's topology, the status and functions of each node[6]. Planar routing protocolsconsist ofa sink node, and other nodes in the network can only communicate with the sink node, and they have the same status and functions. In this protocol, all of the nodes cooperate with each other to do data processing and transmission in the network[7]. Hierarchical routing protocol adopts a hierarchical management mechanism.Its main idea is dividing the whole network into a plurality of relative small collection of network nodes.Each collection is calleda cluster[8]. Normally each cluster has a node which iselected as the cluster head according to certain rules.The cluster head is used to manage or control the collection of nodes.Apart from the cluster head node, other nodes within the cluster are called member nodes.In the cluster, the head node is responsible for collecting the data which are gathered by member nodes and doing aggregation. Finally, it sends theresults to the base station or sink node.

The network which is implemented by the planar protocol is simple and has a good robustness. But sink node has excessive communication and computing pressure, leading to the rapid depletion of the battery energy,and eventually demise of sink node will result to destruction of the entire network.Hierarchical routing protocol put forward the idea of data fusion in cluster heads, reduces power consumption of sink node and decreases the transmission distance between each peer nodes. Finally, it greatly improves network stability and energy efficiency.Therefore,hierarchical routing protocol is more advantageous than planar routing protocol [9]. Shown in Figure 1 is a wireless sensor networkbased on hierarchical routing protocol.There are five clusters in the network, the sink node communicate with five cluster heads, each cluster head node does data transmission with three cluster members.

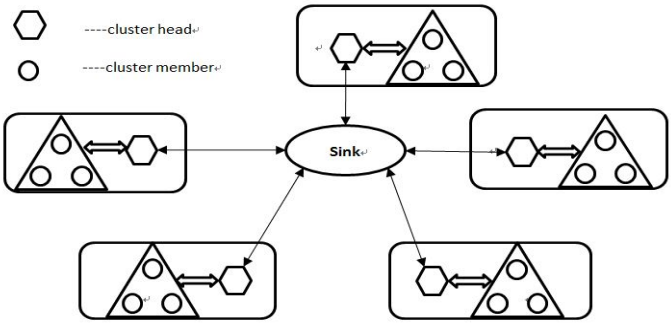


Fig. 1. Awireless sensornetworkbased on hierarchicalrouting protocol

2 Related Works and Motivation

Current hierarchical routing protocols are mainly implemented by clustering-based [10], chain-based [11] and tree-based [12] routing protocols. This paper focuses on the clustering-based hierarchical routing protocol. Heinzelman et al. [13] proposedthe LEACH.It is a low energy adaptive clustering hierarchy protocol. LEACHis the most typical clustering-based hierarchical routing protocol. The basic idea is to select the cluster head randomly in the form of round. Then energy load of the entire network will be distributed equally between all of the nodes. Therefore, it can reduce energy consumption and prolong the network's lifetime. In order to maximize network lifetime andform efficient clusters, Younis [14] proposed Hybrid Energy—Efficient Distributed(HEED) cluster protocol. This protocol is based on the primary and secondary parameters which extend the lifetime of network by distributing energy to the entire network. The HEED's foundation istime synchronization of the entire network.The protocoldivides all the nodes into several levels according to remained energy level by proportion P. Higher-level nodesconsider themselves as the cluster heads and announce other low-level nodes to join thecorresponding clusters by sending a broadcast message. Based on the dataaggregationscheme, Chatterjea et al.[15] proposed the CLUDDA (Clustered Diffusionwith Dynamic Data Aggregation) which performs data aggregation in unfamiliar environmentsby query definitions. The query definitions describe the operations that need to beperformed on the data components in order to generate aproper response. CLUDDA combines directed diffusion [16] with clusteringduring the initial phase of query propagation. Theclustering approach ensures that only cluster heads areinvolved in message transmissions. This techniqueconserves energy, since the regular nodes remain silentunless they are capable of servicing a request.

However, there are some disadvantages in the previous works:

- LEACH is a self-organized protocol to form clusters in the network. The cluster headsare randomly generated. Hence, nodes may be repeatedly selected as the cluster heads. In this case the nodesthat are far away from cluster heads consume moreenergy.The phenomenon of not balanced energy consumption in the network will appear.
- Energy load imbalances [17] between the cluster heads. In fact, energy consumption ofcluster heads is mainly reflected in two aspects: (1) the cluster heads which are far from the sink node require more power to send data, thus expend more energy; (2) there are more nodes in the cluster, so the cluster head consumes more energy.
- The scatter node is defined as a cluster head node which doesn't haveany cluster member. As a result when the base station is far from the scatter nodes more power is necessary to send the data. In addition, when the scatter nodes arecommunicating separately, data aggregation can't be done completely, leading to increasing the total energy consumption of the network.
- Nowadays security is one of the main concerns in WSNs. However, most of the previous works did not consider security beyond energy efficiency.

To overcome these disadvantages, firstly, through clustering algorithm, the hierarchical clustering routing protocol based onoptimalloadbalancing in Wireless Sensor Networks (OLBHC) which is proposed in this paper, produces cluster heads according to certain rules.By this manner, the same node will not berepeatedly selected as the cluster head node and then avoid the imbalance of energy consumption. In addition, each cluster head in the network is allocated, roughly, the same number of nodes, thereby, eliminating the imbalance of load. And then,as cluster heads are allocated, the protocolmakes a full coverage of network, avoiding the generation of scatter nodes.Finally, to consider security issue, encryption and decryption algorithms are applied to the experiments. The cluster member nodes encrypt the sensing date by encryption algorithm; after receiving the data from cluster members, cluster heads decrypt the packets and encrypt all data and then send encrypted data to the sink node.

The clustering algorithm in the OLBHC is an equalization algorithm that expends node energy evenly, reduce network energy consumption and prolong the demise time of a single node so that extendlife cycle of the entire network.

3 Protocol Design

3.1 Prior Knowledge

3.1.1 LEACH Based on the "Round" Thinking

In this paper, OLBHCprotocol is based onthe "round" thinking of LEACH. In fact,in each round the most appropriate nodes are selected as the cluster heads.After determining the structure of each cluster, the cluster members send perception information to the cluster headsand consume energy.After receiving information and perform corresponding data aggregation algorithm, cluster heads consume a certain amount of energy. The concrete steps of LEACH based on the "round" thinking are as follows:

- Setting a threshold T(n), then each node selects a number, randomly, from 0 to 1.If the random numberofthe current round is smaller than T(n), then the node is selected as the cluster head node.
- When a node is selected as the cluster head, it will take the initiative to send a broadcast message to the network. Other nodes selectthe appropriate cluster head which has the strongest signal according to the information they receive, and jointhe cluster where the cluster head is in.
- Finally, the member nodes, in their own time slot, send data to the cluster heads.The cluster headsfuse collected data integration, and send the results to the sink node. Afterwards, LEACH reselects the cluster heads andenters a new round of circulation.

3.1.2 Advanced-Node

The proposed hierarchical clustering routing protocol based onoptimalloadbalancing in Wireless Sensor Networks (OLBHC) in this paper adds the concept of

"advanced-node". The so-called advanced-node is a node that has more initialenergy than ordinary nodes in the network.

3.1.3 The *Flag* Matrix

OLBHC introduces the concept of *Flag* matrix.The matrix is defined as follows: the rows and columns represent the ID of corresponding nodes in the network.Forexample,as shown in the Table 1, when the cell which corresponds to row *i* and column *j* is equal to 1, then node *i* and node *j* are interconnected.

Table 1. The *Flag* matrix

	1	2	3	4	5	6	7	8	9
1	0	0	0	**1**	0	0	0	0	**1**
2	0	0	0	0	0	0	0	0	0
3	0	**1**	0	0	0	0	**1**	0	0
4	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	**1**	**1**	0	0	0
9	0	0	0	0	0	0	0	0	0

Table 1 shows a 9*9 Flag matrix in which the number 9 corresponds to the total number of nodes in the network.In the matrix when cell $(i, j) = 1,(i, j=1,\ldots,9)$,then node *i* is the cluster head and node *j* is the cluster member. Therefore, the correspondingwireless sensor network of theFlag matrix hastotally three clusters, the cluster heads are nodes 1,3 and 8, separately. Figure 2 identifies all of the members in the corresponding clusters.

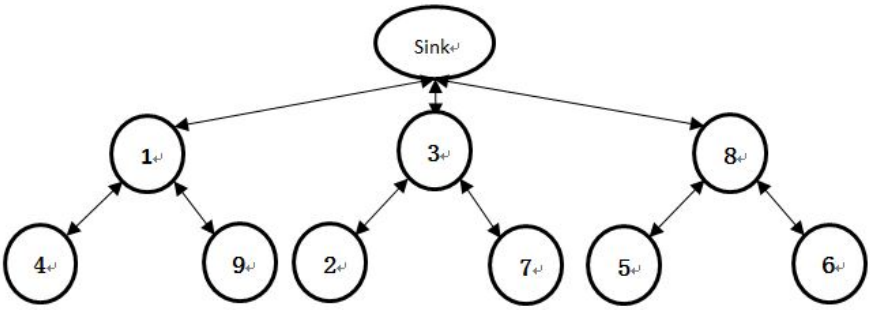


Fig. 2. The proposed topology based on the OLBHC protocol

3.1.4 The *Flagdis* Matrix

The OLBHC protocol also defines a matrix named *Flagdis* which is corresponding to *Flag* matrix.The matrix stores the distance between each peer of nodes. As shown in Table2, the value of the first row and the fourth column is 3.05, it means that the distance between the node 1 and node 4 is 3.05.

Table 2. The *Flagdis* matrix

	1	2	3	4	5	6	7	8	9
1	0	0	0	**3.05**	0	0	0	0	**4.16**
2	0	0	0	0	0	0	0	0	0
3	0	**2.93**	0	0	0	0	**6.77**	0	0
4	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	**8.96**	**9.71**	0	0	0
9	0	0	0	0	0	0	0	0	0

3.2 OLBHC Overall Process

The overall process of the hierarchical clustering routing protocol based onoptimal-loadbalancing in Wireless Sensor Networks (OLBHC) is shown in Figure 3. In this flow chart, *r* represents the number of rounds.During every round, OLBHCadopts a new clustering algorithm which is presented in this paper to select cluster heads and determine the appropriate members of cluster heads.Unlike the LEACH protocol, the determinationof cluster members is not based on the strength of signal, but on the principle of balanced allocation, the specific methodwill be covered in the following.

After selecting the cluster heads and determining the primary corresponding cluster members, there may be a long distance between a node and its cluster head, so cluster head of the nodeswill be selected again. After determining the structure of cluster, the OLBHCcycles this processtentimes and electsthe optimal result. Afterselecting the optimal structure for the clusters, cluster headswill sendinformation to theirs cluster-members, in the process, the cluster heads will consume a certain amount of energy.Likewise, after receiving a message bycluster members,sending their perception information to the cluster head nodes also need to consume energy.The cluster head-saggregate the information which are collected, and thensend the results to the sink node.This process alsoneeds to expendsome energy.

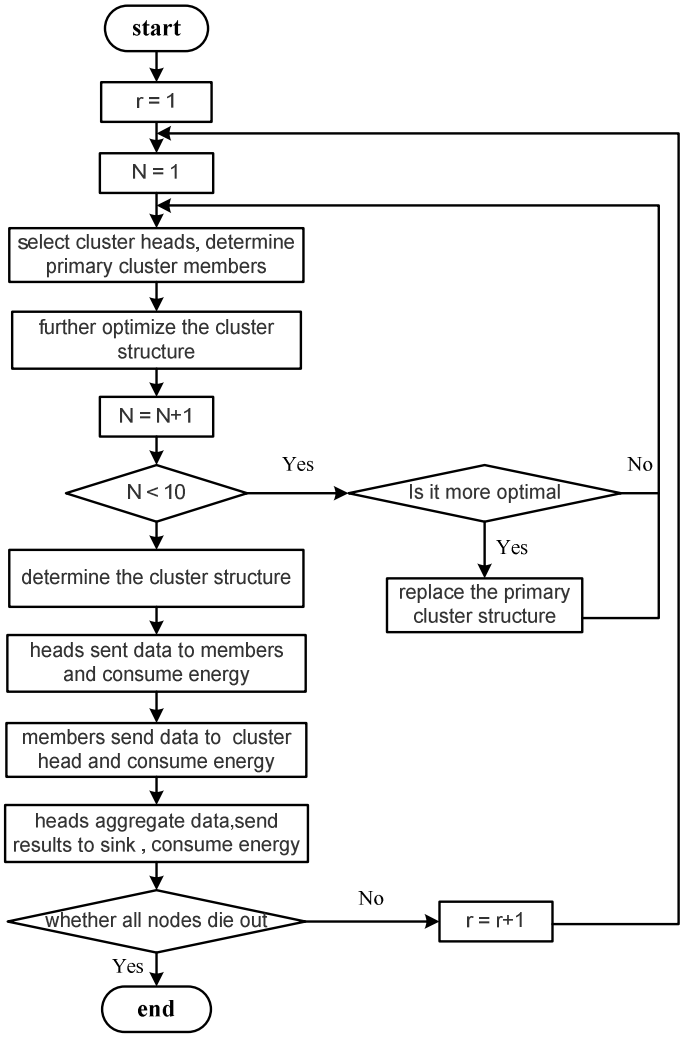


Fig. 3. The flow chart of OLBHC protocol

3.2.1 Cluster Nodes Selection

The experiments show that if in each round the advanced-nodes are all selected as the cluster heads, they will be destroyed in a short time which decreases the lifetime of the network. Traditional LEACH protocol does not take full advantage of the characteristics of advanced-nodes.Based on the experiments,after destruction of all ordinary nodes, finally, only a few advanced-nodes will work.

Suppose the total number of nodes in the wireless sensor network is n, the number of the demise nodes is d, and the percentage of advanced-nodesis a. Select x nodes

from the advanced-nodes, randomly, as part of cluster heads.Select *y* nodes from the remaining ordinary nodes, randomly, as the remaining part of cluster heads. For determining the cluster head nodes,*x* and *y*could be obtained by the following equations:

$$x = \lceil a \times n \div 4 \rceil \tag{1}$$

$$y = \lceil \sqrt{n - d} - x \rceil \tag{2}$$

3.2.2 Determining the Corresponding Cluster Members

For determining the corresponding cluster members, there are several steps as following:

- **Step1:** Calculate the Euclidean distance [18] between all of the *n* nodes in the network.Suppose that there are two nodes A and B, their coordinates are (*X1, Y1*) and (*X2, Y2*), respectively.The Euclidean distance *EU* between A and B is shown in Equation 3. Then all of *EU* distances will be stored in the Matrix *D*.

$$EU = \sqrt{(X1 - X2)^2 - (Y1 - Y2)^2} \tag{3}$$

- **Step2:** Calculate the *CN* value for defining the number of members for the clusters in the network.

$$CN = \lfloor (n - d) \div (\lceil \sqrt{n - d} \rceil) \rfloor - 1 \tag{4}$$

- **Step3:** Based on the *CN* value from previous step, construct the *Flag* matrix and *Flagdis* matrix from matrix *D*. Suppose thatthe node *i* is the cluster headin matrix *D*. Therefore, row *i*will be looked at to identify cluster members. If *CN* = *m*, it means there are *m* cluster members in the row *i*. Then, choose *m* minimum values in form of cell(*i,j*). Finally, in the *Flag* matrix and *Flagdis* matrix, set the corresponding cell(*i, j*) to 1 and the cell(*i, j*)'s value, respectively.
- **Step4:** Denote the dead nodes. Then set all of the cells in the corresponding columns in the *Flag* matrix and *Flagdis* matrix to 0. So because in the corresponding rows, the number of cluster members is lower than *CN*value, based on the difference with *CN* value in each row, choose suitable number of members which have the lowest values among other candidate members.
- **Step5:** Add the values of all the columns in the *Flag* matrix. Then, make two new matrices called *Judge0* and *Judge1*. Add the columns that their sum values are greater than 1 into *Judge1* matrix, and add the columns that their sum values are equal to 0 into *Judge0* matrix. *Judge1* matrix stores the nodes which areconnected with multiple cluster heads, and *Judge0* matrixstores the nodes which are not connected to any node.

- **Step6**: Empty *Judge0* matrix and *Judge1* matrix through a certain method. The specific processing method can be ascertained from the following code:

```
Empty Judge0 and Judge1 Matrixes Method
G1Node = Judge1(1);
G1CCol = FIND(Flag(:, G1Node) == 1);
MAXValue = Max(D(G1CCol(:), G1Node));
MAXCol = FIND(D(:, G1Node) == MAXValue);
Flag(MAXCol, G1Node) = 0;
Flagdis(MAXCol, G1Node) = 0;
MINValue = MIN(D(MAXCol, Judge0(:)));
MINRow = FIND(D(MINValue, :) == MINValue);
Flag(MAXCol, MINRow) = 1;
Flagdis(MAXCol, MINRow) = D(MAXCol, MINRow);
```

Fig. 4. Empty *Judge0* and *Judge1* Matrixes Method. Firstly, select the first node of *Judge1* anddetermine cluster nodes connected with*G1Node*. Secondly, determine the longest distance and determine the longest cluster head connected with *G1Node*. Thirdly, set the mark whether *G1Node* and *MAXCol* are connected to 0 and determine the shortest distance connected with *MAXCol* cluster head. Finally, determine the shortest cluster member connected with *MAXCol* and set the mark whether *MAXCol* and*MINRow* are connectedto 0.

3.2.3 Further Optimization in the Structure of Clusters

For further optimization inthe structure of clusters, there are several steps as following:

- **Step1**: Find the maximum valuein the *Flagdis* matrix. And then identify its corresponding row and column, call r and c. This step can determine the longest path in the network which consumes the largest amount of energy. After that, set the corresponding cell in the *Flag* matrix and *Flagdis* matrix to 0.
- **Step2:** Calculate the minimum value in the column c of matrix D in corresponding to the rows which are cluster heads and based on its position , (i, j), in the matrix D, set corresponding cell in the *Flag* matrix and *Flagdis* matrix to 1 and its value, respectively.
- **Step3:** Calculate the maximum value in the row$i$ of *Flagdis* matrix. Then, write down the position of the value and set it to 0 in the same position in*Flag* matrix and *Flagdis* matrix.
- **Step4:** Calculate the sum of all values in the *Flagdis* matrix.If the sum value is greater than the previous summed value in the *Flagdis* matrix, then replace the *Flag* matrix and *Flagdis* matrix with previous ones.
- **Step5:** Skip to step (2) until the cycle index is$\lceil\sqrt{n-d}\rceil$.

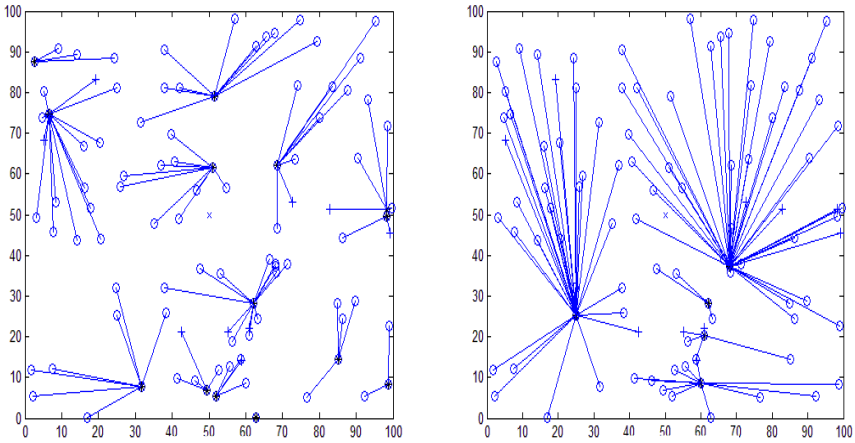
- **Step6:** Skip to step (1) until the cycle index is*RN*,*RN* is determined by equation (5).

$$RN = \lfloor (n - d) \times 2 \div 3 \rfloor \tag{5}$$

4 Experimental Results

This research used MATLAB to simulate the proposed protocol (OLBHC) in the wireless sensor network. The simulation environment is an area of 100*100m^{2}where 100 nodes are randomly distributed. Totally there are 100 nodes which 10 of them are advanced-nodes with 1J energy and other ordinary nodes have 0.5J energy. The coordinate of sink node is (50, 50).

Figure 5 shows the cluster distribution in one of the roundsin traditional LEACH protocol. The empty circles represent ordinary nodes, and the plus signs represent advanced-nodes.As it is obvious, in the Figure 5-a there is a cluster which has 15 members, in contrast to some clusters with only one member. In the figure 5-b, there are two clusters with more than 30 nodes, versus, some clusters which have only three members.In the figure 5-b, there are two clusters with more than 30 nodes, versus, some clusters which have only three members. Figure 5 shows that the traditional LEACH protocol has a lot of randomness when it selects the cluster head nodes. The cluster head node will consume too much energy, when it selects too many nodes as cluster heads. The individual cluster head node will connect excess cluster members when the protocol selects too little nodes as cluster nodes, in this case there will be a substantial of energy consumed.



(a) (a)

Fig. 5. Thecluster distribution of traditional LEACH protocol

Figure 6 shows the cluster distribution of the hierarchical clustering routing protocol based on optimal load balancing in Wireless Sensor Networks (OLBHC) which is proposedin this paper. The number of cluster heads is calculated according to$\lceil\sqrt{n-d}\rceil$. In figure 6, the total number of nodes is 100. The number of dead nodes is 0.Therefore, there are10 clusters. Every cluster contains 10 nodes, equally. Each node connects to the cluster head with an optimal distance. Therefore, the number of clusters selected in each round will not change frequently. Consequently, the OLBHC protocol ensuresthat all of the nodes in the network consume a reasonable amount of energy.

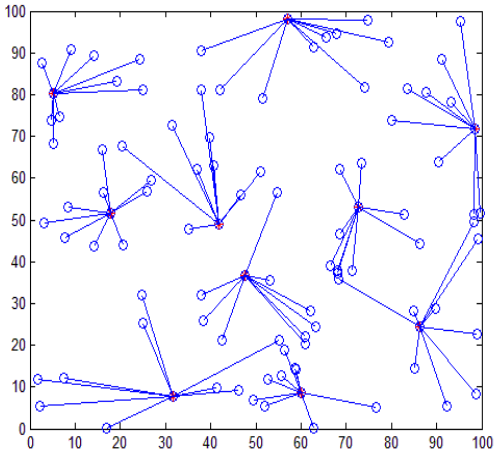


Fig. 6. The cluster distribution of OLBHC protocol

Figure 7 shows the number of active nodes intraditional LEAHC protocol and the hierarchical clustering routing protocol based on optimal load balancing in Wireless Sensor Networks (OLBHC). The experiment results indicate that nodes begin to diefrom 942throundin the traditional LEACH protocol. There were only 15 activenodes at 1420thround. At the same time, there are nodes that begin to die at 1520thround and there are 20 nodes survival until 1700th round. From 942th to 1520th round, in the traditional LEACH protocol nodes start to die dramatically, in contrast to the OLBHC protocol proposed in this paper which increases the efficiency of network immensely. The simulation results show that the average lifetime of the nodes in OLBCH is about 160%longer than LEACH and then OLBHC greatly extends the lifetime of network.

Figure 8 shows the number of active nodes intraditional LEAHC protocol without applying any security mechanism, OLBHC protocol which applied security mechanism and OLBHC protocol without applying security mechanism. The result shows that the lifetime of the OLBHC protocol which uses the security mechanism is much longer than traditional LEACH protocol.

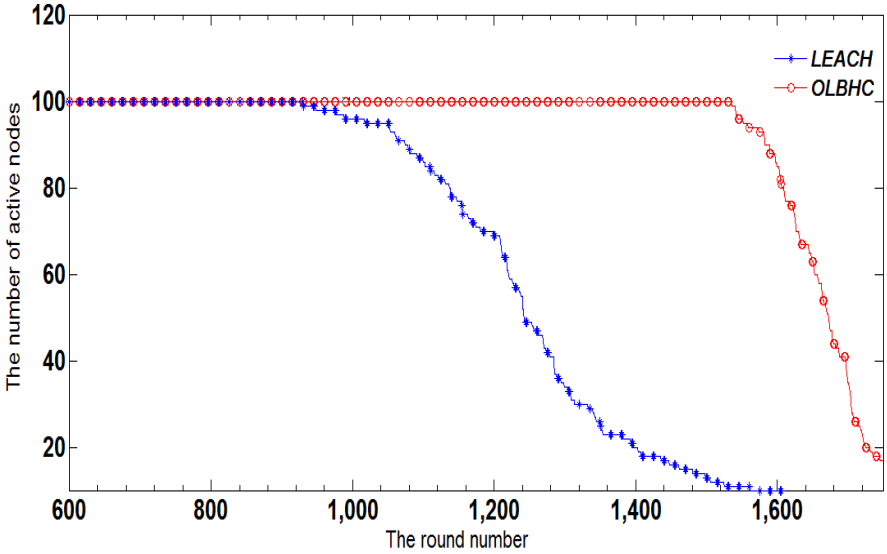


Fig. 7. The number of active nodes in LEACH protocol and OLBHC protocol

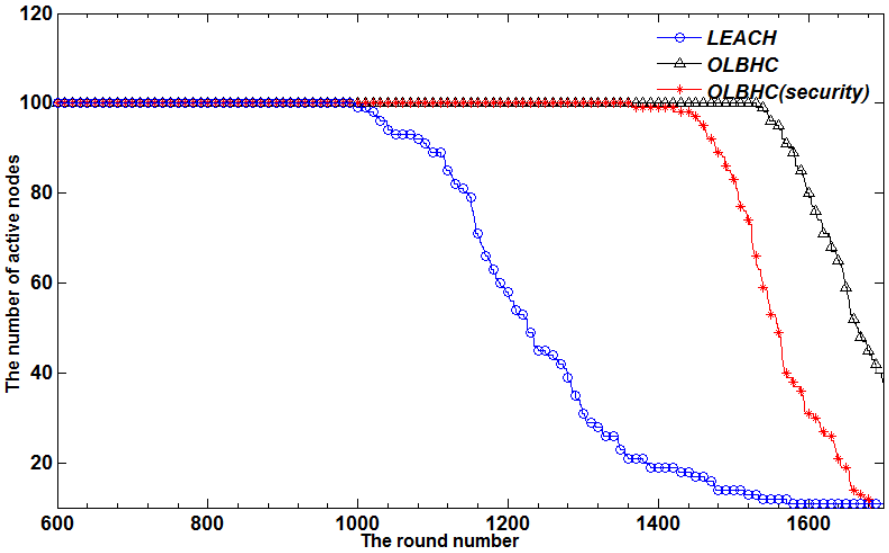


Fig. 8. The number of active nodes in LEACH, OLBHC and OLBHC (security)

5 Conclusion and Future Work

In this paper, a new routing protocol is proposed in the Wireless Sensor Network called OLBHC. It aims at extending the lifetime of the network by providing a new clustering algorithm. In fact, this proposed protocol selects a fixed number of nodes as the cluster heads at each round and then uses a Flag matrix representation for connection status of nodes. In contrast to traditional methods, it applies the idea of equalization to construct appropriate clusters for avoiding the generation of scatters.The simulation results show this protocol's efficiency because it can extend the lifetime of the network about 160 percent more than other traditional methods.

The OLBHC protocol proposed in this paper has been greatly improved with respect to traditional protocols, but still it selects the cluster heads randomly. The future work should consider a certain algorithm to select more appropriate cluster heads according to the physical location and the residual energy of the nodes in the network. In addition, bandwidth and fault tolerance are very important issues in WSNs. Next research will be about developing a new protocol to consider them.

Acknowledgement. This paper is supported by the National Natural Science Foundation funded project No.61272420 and the Natural Science Foundation of Jiangsu Province project No.BK2011022.

References

1. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., et al.: A survey on sensor networks. IEEE Communications Magazine (2002)
2. Yick, J., Mukherjee, B., Ghosal, D.: Wireless sensor network survey. Computer Networks (2008)
3. Sudha Anil Kumar, G., Manimaran, G., Wang, Z.: Energy-aware scheduling with probabilistic deadline constraints in wireless networks. Ad Hoc Networks (2009)
4. Ordonez, F., Krishnamachari, B.: Optimal information extraction in energy-limited wireless sensor networks. IEEE Journal on Selected Areas in Communications 22(6), 1121–1129 (2004)
5. Liu, J., Chen, J., Kuo, Y.: Multipath Routing protocol for networks lifetime maximization in ad-hoc networks. In: The 5th International Conference on Wireless Communications, Networking and Mobile Computing (2009)
6. Mostafizur, M., Mozumdar, R., Nan, G., et al.: An efficient data aggregation algorithm for cluster-based sensor network. Journal of Networks 4(7), 598–606 (2009)
7. Rajagopalan, R., Varshney, P.K.: Data aggregation techniques in sensor networks: A survey. IEEE Comm. Surveys & Tutorials 8, 48–63 (2006)
8. Chen, Y.P., Liestman, A.L., Liu, J.: A hierarchical energy-efficient framework for data aggregation in wireless sensor networks. IEEE Transactions on Vehicular Technology (2006)
9. Sekine, M., Sezaki, K.: Hierarchical aggression of distributed data for sensor networks. In: TENCON (2004)
10. Yu, M., Leung, K.K., Malvankar, A.: A dynamic clustering and energy efficient routing technique for sensor networks. IEEE Transactions on Wireless Communications (2007)

11. Yu, J.D., Kim, K.T., Jung, B.Y., Youn, H.Y.: An Energy Efficient Chain-Based Clustering Routing Protocol for Wireless Sensor Networks. In: Advanced Information Networking and Applications Workshops, pp. 383–384 (2009)
12. Kim, S.H., Park, P.G.: An Efficient Tree-Based Tag Anti-Collision Protocol for RFID Systems. IEEE Communications Letters (2007)
13. Heinzelman, W.R., Chandrakasan, A., Balakrishnan, H.: Energy-efficient communication protocol for wireless microsensor networks. In: Proceedings of the 33rd Hawaii International Conference on System Sciences, pp. 1–10 (2000)
14. Younis, O., Fahmy, S.: HEED: A Hybrid, Energy-Efficient, Distributed Clustering Approach for Ad Hoc Sensor networks. IEEE Trans. Mobile Computing 3(4), 366–379 (2004)
15. Chatterjea, S., Havinga, P.: A Dynamic Data Aggregation Scheme For Wireless Sensor Networks. In: Proc. Program for Research on Integrated Systems and Circuits, Veldhoven, The Netherlands (November 2003)
16. Intanagonwiwat, C., Govindan, R., Estrin, D., et al.: Directed diffusion for wireless sensor networking. IEEE ACM Transactions on Networking (2003)
17. Zhang, Z., Zhang, X.: Research of Improved Clustering Routing Algorithm Based on Load Balance in Wireless Sensor Networks. In: IET International Communication Conference on Wireless Mobile and Computing, pp. 661–664 (2009)
18. Yang, J., Xu, M., Xu, B.: Multipath routing algorithm based on parametric probability for wireless sensor networks. In: Proceedings of the 2nd International Conference on Intelligent Computation Technology and Automation (2009)

An Efficient Parallel Mechanism for Highly-Debuggable Multicore Simulator

Xiaochun Ye, Dongrui Fan, Da Wang, Fenglong Song,
Hao Zhang, and Zhimin Tang

State Key Laboratory of Computer Architecture, Institute of Computing Technology
Chinese Academy of Sciences, Beijing, China
{yexiaochun,fandr,wangda,songfenglong,zhanghao,tang}@ict.ac.cn

Abstract. Fast multicore simulators are extremely useful in evaluating design alternatives and enabling early software development. Among the state-of-the-art multicore simulators, Simics is a very popular used one both in academia and industry. It has a powerful debugging system, and also provides an accelerator to support multithreaded or distributed simulation. However, this kind of parallel mechanism mainly aims at speed up distributed systems. It is not suitable for the shared-memory multicore systems which are much more commonly used. In this paper, we propose a novel parallel mechanism to improve the simulation speed of shared-memory multicore systems. More importantly, our approach is compatible with other optimizations and exist debugging systems used in Simics. Experimental results show that our parallel approach achieves an average speedup of 9.6× (up to 12.2×) when running SPLASH-2 kernel on a 16-core host machine.

Keywords: Parallel Simulator, Multicore Simulator, Debuggable.

1 Introduction

Instruction-Set Simulator (ISS) is an essential system-level design tool for both architecture design and software development. With the simulators, software developers can validate their programs without the need of real target machines and thus significantly shorten design turnaround time. Also, the transparency and debuggability of the simulator can help developers quickly converge on design problems. After years of development, the single-core IIS is close to be ideal, i.e. accurate and fast. However, multicore architecture gradually replaces single-core architecture due to the advance in semiconductor manufacturing process. As a result, multicore simulator is becoming more and more important.

Among the state-of-the-art multicore simulators, Simics [1] is a proven, stable, and efficient simulator framework, which has been very extensively used in both academia and industry. It has a large library of simulated components available for users to use to construct system modeling, which reduces the time to build a model of a particular system. Examples include the X86, ARM, SPARC V8/Leon2, PowerPC, and other processors. There are also models of common system components such as serial

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 241–253, 2013.
© Springer-Verlag Berlin Heidelberg 2013

ports, SDRAM memories, FLASH memories, and system controllers. Various buses and networks are available, including MIL-STD 1553, ARINC 429, and Ethernet. Simulated buses and networks can be connected to real-world networks for mixed simulations involving both virtual and physical nodes.

Simics also offers a convenient software test and debug environment, with features like instant stop of execution, scripted test cases, full determinism, checkpoint and restart, as well as reverse debugging and reverse execution. As a simulator, Simics offers complete control over the simulated computer systems, enabling precise injection of faults and scripting of tests. Faults can be injected both inside devices, memories, and processors, and on the buses and networks connecting them. Simics includes a full Python language interpreter, enabling powerful scripting.

Simics fully supports simulation setups containing multiple cores, multiple chips, or multiple boards, all within a single Simics process running on a single workstation. Different types and speeds of processors can be freely mixed if desired, with no limitations. It is possible to simulate both a central computer unit and the payload computers of a satellite system within a single simulation, if that is desired.

In order to improve the simulation speed, Simics provide a tool named *Simics Accelerator* [17]. It takes advantage of multicore hosts to improve the execution speed of large target system simulations. However, Multithreading is only supported for models marked as thread-safe, using a configuration partitioned in simulation *cells* [18]. A *cell* is a group of configuration objects that are tightly connected to each other, and they should never share memory. Thus in shard memory multicore target systems, core and memory have to be put into different *cells* in order to use multithreading technique. However, memory access operations usually occupy more than 1/3 of the total instructions. This leads to a very inefficient speedup because the interactions between core and memory are very frequent. The multithreaded version may execute even slower than the sequential one. Obviously, the parallel mechanism of Simics Accelerator mainly aims at speed up distributed target systems. It is not suitable for the shared-memory multicore systems which are much more commonly used.

The inefficiency we just mentioned heavily affects the use of Simics, which is a highly-debuggable multicore simulator. It is crucial for Simics to efficiently simulate a shared-memory multicore target system in parallel. However, there are several challenges to do that: First, the module scheduling mechanism is controlled by Simics and cannot be changed because Simics is not open source. Second, the powerful debugging system is very useful and should be keep untouched when parallelizing Simics. Third, Simics already provides a lot of optimization methods to improve the simulation speed, and our parallel mechanism should be compatible with them.

In this paper, we propose a parallel mechanism to further improve the performance of shared-memory multicore simulation in Simics. The main contributions of this paper include:

(1) We present a novel parallel mechanism for shared-memory multicore system simulation in Simics, and get a considerable performance improvement compared with traditional method. As far as we know, this is the first effective solution for this problem.

(2) We support the debugging methods provided by Simics when simulating multicore system in parallel. This is very critical for software developers.

(3) Our parallel approach is compatible with the accelerate methods, such as distributed simulation and binary translation techniques already provided by Simics.

The rest of this paper is organized as follows: We briefly review the scheduling mechanism of Simics, introduce our parallel approach, and discuss its compatibility with existing optimizations and debugging systems in Section 2. Section 3 presents the experimental results. Section 4 discusses related work. Finally, we summarize the paper and present future work in Section 5.

2 Parallel Mechanism

This section discusses the design of our parallel mechanism based on Simics simulator. We first describe the scheduling method of Simics. Then we present our parallel mechanism for shared-memory multicore target systems. Finally, we discuss its compatibility with the debugging system and other optimizations provided by Simics.

2.1 Simics Scheduling Method

In Simics, there is a scheduler responsible to schedule all the modules in the target system. As shown in Figure 1, only those modules that have implemented *execute interface* will be scheduled [22]. Simics schedules them in a round-robin style, and keep them within a time window defined by the *time_quantum* attribute which can be set by the users. After being simulated for *time_quantum* cycles, each target module gives up the control and returns to Simics scheduler through event callback mechanisms. Having a *time_quantum* that is significantly larger than a single cycle allows for faster simulation since optimizations such as binary translation can then be employed.

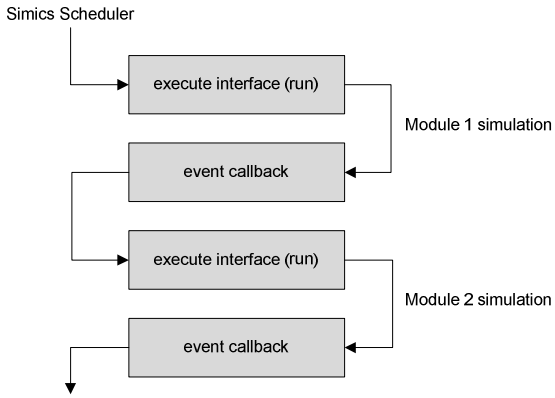


Fig. 1. Scheduling mechanism of Simics

When simulating a shared-memory multicore system, usually Simics puts them into one *cell*, and lets each processor implement its own *execute interface*, so that every core can be scheduled periodically. However, only systems belong to different *cells*

can be parallelized by the multithreaded mechanism of Simics. This kind of shared-memory multicore systems can only be simulated sequentially, leading to a very slow simulation speed.

In order to use Simics Accelerator to parallelize shared-memory target systems, one possible way is to map the core and memory modules into different *cells*. However, this leads to a very inefficient speedup because the interactions between core and memory are very frequent. As a result, we present a new parallel mechanism to effectively parallelize shared-memory multicore target systems.

2.2 Parallel Mechanism

Because Simics is not open source, we cannot modify the module scheduling mechanism. When parallelizing multicore system in Simics, we have to follow its scheduling rule. As we mentioned before, only the modules that have implemented *execute interface* can be seen and will be scheduled by Simics.

Based on this observation, we defined a novel parallel mechanism based on the implementation of a new module named *shadow_core*. Figure 2 shows the parallel mechanism of our method. As we can see, shadow_core is the key of the parallelization. Next we will give some detailed description of shadow_core.

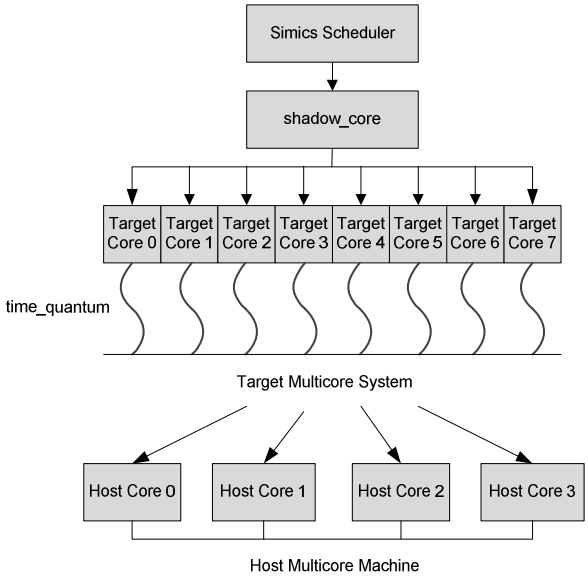


Fig. 2. The parallel scheduling mechanism of using shadow_core

Shadow_core is actually a virtual core which is used to cheat Simics scheduler. If we implement all the target cores of shared memory system with *execute interface*, they will be scheduled sequentially by Simics. In order to avoid this, we define a shadow_core with *execute interface,* and all the target cores being simulated do not implement *execute interface.* In this way, only the shadow_core can be seen and

scheduled by Simics. As for the target cores, they will be scheduled by the shadow_core so that they can be simulated in parallel.

Note that the shadow_core is not a real core, so it does not have to define all the context of the target core. Actually, only the interfaces which are necessary for the interaction between Simics and the target cores need to be implemented.

It is obvious that all the target cores are invisible to Simics, and the shadow_core is responsible for the target core scheduling and synchronization. Roughly, there are three main tasks for the shadow_core. First, shadow_core needs to create threads and schedule the target cores. Second, when running in parallel, the shadow_core should keep the right synchronization of all the target cores. Third, the shadow_core needs to pass the commands, which are mainly debugging commands, from Simics to the actual target cores. In the next three subsections we will discuss how the shadow_core support core scheduling, synchronization, and debugging mechanism.

Core Scheduling

As we mentioned before, only the shadow_core is registered and seen by Simics. After Simics starts, it will schedule the shadow_core and keep it running for *time_quantum* cycle before switching to next registered module.

When the shadow_core starts to run, it creates many new threads, and each thread is responsible for simulating one or several target cores. For example, if the target system has 8 cores and host system has 4 physical cores, the shadow_core can create 4 threads and each thread simulates 2 target cores. The relation between host threads and target cores can be defined by the users. They can map the target cores into any thread at will.

Note that, this scheduling process is totally invisible to Simics. Depending on the different implementation, the shadow_core can use different scheduling methods

Synchronization

In Simics system, users define *time_quantum* attribute to indicate the synchronization frequency of different cores. Each core will keep running for *time_quantum* cycles before synchronize with other cores. However, only the shadow_core can be seen by Simics in our parallel mechanism, and all the target cores are invisible. Thus Simics only makes sure that shadow_core will be scheduled to run *time_quantum* cycles and then synchronize with other modules which have registered *execute interface*. It is not possible for Simics to synchronize all the invisible target cores automatically. In order to solve this problem, the shadow_core must make sure that all the target cores being simulated in different threads synchronize with each other every *time_quantum* cycles. Thus each time the shadow_core is scheduled by Simics, it lets each target core keep running *time_quantum* cycles and then wait for the next scheduling. By defaut, the shadow_core uses a barrier-based synchronization mechanism.

As we mentioned before, Simics has to map cores and memory modules into different *cells* in order to run them in parallel. However, in our method, cores and memory modules are mapped into the same *cell*. In the shared-memory multicore system, all simulated cores share a global, cache-coherent memory. All the cores access the memory module directly. This poses a challenge to scalable parallel simulation. Specifically, processor exports a set of atomic instructions executed atomically, which are usually used to implement synchronization primitives. An efficient simulation of

atomic instructions is critical to parallel simulation. In our system, we perform an identical translation that maps the simulated atomic instruction to one on the host architecture, this solution works well in most situations. As for some special ISAs, we use the lightweight memory transactions method presented by COREMU [2].

Debugging Support

While simulating various of target systems, Simics also provides a powerful debugger system. For example, Simics can run user binaries directly, allowing the user to set breakpoints, inspect state, single step, etc. Some difficult bugs are easier to find using various esoteric breakpoint types. In Simics you can set breakpoints on memory access, time, instruction types, device accesses, output in the console, etc.

More importantly, Simics is the first general-purpose development tool for reversible execution of arbitrary software running on arbitrary systems. With the reversible execution and debugging features of Simics, it is now possible to execute a program in reverse and step backwards through the code. This makes debugging very efficient.

For the convenience of debugging, we make these debugging methods still work correctly under our parallel mechanism. However, it is a great challenge to do that because Simics cannot see the target cores at all. In order to solve this problem, we make a context mapping between shadow_core and target core. Because the shadow_core can be seen by Simics, if we can map the context of the target core into shadow_core exactly, then Simics is actually controlling the target core when it schedules the shadow_core. In order to deceive Simics, it is critical to identify all the context information that needed by the debugging system. We find that it is mainly composed of the interfaces and event queues. Table 1 shows the context we have identified.

Table 1. The context of target core

Context	Description
time queue	the cycle queue of target cores, responsible for the management of events related to time. For example, it is used to record the events send by Simics or external devices.
step queue	the step queue of target cores, responsible for the management of events related to step. It is used in the step debugging.
iface_context_query	the context query interface of target cores, responsible for the query operations related to the context of cores.
iface_context_trigger	the context trigger interface of target cores.
obj_context	the object of target core's context.
iface_phys_mem_page	the physical memory page interface of target cores.
iface_phys_mem_space	the physical memory space interface of target cores.
iface_phys_trigger	the physical memory trigger interface of target cores, responsible for the trigger operations related to the physical memory.
obj_phys_mem	the object of target core's physical memory.
cache_page	the cache pages of target cores.

In our parallel simulator, we provide the user a new command interface to select the target core number being debugged. When shadow_core creates threads and schedules the target cores, it checks whether the target core is the debugged one. If so, shadow_core will copy all the context pointers of the debugged core into itself, so that the Simics can control the running of the debugged core.

Note that although shadow_core does not perform any real instructions, it is still necessary to define a context that is the same as real target core. After that, we can make a consistent map between the shadow_core and target core.

Because multicore system debugging is more complicated than single core debugging, we extend the basic debugging mechanism of Simics in shadow_core. For example, we add new commands to select the target core being debugged.

2.3 Compatibility with Other Optimizations

In Simics, there are many kinds of optimization strategies, such as multithreaded/distributed simulation of different *cells*, binary translation technique, etc. Our parallel mechanism is also compatible with them.

When simulating a cluster target system, we can parallelize it in two hierarchies. First, we use Simics Accelerator to distribute multiple machines into different host cores or host machines. Then we can further map the multiple cores within a target machine into different host threads.

Another powerful simulation optimization technique, binary translation, can also be used combined with our parallel mechanism. Binary translation is a very popular technique to improve the simulation speed of single core. Thus in our parallel simulator, when the target cores are simulated in different host cores, we can use binary translation to further improve its simulation speed.

We currently have not finished the work of combing our parallel simulator with these optimization methods. The evaluation of it will be our future work.

3 Experimental Results

In this section, experimental platform and benchmarks are introduced first. Then we evaluate the performance and scalability in section 3.2. Finally, we demonstrate its compatibility with exist debugging methods of Simics.

3.1 Experimental Setup

The configuration of the systems and the benchmarks we use are listed in Table 2. The host platform has 4 quad-core AMD Opteron Processor 8347 HE running at 1.9GHz and 64GB of DRAM. The host OS is linux-2.6.9 (x86_64). The benchmark we used is SPLASH-2 [17] kernel and a hybrid benchmark. We evaluate the performance of a 16-core target system on Simics 4.2.

Table 2. Evaluation system configuration

	Parameters	Configuration
Host system	CPU type	16-core (4 quad-core AMD Opteron 8347 SMP)
	Memory size	64GB
	OS	Linux 2.6.9, x86_64
	Compiler	Gcc 4.1.1
	Simics	Version 4.2
Benchmark	SPLASH-2	Fft, lu, cholesky, radix
	Hybrid benchmark	Multithreading program consists of all the kernel computation of splash2 benchmarks
Target system	CPU & memory	16-core processor, ARM ISA, Shared 4GB memory

3.2 Performance Evaluation

Uniprocessor Scheduling Overhead

In our method, we introduce a new scheduling layer to parallelize the simulation of multicore target systems. Obviously this scheduling layer will lead to some additional overhead. In order to evaluate the scheduling overhead, we set only one core in the target system.

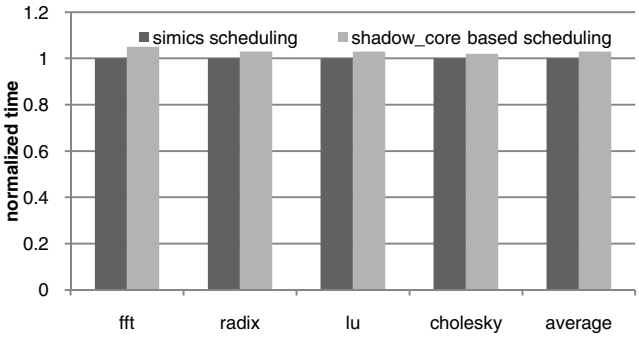


Fig. 3. Normalized uniprocessor scheduling overhead

Figure 3 depicts the relative performance overhead to native Simics scheduling for splash-2 benchmark suite. As shown in the figure, our scheduling method incurs negligible performance overhead compared to Simics scheduling, within 5% for all the benchmarks.

The additional overhead we introduced mainly comes from the thread creation overhead and context mapping of target core. However, as the simulated target cores run longer, the incurred thread creation overhead is negligible. As for the context mapping, it happens only when we change the debugged target core, which means the target system is under debug mode, and not running at full speed. Compared with the very slow manual interactive speed, the context mapping overhead is also negligible.

Speedup and Scalability

When simulating multicore target system in parallel, the value of *time_quantum* is a key factor, which determines the frequency of synchronization between different

threads, thereby affects the final performance. Normally, a large *time_quantum* reduces the frequency of scheduling and synchronization, thus improving the performance. However, larger *time_quantum* means longer synchronization period, which usually incurs larger accuracy errors. Note that the functional correctness is still guaranteed as long as the benchmarks running on the simulator are synchronized correctly. In this paper, we only evaluate the speedup and scalability of splash-2 kernels with the *time_quantums* which are not more than 1000.

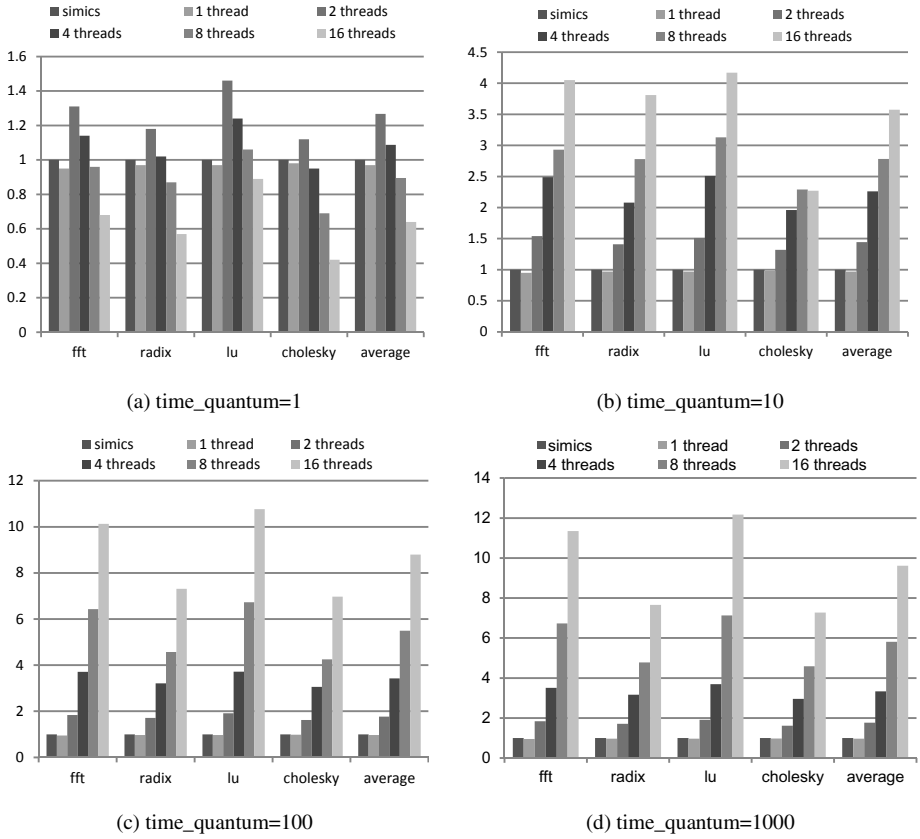


Fig. 4. Speedup of splash-2 kernels across different host thread numbers when simulating 16 target cores

Figure 4 shows the speedup results of simulating a 16-core target system with splash-2 benchmarks running on it. The number of threads used in the host machine varies from 1 to 16, and the *time_quantum* used in the parallel mechanism varies from 1 to 1000. Note that the performance of 1 thread in our method is a little worse than the original performance of sequential scheduling in Simics. This is because of the additional scheduling overhead introduced by our method.

We can see that the value of *time_quantum* affects the final parallel performance significantly. Small *time_quantum* leads to an unsatisfactory speedup because of the frequent synchronization operations. Figure 4(a) shows that the performance even gets worse when using more host threads. However, when the *time_quantum* is larger than 100, our method produces a good scalability. For example, when the *time_quantum* equals to 1000, we can see that most of the benchmarks get linear speedup. The average speedup of 16-thread is 9.6×.

Normally the speedup is in negative relation with the percent of thread synchronization in the applications. From figure 4, we can see that *cholesky* gives a relative worse speedup because it has more synchronization operations. However, the average performance is still quite noticeable. The results show that our parallel mechanism can produce a considerable speedup for typical multithreading applications, and it is a very attractive option for achieving fast multicore processor simulation.

3.3 Debugging Function Verification

To verify the compatibility with exist debugging mechanism of Simics, we make several case studies by using our parallel simulator to diagnose and debug the bugs in user applications. The following four debugging functions are intensively tested.

Target Core Selection: We randomly set different core as target debugged core during our test, and each core has been selected at least 10 times.

Core Information Display: We test all the display interfaces provided by each target core. Each target core's inner information, such as register value, can be correctly displayed.

Breakpoint Operations: Similar to the core information display, we test all the breakpoint interfaces provided by the target cores, including adding, removing, enabling, disabling breakpoints. All the functions work well.

Step Execution: With the former debugging mechanisms and step execution, we quickly locate the function and execution context causing the bug.

4 Related Work

Simulation is an important technique to explore new computer architectures ranging from micro-processors to parallel computers. A variety of different simulators have already existed. Due to the space constraint, we only give a brief introduction of multicore simulators.

GEMS [11] is a very popular multicore simulator. It uses Simics for functional modeling plus their own models for memory systems and core interactions. However, GEMS is a sequential simulator, and the simulation speed is not fast. Similar to GEMS, GEM5 [12] is also a sequential simulator, with the difference of using M5 [16] instead of Simics for functional modeling.

There is also much existing research in the field of parallel simulation of multicore target systems. The Wisconsin Wind Tunnel [6] is one of the earliest parallel simulators. It requires applications to use an explicit interface for shared memory and only runs on CM-5 machines, making it impractical for modern usage.

P-Mambo [20] is a multithreaded implementation of IBM's full-system simulator Mambo. It uses a user-level thread-scheduler to adapt multithreaded execution. However, the scale of the P-Mambo simulation target is only 4 cores.

COREMU [2] clusters multiple mature sequential simulators (QEMU in their working prototype) using a thin library layer, hence decouples the complexity of supporting parallel emulation from building an optimizing sequential emulator.

COTSon [8] is a system-level simulator for modeling clusters of multicore CPUs, networking and I/O. It uses AMD's SimNow! [5] for functional modeling. It also uses Parallel Discrete Event Simulation (PDES) [15] to synchronize different COTSon node instances. The sequential instruction stream coming out of SimNow! is demultiplexed into separate threads before timing simulation.

Penry [9] presents techniques to perform automated simulator parallelization and hardware integration for CMP structural models by generating the simulator from a concurrent, structural model of the CMP.

Graphite [4] provides user-level parallel functional simulation using a multi-machine distributed method, which provides a good scalability. However, it is based on Pin tools [21], and does not support full system simulation.

SlackSim [10] accelerates the parallel simulation of CMPs by relaxing the tight synchronization enforced between simulation threads in cycle-by-cycle (cycle accurate) simulation. It allows all threads to run freely as long as their local clocks remain within a specified window.

Hornet [19] is a highly configurable, cycle accurate network-on-chip simulator, which can scale multicores and their on-chip networks to thousand core levels. Hornet's parallelized simulation engine that can scale well with the number of physical cores in the processor. It also allows the user to obtain even more speed via loose synchronization.

Transformer [13] is an extensible, cycle-accurate loosely-coupled full-system multicore simulator. It leverages an architecture-independent interface between function mode (FM) and timing mode (TM) and uses a lightweight scheme to detect and recover from execution divergence between FM and TM. To improve performance, it parallelizes FM and TM by putting them into two threads. Different with our method, parallelized transformer can use only two threads and the performance improvement is limited.

Wu [14] proposes a new distributed scheduling mechanism for a parallel compiled Multi-Core Instruction-Set Simulator (MCISS). The distributed scheduling with his prediction method shortens the waiting time an ISS spends on synchronization.

Different with the multicore simulators we mentioned above, Simics is a commercial simulation framework and has many advantages. For example, it supports more kinds of target platforms, provides powerful debugging system and accelerator. These features make Simics succeed both in academia and industry. However, Simics does

not provide an effective parallel mechanism for shared-memory multicore target systems, which heavily affects the use of this highly-debuggable multicore simulator.

In this paper, we propose a novel parallel mechanism to parallelize shared-memory multicore system simulation in Simics. As far as we know, this is the first effective solution for this problem.

5 Conclusion and Future Work

In this paper, we propose a novel parallel mechanism to improve the simulation speed of shared-memory multicore systems in Simics. Instead of scheduling the target cores by Simics, we define a new module named shadow_core to do that. Thus in our parallel simulator, shadow_core is responsible for the multi-threading scheduling of the simulated cores. More importantly, our approach is compatible with other optimizations and exist debugging systems used in Simics. When using a 16-core host machine, our experiments showed that it achieved an average speedup of 9.6× for SPLASH-2 kernels.

There are mainly two directions in our future work. First, we plan to combine our approach with other optimizations such as binary translation to further improve the simulation speed. Second, we will extend our parallel mechanism to support larger scale multicore/many-core systems.

Acknowledgment . This work is in part supported by the National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302501, the National Science Foundation for Distinguished Young Scholars of China under Grant No. 60925009, the Foundation for Innovative Research Groups of the National Natural Science Foundation of China under Grant No. 60921002, the Beijing science and technology plans under Grant No.2010B058 and the National Natural Science Foundation of China under Grant No.(61173007, 61100013, 61100015, 61204047 and 61202059), the National High-Tech Research & Development Program of China（2012AA010303）

References

[1] Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A full system simulation platform. Computer 35, 50–58 (2002)

[2] Wang, Z., Liu, R., Chen, Y., Wu, X., Chen, H., Zhang, W., Zang, B.: Coremu: A scalable and portable parallel full-system emulator. In: PPoPP 2011, pp. 213–222 (2011)

[3] Mukherjee, S.S., Reinhardt, S., Falsafi, B., Litzkow, M., Huss-Lederman, S., Hill, M.D., Larus, J.R., Wood, D.A.: Wisconsin Wind Tunnel II: A Fast, Portable Parallel Architecture Simulator. IEEE Concurrency 8(4), 12–20 (2000)

[4] Miller, J., Kasture, H., Kurian, G., Gruenwald III, C., Beckmann, N., Celio, C., Eastep, J., Agarwal, A.: Graphite: A Distributed Parallel Simulator for Multicores. In: Proc. HPCA (2010)

[5] Bedicheck, R.: SimNow: Fast platform simulation purely in software. In: Hot Chips 16 (August 2004)
[6] Reinhardt, S.K., Hill, M.D., Larus, J.R., Lebeck, A.R., Lewis, J.C., Wood, D.A.: The wisconsin wind tunnel: virtual prototyping of parallel computers. In: SIGMETRICS 1993: Proc. of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp. 48–60 (1993)
[7] Over, A., Clarke, B., Strazdins, P.E.: A comparison of two approaches to parallel simulation of multiprocessors. In: Proc. ISPASS, pp. 12–22 (2007)
[8] Argollo, E., Falcón, A., Faraboschi, P., Monchiero, M., Ortega, D.: Cotson: infrastructure for full system simulation. SIGOPS Oper. Syst. Rev. 43(1), 52–61 (2009)
[9] Penry, D.A., Fay, D., Hodgdon, D., Wells, R., Schelle, G., August, D.I., Connors, D.: Exploiting parallelism and structure to accelerate the simulation of chip multi-processors. In: HPCA 2006: The Twelfth International Symposium on High-Performance Computer Architecture, pp. 29–40 (February 2006)
[10] Chen, J., Annavaram, M., Dubois, M.: SlackSim: A Platform for Parallel Simulations of CMPs on CMPs. SIGARCH Comput. Archit. News 37(2), 20–29 (2009)
[11] Mauer, C.J., Hill, M.D., Wood, D.A.: Full-system timing-first simulation. SIGMETRICS Perform. Eval. Rev. 30, 108–116 (2002)
[12] Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. Computer Architecture News (2011)
[13] Fang, Z., Min, Q., Zhou, K., Lu, Y., Hu, Y., Zhang, W., Chen, H., Li, J., Zang, B.: Transformer: A Functional-Driven Cycle-Accurate Multicore Simulator. In: DAC 2012, pp. 106–111 (2012)
[14] Wu, M., Wang, P., Fu, C., Tsay, R.: A High-Parallelism Distributed Scheduling Mechanism for Multi-Core Instruction-Set Simulation. In: DAC 2011, pp. 339–344 (2011)
[15] Fujimoto, R.M.: Parallel discrete event simulation. Commun. ACM 33(10), 30–53 (1990)
[16] 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 26, 52–60 (2006)
[17] Engblom, J.: Simics Accelerator. VIRTUTECH White Paper (2009)
[18] Modeling your system in Simics, version 4.2. Virtutech Inc. (2010)
[19] Lis, M., Ren, P., Cho, M., Shim, K., Fletcher, C., Khan, O., Devadas, S.: Scalable, accurate multicore simulation in the 1000-core era. In: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (April 2011)
[20] Wang, K., Zhang, Y., Wang, H., Shen, X.: Parallelization of IBM mambo system simulator in functional modes. SIGOPS Oper. Syst. Rev. 42(1), 71–76 (2008)
[21] 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. In: PLDI 2005, pp. 190–200 (June 2005)
[22] Virtutech. Simics Processor API. Version 4.2

Data Access Type Aware Replacement Policy for Cache Clustering Organization of Chip Multiprocessors

Chongmin Li, Dongsheng Wang, Haixia Wang, Guohong Li, and Yibo Xue

Tsinghua National Laboratory for Information Science and Technology,
Department of Computer Science & Technology, Tsinghua University,
Beijing 100084, China
{liismn,wds,hx-wang,yiboxue}@tsinghua.edu.cn,
liguohong2012@163.com

Abstract. Chip multiprocessors (CMPs) are becoming the trend of mainstream computing platforms. The design of an efficient on-chip memory hierarchy is one of the key challenges in computer architecture. Tiled architecture and non-uniform cache architecture (NUCA) is commonly adopted in modern CMPs. Previous efforts on cache replacement policy usually assume an unified last-level cache or running multiprogrammed workloads. However, few researches focus on the replacement policy of cache clustering scheme running parallel workloads. Cache clustering scheme can improve the system performance on parallel performance, which is a tradeoff between shared cache organization and private cache organization which adopts cache replication. In cache clustering scheme, cache blocks in last-level cache can be subdivided into eight types.

In this work we propose *Data access Type Aware Replacement Policy* (DTARP) for cache clustering organization, DTARP classifies data blocks in last-level cache into different access types, and designs the insertion and the victim selection policies according to different data access types based on traditional LRU policy. The global shared data will be kept in last-level cache longer than before. Simulation results show that DTARP can improve the system performance of cluster scheme using LRU policy by 10.9% on average.

1 Introduction

With the advance of semiconductor technology, chip multiprocessors (CMPs) are becoming the trend of mainstream computing platforms. Intel® has recently demonstrated a 48-core cloud chip prototype [2]. The design of an efficient on-chip memory hierarchy is one of the key challenges in computer architecture. Tiled architecture is commonly adopted in modern CMPs because of its low design complexity as well as high scalability. Non-uniform cache architecture (NUCA) [3] has pioneered the effort of addressing the scalability problem of traditional uniform cache architecture in large CMPs. In NUCA, the access latency to a cache block is the sum of the static bank access latency and the variable on-chip communication latency to different cache banks.

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 254–268, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Traditional last level cache organizations of CMPs are either shared or private. Shared scheme has advantage of high capacity utilization, while suffers from long average on-chip access latency. The main advantage of the private scheme is the proximity of data to the requestor cores, but private scheme suffers poor capacity utilization. Cache clustering scheme is a tradeoff between shared scheme and private scheme, where several nearby tiles are clustered together into a region. Cache blocks inside last-level cache is shared by all the cores within the region. Cache clustering scheme is a technique of cache replication, which replicates data from home node to a L2 bank close to the requestor cores and thus reduces the following access latency. Fig. 1 gives the performance of cluster scheme over shared scheme in a 64-core CMP with 16×4 clusters, cluster scheme gets about 48.9% higher performance than shared scheme on average. Both schemes adopt traditional LRU replacement policy.

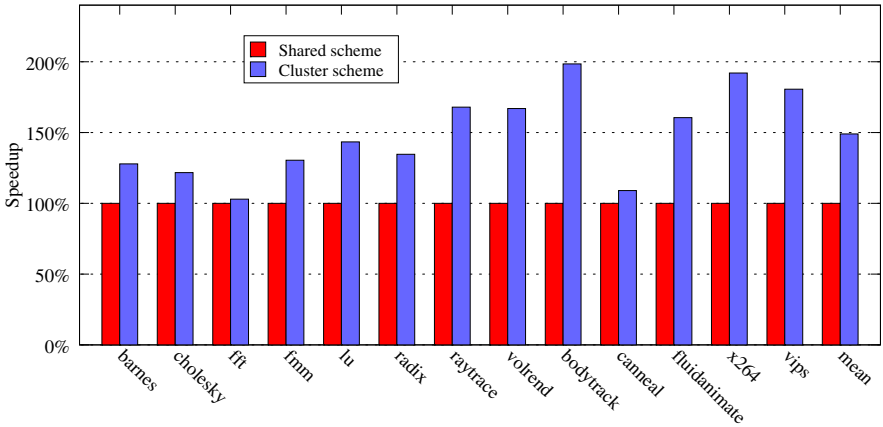


Fig. 1. Performance of cluster scheme

Cache replacement policy choose a victim cache block to evict when there has a need to fetch a new data block to the cache. Recent representing researches include DIP [4], TADIP [5] and RRIP [6], However, these techniques assumed a unified last-level cache or running multiprogrammed workloads, where a core access will suffer a stable latency. Traditional LRU policy treats each cache request as the same. However, for parallel workloads, such as scientific computing, different type cache blocks in L2 cache get different access probability, especially for cluster scheme. Unfortunately, few researches focus on the replacement policy of cache clustering scheme running parallel workloads.

In this work we propose *Data access Type Aware Replacement Policy* (DTARP) for cache clustering organization, DTARP classifies data blocks in last-level cache into different access types, and designs the insertion and the victim selection policy according to different data access types based on traditional LRU policy. The global shared data will be kept in last-level cache longer than before. Simulation

results show that DTARP can improve the system performance of cluster scheme using LRU policy by 10.9% on average.

The rest of this paper is organized as follows. Section 2 provides some background information. Section 3 describes classification of data type in last-level cache and the data access distributions, then introduces the data access type aware replacement policy. Section 4 provides the experimental methodology and simulation results. Section 5 presents related work. This paper will be summarized in Section 6.

2 Background

2.1 Cache Clustering Scheme

Most modern chip multiprocessors adopt TILE architecture. A tiled CMP has the advantages of modularity for low design complexity and high scalability. As shown in Fig. 2, this work considers a tiled CMP. Each tile has a processor core, private L1 instruction and data caches, a unified and L2 cache slice organized as cluster, and a router connecting to the on-chip network. All the tiles are connected by a 2D mesh on-chip network, last-level caches distribute across the chip. For cluster organization, every 4 nearby tiles construct a cluster. In such a tiled CMP, each L2 slice can serve requests from any processor with in the cluster through the on-chip network. For simplicity, we assume there is no congestion in the on-chip network, and L2 caches are last-level caches and all data in the L1 cache must also be somewhere in the L2 cache (inclusive). Access to remote cache suffers longer on-chip latency.

The L2 caches are last-level caches in this work, although the discussion can be applied to any cache organizations with multiple cache hierarchies. Each L2 slice is the regional home node or the global home node for a fraction of the statically distributed physical address space. On an L1 miss, a regional home node will look up its local L2 slice for the requested cache line. On a regional cache miss, the remote global home node will repeat the look up process, and access off-chip memory on a miss. When an L1 replacement occurs, the regional home node will be notified.

Huh *et al.* [15] precisely defined the concept of sharing degree (SD) as the number of processors that share a pool of L2 cache banks. In this terminology, private scheme means an SD of 1 where each core maps and locates a requested cache block to and from its corresponding L2 bank. Shared scheme, on the other hand, means each core shares with all other 63 cores the total L2 banks [16]. Cluster scheme is intermediate between private scheme and shared scheme, which SD can be one of 2, 4, 8, 16 and 32. In this paper, we choose SD as 4 for simplicity, as shown in Fig. 2. The proposed design in this work can be applied to other cluster scheme with different SDs.

2.2 Data Access Types in LLC

In shared scheme, last-level cache accesses, as well as the cache blocks, can be classified into four access types, Instructions, Data-Private, Data-Shared-Read-Only,

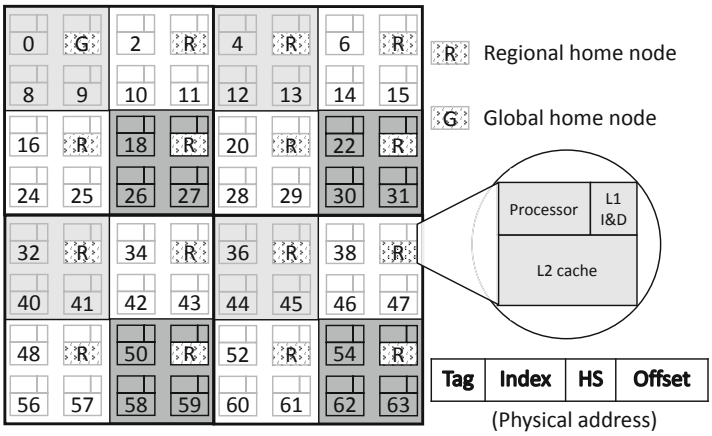


Fig. 2. Baseline architecture

and Data-Shared-Read-Write [19]. For cluster scheme, we can further divided each access type into two categorizations, accesses to global L2 cache bank or accesses to regional L2 cache bank, which means overall eight access types. Tab. 1 gives the access type names and descriptions.

Table 1. Data type in clustered last-level cache

Data access type	Description
Global private (G_PRIVATE)	L2 bank is the global home, cache block is accessed by one and only one requestor
Global Instruction (G_INS)	L2 bank is the global home, data in cache block belongs to instruction
Global shared read only (G_RO)	L2 bank is the global home, cache block is read by more than one requestors
Global shared read write (G_RW)	L2 bank is the global home, cache block is read or write more than one requestors, at lest a write requetor
Regional private (R_PRIVATE)	L2 bank is the regional home, cache block is accessed by one and only one requestor
Regional Instruction (R_INS)	L2 bank is the regional home, data in cache block belongs to instruction
Regional shared read only (R_RO)	L2 bank is the regional home, cache block is read by more than one requestors
Regional shared read write (R_RW)	L2 bank is the regional home, cache block is read or write more than one requestors, at lest a write requetor

2.3 LRU Replacement Policy

The Least Recently Used (LRU) replacement policy is commonly used in the on-chip memory hierarchies and is the de-facto standard of replacement policy for researchers. In the LRU policy, a cache block must traverse from the MRU position to the LRU position before being evicted from the cache. The LRU policy can get high performance for workloads which working sets are smaller than the on-chip cache size or have high locality.

A cache replacement policy mainly concerns about two issues: the victim selection policy and the insertion policy. The victim selection policy determines which cache block to be evicted when a new data block needs to be accessed from a lower level memory hierarchy. The insertion policy decides where to place the incoming data block in the replacement list. Traditional LRU replacement policy inserts a new cache block at the MRU position of the replacement list to keep it on-chip for a long time. When an eviction is needed, the cache block at the LRU position is chosen as the victim.

3 Implementation of Data Access Type Aware Replacement Policy

3.1 Data Access Type Classification

The classification of L2 cache blocks' data access type is easy to realize with little modification on current cache coherent protocols. We can follow the steps below

- To determine a cache block is an instruction or not. Which can be achieved by tracing the access type to L2 cache. If the access type is Instruction-Fetch, it means the corresponding data in the cache block is instruction.
- To determine a cache block is private or not, Which can be achieved by tracing the requestor of a data block. If a cache block is accessed by one and only one requestor, the block is private, otherwise the cache block is shared.
- To determine a cache block is read-only or not. Which can be achieved by tracing whether there is ever a write request to the cache block. If there is ever a write request to the cache block, the cache block is read-write, otherwise it is read-only.
- To determine a cache block is global or regional, which can be determined when a cache block is fetched to the L2 slice. If the global home id is equal to the L2 slice id, the cache block is global, otherwise it is regional.

The data access type determination of each cache block can be accomplished with the maintenances of L2 cache coherence protocol. The data access type of each cache block can be stored and updated together with the coherence states. For each cache block, only 3 additional bits are needed. For L2 caches with 64B cache line, the storage overhead is increased by about 0.6%. Despite instructions, all off-chip accessed data are initialized as G_PRIVATE, and then will be updated according to their later accesses. The data access type information of a cache block is cleared when evicted from L2 cache. In this work, we don't concern those rarely happened cases such as writing to instruction code which have little impacts on system performance.

3.2 Data Access Type Distribution in Shared Scheme

Fig. 3 gives the access type distribution of different workloads in shared schemes at the end of parallel execution. Detailed system configurations are given in

Section 4. Private cache blocks occupy about 57% of the overall cache capacity on average. The percentage of instruction cache blocks is relatively small. *Volrend* gets the maximum 6.6% percentage of instructions. Shared cache blocks occupy about 40% percent of the overall L2 cache capacity on average. For most workloads, the capacity of shared read only cache blocks are larger than shared read-write cache blocks. Fig. 4 gives the distribution of accesses to different L2 cache blocks during the parallel execution. More than 85% percent of accesses are belong to shared cache blocks. The percentage of requests for private cache blocks is less than 3% on average. About 10% accesses are for instructions.

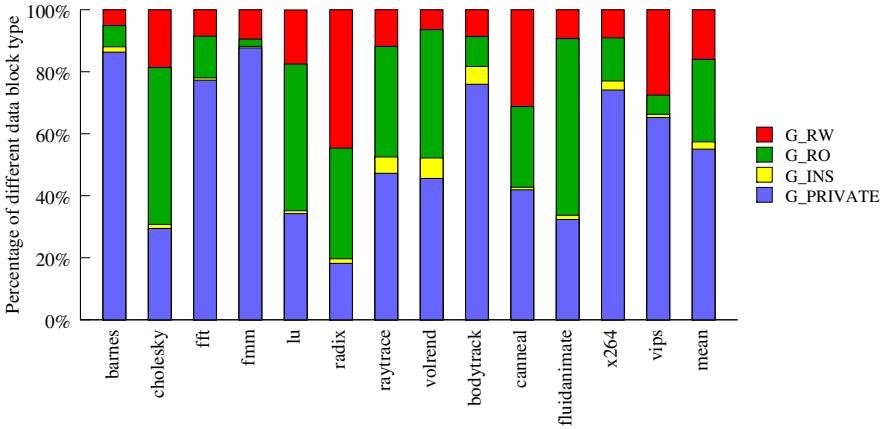


Fig. 3. Cache blocks distribution in shared scheme

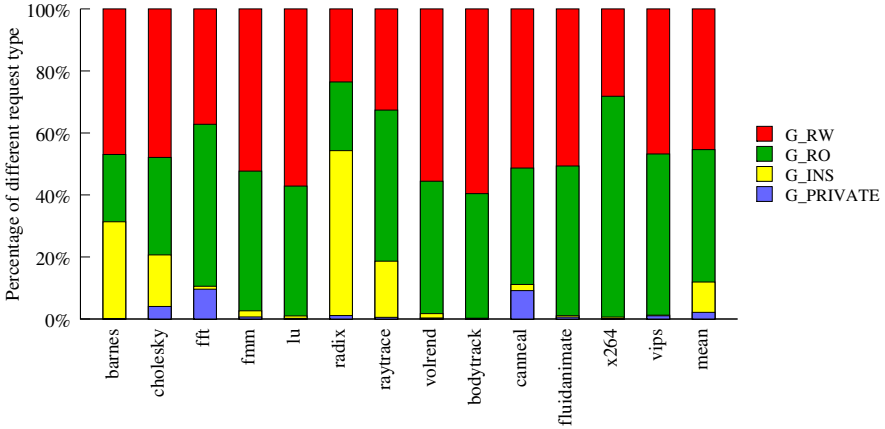


Fig. 4. Accesses distribution in shared scheme

3.3 Data Access Type Distribution in Cluster Scheme

Fig. 5 gives the data access type distribution of different workloads in cluster scheme at the end of parallel execution. In cluster scheme, there are overall eight data access types. On average, global cache blocks occupy about 33% of L2 capacity. The percentage of global instruction cache blocks is relatively small, volrend gets the maximum 3.46% percentage. Other cache blocks belong to regional data blocks. Regional private cache blocks occupy about 41% of the overall cache capacity on average. regional shared L2 cache blocks occupy about 15% percent of L2 capacity, and most of them are regional read-write. *Fluidanimate* gets the maximum 9.3% percentage of regional shared read-only cache blocks. Other workloads' percentage of shared read-only cache blocks are less than 2%. Fig. 6 gives the accesses distributions on different L2 cache blocks during the parallel execution. More than 85% percent of the accesses are belong to regional cache blocks. Percentage of regional private accesses are more than 60% on average.

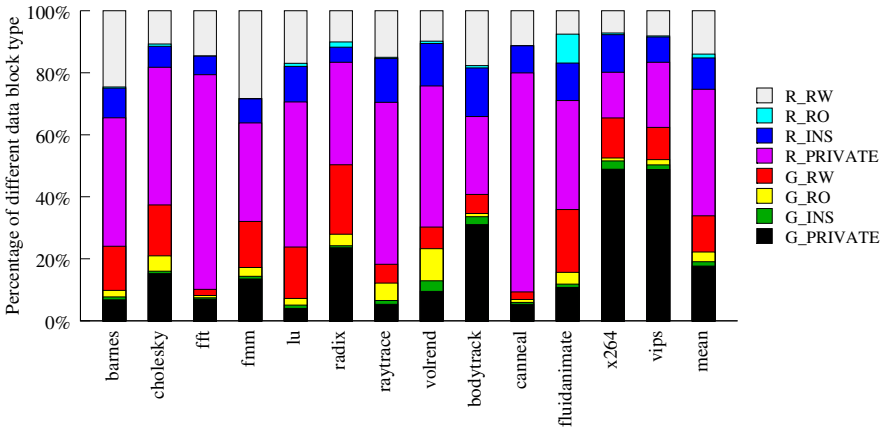


Fig. 5. Cache blocks distribution in cluster scheme

3.4 Data Access Type Aware Replacement Policy

The main idea of data access type aware replacement policy is to identify the access type of each cache blocks in last-level cache, and then design the insert policy and the victim selection policy both according to the access type of cache blocks. In this way, a more important cache block can stay in the cache set longer than a less important one, which may get further accesses.

To implement data access type aware replacement policy, the importance of each data access type should be determined first. As mentioned in Section 3.2, for parallel workloads running under shared cache organization, the global private cache blocks are least important, as they are rarely re-referenced while occupying

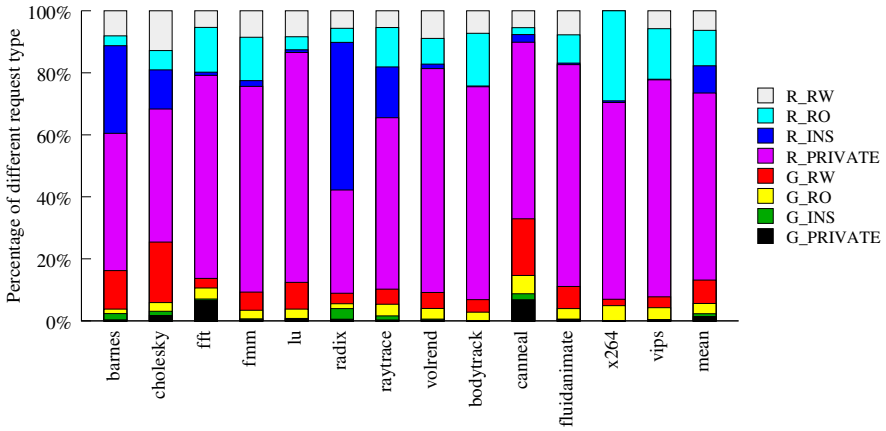


Fig. 6. Accesses distribution in cluster scheme

the majority of the overall L2 cache capacity. For cluster scheme, accesses to global instructions and shared data are less than under shared scheme because most accesses hit at regional home node. However, these cache blocks are as important as in shared scheme to avoid expensive off-chip accesses. In similar way, we can determine the importance of regional cache blocks in each type. Where regional read only cache blocks are the most important, as its capacity is small compared with its accesses. The capacity percentage of regional read-write cache blocks is higher than its access percentage, so it is unimportant.

Data access type aware replacement policy is based on traditional LRU policy, and is optimized through classification of different data access types. When a data block is fetched to the L2 cache slice, or its access type is updated by a cache access, its LRU position is set according to the updated access type. Global instructions and global shared data are placed at the MRU position. The LRU positions of other access types are set according to their importance, the more important, the closer to the MRU position. So the initial LRU position of global private is the most faraway from the MRU position. On victim selection policy, unlike LRU policy, DTARP choose a victim from several (4 in the work) candidates at the top of LRU stack. the final priority of a cache block is its LRU position plus its evict priority given in Tab. 2. The cache block with the highest priority will be selected as the victim. If two or more cache blocks have same priorities, their distances to the global home are compared and the nearest one is chosen as the victim. Otherwise the victim is the cache block with the

Table 2. Eviction priorities of different data access type

Data access type	G_PRIVATE	G_INS	G_RO	G_RW	R_PRIVATE	R_INS	R_RO	R_RW
Eviction priority	6	0	0	0	5	2	2	4

highest LRU value. When there is a cache hit without access type change, its LRU position is updated as traditional LRU policy.

4 Evaluation

In this section, we first describe the simulation environment. Then the experimental results are utilized to evaluate the effect of DTARP.

4.1 Simulation Setup

We use GEMS[13] tool sets, which is based on a full system simulator simics [14], to evaluate our proposal. The parameters of simulated system configuration are given in Tab. 3. We simulate a 64-core Tiled CMP, and the network-on-chip is modeled in detail, including all messages required to maintain cache coherence. We evaluate our design using eight workloads from SPLASH-2 [17] and five workloads from PARSEC [18] benchmarks. Tab. 4 gives the workload's name and problem size.

Table 3. Configurations of s imulated system

Component	Parameter
CMP size	64-core
Processor model	Sparcv9
CMP line size	64B
L1 I-Cache Size/ Associativity	32KB /2-way
L1 I-Cache Size/ Associativity	32KB /2-way
L1 Load-to-Use Latency	2 cycle
L1 Replacement Policy	LRU
L2 cache Size/Associativity (per tile)	256KB/16-way each tile
L2 Load-to-Use Latency	15 cycles
L2 Replacement Policy	LRU (***)
Network Configuration	8×8 2d mesh
One-hop Latency	3 cycles
External Memory Latency	300 cycles

Table 4. Benchmarks

Workload	Problem Size	Workload	Problem Size
barnes	32768 particles	bodytrack	simmedium
cholesky	tk 29.0	canneal	simlarge
fft	-m20	fluidanimate	simmedium
fmm	32768 particles	x264	simmedium
lu	1024×1024	vips	simsmedium
radix	1M-keys, 1024-radix, 2M-maxkey		
raytrace	teapot.env		
volrend	head		

We compare our design (Cluster-DTARP) with other three schemes: shared scheme using traditional LRU policy (Shared-LRU), cluster scheme using LRU policy (Cluster-LRU) and cluster scheme using RRIP [6] policy (Cluster-RRIP).

4.2 System Performance

Fig. 7 gives the system performance for four schemes: Shared, Cluster, RRIP and DTARP schemes. Shared scheme is the baseline system with traditional LRU policy. Both RRIP and DTARP get the maximum system performance with about 220% speedup for *x264* benchmark. The average system performance improvement of DTARP is 59.8%, which is about 10.9% higher than Cluster scheme. RRIP gets about 57.7% system performance improvement over Shared scheme on average, which is about 8.8% higher than Cluster scheme. For most workloads, RRIP and DTARP have similar performance, except *cholesky* and *canneal*, where the system performance of RRIP scheme are close to or even worse than Cluster scheme.

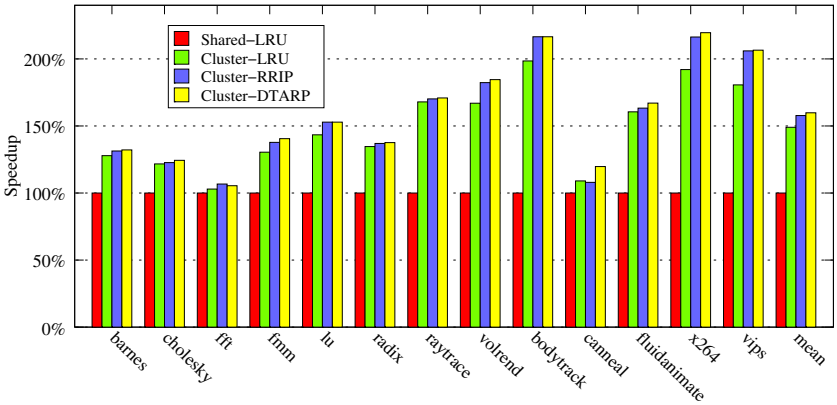


Fig. 7. System performance of 4 schemes

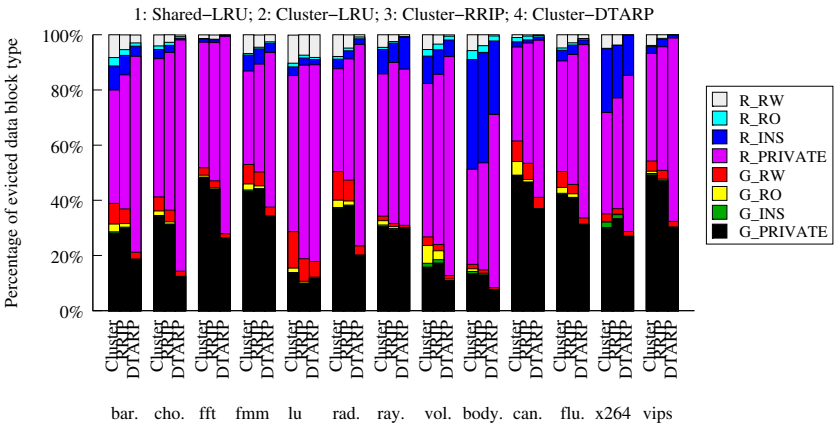


Fig. 8. Distribution of evicted cache block type

4.3 Evicted Data Distribution

Fig. 8 shows the data block type distribution of evicted cache blocks for Cluster, RRIP and DTARP schemes. DTARP scheme evicts least G_PRIVATE cache blocks for all workloads except *lu* and *raytrace*. At the same time DTARP scheme evicts most P_PRIVATE cache blocks for all workloads except *raytrace*. Cluster scheme has most evictions of instructions and shared data, either global or regional. RRIP schemes evicts more global instructions and global shared cache blocks than DTARP scheme.

4.4 L2 Capacity Distribution

Fig. 9 illustrates the L2 cache capacity occupied by different cache blocks for Cluster, RRIP and DTARP schemes. RRIP scheme keeps most R_RW cache blocks for all workloads, while DTARP scheme keeps lest R_RW cache blocks. For 12 of the 15 workloads, RRIP gets lest G_PRIVATE cache blocks, DTARP gets lest G_PRIVATE cache blocks in other 3 workloads. For most workloads, DTARP keeps most global instructions and global shared cache blocks, which is because DTARP scheme gives cache blocks of such access types higher importance than other access types.

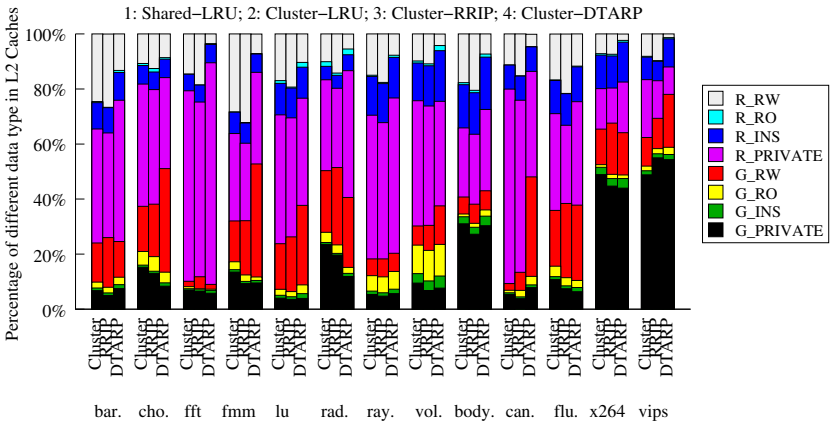


Fig. 9. Data access type distribution in L2 caches

4.5 LRU Hit Curve

Here we give the hit curve of Cluster, RRIP and DTARP schemes for workloads *fft* and *x264*, as shown in Fig. 10, the missed accesses are not considered. The hit curves of other workloads present similar. As cache blocks in RRIP scheme only has 4 LRU positions, it hit curve reach 1 when LRU position is 3. The hit curves of Cluster Scheme are close to RRIP, the hit counts on LRU position large than

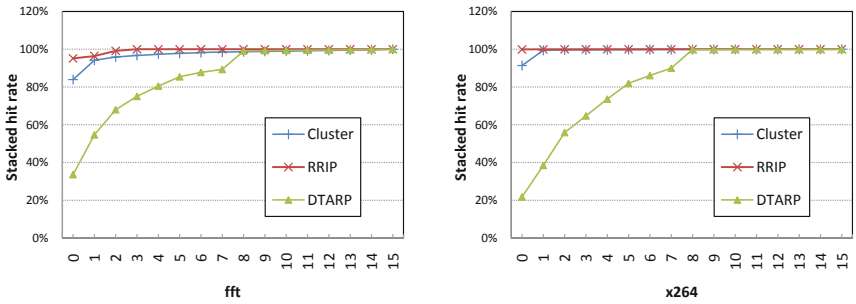


Fig. 10. Hit curve of different cache schemes

3 are relatively small. The hit curves of DTARP vary greatly from other two schemes. The stack hit rate is not close to 1 until LRU position approaching 8.

5 Related Work

There is an impressive amount of research work attempting to improve the cache performance through cache replication/migration or cache replacement policies. We describe the work which is most relevant to our proposal in the following paragraphs.

Lee et al. [1] proposed Least Frequently Used (LFU) replacement policy which predicts the blocks frequently accessed have a *near-immediate* re-reference interval, and the blocks infrequently accessed have a *distant* re-reference interval. LFU is suitable for workloads with frequent scans, but it doesn't work well for workloads with good recency. Then LRFU [1] is proposed to combine recency and frequency to make replacement.

Qureshi et al. [4] proposed Dynamic Insertion Policy (DIP) which places a few of the incoming lines in the LRU position instead of MRU position. DIP can preserve some of the working set in cache, which is suitable for thrash-access patterns. However, inserting lines at LRU position makes cache blocks are prone to be evicted soon and have no time to learn to retain active working set. Since DIP adopts a single insertion policy for all references of the workload, the LRU component of DIP will discard active cache blocks with single used cache lines. Jaleel et al. [5] extends DIP to make it thread-aware for shared cache multi-core.

RRIP [6] is the mostly recent proposal that predicts the incoming cache line with a re-reference interval between *near-immediate* re-reference interval and a *distant* re-reference interval. The prediction interval is updated upon re-reference. The policy is robust across streaming-access patterns and thrash-access patterns. However, it is not suitable for recency-friendly access patterns. The proposed DAI-RRP [12,11] is more robust than RRIP for it is also suitable

for access patterns with high temporal-locality. In addition, the hardware overhead of DAI-RRP is comparable to RRIP, which is identical to LRU policy.

Kim *et al.* pioneered the design of Non-Uniform Cache Architecture [3]. Huh et al. [15] presented a flexible NUCA design that supports a spectrum of sharing degrees. Hammoud et al. [16] proposed to dynamically adjust the sharing degree according to the L2 miss rate and average access latency. Hardavellas et al. [19] proposed R-NUCA that coordinates with OS support to classify the types of cache accesses, and used rotational interleaving for fast looking up. Marty et al. proposed Virtual Hierarchies to partition L2 caches for multiprogrammed workloads [20].

Cache partitioning is also one popular way of managing shared caches among competing applications. Stone et al. [7] investigated optimal (static) partitioning of cache resources between multiple applications when the information about change in misses for varying cache size is available for each of the competing applications. Dynamic partitioning of shared cache was first studied by Suh et al. [8]. Qureshi *et al.* [9] improved on [8] by separating the cache monitoring circuits outside the cache so that the information computed by one application is not polluted by other concurrently executing applications. Xie and Loh proposed Promotion/Insertion Pseudo-Partitioning [10] to partition shared cache by insertion policy. Jaleel *et al* proposed cache replacement and utility-aware scheduling [21] which is a hardware/software co-designed approach for shared cache management.

6 Conclusions and Future Work

As the scale of CMPs keeps increasing, the pressure on the on-chip memory hierarchies also expands. Cache clustering scheme combines the advantage of shared organization and private organization, which is a promising choice of CMPs last-level cache organization. Through analysis on data type distribution, as well as accesses in last-level cache, we point out that different data type get different access patterns. Shared data and instructions may be accessed many times, while most of the cache capacity is occupied by private cache blocks which are accessed infrequently. In this work, we propose *Data access Type Aware Replacement Policy* (DTARP) for cache clustering organization. DTARP classifies data blocks in last-level cache into different data types, and designs insertion and victim selection policy according to different data types based on traditional LRU policy. The global shared data will be kept in last-level cache longer than before. Simulation results show that DTARP can improve the performance of cluster scheme using LRU policy by 10.9% on average.

In this study, the eviction priorities of different data type are fixed, which may cause cache performance degradation for some applications. Design a more flexible mechanism by tracing the accesses together with L2 capacity usage by different cache block type, and dynamically update different data type priorities can be our future work.

Acknowledgements. The authors would like to thank the anonymous reviewers for their feedback in improving this paper. This work is supported by the Natural Science Foundation of China under Grant No.61373025, 61303002, and the National 863 High Technology Research Program of China(No.2012AA010905, 2012AA012609).

References

1. Lee, D., Choi, J., Kim, J.H., Noh, S.H., Min, S.L., Cho, Y., Kim, C.S.: LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. IEEE Transactions on Computers 50(12), 1352–1361 (2001)
2. Howard, J., Dighe, S., Hoskote, Y., et al.: A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In: ISSCC 2010, pp. 108–109 (2010)
3. Kim, C., Burger, D., Keckler, S.W.: An Adaptive, Non-uniform Cache Structure for Wire-delay Dominated On-chip Caches. In: ASPLOS-X, pp. 211–222 (2002)
4. Qureshi, M., Jaleel, A., Patt, Y., Steely Jr., S.C., Emer, J.: Adaptive Insertion Policies for High Performance Caching. In: ISCA-34, pp. 167–178 (2007)
5. Jaleel, A., Hasenplaugh, W., Qureshi, M., Sebot, J., Steely Jr., S.C., Emer, J.: Adaptive insertion policies for managing shared caches. In: PACT-17, pp. 208–219 (2008)
6. Jaleel, A., Theobald, K.B., Steely Jr., S.C., Emer, J.: High performance cache replacement using re-reference interval prediction (RRIP). In: ISCA-37, pp. 60–71 (2010)
7. Stone, H.S., Turek, J., Wolf, J.L.: Optimal Partitioning of Cache Memory. IEEE Transactions on Computers 41(9), 1054–1068 (1992)
8. Suh, G.E., Rudolph, L., Devadas, S.: Dynamic Partitioning of Shared Cache Memory. Journal of Supercomputing 28(1), 7–26 (2004)
9. Qureshi, M.K., Patt, Y.: Utility Based Cache Partitioning: A Low Overhead High-Performance Runtime Mechanism to Partition Shared Caches. In: MICRO-39 (2006)
10. Xie, Y., Loh, G.H.: PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches. In: ISCA-36, pp. 174–183 (2009)
11. Zhang, X., Li, C., Wang, H., Wang, D.: A Cache Replacement Policy Using Adaptive Insertion and Re-reference Prediction. In: SBAC-PAD-22, pp. 95–102 (2010)
12. Zhang, X., Li, C., Liu, Z., Wang, H., Wang, D., Ikenaga, T.: A Novel Cache Replacement Policy via Dynamic Adaptive Insertion and Re-Reference Prediction. IEICE Transactions on Electronics E94-C(4), 468–478 (2011)
13. Martin, M.M., Sorin, D.J., et al.: Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. Computer Architecture News (CAN) 33(4), 92–99 (2005)
14. Magnusson, P., Christensson, M., et al.: Simics: A Full System Simulation Platform. Computer 35(2), 50–58 (2002)
15. Huh, J., Kim, C., Shafi, H., Zhang, L., Burger, D., Keckler, S.W.: A NUCA Substrate for Flexible CMP Cache Sharing. In: ICS-19, pp. 31–40 (2005)
16. Mohammad, H., Sangyeun, C., Rami, M.: Dynamic Cache Clustering for Chip Multiprocessors. In: ICS-23, pp. 56–67 (2009)

17. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations. Computer Architecture News 23(2), 24–36 (1995)
18. Bienia, C., Kumar, S., Clara, S., Singh, J.P., Li, K.: The PARSEC Benchmark Suite: Characterization and Architectural Implications. In: PACT-17, pp. 272–281 (2008)
19. Nikos, H., Michael, F., Babak, F., Anastasia, A.: Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches. In: ISCA-36, pp. 184–195 (2009)
20. Marty, M.R., Hill, M.D.: Virtual Hierarchies to Support Server Consolidation. In: ISCA-34, pp. 46–56 (2007)
21. Jaleel, A., Najaf-abadi, H.H., Subramaniam, S., Steely Jr., S.C., Emer, J.: CRUISE: Cache Replacement and Utility-aware Scheduling. In: ASPLOS-XVII, pp. 249–260 (2012)

Agent-Based Credibility Protection Model for Decentralized Network Computing Environment

Xiaolong Xu[1,2], Qun Tu[1], and Xinheng Wang[3]

[1] College of Computer, Nanjing University of Posts and Telecommunications, Nanjing 210003, China
[2] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210046, China
[3] School of Computer, University of the West of Scotland, Paisley PA1 2BE, United Kingdom
{xuxl,B09040307}@njupt.edu.cn, Xinheng.Wang@uws.ac.uk

Abstract. Decentralized networks have some unique characteristics, such as heterogeneity, autonomy, distribution and openness, which lead to serious security issues and low credibility. In this paper, the agent technology is utilized to construct a novel credibility protection model, which efficiently makes open, disordered, decentralized networks evolve gradually as an orderly, stable and reliable computing environment. Within this model, our research is focused on the credibility measurement mechanism, and a novel behavior-credibility evaluation algorithm is proposed. The evaluation value of behavior-credibility is calculated based on the current and historical performances of nodes as service providers and consumers to make an accurate prediction. Based on a decentralized network simulation environment, we conducted several rounds of simulation experiments to test the function, performance and validity of the model, mechanism and algorithm.

Keywords: decentralized networks, agent, credibility.

1 Introduction

Many research and practice have indicated that decentralized network computing environments have some unique characteristics, such as heterogeneity, autonomy, distribution and openness, which lead to serious security issues and low credibility[1]. Without the reliable management of central node, malicious nodes could pose serious threats to the normal operation of network and damage the credibility of network through fake services, conspiracy, non-cooperation and the spread of malicious codes, etc. The construction and operation of decentralized-network-oriented software system faces greater uncertainty, which brings more serious issues of system security, reliability and availability.

On the other hand, new network applications continue to emerge, showing a more diverse and flexible features. The traditional study of software credibility focuses on building safe, reliable and available software in closure computing

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 269–276, 2013.
© Springer-Verlag Berlin Heidelberg 2013

environment[2], which is unable to meet for open, cross-organization computing environments.

In this paper, we provide a new solution to protect the credibility of decentralized network in order to make resource sharing and collaboration efficient and reliable in dynamic environments. We constructed the credibility protection model for decentralized network based on agent, which is able to support all kinds of application operating in decentralized network computing environments. Within this model, our research is focused on the credibility measurement mechanism, and a novel credibility evaluation algorithm is proposed, which is able to evaluate behavior-credibility. Our research result can promote open, chaotic, decentralized networks gradually into orderly, stable and reliable computing environments.

2 Analysis of Decentralized Networks

The basic element of decentralized network computing environment is the autonomous node. It can be found that decentralized network computing environments and their nodes have the following features[3]:

- Resource redundancy: the number of nodes decentralized network (such as Internet-based P2P network) might be very large, which means resources are always redundancy, but the stability of the resources sharing is enhanced.
- Capacity difference: the software and hardware resources of nodes are very different in quality and quantity.
- Node autonomy and network instability: nodes are free to join and withdraw from the network, and determine all by their own will how and when to use those resources owned by themselves, which makes the network very unstable.
- Lacking of centralized control node: decentralized networks don't have the centralized control node usually deployed in traditional distributed computing environments, resulting in low security management problem.
- Node anonymity and selfishness: the majority of nodes in the selfish decentralized network computing environment are selfish. All their activities (to provide services for others or obtain others' services) are to maximize their own interests and meet their default targets, not volunteer to provide any services and resources selfless; many nodes demand for anonymity and hope their actions hard to track, which also makes it easier for the existence of malicious nodes.

In short, decentralized networks show an unstable anarchy, bringing a series of serious problems, cumbering the normal operation of network systems. There are a large number of malicious nodes providing false and unreliable services, conspiring, attacking other nodes or even the whole network.

3 Agent-Based Credibility Protection Model for Decentralized Networks

Agent-based Node (AbN) of decentralized network is an independent, self-government software and hardware integrated system[3,4,5]. The introduction of Agent technology is to build a better system, which could manage resources and regulate actions of node in order to maximize the benefit of the whole network system. AbN is a kind of rational agent, of which the model can be expressed as follows:

$$AbN = (ID, Goal, Action, Capability, Policy, Creditability, Group) \quad (1)$$

Of which, AbN obtains its own ID when it adds network at the first time, which is the basis of the identity-credibility. *Goal* is the set of goal on behalf of all the objectives of AbN. *Action* is the set of collaborative action as follows:

$$Action = \{search, choose, commit, investigate, evaluate, execute, pay, gain\} \quad (2)$$

Action *search* is used to find some ability needed and its owner; *choose* is used to select the best suitable AbN; *commit* is to agree to implement the task sent from others; *investigate* refers to the investigation of the credibility of another node's action and ability; *evaluate* is to give a valuation to the node based on the interaction just happened; *execute*, *pay* and *gain* are easy to understand.

Capability is the set of the actual resources owned by AbN, including computing resources (exp. CPU and memory), storage resources (exp. hard drives) and software resources (exp. programs, documents, data):

$$Capability = \{c_1, c_2, ..., c_i, ..., c_m\} \quad (3)$$

Capability c_i contains the following information:

$$c_i = (subject_i, category_i, quantity_i, remainder_i) \quad (4)$$

$quantity_i$ is the total amount of Capability c_i; $remainder_i$ is the available amount of Capability c_i. For example, the number of resources in memory refers to the size of its physical space, and $category_i$ of memory is the type of reusable computing resources.

Policy is the set of strategy adopted by AbN who is executing some tasks:

$$Policy = \{P_1, P_2, ..., P_i, ..., P_m\} \quad (5)$$

P_i is a piece of policy based on c_i one-to-one, including $priority_i$, $timing_i$ and $quota_i$.

Creditability should objectively reflect the credibility of the behavior-credibility and ability-credibility of AbN.

Node performance is mainly determined by its ability and behavior. In addition, it is very possible that active nodes could fail sometimes in interactive transactions because of objective reasons, while those nodes seldom interacting

with others are certainly not easy to fail. It is unfair to say that nodes who seldom provide services are more trustworthy than those actively providing services. The credibility of nodes can be described as follows:

$$Creditability = (ability-credit, behavior-credit, number_1, number_2) \quad (6)$$

$number_1$ is the number of node providing services and $number_2$ is the number of node obtaining services.$Group$ is used to depict the logical group AbNs belong to.

4 Prediction-Oriented Behavior-Credibility Evaluation Algorithm

The connotation of credibility is manifested in behavior analysis through the quantitative measurement. The AbN with credible identity (that is, through authentication) might not have the credible behavior. Behavior-credibility not only involves the credibility of service providers, but also involves services consumers. Research on the credibility of the node behavior can forecast behavior of nodes in advance of it conducting any acts of vandalism in order to reduce or even avoid contact with malicious nodes, enhance their potential to complete tasks by establishing mutual credibility, and reduce the additional overhead brought about by monitoring and prevention of no-confidence. Certainly, the behavior-credibility is assessed on past evidence of nodes conduct to make an accurate prediction. The behavior-credit of dishonest node should decline so rapidly that other nodes do not want to interact with it.

Behavior-credibility can be described as the following formula:

$$behavior-credit = (pcredit, pnumber, ccredit, cnumber) \quad (7)$$

Behavior-credibility of AbN contains two pairs of factors: $pcredit$(AbN credibility acting as service provider) and $pnumber$(the number of service provided) as well as $ccredit$(AbN credibility acting as service consumer) and $cnumber$(the number of service obtained).

Both $pcredit$ and $ccredit$ can be divided into 7 levels: Distrust highly, Distrust moderately, Distrust slightly, Trust neutrally, Trust slightly, Trust moderately and Trust highly. Trust neutrally is given to the node just joining the network and not interacting with other nodes yet.

If a node always contributes valuable resources honestly, actively and stably, it would get a higher evaluation of other nodes interacting with it. Inspired by the EigenRep model of Stanford[6], a new prediction-oriented behavior-credibility evaluation algorithm is proposed, with the following description.

Suppose that in an decentralized network there are a series of interactive activities between AbN_A and AbN_B (AbN_A obtains service from AbN_B) in a certain period of time Δ_x; it is allowed that AbN_A stores the number of successful transactions it has had with AbN_B, S_{AB} , and the number of failure transactions it has had with AbN_B, F_{AB}. A normalized local direct evaluation value of AbN_B in provided by AbN_A, E_{AB}, is defined as follows:

$$E_{AB} = \begin{cases} \frac{\max(S_{AB}-\mu_B F_{AB},0)}{\sum_B max(S_{AB}-\mu_B F_{AB},0)}, \sum_B \max(S_{AB}-\mu_B F_{AB},0) \neq 0 \\ 0, \sum_B \max(S_{AB}-\mu_B F_{AB},0) = 0 \end{cases} \tag{8}$$

Parameter μ_B is a very important factor in this formula, which is used as the adjustable penalty factor. $pcredit_{B,\Delta_x}$ is set as the global behavior-credibility evaluation value of AbN_B as a service provider in the period of Δ_x. The value of μ_B is determined with $pcredit_{B,\Delta_{x-1}}$, which is the global behavior-credibility evaluation value of AbN_B in the period of Δ_{x-1}(that is the previous time period before the current period of Δ_x):

$$\mu_B = 1+\alpha^{pcredit_{B,\Delta_{x-1}}}, 0 < \alpha < 1 \tag{9}$$

This means that node with the worse performance in the previous period, will be punished more seriously in this current period. The global behavior-credibility evaluation value of AbN_B as a service provider in the period of Δ_x is:

$$pcredit_{B,\Delta_x} = (1-\lambda)\Sigma_m E_{mB} \times ccredit_{m,\Delta x} + \lambda pcredit_{B,\Delta_{x-1}}, 0 < \lambda < 1 \tag{10}$$

$pcredit_{B,\Delta_{x-1}}$ is also considered above, indicating that AbN_B have to keep acting honestly, reliably in all transactions. $ccredit_{m,\Delta x}$ indicates that the important degree of evaluation on AbN_B is based on the its trading partners behavior-credibility as service consumers.

Similarly, the global behavior-credibility evaluation value of AbN_B in the period of Δ_x as a service consumer is:

$$ccredit_{B,\Delta x} = (1-\lambda)\Sigma_m E_{mB} \times pcredit_{m,\Delta x} + \lambda ccredit_{B,\Delta_{x-1}}, 0 < \lambda < 1 \tag{11}$$

The above algorithm considers the dynamic adjustable penalty factor and the nodes history behavior, which makes it more reasonable and efficient than EigenRep.

5 Experimental Analysis

We designed and built a Decentralized network simulation environment containing 100 AbN nodes. In this environment, we conducted several rounds of simulation experiments to test the function and performance of the model, mechanism and algorithm proposed in this paper. In order to facilitate comparative experiments, we have set a fuzzy interval $[\beta_1, \beta_2]$ based on the average value of the node credibility, and defined AbN nodes by the credibility into three types simply: TH AbNs, which overall evaluation values are higher than the value β_2;

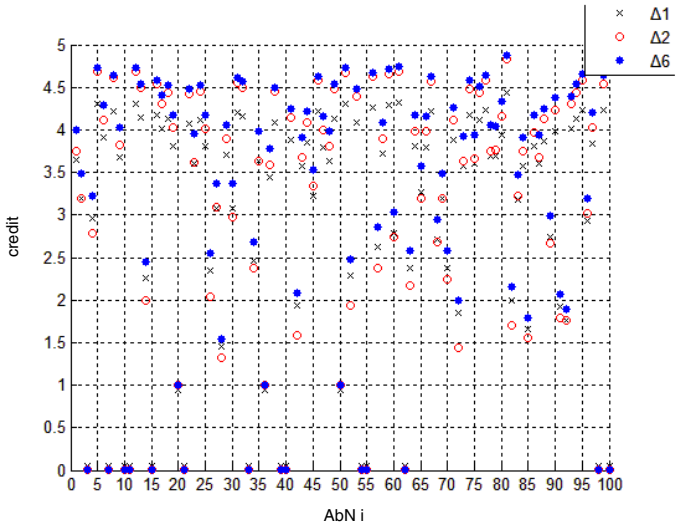


Fig. 1. The changes and distribution of global credibility evaluation value of AbN ($\alpha = 0.1, \lambda = 0.1$)

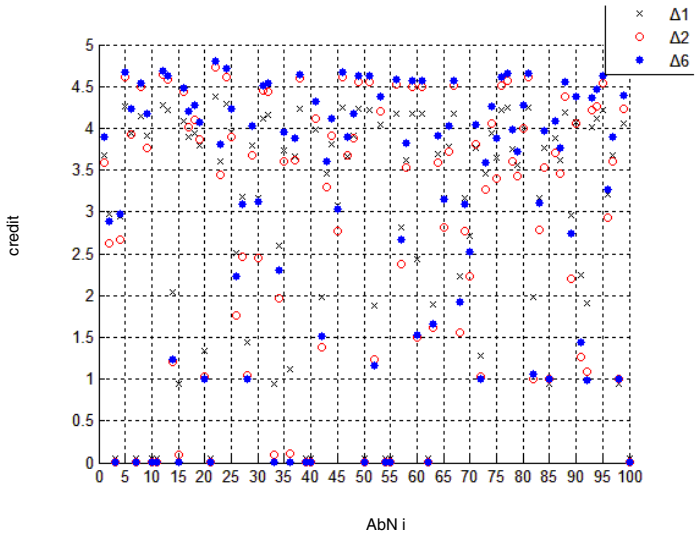


Fig. 2. The changes and distribution of global credibility evaluation value of AbN ($\alpha = 0.7, \lambda = 0.1$)

TN AbNs, which overall evaluation values are equal to the average value and between; DH AbNs, which overall evaluation values are lower than the value β_1. In the following part, we focused on analysis and testing on the new credibility evaluation algorithm. We studied how different adjustment parameter α of the

penalty factor μ affecting the global credibility evaluation values of AbNs. We compared the global credibility evaluation values of AbNs of three time sections, Δ_1, Δ_2 and Δ_6, As Fig. 1 and Fig. 2 show. Supposing $\lambda = 0.1$, the new credibility evaluation algorithm proposed in this paper with different α can all impove the global credibility evaluation values of TH AbNs, and low the global credibility evaluation values of DH AbNs, which proves the validity of the algorithm. From Fig.1 and Fig.2 we can find that if parameter α is set higher, the effect of penalty factor μ to the global credibility evaluation value of AbN is smaller; if parameter α is set lower, the effect of penalty factor μ to the global credibility evaluation value of AbN is larger, which is able to distinguish between TH AbNs and DH AbNs sooner.

6 Conclusion

Lacking of credibility and security mechanisms, current decentralized network computing environments can not fully utilize the enormous computing power and the mass of types of resources, and provide really convenient, stable cooperation work platform to share resources and address needs for large computing, large storage space and valuable projects. Currently, domestic and overseas research institute have not studied the credibility mechanism of resource sharing and collaboration in decentralized network computing environment well enough, still at the initial stage. It is necessary to consider the credibility problems emerging at decentralized network computing environment, and included the trust-computing architecture, avoiding decentralized network computing system in a relative reliable and isolative status. Results of this study can be widely applied to Grid computing, P2P computing, cloud computing and other network computing systems and applications. It is sure that the specific application methods and application results are also worth further study.

Acknowledgments. This work was supported by the National Natural Science Foundation of China(61202004), the China Postdoctoral Science Foundation funded project (2011M500095, 2012T50514), the Natural Science Foundation of Jiangsu Province (BK2011754), the Jiangsu Postdoctoral Science Foundation funded project (1102103C), the Natural Science Fund of Higher Education of Jiangsu Province (12KJB520007) and the Project Funded by the Priority Academic Program Development of Jiangsu Higher Education Institutions(yx002001).

References

1. Shen, C., Zhang, H., Wang, H.: Research on Trusted Computing and its Development. Science China: Information Sciences 3, 405–433 (2010)
2. Liu, K., Shan, Z., Wang, J.: Survey of Basic Research in Trusted Software. Sci. Found China 3, 145–151 (2008)

3. Xu, X., Wang, R.: The Collaboration Alliance Mechanism of P2P Based on Mobile Multi-agent Technology. Journal of Electronics & Information Technology 2, 345–349 (2007)
4. Wang, H., Tang, Y., Yi, G.: Trustworthiness of Internet-based software. Science in China Series E: Information Sciences 10, 759–773 (2006)
5. Jiang, W., Zhang, L., Wang, P.: Dynamic Scheduling Model of Computing Resource Based on MAS Cooperation Mechanism. Science in China Series F: Information Sciences 8, 1302–1320 (2009)
6. Dou, W., Wang, H., Jia, Y., Zou, P.: A Recommendation-Based Peer-to-Peer Trust Model. Journal of Software 4, 571–583 (2004)

A Vectorized K-Means Algorithm for Intel Many Integrated Core Architecture

Fuhui Wu, Qingbo Wu, Yusong Tan, Lifeng Wei, Lisong Shao, and Long Gao

School of Computer Science, National University of Defense Technology,
Changsha 410073, China
{fuhui.wu,qingbo.wu,yusong.tan,lifeng.wei,
lisong.shao,long.gao}@nudt.edu.cn

Abstract. The K-Means algorithms is one of the most popular and effective clustering algorithms for many practical applications. However, direct K-Means methods, taking objects as processing unit, is computationally expensive especially in Objects-Assignment phase on Single-Instruction Single-Data (SISD) processors, typically as CPUs. In this paper, we propose a vectorized K-Means algorithm for Intel Many Integrated Core (MIC) coprocessor, a newly released product from Intel for highly parallel workloads. This new algorithm is able to achieve fine-grained Single-Instruction Multiple-Data (SIMD) parallelism by taking each dimension of all objects as a long vector. This vectorized algorithm is suitable for any-dimensional objects, which is little taken into consideration in preceding works. We also parallelize the vectorized K-Means algorithm on MIC coprocessor to achieve coarse-grained thread-level parallelism. Finally, we implement and evaluate the vectorized method on the first generation of Intel MIC product. Measurements show that this algorithm based on MIC coprocessor gets desired speedup to sequential algorithm on CPU and demonstrate that MIC coprocessor owns highly parallel computational power as well as scalability.

Keywords: K-Means, Vectorization, Locality, MIC Coprocessor, Algorithm.

1 Introduction

1.1 K-Means

Clustering is considered as one of the most important unsupervised learning [1] problem. The target of clustering is to find a structure in a collection of unlabeled data. There are a number of published algorithms to achieve this target. Among all these algorithms, K-Means [2] is an effective and widely applied one in many applications for its simplicity, and was identified as one of the top 10 algorithms in data mining [3].

The K-Means problem offers no accuracy guarantees, it is NP-hard to solve this problem exactly [21]. Nevertheless, K-Means execute time is still very appealing in applications; especially with data scale expands at an amazing speed nowadays. Over the past decades, a number of efficient K-Means algorithms have been studied.

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 277–294, 2013.
© Springer-Verlag Berlin Heidelberg 2013

By augmenting k-means with a very simple, randomized seeding technique, K-Means++ [5] obtained an algorithm that is $O(\log k)$ competitive with the optimal clustering. K-Means|| [6] proposed an efficient parallel version of the sequential K-Means++. Both of them focus on the initialization of K-Means, which is crucial for obtaining good final solutions. K-Means# [7] provided a clustering algorithm that approximately optimizes the k-means objective in the one-pass streaming setting based on K-Means++.

Another idea of K-Means acceleration is to explore the parallelism feature of input data, which is also the foundation of this paper. Paper [8] studied the performance of general-purpose applications as well as K-Means on graphics processors using Compute Unified Device Architecture (CUDA). Paper [9] [10] implemented K-Means on GPUs with CUDA, and got speedups. The speedups are obtained through a vast number of simple, data-parallel, deeply multithreaded cores and high memory bandwidths. Mahout [11] implement a parallel version of K-Means algorithm on top of Hadoop [12] using the MapReduce [13] paradigm.

Preceding parallel optimization [9][10][11] can be summarized as optimization at task level. Most of them adopt objects oriented scheme, which is nature to be parallelized by uniformly splitting objects into chunks. And then, each chunk is processed in sub-tasks. These parallel methods do not need to modify computational logic of sub-tasks. They just spawn a number of processes on cluster nodes or threads on GPU at runtime for all sub-tasks. All these processes and threads execute the same logic as in common K-Means algorithm.

However, preceding parallel methods do not take the dimension parameter into account. The GPU based methods are just suitable for one-dimensional objects. So, they just achieve coarse-grained parallelism. This paper proposes a new vectorized K-Means algorithm that suits any-dimensional objects and is particularly suitable for MIC Architecture supporting multi-level parallelism.

1.2 Many Integrated Core Architecture (MIC)

There is a strong trend to use heterogeneous architecture with moderate amounts of CPU cores and a number of accelerators or coprocessors in High Performance Computing (HPC) recently. This trend has been announced from TACC [14] and ORNL [15]. From the top500 Supercomputing list we also observe that both Titan - Cray XK7 (Top 1 in November 2012) and Tianhe-1A - NUDT YH MPP (Top 1 in November 2010) used NVIDIA GPUs as accelerators. From the Top10 list in Nov. 2012 (www.top500.org), we can find the Intel Xeon Phi coprocessor, of Intel MIC Architecture, has already been used as accelerators in Stampede.

The Intel MIC Architecture [16] was announced in this context in 2010. Its first generation of Intel Xeon Phi product, codenamed Knights Corner (KNC), was just release in the late of 2012. It is a general-purpose, many-core architecture that supports shared memory execution model based on Intel's previous Larrabee [17] design. Intel MIC coprocessor supports developers to run on standard, existing programming

tools and methods. Each Intel MIC coprocessor is consisted of many Intel CPU cores. Each core can execute 512-bit width SIMD instructions, which is a key feature contributing to MIC coprocessor's computational power.

The first generation of Intel Xeon Phi product has key specifications of: up to 1 teraflops double-precision performance, exceptional performance-per-watt for highly parallel workloads and familiar programming mode. Nevertheless, to port applications on MIC is not straight forward. Up to now, there are little published studies on MIC as [18] [19]. To take advantages of MIC, the key technologies are vectorization, improving data locality and parallel on many integrated cores. Through this case study of using MIC to accelerate K-Means, we make an early exploration of Intel MIC architecture in real-world application.

1.3 Our Contributions

In this paper we obtain a vectorized version of the K-Means algorithm. The main idea is to further exploit data parallel characters in K-Means problems. There are variations of parallel K-Means methods based on MapReduce or GPU technologies [20] mentioned in section-1.1. However, they just exploit the parallelism between objects. We further exploit the parallelism between elements of multi-dimensional object.

On the other hand, this paper is largely inspired by the newly announced MIC architecture. This vectorized K-Means algorithm is suitable for this architecture, because we can exploit both element-level and object-level parallelism. We then evaluate the performance of our algorithm on heterogeneous architecture of CPU and MIC.

Our key contributions in this paper are:

- We firstly describe the execute characteristic of K-Means algorithm. And then, propose a vectorized K-Means algorithm to further exploit data parallelism.
- We design a special data layout scheme for this algorithm, because data locality plays an important role in both vectorization execution and thread-level parallel on shared memory.
- Finally, we port the vectorized K-Means algorithm on heterogeneous architecture of CPU and MIC, and then make evaluation. We also exploit the computational power and scalability of MIC architecture.

2 A Vectorized K-Means Algorithm

In this section we present our vectorized algorithm based on a classic K-Means algorithm, called Lloyd's iteration [4]. Firstly, we make an introduction of the Lloyd's iteration. Then, we set up the notations that will be used throughout this paper. After that, we present our vectorized algorithm in the trend towards fine-grained parallelism by using more complicated SIMD vector instruction sets.

2.1 Lloyd's Iteration and Notations

A simple and popular implementation of K-Means is called Lloyd's iteration, which is made up of two steps: Objects-Assignment and Centroids-Recalculation. As an initialization, k centroids are chosen randomly in the first iteration. And then, all n objects in R^d are assigned to the nearest centroid. So, n objects are divided into k clusters. In the second step, k centroids are re-calculated for k clusters. The iteration is then repeatedly until k centroids converge to a stable state that no longer moves.

In the following section, we set up some notations that will be used throughout the whole paper:

Let $O = \{o_1 \dots o_n\}$ be a set of objects to be clustered into k clusters $C = \{c_1 \dots c_k\}$ in the d-dimensional Euclidean space, where c_i is the centroid of ith cluster. The Euclidean distance, denoted as $||o_i - c_j||$, is used to measure the distance from o_i to jth cluster.

For each object in O, we define the assignment of o_i to one cluster in C follows the formula of:

$$if\ \forall c_l \in \{c_1 \dots c_k\}, ||o_i - c_j|| \leq ||o_i - c_l||, \text{membership}'(\text{o}_i) = j \tag{1}$$

After the assignment of all objects, we get k new clusters $C' = \{c'_1 \dots c'_k\}$. For each new cluster, the new centroid can be compute as:

$$c'_i = \frac{\sum o_l}{|o_l|}\ \text{, where } \{o_l : membership'(o_l) = i\} \tag{2}$$

We also define a threshold $\emptyset_0$, telling the program when to terminate in practice. Whether the centroids converge to a stable state is checked by a parameter $\emptyset$ given by:

$$\emptyset = {}^{\Delta}/_{|O|}, where\ \Delta = |\{o_i : membership'(o_i)\ \textbf{not equal to}\ membership(o_i)\}| \tag{3}$$

In formula (3), delta is the total number of objects whose membership has changed in current iteration compared to previous iteration.

With these notations, we can present the pseudo code of Lloyd's iteration in Algorithm-1.

Algorithms 1. Lloyd's iteration

```
C ← random chosen from O
initialize membership(o_{1...n}) = −1
while ∅ > ∅_0 do
    for each o_i ∈ O
        compute membership'(o_i)                (1)
    for each c'_i ∈ C'
        compute c'_i                            (2)
    compute ∅                                   (3)
end while
```

2.2 Execute Characteristics of K-Means

In order to analyze the execute characteristic of K-Means, we implement the Lloyd's iteration in C++, and examine the execute time in different datasets. We breakdown the execute time into two parts corresponding to two phases of Objects-Assignment and Centroids-recalculation.

We use five synthetic datasets to make the evaluation. All datasets compose of 100,000 objects, using float as element data type. They are different from each other in dimension configurations as shown in Table 1. The figures in brackets from dimension column denote the total size of five datasets. For each dataset, we carry out three experiments with the parameter k tuning from 50 to 150. From Table 1, we find that the run time increase astonishingly with the increase of dimension and k parameters. We can also get that Objects-Assignment time occupies the most of running time. Specially, with the increase of dimension and k, the value of $(Assign. / ReCalculation.)$ enlarges drastically. Keeping these characteristics in mind, we propose our vectorized K-Means algorithm.

Table 1. Time breakdown of Lloyd's iteration

Time Breakdown	Assign Time(s)			Centroids Re-Calculation Time(s)			Assign. / ReCalculation.		
d \ k	50	100	150	50	100	150	50	100	150
1(1.6M)	10.2	31.7	45.6	0.60	0.99	0.98	16.8	31.9	46.4
5(5.0M)	32.7	96.3	62.4	1.11	1.67	0.72	29.4	57.4	86.3
10(9.3M)	51.0	76.1	203	1.48	1.11	2.00	34.4	68.2	101
15(14M)	122	161	307	3.23	2.19	2.75	37.7	73.6	111
20 (18M)	126	203	235	3.15	2.56	1.99	40.0	79.3	117

2.3 Vectorization of K-Means Algorithm

The vector processing mode of single-instruction multiple-data (SIMD) has always been an important branch of parallelism as well as thread-level parallelism in multi-cores and MapReduce-like parallelism in clusters. The latter two are much simpler in programming by splitting a large job into small tasks. However, the SIMD parallelism on vector processing unit shares the advantages of power and area efficiency on single chip. Nowadays, most major high-performance CPUs support short SIMD instruction set extensions [22], such as SSE and AVX introduced by Intel. Recently, Intel extends SIMD instructions width to 512-bit with the release of Intel MIC coprocessor. The width is going to be wider in Intel's roadmap. For K-Means problems, however, a high vectorization ratio cannot be expected in the algorithms mentioned above.

In section 2.2, we observe that the Objects-Assignment phase occupies most of the running time. In theoretical, the running time of Objects-Assignment is $O(n \times k \times d)$, while the running time of Centroids Re-Calculation is $O((n+k) \times d)$. So, the value of $(Assign. / ReCalculation.)$ equals to $O\left(\frac{(n \times k \times d)}{(n+k) \times d}\right) \cong O(k), if\ n \gg k$, which can be verified in experimental results in Table 1. In this section, we focus on the vectorization

of Objects-Assignment phase. In section 3, a full implementation will be presented based on the heterogeneous architecture of CPU and MIC.

As we all know, there are two critical factors to attain high vectorization ratio: (i) the vector length must be long sufficiently, (ii) there should not be operation dependence between vector elements in one operation. For wide width vector instructions in MIC coprocessor, data locality is another factor that also influences performance obviously.

To meet these requirements, there are two choices in Objects-Assignment phase:

1) Objects-oriented Objects-Assignment

Objects-oriented mode is a natural strategy. The intuition here is that k distances from one object to k centroids are calculated in one loop, and then the centroid of the shortest distance is chosen as the cluster, to which current object will be assigned. After the assignment of current object, the next object is processed and so on. This strategy is easy to scale out by splitting objects into multi-chunks and spawning multi-processes on cluster or multi-threads on multi-core processor to process these chunks. Most sequential and parallel K-Means methods in the preceding adopt this strategy.

However, this strategy is not adapted to the vectorization requirements mentioned above. First, the elements number, equals to d, is not long sufficient required in the first premise. Unless the objects are of 1-dimensional, then the whole objects are taken as one vector as in [10] method, that assume all objects are 1-dimensional. Second, the operations on vector elements are not independent. To compute distance from objects to centroids, a classic distance mode is the Euclidean-Distance denoted as $\sqrt{\sum_{i=1}^{d}(o_i - c_i)^2}$, where o_i and c_i is the ith element of object o and centroid c respectively. The subtraction and square operations on different elements can execute separately, while the sum operations of different elements need to write to the same intermediate result.

2) Centroids-oriented Objects-Assignment

Centroids-oriented mode is another strategy that is used in this paper. The intuition here is that n distances from n objects to the same centroid are calculated in one loop, and then n new distances from n objects to the next centroid are calculated in the next loop. At the end of each loop, n new distances are compared to previous n distances correspondingly to n objects except the first loop as an initializer. If the new distance is shorter, the corresponding object is reassigned to a new centroid. The details are presented in Algorithm 2.

Algorithm 2. Centroids-oriented Obj.-Assign.

1: **initialize** $distOld(o_{1\ldots n}) = computeDist(o_{1\ldots n}, c_1)$
2: **for** each c_i **from** 2 **to** k
3: $\quad distNew(o_{1\ldots n}) = computeDist(o_{1\ldots n}, c_i)$ (4)
4: $\quad$ **for** each o_j **from** 1 **to** n (5)
5: $\quad\quad$ **if** $distNew(o_j) < distOld(o_j)$
6: $\quad\quad\quad membership(o_j) = i$
7: $\quad\quad\quad distOld(o_j) = distNew(o_j)$

We find an obvious characteristic that the arithmetic (4) and logic (5) operations are separated in this strategy, which will be utilized in the full implementation on MIC coprocessor.

In this strategy, it owns potential to meet two vectorization requirements if we calculate the distances of all objects to current centroid in the manner of Algorithm-3, which take each dimension of all objects as one long vector. First, the vector length, equals to objects scale of n, is long enough. Second, the operations of subtraction, square, sum and square root are independent, because they are carried out on different objects.

From the pseudo code we can see, that all arithmetic operations in Algorithm-3 can be implemented in a vectorization way.

Algorithm 3. Distances calculation

1: **initialize** $dist(o_{1...n}) = \{0\}$
2: **for** dim **from** 1 **to** d
3: **for** $each\ o_i$ **from** 1 **to** n
4: $dist(o_i) += (o_i(dim) - c(dim))^2$
5: **for** $each\ o_i$ **from** 1 **to** n
6: $dist(o_i) = \sqrt{dist(o_i)}$

2.4 Data Locality Optimization

A typical set of d-dimensional objects are distributed in Rd space as shown in Fig. 1(a), which is also the layout of data in hard disk files.

In memory, a common layout mode of all objects is shown in Fig. 1(b), in which the serial numbers denote the element offset to the first element of the first object. This layout mode is used in most preceding sequential or parallel variations of K-Means method. As in the parallel variation for example, the ith chunk starts at $((i - 1) \times chunk_size) \times d + 1$, where chunk_size is the splitting granularity of chunks.

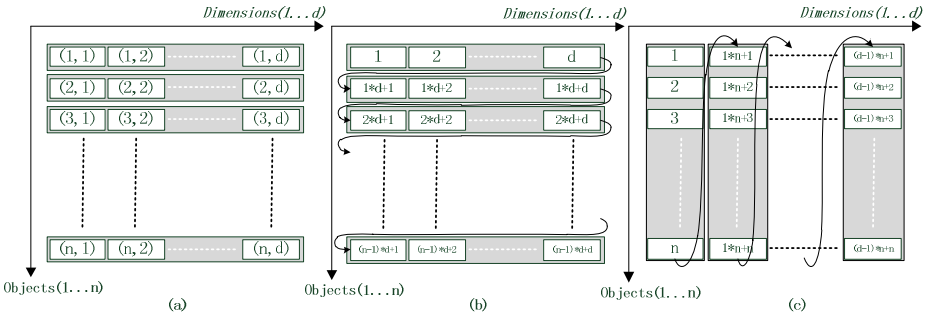


Fig. 1. Data layout for locality optimization

The layout mode in Fig. 1(b) works well for objects-oriented strategy, because the data access pattern is also serial in that direction. However, this layout mode is not suitable for vectorized K-Means algorithm. We designed a new data layout mode as shown in Fig. 1(c), following SoA (Struct of Array) methodology. It is unlike the

layout mode in Fig. 1(b), following AoS (Array of Struct) methodology. This data layout mode performs better in utilization of memory bandwidth and cache, because Gather and Scatter operations [23] are not needed to prepare vectors for vector operations in the vectorized K-Means method.

Based on the vectorized K-Means algorithm, we then implement the algorithm on MIC coprocessor to achieve both fine-grained parallelism through 512-bit SIMD instructions and coarse-grained parallelism through multi-threads parallel.

3 Implement Vectorized K-Means on MIC

3.1 MIC Architecture Overview

Intel MIC coprocessor combines many Intel CPU cores onto a single chip. All the cores are general-purpose cores that can execute 64-bit scalar instructions and 512-bit vector instructions (16 single-precision or eight double-precision floating-point values per vector instruction) as well. The vector instructions run on vector procession unit (VPU) of each core. Each core supports four hardware threads with round-robin scheduling between instruction streams. All the cores share the coprocessor memory via a ring bus. It uses the cache structure of per-core L1 32KB Instruction/Data cache and L2 512KB cache. All caches and the shared memory are fully coherent. This Architecture satisfies the increasing computational demand and enables a higher ratio of computation per watt.

The MIC card is designed for highly parallel part of a program. It works as coprocessor that is connected to CPU through PCI-e bus. MIC supports two easy Programming Modes, called offload and native. In offload mode, hot parts of calculation are offloaded to coprocessor just by adding a directive: "#pragma offload …" In the native programming model, it compiles applications to run directly on the coprocessor. Because there is a Linux-based operating system running on MIC coprocessor, it supports to run all code on coprocessor just by adding a "–mmic" option at compiling time.

For parallel Programming Modes, most options available in the host systems are available in Intel MIC coprocessor. These include: pThreads, OpenMP, Intel Threading Building Blocks, and Intel Cilk Plus.

In general, the main advantages of MIC are as follows:

1) It owns strong parallel computational power. One MIC card is consisted of many integrated cores, and each core can execute 512-bit wide SIMD instructions offering high FLOP rates for computationally dense workloads.
2) It provides an easy programming-mode for user who is familiar with C/C++ or FORTRAN. Thread-level parallel is easy to achieve through most parallel programming language as OpenMP, TBB, and Intel Cilk Plus.
3) It is an energy efficient design. In highly parallel applications, there are frequently inner loops that step regularly through memory and perform the same math operations repeatedly. The energy overhead for processing instructions repeated is impressive if these loops are assembled with scalar instructions. While SIMD amortizes that cost by doing all of the bookkeeping once and performing many math operations in just one instruction.

3.2 Vectorized K-Means on MIC: System Design

Fig. 2 shows the basic work flow of vectorized K-Means algorithm on heterogeneous architecture of CPU and MIC.

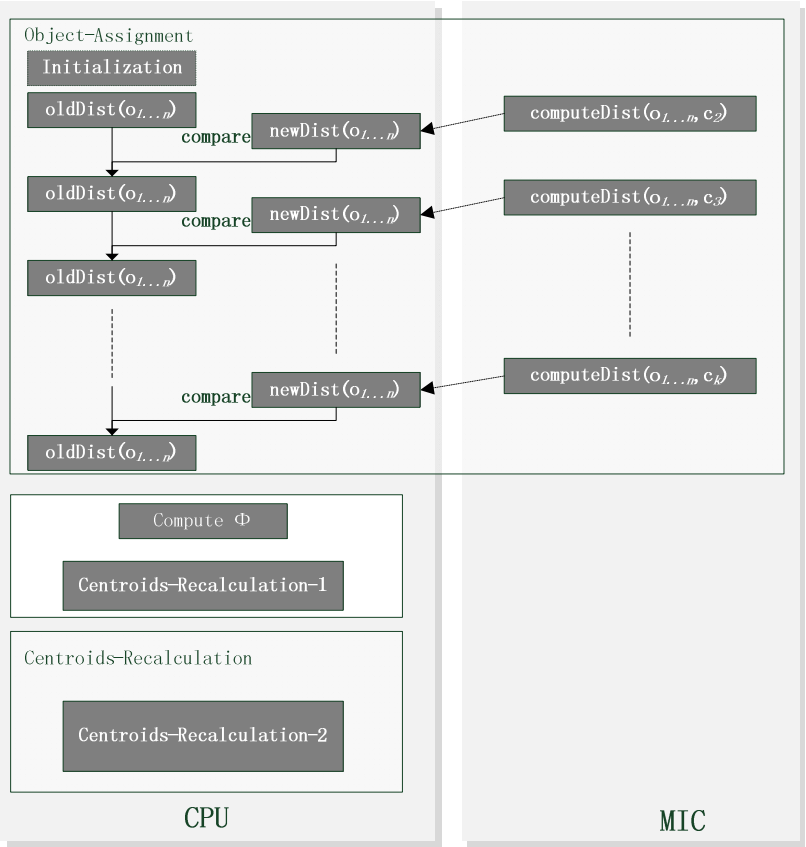


Fig. 2. The basic work flow of vectorized K-Means on CPU and MIC

We adopt offload programming mode to offload highly parallel part to MIC coprocessor. However, the algorithm presented in previous section contains two main processes that run in a sequential order. And not all of them are suitable to be offloaded to MIC, because the MIC coprocessor does not perform well in logic operation. So, we divide the processes into logic operation and arithmetic operation firstly, and run them on host and coprocessor separately.

1) We separate the $\emptyset - computation$ operations from Objects-Assignment. There is a logic operation of deciding whether the membership of object has changed.
2) In Object-Assignment phase, the distance calculation and membership modifying is separated. The distance calculation can run parallel but membership modifications need to compare old distances with new distances.

Secondly, the processes of membership modification and distance calculation already execute on CPU and MIC separately to take advantages of CPU's logic computational power and MIC's parallel arithmetic computational power. But the two processes do not have to execute serially. Because the modification of membership for ith centroid can run simultaneously with the distance computation for (i+1)th centroid. After the distances to ith centroid be calculated, it doesn't have to wait for membership modification but transfer niwDist(o1…n) back to host asynchronously. In host side, the membership modification is triggered when niwDist(o1…n) is received.

Finally, in Centroids-Recalculation phase, we don't offload it to MIC, because there are logic operations to sum the value of each dimension in a cluster, and it only occupies a small part of running time, especial in situation of large number of clusters. We separate these logic operations from Centroids-Recalculation, and execute them with membership modification as Centroids-Recalculation-1, because they share the same logic compare operation. In Centroids-Recalculation-2 phase, the new centroid is calculated by averaging the summation from Centroids-Recalculation-1.

3.3 Implementation

In this section, we describe the full implementation of the algorithm for MIC.

1) Vectorization and Data layout

In order to meet two vectorization requirements in section 2.4, we use the centroids-oriented strategy and layout all objects in the manner of Fig. 1(c). So, distances from all objects to centroid c can be calculated in vectorization way. There are d vectors, elem1…elemd, made up of dth dimension of all objects, and d scalar value, c1…cd, made up of dth dimension of centroid c. Then the distance vector, denoted as dist, of all objects to centroid c, is calculated as:

$$dist = \sqrt{\sum_{i=1}^{d}(elem_i - c_i)^2}$$

The length of dist and elem1…elemd are n, equals to the number of objects. As the MIC's SIMD instructions are 512-bit wide, every long vector operation is then divided into $(n/(\frac{512}{WidthOfElement}))$ short vector operations on MIC's VPUs.

There are two practical choices to vectorize highly parallel code on MIC. One way is auto-vectorization, which is achieved by the compiler without any code modification. But it requires the code to be suitable for efficient auto-vectorization. Another way is to explicitly call SIMD intrinsics, which is usually the most efficient way to use the SIMD capabilities of MIC coprocessor. But it needs some programming skills to use SIMD intrinsics explicitly.

2) Multi-Threading parallelism

The MIC coprocessor exposes many computational cores, each featuring 4-way SMT, providing developer hundreds of threads in total. To utilize all the threads is a key factor for parallel applications. Although developers can manage all the threads themselves, the managements of raw threads are a lot of work and it is very hard to maintain.

Fortunately, OpenMP, TBB, and Intel Cilk Plus are three parallel programming options that are available in Intel MIC coprocessor. In this paper, we use OpenMP to achieve thread-level parallelism for our K-Means algorithm on MIC coprocessor.

OpenMP is an API that supports shared memory parallel programming in C, C++ and FORTRAN on multi-platform. It consists of a set of compiler directives and library routines. The compiler directives are used to annotate loop bodies and code blocks for parallel execution and marking variables as local or shared as well. The runtime behavior can be controlled by environment variables. In this paper, we use a for compiler directive to execute the iterations of the loop concurrently. We also set thread schedule mode as static and thread affinity mode as scatter to control the runtime behavior. In static scheduling mode, the iterations will be partitioned in chunks which are allocated to the threads in a round-robin manner. In scatter thread affinity mode, the target is to make the best use of every core first. So, the first thread is combined to core-1, the second thread is combined to core-2 and so on.

Applying these technologies on vectorized K-Means algorithm, we can now port it on heterogeneous architecture of CPU and MIC. The pseudo code of Objects-Assignment is shown in Algorithm-4.

Algorithm 4. Obj.-Assign. On CPU + MIC

```
1:  initialize distOld(o_{1...n}) = {MAX}
2:  initialize distNew(o_{1...n}) = computeDist(o_{1...n}, c_1)
3:  for each c_i from 2 to k
4:     swap(distTmp(o_{1...n}), distNew(o_{1...n}))
5:  #pragma offload target(mic) ... signal(distNew)
6:  {                                                    MIC
7:     initialize distNew(o_{1...n}) = {0}
8:     for dim from 1 to d
9:  #pragma omp parallel for num_threads(*)
10:        for each o_i from 1 to n
11:           distNew(o_i)+= (o_i(dim) − c(dim))^2
12: #pragma omp parallel for num_threads(*)
13:    for each o_i from 1 to n
14:       distNew(o_i) = √distNew(o_i)
15: }
16:    for each o_j from 1 to n                          Host
17:       if distTmp(o_j) < distOld(o_j)
18:          membership(o_j) = i
19:          distOld(o_j) = distTmp(o_j)
20: #pragma offload_wait target(mic) wait(distNew)
21:       null
```

The "MIC" code block is offloaded to MIC coprocessor through an "offload" as clause of #pragma, while the "Host" code block runs on common CPU at host side. The overlap executing of these two blocks is achieved by using "signal" and "wait" as offload clauses of #pragma. When the program runs to line-5, it tells the runtime to

execute the "MIC" block on MIC, and gives the control back to CPU. So, the CPU thread goes on executing "Host" code block. When the program runs to line-20, it blocks the CPU thread until distNew(o1…n) is calculated on MIC and transferred back to host. It is possible that current distNew(o1…n), distances to ith centroid, be overwritten by next distNew(o1…n), distances to (i+1)th centroid, when it is still being read at host side. To avoid this overwritten, we define distTmp of the same size as distNew, and make a swap with distNew at the beginning of iterations for different centroids. This process doesn't introduce memory copy overhead just by swapping address.

In order to utilize the hardware resource of many cores, we use OpenMP to get a coarse parallel by spawning a number of threads on many cores. As shown in line-9 and line-12, all objects are split into chunks to be processed on different threads.

4 Performance Evaluations

In this section we present the experimental evaluations for vectorized K-Means algorithm and MIC coprocessor.

4.1 Experiment Setup

All experiments were conducted on a single server made up of Intel Xeon CPU and Intel MIC coprocessor connected to CPU through PCI-e bus.

The host side features two-way processors of Intel Xeon Processor E5-2670. Each processor consists of 8 physical cores with clock speed of 2.60GHz. Based on Intel Hyper-threading technology, that delivers two processing threads per physical core, the host side can spawn up to 32 threads each time. Moreover, the E5-2670 processor can execute vector instructions of AVX Instruction Set Extensions.

In the coprocessor side, we choose the first generation of Intel Xeon Phi product, codenamed Knights Corner (KNC). The Intel Xeon Phi card in our experiment provides in a single chip 57 cores with clock speed of 1.094GHz. Each core can execute four hardware threads in round-robin scheduling between instruction streams, and features a 512-bit wide vector processing unit. It uses the cache structure of per-core L1 32KB Instruction/Data cache and L2 512KB cache. All caches and the shared 8GB GDDR5 memory (up to 320GB/s) are fully coherent. This coprocessor can spawn up to 224 threads each time.

4.2 Dataset and Baseline

The 512-bit wide SIMD instructions of each core in MIC coprocessor can process sixteen 32-bit single-precision float or eight 64-bit double-precision double operations in one instruction. We produce sample objects, consisted of random 32-bit float as elements of an object. We change the parameters of means, number of dimensions as well as object-scale and evaluate the performance in different settings.

For the rest of the experiments, we take the scalar version of Lloyd's iterations as baseline. We evaluate the vectorization on CPU firstly. And then, we evaluate the performance of vectorized K-Means algorithm on MIC coprocessor. We also survey the scalability of MIC coprocessor as well as the overhead to use MIC coprocessor.

4.3 Basic Evaluation on CPU

In order to evaluate the vectorization performance of K-Means algorithm and the speed up achieved from MIC coprocessor, we compare the scalar and vector version of K-Means method on CPU first. The scalar version uses objects-oriented strategy in Objects-Assignment phase, while the vector version uses centroids-oriented strategy. Fig. 3 plots the time elapse of both strategies in Objects-Assignment phase. The experiments were conduct on a dataset consist of 100,000 1 to 20-dimensional objects. All the objects are assigned to 50-150 clusters. The iteration times are the same in two strategies with the same configurations of #dimension and k. Fig. 3 shows that the vectorized variation gets a speed-up of 2 in Objects-Assignment phase with different settings.

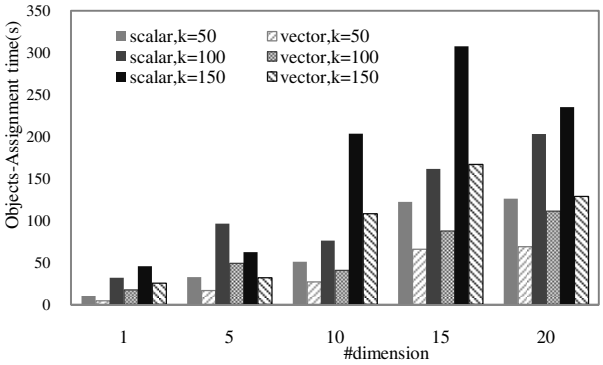


Fig. 3. Comparison of Objects-Assignment time between our vectorization version and scalar version on CPU

4.4 Computational Power and Scalability of MIC Coprocessor

Fig. 4 compares the computational power of MIC coprocessor to CPU on host. In this experiment, we take the vectorized K-Means method on CPU as baseline, and then compare the distances calculation time on both CPU and MIC coprocessor. The y-axis is the speedup to baseline on CPU. We use two vectorization ways on MIC coprocessor as shown in Fig. 4. Both the baseline and auto-vectorization version use compiler parameter "-O3" as optimization option for vectorization. In intrinsic version, we unroll the loop 4 times to use most of the vector registers. We also use prefetch intrinsic and broadcast intrinsic to make the most of cache resources. We observe that MIC coprocessor works faster than CPU to calculate distances unless objects are 1-dimensional, and get a maximum speed up of 1.5.

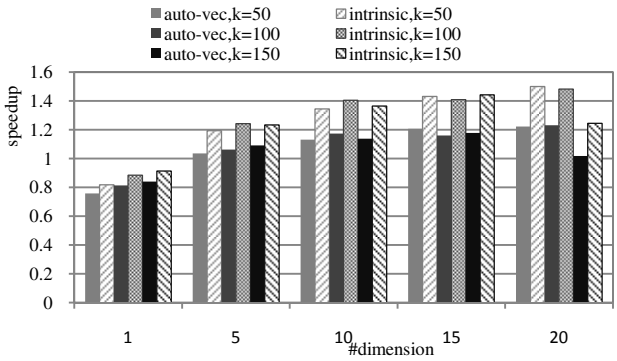


Fig. 4. Computational power comparison between MIC and CPU

Fig. 5 plots the scalability as we scale the number of threads in both MIC coprocessor and host. As mentioned above, the host features two physical E5-2670 CPUs, with each consisting of 8 cores. As E5-2670 CPU use Hyper-threading technology, the host supports 32 hard threads at most. The MIC coprocessor consists of 57 cores, with each supports 4 hard threads. One MIC coprocessor supports 224 hard threads at most on 56 cores, with one core used to run the uOS. All experiments were on a dataset of 5,000,000 10-dimensional objects, which were clustered into 50 clusters.

The experiment results shows that MIC coprocessor scales out near linearly before #thread 32, and gets a maximum speed up of 23 at 112-threads to 1-thread setting. The maximum speed up of CPU is 10.5 at 32-threads settings. There is a shock from 8-threads to 16-threads settings on CPU, because every 8 cores are on two different physical CPUs. We observe that in the same setting before 8-threads, host side performs little better than MIC coprocessor. This phenomenon is different from the computational power comparison. It might be influenced by parallel programming mode of OpenMP, and suggests optimization for parallel programming mode on MIC coprocessor.

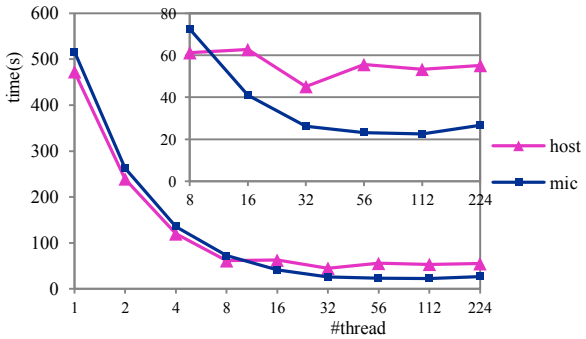


Fig. 5. Scalability of MIC coprocessor (the inset figure shows the explicit runtime of the outset figure at the configuration from 8 to 224)

4.5 Data Locality

To evaluate the influence of data locality on vectorized K-Means algorithm, we compare two data layout strategies in Fig. 1(b) and Fig. 1(c). All experiments are conducted on a dataset of 100,000 objects. We compare the Objects-Assignment time for both data layout strategies on MIC coprocessor. As the number of iterations varies in different parameter settings, we take the Objects-Assignment time of scalar variation on CPU as baseline, and compute the ratio of Objects-Assignment time to baseline as y-axis for both data layout strategies as shown in Fig. 6. We observe from the experiment results that the second data layout strategy gets a steady ratio of 0.5, meaning a speedup of 2 to baseline. While in the first data layout mode, the Objects-Assignment time equals to baseline when the dimension is 5. Since then, the Objects-Assignment time grows near linearly with the increase of dimension.

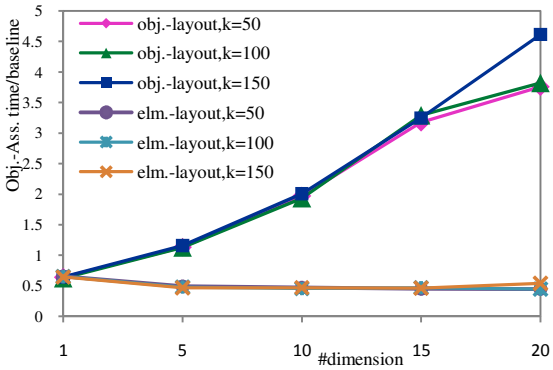


Fig. 6. Data locality in two data layout modes

4.6 Total Speed Up and Execution Time Breakdown

To test the total speedup achieved by this algorithm on heterogeneous architecture. We test the execution time breakdown firstly.

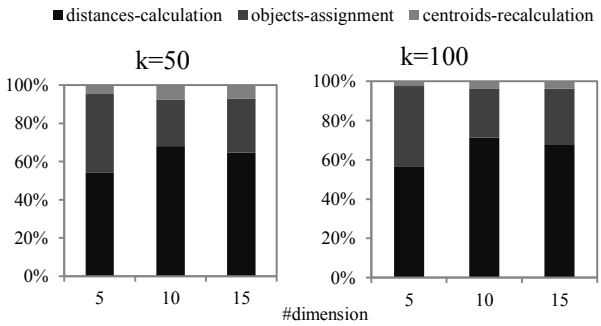


Fig. 7. Execution time breakdown of vectorized K-Means on CPU and MIC

Fig. 7 shows the time breakdown of three parts of the algorithm. The distances-calculation is finished on MIC, while the other two are on host. These experiments are conducted on the most optimized setting, meaning #thread is set to 112. From the Figure we observe that distances-calculation time is usually longer than objects-assignment, which inspires us to hide the objects-assignment from distances-calculation.

Fig. 8 shows the total speed up to the baseline of sequential Lloyd's iteration against 1,000,000 objects. The synchronous mode means distances-calculation and objects-assignment execute in sequential way, while asynchronous mode means they execute asynchronously. The ideal mode is based on a hypothesis that objects-assignment can be full hided from distances-calculation without overheads. The speed up peaks at 23.8 in the 100-k configuration and 15-dimension configuration.

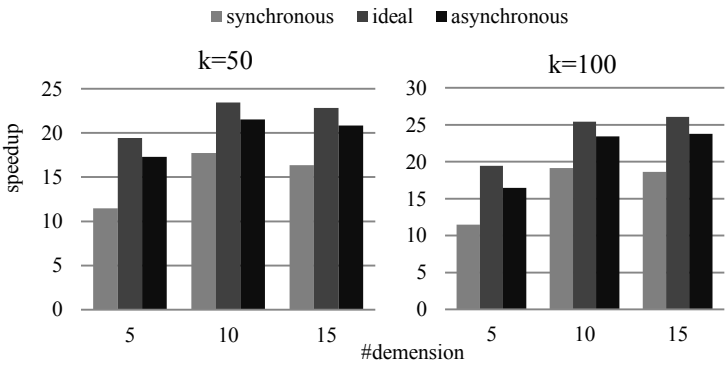


Fig. 8. Total speed up

4.7 Overheads

As previously noted, we cannot expect an ideal speed up. There are two main overheads at runtime.

Firstly, threads management might affect the performance. As mentioned in section 4.4, the single thread performance of MIC is better than CPU. But it is not the same when using OpenMP to manage threads on MIC and CPU. Moreover, the scalability decreases dramatically after 56-thread configuration.

Secondly, data transfer between host and MIC will affect the performance as shown in section 4.6. It is hard to achieve ideal asynchronous execution between host and MIC. Because the memory on MIC coprocessor is shared by hundreds of threads, the memory mapping technologies between host and MIC are worthy of further research.

5 Conclusion

To extract useful information from massive data and use it to facilitate our decision making are challenging scientific problems in big data technologies. The key to accelerate massive data processing is parallelism. Traditional strategies are to use more

hardware to divide massive data into small tasks, which can be executed parallel. But we think the essence is to explore data parallel to the most fine-grained level. In the context of heterogeneous architecture, we have already got the hardware support.

In this paper, we presented a vectorized K-Means algorithm that achieves both fine-grained and coarse grained parallelism. This algorithm further explores the data parallel at element-level of objects in vectorization manner. We then introduce a data layout mode that is suitable for this algorithm to improve data locality in shared memory.

Moreover, we implement the algorithm on heterogeneous architecture of CPU and MIC coprocessor. As newly designed coprocessor architecture, there will surely be a lot of following studies on Many Integrated Core architecture that supports wide SIMD instructions. Through this work, we make an early study of MIC coprocessor. We achieve desired speedup in real-world application as well as scalability on the Intel MIC architecture. From the implementation, we also get a strong impression of the simple programming mode of MIC coprocessor.

Acknowledgment. This work was supported by project (2013AA01A212) from the National 863 Program of China, project (61202121) from the National Natural Science Foundation of China and project (20114307120013) from the New Teachers' Fund for Doctor Stations, Ministry of Education of China.

References

1. Dayan, P.: Unsupervised Learning. The MIT Encyclopedia of the Cognitive Sciences
2. MacQueen, J.B.: Some Methods for classification and Analysis of Multivariate Observations. In: Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability, vol. 1, pp. 281–297. University of California Press, Berkeley (1967)
3. Wu, X., Kumar, V., Ross Quinlan, J., Ghosh, J., Yang, Q., Motoda, H., McLachlan, G.J., Ng, A., Liu, B., Yu, P.S., Zhou, Z.-H., Steinbach, M., Hand, D.J., Steinberg, D.: Top 10 algorithms in data mining. Knowl. Inf. Syst. 14, 1–37 (2007)
4. Lloyd, S.: Least Squares Quantization in PCM. IEEE Transactions on Information Theory 28(2), 129–136 (1982)
5. Arthur, D., Vassilvitskii, S.: k-means++: the advantages of careful seeding. In: SODA (2007)
6. Bahman, B., Moseley, B., Vattani, A.: Scalable K-Means++. Proceedings of the VLDB Endowment 5(7)
7. Ailon, N., Jaiswal, R., Monteleoni, C.: Streaming K-Means approximation. In: NIPS (2009)
8. Che, S., Boyer, M., Meng, J., Tarjan, D.: A performance study of general-purpose applications on graphics processors using CUDA. Journal of Parallel and Distributed Computing 68(10), 1370–1380 (2008)
9. Bai, H.-T., He, L.-L., Ouyang, D.-T., Li, Z.-S., Li, H.: K-Means on commodity GPUs with CUDA. In: World Congress on Computer Science and Information Engineering (2009)
10. Farivar, R., Rebolledo, D., Chan, E., Campbell, R.: A parallel implementation of K-Means clustering on GPUs. In: Proceeding of International Conference on Parallel and Distributed Processing Techniques and Applications (2008)

11. Mahout, http://mahout.apache.org/
12. Hadoop, http://hadoop.apache.org/
13. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)
14. Feldman, M.: TACC Steps Up to the MIC. HPCwire (April 21, 2011), http://www.hpcwire.com/hpcwire/2011-04-21/tacc_steps_up_to_the_mic.html
15. Nvidia. Oak Ridge National Lab Turns to NVIDIA Tesla GPUs to Deploy World's Leading Supercomputer. HPCwire (October 11, 2011), http://www.hpcwire.com/hpcwire/2011-10-11/oak_ridge_national_lab_turns_to_Nvidia_tesla_gpus_to_deploy_world_s_leading_supercomputer.html
16. Intel. Introducing Intel Many Integrated Core Architecture. Press release (2011), http://www.intel.com/technology/architecture-silicon/mic/index.htm
17. Seiler, L., et al.: Larrabee: A Many-Core x86 Architecture for visual Computing. ACM Trans. Graphics 27(3), 18:1–18:15 (2008)
18. Saule, E., Catalyurek, U.V.: An early evaluation of the scalability of graph algorithms on the Intel MIC Architecture. In: IEEE IPDPSW (2012), doi:10.1109
19. McFarlin, D.S., Arbatov, V., Franchetti, F., Puschel, M., Zurich, E.: Automatic SIMD vectorization of fast fourier transforms for the Larrabee and AVX Instruction sets. In: ACM ICS (2011)
20. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU Computing. Proceedings of the IEEE 96(5) (May 2008)
21. Aloise, D., Deshpande, A., Hansen, P., Popat, P.: NP-hardness of Euclidean sum-of-squares clustering. Machine Learning 75(2), 245–248 (2009)
22. Henretty, T., Stock, K., Pouchet, L.-N., Franchetti, F., Ramanujam, J., Sadayappan, P.: Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 225–245. Springer, Heidelberg (2011)
23. He, B.S., Govindaraju, N.K., Luo, Q., Smith, B.: Efficient Gather and Scatter operations on graphics processors. In: Proceeding of the 2007 ACM/IEEE Conference on Supercomputing, p. 46. ACM (2007)

Towards Eliminating Memory Virtualization Overhead

Xiaolin Wang[1], Lingmei Weng[1], Zhenlin Wang[2], and Yingwei Luo[1]

[1] Peking University
[2] Michigan Technological University
{wxl,lyw}@pku.edu.cn, zlwang@mtu.edu

Abstract. Despite of continuous efforts on reducing virtualization overhead, memory virtualization overhead remains significant for certain applications and often the last piece standing. Hardware vendors even dedicate hardware resources to help minimize this overhead. Each of the two typical approaches, shadow paging (SP) and hardware assisted paging (HAP), has its own advantages and disadvantages. Hardware-dependent HAP eliminates most VM exits caused by page faults, but suffers from higher penalties of TLB misses. On the other hand, the software-only approach, SP, enjoys shorter page walk latencies while paying for the VM exits to maintain the consistency between the shadow page table and the guest page table. We observe that, although HAP and SP each holds its ground for a set of applications in a 32-bit virtual machine (VM), SP almost always performs on a par with or better than HAP in a 64-bit system, which steadily gains its popularity. This paper examines the root cause of this inconsistency through a series of experiments and shows that the major overhead of shadow paging in a 32-bit system can be substantially reduced using a customized memory allocator. We conclude that memory virtualization overhead can be minimized with software-only approaches and therefore hardware-assisted paging might no longer be necessary.

Keywords: Memory virtualization, shadow paging, hardware assisted paging.

1 Introduction

Server consolidation, performance isolation and maintenance advantages of system virtualization makes it a backbone technique for data center and cloud computing. However, virtualization brings an additional layer of abstraction that can result in performance penalty. Previous studies in both software and hardware have successfully minimized the virtualization overhead for CPU, I/O and network [1, 2, 3]. Recent research proposes a selective approach to mitigate memory virtualization overhead by taking advantage of hardware support and software page table management [4]. Enhancements of hardware assistance are discussed in [5, 8].

Each of the two conventional memory virtualization approaches, shadow paging (SP) and hardware assisted paging (HAP) such as Extended Page Table (EPT) and Nested Page Table (NTP) [6, 7, 8], has its own advantage. Figure 1 illustrates the key mechanism of SP and HAP. SP is a pure software approach. In implementation, VMM maintains a shadow page table which maps virtual addresses directly to machine

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 295–306, 2013.
© Springer-Verlag Berlin Heidelberg 2013

addresses while a guest OS maintains its own virtual to physical page table. Since the shadow table is the actual page table loaded into the MMU, operations like reloading CR3, invalidating TLB entries and modifying page table result in multiple VM exits in order to synchronize between the two page tables. To avoid expensive VM exits, HAP introduces a two-dimensional page table by extending the guest pages table with EPTs or NPTs, which complete the translation from guest physical address to machine address. When loaded into the MMU, a guest OS can convert a guest virtual address to machine address through guest and extended page tables. Therefore, guest page fault, CR3 change and invalidation of TLB entries can be handled within the guest OS itself without triggering VM exits. However, the additional layer of translation adds overhead to the page walk on a TLB miss, which becomes remarkable in 64-bit system with multi-level page table.

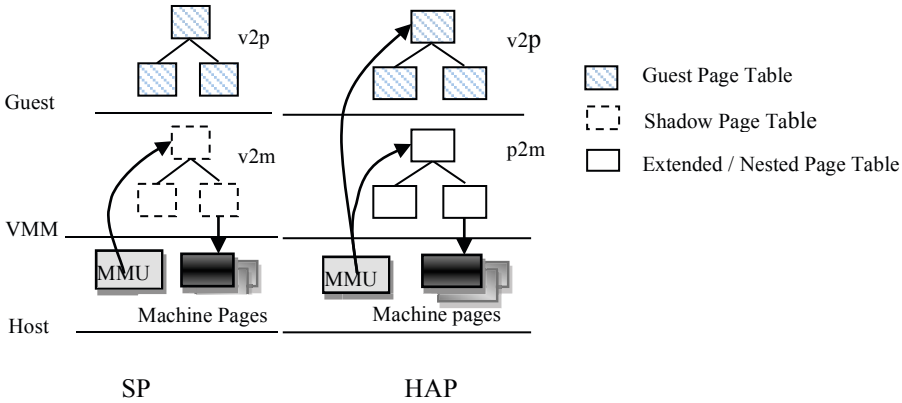


Fig. 1. Memory virtualization

Figure 2 shows that neither HAP nor SP can be a definite winner in a 32-bit system. HAP edges SP with an up to 60% performance gap for gcc while losing 20% to SP for mcf. Interestingly, for a 64-bit system as shown in figure 3, despite that there is no noticeable gap between the two mechanisms in over half of the cases in SPEC CPU 2006, SP performs better in other cases.

This paper investigates empirically the performance discrepancy of SP and HAP between a 32-bit and a 64-bit system. We evaluate the system behavior of VM with SP mode and HAP mode, respectively, and find that the major overhead of SP stems from VM exits, which in turn result from minor faults in Guest OS. We observe that, although a massive number of minor faults have a minimal impact in the native system and HAP, it causes a notable penalty for SP. Further experiments reveal that over 90% minor faults are introduced by virtual memory management in system libraries. We propose an adaptive approach that tunes memory management libraries to eliminate minor faults. Our results show that SP can match the native performance with the software improvement. Although it is still too early to drop HAP as our results are limited to C/C++ applications that use the glibc libraries, we conclude that a software approach to eliminating memory virtualization overhead is apparently more cost effective and promising.

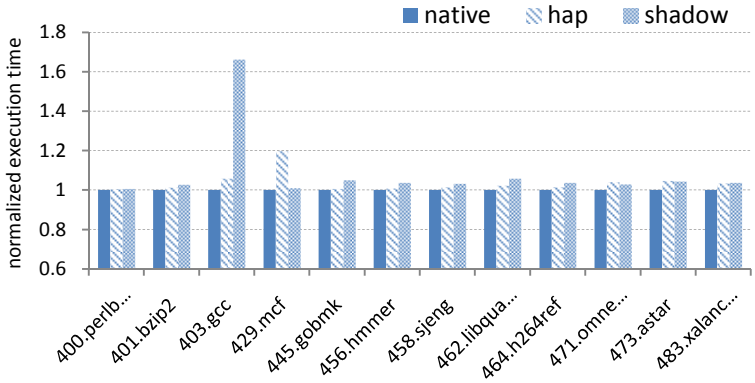


Fig. 2. 32-bit VM normalized execution time

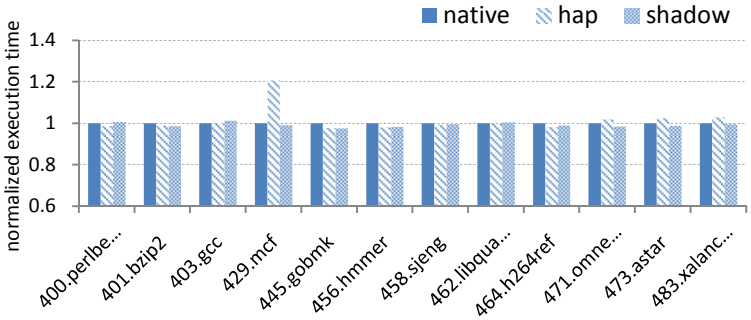


Fig. 3. 64-bit VM normalized execution time

2 Origin of Memory Virtualization Overhead

To understand why SP and HAP behave so differently in a 32-bit VM versus a 64-bit VM, we profile the systems by collecting the number of VM exits and TLB misses, noting that longer latency is required to fill TLB entry for a 64-bit VM in HAP mode because of multi-level page table, and HAP eliminates VM exits caused by page faults, CR3 loading and TLB entry invalidation.

We use hardware performance counters to track TLB misses for all applications. The number of TLB misses of mcf is one magnitude higher in both 32-bit and 64-bit, which explains the significant of performance gain of SP over HAP.

Table 1 lists the number of VM exits. The number of VM exits of gcc is one magnitude higher than all other benchmarks in the 32-bit VM while the gap disappears in the 64-bit VM. We thus conclude that the performance benefit of HAP stems from its ability to reduce certain VM exits. However, when the number of VM exits becomes insignificant, the advantage of HAP no longer exists.

Table 1. VM exits (millions)

Benchmark	32-bit shadow	32-bit hap	64-bit shadow	64-bit hap
400.perlbench	4.493	1.148	2.002	1.091
401.bzip2	10.107	1.712	3.588	1.413
403.gcc	**190.038**	0.857	7.043	1.012
429.mcf	3.004	0.834	2.070	1.094
445.gobmk	2.112	1.353	1.656	1.241
456.hmmer	3.033	2.458	2.253	1.928
458.sjeng	2.151	1.443	1.710	1.324
462.libquantum	3.998	1.570	2.150	1.312
464.h264ref	4.216	2.065	2.095	1.607
471.omnetpp	1.431	0.921	1.232	0.962
473.astar	3.508	1.335	1.965	1.230
483.xalancbmk	2.073	1.294	1.276	1.257

When comparing the VM exits of the 32-bit VM with the 64-bit VM under SP mode, we observe that all benchmarks yield a lower number of VM exits in the 64-bit VM. The VM profile tells us that major sources of VM exit events are EXCEPTION_NMI, EXTERNAL_INTERRUPT, PENDING_VIRT_INTR, CPUID, HLT, INVLPG, CR_ACCESS, IO_INSTRUCTION and APIC_ACCESS, where EXCEPTION_NMI and INVLPG are two events that dominate the differences between 32-bit and 64-bit. EXCEPTION_NMI and INVLPG correspond to page fault handling and TLB invalidation in SPEC CPU 2006, respectively. These differences can stem from operating system, application memory footprint, runtime library or their interactions. We examine these possibilities in this section.

2.1 Experimental Setup

We use SPEC CPU2006 to explore the origin of the excessive VM exits in the 32-bit VM. We conduct our experiments on an Intel CORE i5 machine with 4 GB of physical memory and two 2.80GHz cores. We run Xen 4.1.2 [13] as the VMM, Linux 2.6.38 with 1 core and 2 GB of physical memory as domain 0, and Linux 2.6.32 with 1 core and 2 GB of physical memory as domain U. For comparison, the native platform also uses 1 core and 2 GB of physical memory.

2.2 Memory Footprint

A 64-bit application and a 32-bit one can demonstrate different memory access pattern even when the source is the same. In Intel X86 architecture, the compiler can reduce memory accesses by using more registers in 64-bit mode. On the other hand, some data structures, such as pointers, might require more memory. When monitoring the memory footprint of the SPEC integer benchmarks, we notice that only mcf shows notable working set size changes, whose working set is doubled in 64-bit as mcf is

pointer intensive[9,10]. Slight increase of memory footprint is irrelevant to the decrease of VM-exits. We conclude that changes in memory footprint cannot explain the differences of VM exits.

2.3 OS Impact

Since most of the VM exits are caused by page faults in a VM, we further investigate the sources of page faults. We find out the VM exit resulting page faults are predominantly minor faults, which makes sense considering the sufficient physical memory we allocate for the VM. Our first guess is the minor faults could be attributed to the different OS behaviors in 32-bit mode and 64-bit mode although both of them are built from the same source code.

To verify this conjecture, we run the benchmarks in a new mode, *blend mode*, where we experiment with 32-bit applications in the 64-bit guest OS. We then compare with existing results: 32-bit application in the 32-bit system (32-bit mode) and 64-bit applications in the 64-bit system (64-bit mode). We observe that the number of page faults under blend mode is roughly the same as 32-bit mode as shown in Table 2 for gcc. The other benchmarks behave the same. We thus conclude the page fault discrepancy must result from user space and the OS impact is minimal.

Table 2. Minor faults statistics of 403.gcc

403.gcc	**32-bit shadow**		**64-bit shadow**		**Blend mode**	
inputs	**# million**	**heap %**	**#million**	**heap %**	**# million**	**heap %**
166.i	0.28	99.67	0.20	99.47	0.28	99.62
200.i	0.27	99.67	0.10	99.03	0.27	99.62
typeck.i	0.34	99.73	0.29	99.63	0.34	99.68
cp-decl.i	1.85	99.95	0.18	99.42	1.85	99.94
expr.i	3.61	99.97	0.32	99.67	3.61	99.97
expr2.i	0.52	99.82	0.50	99.78	0.52	99.79
g23.i	4.79	99.98	0.41	99.72	4.79	99.97
s04.i	9.03	99.99	0.61	99.80	9.03	99.99
scilab.i	0.04	97.80	0.03	97.34	0.04	97.44
sum	20.77	99.96	2.68	99.63	20.77	99.95

2.4 Runtime Library

To identify the source of minor faults at user space, we attempt to pinpoint the exact location in code by analyzing the type and address of page fault. Table 2 shows the numbers of minor faults and their distribution when gcc runs on 32-bit mode, 64-bit mode, and blend mode, all with shadow paging. Note that most of the minor faults are located in heap for gcc (similar results for other applications). We can be certain that the minor fault discrepancy come from the memory management library.

This conclusion is also supported by a study by Phillip Ezolt et al. who pointed out that minor faults are the source of performance difference between Tru64 Unix and Linux/Alpha [11].

Both the 32-bit and 64-bit systems use the same glibc that implements the Doug-Lea allocator [12]. The difference locates at the two tunable parameters *mmap_threshold* and *trim_threshold*, which are set to different default values in the two systems. *mmap_threshold* is the request size threshold for using *mmap()* to service a request. Requests of at least this size will be serviced via *mmap()*. In Linux, the *mmap()* support routine is expensive because it needs to zero out the mapped space and restart corresponding page table mappings. *trim_threshold* is the maximum amount of unused top-most heap memory to keep by a process. The top-most available chunk is released back to the system if its size is larger than the *trim_threshold*. Both thresholds are dynamically adjusted when the application frees memory that was allocated via *mmap()*. *mmap_threshold* is set to the size of the recently unmapped space if the size no bigger that DEFAULT_MMAP_THRESHOLD_MAX (the default upper bound of *mmap_threshold*) and bigger than the current *mmap_threshold*. *trim_threshold* is always maintained as twice of *mmap_threshold* when an adjustment takes place.

For a 32-bit application, the default upper bound of *mmap_threshold* is 512K, while for a 64-bit application it is 32M. A request size between 512K and 32M will likely trigger two different allocation processes. The request of this size in a 32-bit application will be likely served by more expensive *mmap()*, while the same size request from a 64-bit application is likely served by the chunks in free bins which can reuse the existing page table entries and do not trigger minor faults [12].

```
            brk          allocated  area

malloc a   218718208       218718208 : 218996736
malloc b   218996736       218996736 : 219275264
free a
free b
malloc c   218718208       218718208 : 218996736
malloc d   218996736       218996736 : 219275264

.............
```

With relatively smaller *mmap_threshold* and *trim_threshold*, by observing the glibc allocations for gcc, we find a typical operation pattern where trim operations release the top-most chunk back to the system and malloc operations reallocate the same virtual memory region later. Above is a snapshot of trim operation effects. The first column stands for the memory operation; the value of heap top address is in the second column; and the third is the allocated area. After malloc b, the heap top address is 219275264, but before malloc c, due to free a and free b, whose combined size is bigger than the *trim_threshold*, the heap brk is trimmed to 218718208. Then the same virtual space is allocated to c that needs to be mapped again. Accesses to c will trigger minor faults.

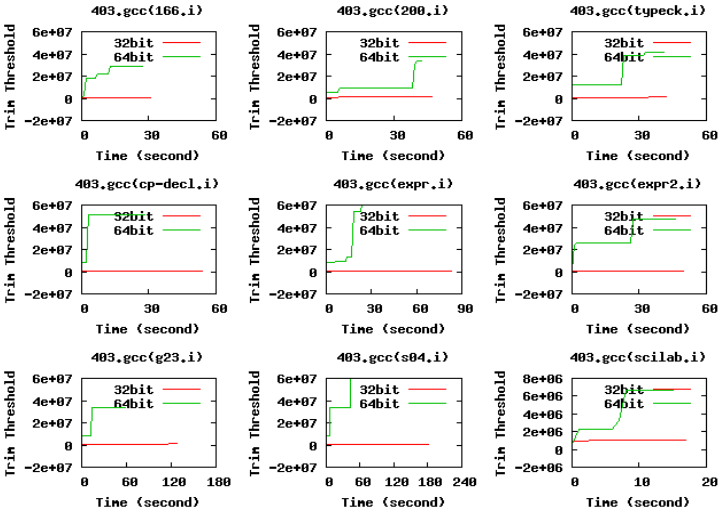


Fig. 4. Trim threshold

Figure 4 shows the trends of trim threshold values for gcc. The trim threshold for 32-bit stays flat while it quickly increases in 64-bit.This trends and the aforementioned example partially explain the minor fault difference between 32-bit and 64-bit. A bigger threshold that prevents releasing a and b will avoid these faults.

We also track the number of memory management related system calls by glibc. Table 3 shows that system operations invoked from glibc, such as mmap, sbrk, and unmap, are many more in 32-bit than in 64-bit. Due to the relatively smaller thresholds, a 32-bit application usually allocate and free the same region back and forth, leading to more calls to mmap, unmap, and sbrk.

Table 3. Glibc system calls

403.gcc	**mmap**		**sbrk**		**unmap**	
inputs	**32-bit**	**64-bit**	**32-bit**	**64-bit**	**32-bit**	**64-bit**
166.i	340	45	526	678	317	20
200.i	900	37	1,334	1,043	878	14
typeck.i	1,599	48	804	839	1,577	24
cp-decl.i	95	35	70,294	788	73	12
expr.i	97	51	119,431	1,226	75	28
expr2.i	142	56	1,910	2,542	119	33
g23.i	1,261	43	106,539	2,828	1,238	19
s04.i	124,004	56	960	1,054	123,979	32
scilab.i	80	37	478	482	62	14
sum	128,518	408	302,276	11,480	128,318	196

To further verify the impact of the thresholds, we set environment variables M_TRIM_THRESHOLD_ to -1 and M_MMAP_MAX_ to 0 to forbid the system libraries from returning dynamically allocated memory to OS and thus minimize

heap-related minor faults. Table 4 compares the results for gcc. With disabled thresholds, the number of minor faults is reduced by 95%. The other applications in SPEC2006 are also reduced to different extent. However, simply disabling the threshold is not a good idea as it increases physical memory pressure and could result in more hard faults that will significantly drag down the performance. Section 3 proposes a software approach that can effectively reduce minor faults without increasing memory pressure.

Table 4. Minor faults with disabled thresholds

403.gcc	**minor fault**	
inputs	**original**	**disabled thresholds**
166.i	283,502	59,397
200.i	278,109	45,850
typeck.i	343,063	107,834
cp-decl.i	1,853,783	77,391
expr.i	3,616,901	105,113
expr2.i	528,185	144,304
g23.i	4,791,573	210,420
s04.i	9,036,055	237,832
scilab.i	40,238	17,200
sum	20,771,409	1,005,341

3 Software Approach

3.1 Adaptive Trim Threshold

As analyzed in Section 2, excessive minor faults of 32-bit applications are caused by the selection of thresholds. A small trim threshold can lead to fluctuations of the heap top. The result is that the library returns the top chunk of heap and asks it back immediately. To eliminate the unnecessary operations and the resulting minor faults, we propose to adaptively increase the trim threshold when the heap top thrashing is observed.

Our implementation introduces a new parameter, *last_trim*, to keep track of the most recent break or heap top address before the most recent trimming attempt. On a free, if the size of the unused top-most memory is larger than *trim_threshold*, which suggests trimming, we compare the current break address with *last_trim*. If the current break address is larger than or equal to *last_trim*, which means that the memory region released by last trimming has been reused again, we adjust the trim threshold with an increment of *trim_threshold*. The current trimming is determined by the new threshold. Otherwise, we keep the current trim threshold intact to allow the top chunk to be released. In this case, the new break will go down further, demonstrating continuous memory release actions.

3.2 Delayed Unmap

Another source of minor faults stems from the mmapped space that reuses the recently unmapped area. The *mmap_threshold* is designed with the assumption the large chunk of allocated memory often lives long and thus is more persistent. However, in the 32-bit library, the *mmap_threshold* is set small considering the limited virtual address space. An unmapped area often gets mapped again, yielding unnecessary minor faults because the remapped area uses the previously unmapped physical pages.

To make up for the deficiency of superfluous *mmap* and *unmap* in 32-bit applications, we propose a new *mmap/unmap* scheme, called delayed unmap. Delayed unmap does not change *mmap_threshold* and thus will not increase the pressure to the main heap. When a chunk of memory is to be unmapped, we do not unmap it directly. Rather, we keep track of this chunk in a pointer called *delayed_unmap_chunk*. To simplify the algorithm, we only track the largest chunk. If the current chunk is bigger than the *delayed_unmap_chunk*, the delayed chunk is unmapped and the current chunk is marked as delayed. When an allocation request exceeds the *mmap_threshold* and thus needs to be mmaped, we check if the request can be satisfied by the delayed chunk. If so, it is served right after memseting the allocated region to emulate the zeroing out memory operation of unmap. The remaining chunk, if any, is still reserved in the *delay_unmap_chunk* with updated size and location information. If a reserved chunk cannot satisfy *mmap* requests for more than five times, it is released, which rarely happens in our experiments.

4 Experiment Evaluation

We implement our adaptive threshold and delayed mmap in glibc and evaluate its impact on the 32-bit systems. In the results, *shadow simple* stands for disabled thresholds in glibc to forbid returning dynamic memory to OS, while *shadow adaptive* is our implementation.

Figure 5 shows the normalized VM exits of SP when the adaptive approach is applied or the thresholds are disabled. The performance improvements as shown in Figure 6

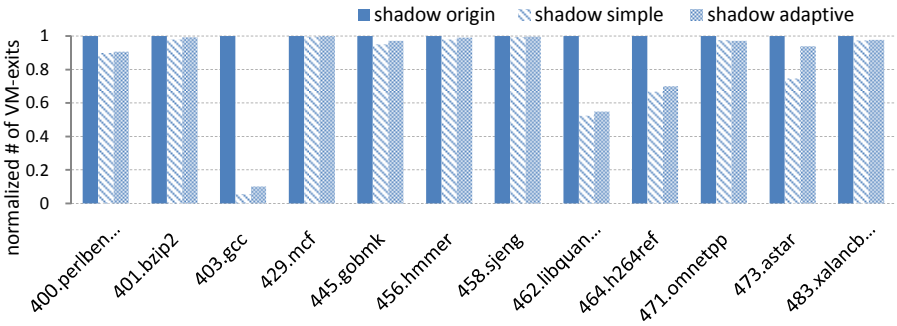


Fig. 5. Normalized times of VM-exits

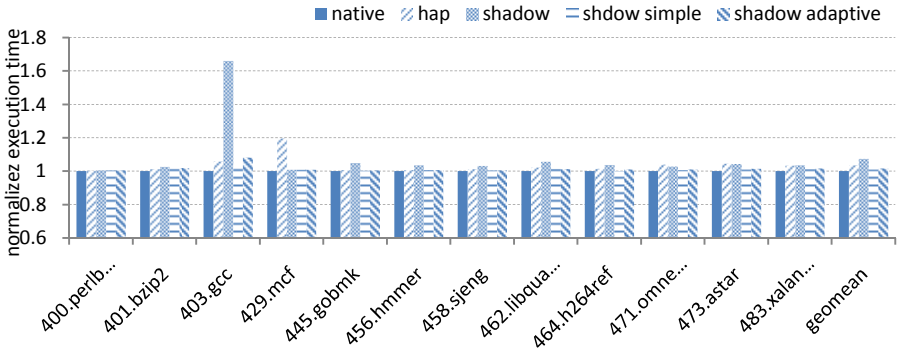


Fig. 6. Normalized Execution time

correlate well with the reduction ratios of VM exits. Note that gcc , libquantum and h264ref all have remarkable VM exits reduction and thus performance improvement. It shows the normalized execution time of SPEC INT2006 in VM with HAP, SP, SP with disable thresholds which serves an upper bound for software approaches, and SP plus delayed unmap and adaptive threshold.

Our software approach essentially treats the pathological case of gcc. It also brings notable improvement over SP for several other benchmarks, namely 3% for h264ref, and 5% for libquantum. The software approach brings SP to within 2% of the native performance, on average, compared to 8% before and %4 for HAP.

Figures 7 and 8 show the physical memory footprints of different memory management schemes under SP mode for gcc and libquantum, two representative cases, on 32-bit system. Note that simply disabling the thresholds will result in significant physical memory footprint increase and thus increase the memory pressure. Our software method does not cause notable increase of footprints.

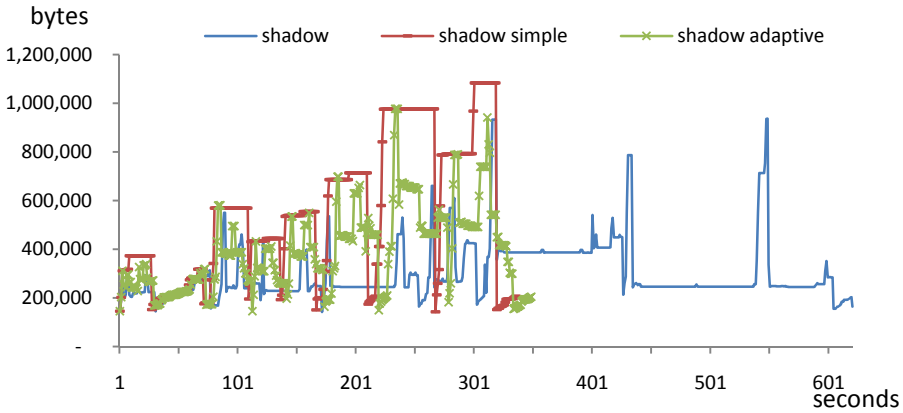


Fig. 7. Footprint of gcc

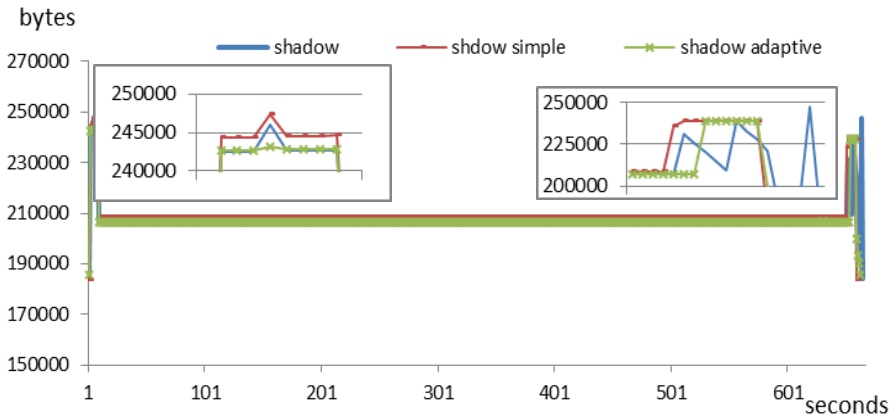


Fig. 8. Footprint of libquantum

5 Conclusion

This paper analyzes the overhead of memory virtualization and finds that the major overhead of shadow paging comes from minor page faults in 32-bit systems that mostly disappear in 64-bit systems. Through a series of experiments, we show that the minor faults can mostly be eliminated with careful tuning of glibc memory management thresholds. We propose a software solution that can adaptively adjust the thresholds and reuse the unmapped area to minimize minor faults. Our experimental results show that, with the software scheme, shadow paging can exhibit close to native performance. On the other hand, HAP cannot avoid its penalty for long page walk and thus is less effective.

Acknowledgements. This work is supported by the National Natural Science Foundation of China (Grant No. 61170055, 61272158 , 61232008), the National High Technology Research 863 Program of China (2012AA010905), the Research Fund for the Doctoral Program of Higher Education of China (20110001110101), the Special Foundation of Industry Development for Biology, Internet, New Energy and New Material of Shenzhen (JC201104210107A). Zhenlin Wang is also supported by National Science Foundation Career (CCF0643664).

References

1. Adams, K., Agesen, O.: A comparison of software and hardware techniques for x86 virtualization. ACM SIGOPS Operating Systems Review, 2–13 (2006)
2. Santos, J.R., et al.: Bridging the gap between software and hardware techniques for i/o virtualization. In: Proceedings of the USENIX 2008 Annual Technical Conference (2008)
3. Menon, A., Cox, A.L., Zwaenepoel, W.: Optimizing network virtualization in Xen. In: USENIX Annual Technical Conference, pp. 15–28 (2006)

4. Wang, X., et al.: Selective hardware/software memory virtualization. ACM SIGPLAN Notices, 217–226 (2011)
5. Barr, T.W., Cox, A.L., Rixner, S.: Translation caching: skip, don't walk (the page table). ACM SIGARCH Computer Architecture News, 48–59 (2010)
6. Bhargava, R., et al.: Accelerating two-dimensional page walks for virtualized systems. ACM SIGARCH Computer Architecture News 36(1), 26–35 (2008)
7. Gillespie, M.: Best Practices for Paravirtualization Enhancements from Intel® Virtualization Technology: EPT and VE-d (June1, 2009), `http://software.intel.com/en-us/articles/best-practices-for-paravirtualization-enhancements-from-intel-virtualization-technology-ept-and-vt-d/`
8. Ahn, J., Jin, S., Huh, J.: Revisiting hardware-assisted page walks for virtualized systems. In: 2012 39th Annual International Symposium on Computer Architecture (ISCA), pp. 476–487. IEEE (2012)
9. Zhao, W., Wang, Z., Luo, Y.: Dynamic memory balancing for virtual machines. ACM SIGOPS Operating Systems Review 43(3), 37–47 (2009)
10. Gove, D.: CPU2006 working set size. ACM SIGARCH Computer Architecture News 35(1), 90–96 (2007)
11. Ezolt, P.: A study in Malloc: A case of excessive minor faults. In: Proceedings of the 5th Annual Linux Showcase and Conference (2001)
12. Doug Lea, `http://g.oswego.edu/dl/html/malloc.html`
13. Barham, P., et al.: Xen and the art of virtualization. ACM SIGOPS Operating Systems Review 37(5), 164–177 (2003)

An Improved FPGAs-Based Loop Pipeline Scheduling Algorithm for Reconfigurable Compiler

Zhenhua Guo, Yanxia Wu*, Guoyin Zhang, and Tianxiang Sui

College of Computer Science and Technology,
Harbin Engineering University, Harbin, China
{hrbeu.guozhenhua,wuyanxia98621}@gmail.com

Abstract. Reconfigurable compilers have shown significant promise in the field of reconfigurable computing, and pipeline scheduling algorithms are typically concerned with improving iteration performance or saving the resources. However, the lack of loop pipeline scheduling algorithm for reconfigurable systems hampers the widespread adoption of fine-grained reconfigurable compilers. This paper presents an improved FPGAs-based loop pipeline scheduling algorithm and has realized it in ASCRA (Application-Specific Compiler for Reconfigurable Architecture) compilation framework. In FPGAs-based loop pipeline scheduling algorithm, the adequate consideration of hardware operation logic delay can save the resources of pipelining and ensure the performance of reconfigurable systems. Both of iterations with carried dependencies and without carried dependency have been considered. The preliminary experiment results show that it can economize more than 20% of the register resources by combining the adjacent pipeline stages without influencing the performance, and the algorithm is feasible for the other fine-grained reconfigurable compilers.

Keywords: Reconfigurable compiler, FPGAs, Pipelining algorithm, loop, LLVM.

1 Introduction

The manufacturing technique of modern field programmable gate arrays (FPGA) has prompted the supercomputing industry to develop high-performance reconfigurable computers that combine general-purpose processors (GPPs) with FPGAs. For general-purpose, reconfigurable compilers have also been developed, which allows scientists and engineers to create FPGA-based reconfigurable computing systems using high-level language (HLL) rather than hardware description language (HDL) [1]. The loop pipeline techniques used in the hybrid platform based on GPPs-FPGAs can take full advantage of parallelism to improve the data processing capacity. Therefore, the researching of efficient loop pipeline scheduling algorithm is necessary to reconfigurable compilers.

* Corresponding author.

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 307–318, 2013.
© Springer-Verlag Berlin Heidelberg 2013

At present, many reconfigurable compilers can generate pipelined architecture on FPGAs, and most of them use the ASAP (As Soon As Possible) scheduling and directed pipelining division method to insert nodes in the LDDG (loop data dependence graph) to pipelines [2–5]. The directed pipelining division method puts nodes that have the same height in the LDDG into the same pipelining stage. This approach is simple, but did not consider the impact of time and resources for pipelining scheduling.

Reference [6] proposes a pipeline stages combined scheduling algorithm. It can combine the adjacent two or more pipeline stages into a single relatively long delay pipeline stage to reduce pipeline stages and optimize pipeline without increasing the pipeline maximum delay. But the pipeline scheduling algorithm does not consider the feedback-directed in the LDDG and the performance of scheduling pipeline circuit is still not satisfied. Reference [7] proposes an optimized pipeline scheduling algorithm on the basis of reference [6], and it has considered how to optimize the maximum clock frequency and can be more effective for pipeline scheduling. But the method of achieving the higher frequency by dividing the multiplication operation is not suitable for the fine-grained reconfigurable compilers.

This paper presents a FPGAs-based loop pipeline scheduling algorithm for fine-grained reconfigurable compilers, and this algorithm is designed in the hardware implementation module of the ASCRA compiler [8]. This paper is organized as follows. Section 2 presents an overview of ASCRA. On the basis, Section 3 presents the FPGAs-based loop pipeline scheduling algorithm. Section 4 presents the preliminary experiment results and the analysis. Section 5 concludes the paper.

2 Overview of ASCRA

The input for ASCRA consists of an algorithm specification written in C language. The ASCRA compiler framework uses the LLVM front-end to convert the input C program to LLVM IR (Intermediate Representation) with selected optimizations that can be applied for the RAU (Reconfigurable Acceleration Units). LLVM is an open-source project designed by University of Illinois [9]. It is not essentially a compiler, but rather a compiler infrastructure. The LLVM, based on GCC front-end or Clang front-end, has its own code optimization system. ASCRA is an reconfigurable compiler based on LLVM. ASCRA compiler can extract compute-intensive loops of ANSI C programs and compile C subset into VHDL running on a tightly-coupled RAU automatically. Then, the selected kernel can be mapped by IR to VHDL transformation module. The output files of ASCRA compiler consist of C program, VHDL Code and the interface files of VHDL and C. The target architecture consists of a single general-purpose processor and on-chip coprocessor based on FPGA to improve the performance of general purpose applications.

Figure 1 outlines the flow of processing information in the ASCRA environment. The target architecture for ASCRA consists of a single GPP (Microblaze)

and RAU (FPGAs) towards improving the performance of general purpose applications. Fast Simplex Link (FSL) bus is adopted as communication channels between the MicroBlaze and FPGA in the platform.

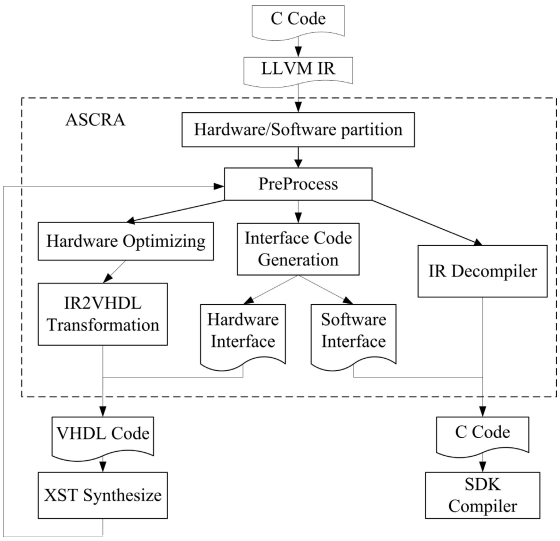


Fig. 1. ASCRA Compilation Flow Outline

As shown in figure 1, the process of each module in the ASCRA is as follows.

(1) Hardware and Software Partitioning. Using the profiling and some computing templates and memory templates which can be adapted (automatically) to suit the application, this module could select the kernels that make the best trade-off between resource allocation and performance in IR for hardware automatically.

(2) Preprocess. This module transforms the selected kernels that composed of a series of BBs (Basic Blocks) to function. And this module also is responsible for generating the hardware/software interface auxiliary information file and the communication function between hardware and software.

(3) Optimization for Hardware. LLVM IR is a target independent SSA (Static Single Assignment) language that uses RISC-like instructions. The optimizations for software do not consider the FPGA features. Therefore its necessary to apply dedicated optimizations to LLVM IR, which are suitable for hardware. The optimizations include: bit-width analysis, illegal name renaming, ram memory access optimization, basic block partitioning according to RAM access behavior, operation parallelization and so on.

(4) IR to VHDL Transformation. This module can transform the IR function into VHDL. According to characteristics of accelerated program, IR to VHDL transformation module transforms program to be accelerated into VHDL with loop pipelining or systolic array architecture.

(5) Interface Code Generation. This module can generate the interface program for software and hardware based on the hardware/software interface auxiliary information file.

(6) IR to C Decompiler. The module transforms the modified IR into C program. The functions transformed into VHDL are commented and the statements and calling information of communication bus is inserted in the modified IR.

This paper presents an advanced loop pipeline algorithm for the optimization for hardware module, and can help to generate the efficient hardware for FPGA.

3 Pipeline Scheduling Algorithm

The ASAP scheduling algorithm has been widely used for loop pipeline partitioning in the reconfigurable compilers, and need set the parameters manually to guide the classification of pipeline [10–12]. Based on the factor of operator delay in the FPGA, this paper extends the work by seeking a compromise between the ASAP and ALAP (as late as possible) scheduling, and presents a FPGA-based loop pipeline scheduling algorithm that can generate the pipelined hardware architecture automatically.

In this paper, the iteration is divided into two groups. Firstly, we consider loops which do not have loop carried dependencies. Finally, we deal with the loops which have feedback-directed edges in the LDDG.

As follows, the original loop code which do not have loop carried dependencies is shown, and figure 2 shows the DFG (data flow graph) generated by the loop body which is scheduled with ASAP algorithm. Although the result of ASAP can improve the performance of reconfigurable system by pipelining the hardware, the ASAP scheduling algorithm isnt perfect for the design of FPGA. There are some principles to improve the efficiency of the pipeline scheduling algorithm which is defined by this paper.

```
for( i=0; i<MAX; i++ ){
    p=x[i]+x[i-5];
    r=x[i]*s;
    if( p>r ){
        y[i]=p*(r-r*t1+q)&0xF;
        z[i]=r*t1*p&0xF;
    }
    else{
        y[i]=(t1-r+s)*q&0xF;
    }
}
```

Principle 1. ***(Maximum Frequency Principle).*** *To speed up the reconfigurable system, the program running in the FPGA platform should be executed as fast as possible. Splitting the long delay pipeline stage into multiple short delay pipeline stages is the key for increasing pipeline frequency.*

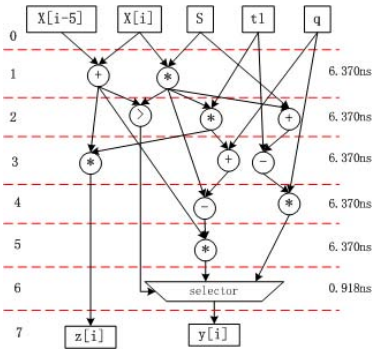


Fig. 2. DFG with ASAP algorithm

The maximized operations delay among the pipeline stages reflects the frequency of the hardware. Based on the result of ASAP scheduling algorithm, the main idea of this paper is only put one node into the pipeline stage in the every data-path. The result of preliminary pipelining is as shown in Figure 2. The maximized delay pipeline stage has the minimized hardware delay among the different divided methods.

Principle 2. ***(Operation Balance Principle).*** *The operation which has the similar delay time should be scheduled to the same pipeline stage. Keeping the operation balance is the benefit of the merging of pipeline for the minimized number of pipeline stages.*

Combining the ALAP scheduling algorithm and the hardware delay about the operations, the main train of this paper is scheduling the operation to the proximity pipeline stages. Without influencing of the frequency and reducing the delay of some stages, the result of preliminary scheduling is as shown in Figure 3.

Principle 3. ***(Mergence Principle).*** *To reduce the resources used by partitioning pipeline and reduce the iteration execution interval, it is necessary merging the adjacent pipeline stages without influencing the maximized pipeline stage delay.*

Based on the result of operation balanced, the next step is merging adjacent stages in the CDFG, and as shown in Figure 4. After merging the stage 3 and stage4, the maximized hardware delay about the stages is also 6.370ns, without affecting the frequency of the system.

Based On the above principles, this paper proposes a basic pipeline partitioning algorithm (as shown in algorithm 1) for the loops that don't have carried dependencies.

When there are carried dependencies in the loop body, it is necessary considering the back-edges in the DFG. A loop code in C language which has carried

Algorithm 1. Basic Pipelining Division Function (DFG dfg)

```
Step 1.Get the starting nodes in the dfg;
Q ← Ø;
for each node dfg.v_i ∈ V do
   dfg.v_i.h ← 0
   if dfg.v_i is the starting node then
      ENQUEUE(Q, dfg.v_i);
   end if
end for
Step 2.Initialize the pipeline stages number about the nodes in the dfg by ASAP
scheduling algorithm;
while Q ≠ Ø do
   dfg.v_k ← DEQUEUE(Q);
   for each edge dfg.e_ij ∈ E do
      if dfg.v_i = dfg.v_k and dfg.v_j is not marked then
         if dfg.v_j.h = 0 then
            dfg.v_j.h ← dfg.v_k.h + 1;
            ENQUEUE(Q, dfg.v_j);
         else if dfg.v_j.h < dfg.v_k.h + 1 then
            dfg.v_j ← dfg.v_k.h + 1;
            UPDATE(dfg.v_j);
         end if
      end if
   end for
end while
Step 3.Compute the max value of combinational logic delay of each pipeline stages;
for l = 0 to max_h do
   max_s[l].d ← 0;
   for each node dfg.v_i ∈ V do
      if dfg.v_i.h = l and dfg.v_i.d > max_s[l].d then
         max_s[l].d ← dfg.v_i.d;
      end if
   end for
end for
Step 4.Combine the adjacent pipeline stages without changing the maximum fre-
quency.
s ← 1;
while s ≤ max_h do
   delay_0 ← max_s[s − 1].d;
   delay_1 ← max_s[s].d;
   if delay_0 + delay_1 ≤ max_delay then
      for each node v_i ∈ V do
         if dfg.v_i.h ≥ s then
            dfg.v_i.h ← s − 1;
         end if
      end for
   else
      s ← s + 1;
   end if
end while
```

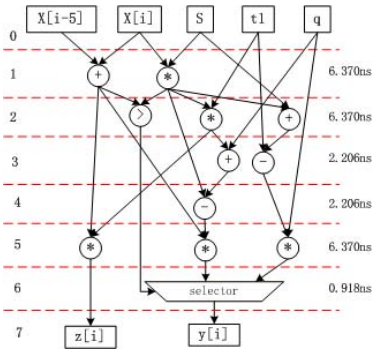


Fig. 3. Delay Balanced DFG

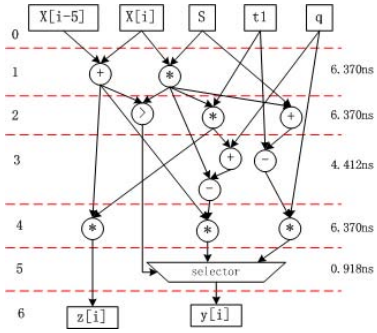


Fig. 4. DFG with Minimized Pipeline Resources

dependencies is shown as follows. According to the partitioning method shown in figure 5, we can get the space-time diagrams of the loop. When the stages number crossed by the back-edge is more than the offset of iteration about the dependency, there would be a bubble in the space-time diagram, as shown in figure 6. It is necessary to reduce the number of pipeline stages inside the back-edges to eliminate the bubble. To ensure the validity of pipeline scheduling algorithm aiming at the loop with back-edges, this paper proposed another principle.

```
for(int i=0 i <N;i++){
    X[i+2]=A[i]+Y[i]+B[i]+C[i]+D[i]+E[i]+F[i]+G[i];
    Y[i+4]=X[i+2]*A[i];
}
```

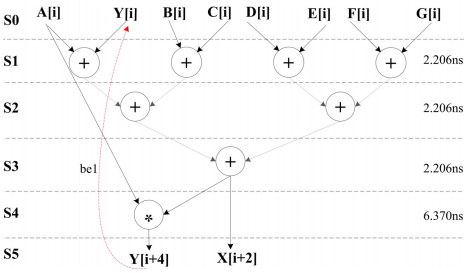


Fig. 5. DFG of the loop with back edges

Principle 4. ***(Back-Edges Principle).*** *The distance of iterator dependency interval for every back-edge should be more than the number of pipeline stages crossed by back-edges. For the nodes v_i and v_j $(i < j)$, the back-edge from v_j to v_i expressed by e_{ji}. S_i is the pipelined stage number about the nodes v_i and the S_j is for the nodes v_j. IDI_{ij} is the presentation of iterator dependency interval between nodes v_i and v_j. The formula (1) should be satisfied.*

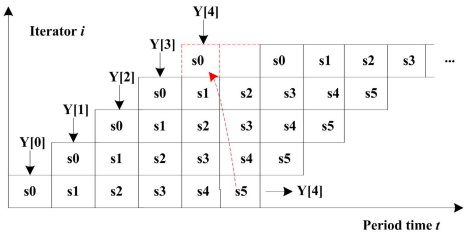


Fig. 6. Space-time diagram of the loop

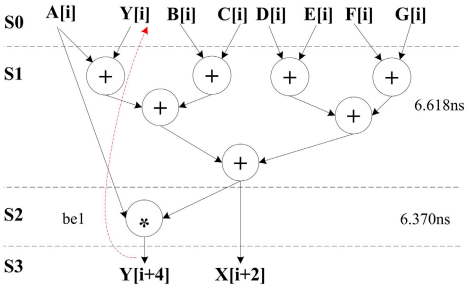


Fig. 7. The result of pipelining divided

$$S_j - S_i \leq IDI_{ij} \tag{1}$$

When there are back-edges in the DFG corresponding to the loop body, the frequency of the system may be influenced by combining the adjacent pipeline stages too much. The pipelining divided result of DFG shown in the figure 5 is shown in figure 7. Although the maximized logic delay among the pipeline stages has been changed, we can get the more perfect space-time diagram, as shown in figure 8. By the way, the pipeline control units will be simpler, and the validity result of pipelining is ensured.

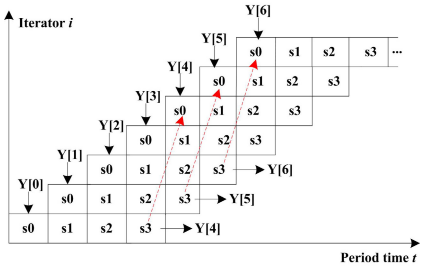


Fig. 8. The improving space-time diagram

Algorithm 2. Improvement Pipelining Scheduling Function (DFG dfg)

Step 1.Run step1 and step2 from the basic pipelining scheduling function for dfg, as shown in algorithm 1, and get the preliminary result of pipeline partition;
Step 2.Get the back-edges in the dfg;
$QE \leftarrow \emptyset$;
for each edge $dfg.e_{ij}(v_i \rightarrow v_j)$ and $(i < j)$ **do**
  if $dfg.v_j.h > dfg.v_i.h$ **then**
    ENQUEUE(QE, $dfg.e_{ij}$);
  end if
end for
Step 3.Build the subgraph set of the DFG dfg according to the back-edge queue QE;
$QG \leftarrow \emptyset$;
while $QE \neq \emptyset$ **do**
  $be_{ij} \leftarrow$DEQUEUE(QE);
  $offset_{ij} \leftarrow IDI_{ij}$;
  if $offset_{ij} < dfg.v_j.h - dfg.v_i.h$ **then**
    for each node $dfg.v_k \in V$ and $dfg.v_i.h \leq dfg.v_k.h \leq dfg.v_j.h$ **do**
      According to the node $dfg.v_k$ and edges $dfg.e_{ij}$ build subgraph of dfg named dfg_sub;
    end for
  end if
  ENQUEUE(QG, dfg_sub);
end while
Step 4.According to the principle 4, eliminate the bubble for every subgraph dfg_sub.

for each back-edge $be_{ij} \in QE$ **do**
  $offset_{ij} \leftarrow IDI_{ij}$;
  while $offset_{ij} < dfg.v_j.h - dfg.v_i.h$ **do**
    $dfg_sub \leftarrow$ DEQUEUE(QG);
    for l in $dfg.v_i.h$ to $dfg.v_j.h$ **do**
      $combinDelay[l] = max_s[l].d + max_s[l+1].d$;
    end for
    Select the minimized $combineDelay[k]$;
    Combine the pipeline stages k and $k+1$;
    Update the height of nodes in the dfg_sub and dfg;
  end while
end for
Step 5.Run the step 3 and step 4 in algorithm 1 for the dfg, and update the pipeline stages height.

On the basic of algorithm 1 and principle 4, this paper has proposed an improvement algorithm for the loop with carried dependencies. As shown in algorithm 2.

According to the algorithm 2, the loop with carried dependencies can be well handled and the result of example code is shown in figure 7. The bubble of space-time diagram is eliminated, as shown in figure 8. The maximal delay of pipeline

stage is decreased from 6.618ns to 6.370ns, and its the least of delay when the bubble would be eliminated about the space-time diagram.

4 Experiment and Analysis

To evaluate the effectiveness of our loop pipeline scheduling algorithm, we have finished the implementation of loop pipeline scheduling algorithm in reconfigurable compiler ASCRA. We generate code for reconfigurable systems based on microblaze and FPGAs using ASCRA, and preliminary experimental results are acquired. Our loops include both perfect and imperfect ones. For example, the Discrete Cosine Transform (DCT), the 5 tap Finite Impulse Response (5-tap-FIR) and the Fast Fourier Transformation (FFT) include the loop without carried dependencies. The seventh test case of Livemore Fortran Kernels benchmark (Kernel-7) has carried dependencies. The result of pipelining by the FPGA-based loop pipeline scheduling algorithm is shown in figure 9. IPD is the improved FPGA-based loop pipeline scheduling algorithm proposed by this paper, and DPD is the directed pipelining division method used by the other reconfigurable compilers which haven't consider the delay of hardware operation. The number of pipeline stages is named PS in figure 9.

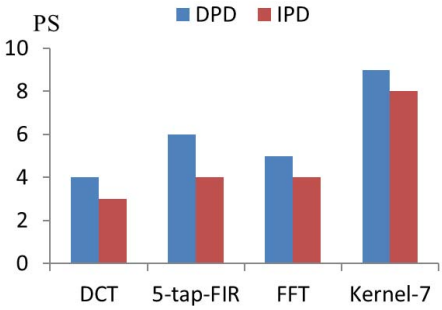


Fig. 9. The number of pipeline stages

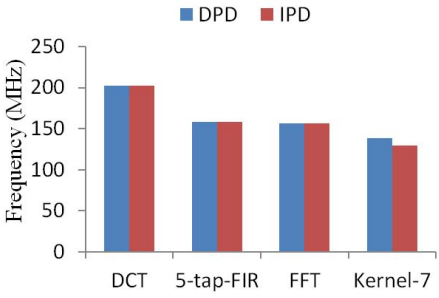


Fig. 10. The frequency of Benchmark

Figure 9 shows that the pipeline stages quantity has been reduced efficiently by our algorithm. The VHDL code generated by ASCRA is synthesized by ISE 14.3 and we obtain the frequency of benchmark as shown in figure 10. Except the loop with carried dependencies, such as Kernel-7, the ASCRA will not decrease the frequency for the loop without carried dependency.

In addition, we have compared the resources of register cost by our improved pipelining division algorithm with DPD. In order to estimate the effects of saving register resources, we have improved the complexity of calculation by unrolling the loop, and obtained the cost of register for various unrolling factors. The result of experimental is shown in figure 11, figure 12, figure 13 and figure 14. The result shows that the more complicated about the calculation, means that the unrolling

factor is larger, the improved algorithm can saving the more resources of register. In this figure, OP means that the optimization percentage. By the end of the experiment, the optimization rate of the register resources used IPD can achieve 20% to 40%.

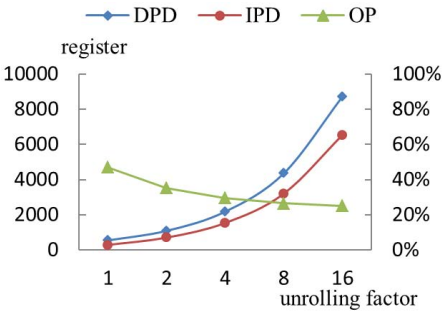


Fig. 11. The result of DCT

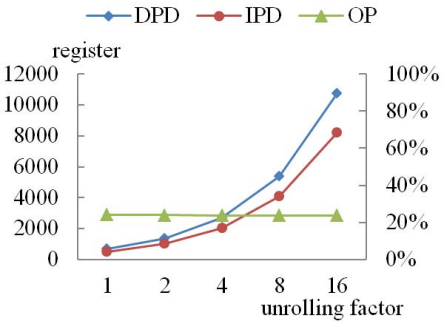


Fig. 12. The result of 5-tap-FIR

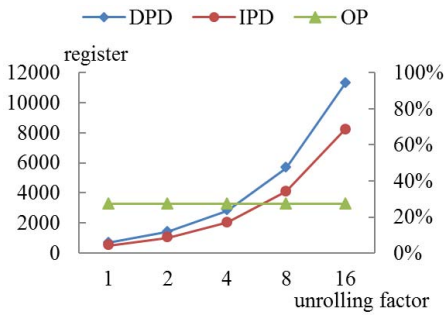


Fig. 13. The result of FFT

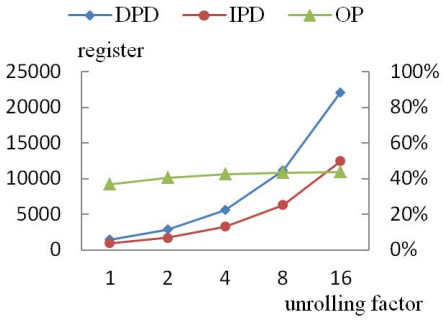


Fig. 14. The result of Kernel-7

All the analysis shows that the combination of adjacent stages based on hardware operation delay is feasible for loop pipeline scheduling algorithm. Without decreasing the performance of reconfigurable system, the algorithm can obtain the more economical hardware architecture pipelined for FPGA.

5 Conclusions

This paper presented an overview of ASCRA compiler which can translate C code to VHDL program automatically for reconfigurable systems. Traditional reconfigurable compilers mainly focus on the directed pipeline division method without considering the influence of operation logic delay for FPGAs. This paper have proposed 4 pinciples for the reconfigurable compilers and improved

pipelining scheduling algorithms based on FPGA to improve method of hardware implementation for ASCRA. The preliminary experiments validates that the algorithm is feasible and can generate the more efficient hardware.

In future work, we will explore the pipelining scheduling algorithm for multi-dimensional iterations and complete the experiment aiming at more benchmarks.

Acknowledgements. This work is supported by the National Natural Science Foundation of China No.61003036 and the Natural Science Foundation of Heilongjiang Province of China under Grant No.QC2010049. An extra special thanks to Xilinx Inc. for designing this board and making this independent study possible.

References

1. Cardoso, J.M.P., Diniz, P.C., Weinhardt, M.: Compiling for Reconfigurable Computing: A Survey. ACM Computing Surveys 42, 1–65 (2010)
2. Gokhale, M.B., Stone, J.M.: NAPA C: compiling for a hybrid RISC/FPGA architecture. In: Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), Napa Valley, California, pp. 126–135 (1998)
3. Callahan, T.: Kernel Formation in Garpcc. In: Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, vol. 04, pp. 308–309 (2003)
4. Stockwood, J., Harr, R., Callahan, T., et al.: Hardware-Software Co-Design of Embedded Reconfigurable Architectures. In: 37th Conference on Design Automation (DAC 2000), pp. 507–512 (2000)
5. Alex, D., Jones, B., et al.: PACT HDL: A C Compiler Targeting ASICs and FPGAs with Power and Performance Optimizations. In: Proc. of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pp. 188–197 (2002)
6. Shen, Y.: Research on the hardware-software code partition technology of reconfigurable computing systems. University of Science and Technology of China, pp. 35–39 (2007)
7. Qu, J., Zhao, R., Liu, T., et al.: The Research of FPGA-based Loop Optimization Pipeline Scheduling Technology. In: Computer and Communication Technologies in Agriculture Engineering International Conference on IEEE (CCTAE 2010), vol. 3, pp. 426–429 (2010)
8. Wu, Y., Gu, G., et al.: Application-Specific Compiler for Reconfigurable Architecture ASCRA. Jisuanji Kexue yu Tansuo 5(3), 267–279 (2011)
9. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization (CGO), pp. 75–86. IEEE (2004)
10. Pellerin, D., Thibault, S.: Practical FPGA programming in C. Prentice Hall Press, Upper Saddle River (2005)
11. `http://www.xilinx.com/products/design-tools/vivado/index.htm`
12. Guo, Z., Najjar, W.: A compiler intermediate representation for reconfigurable fabrics. In: Proceedings of the Conf. on Field Programmable Logic and Applications (FPL 2006), pp. 1–4 (2006)

ACF: Networks-on-Chip Deadlock Recovery with Accurate Detection and Elastic Credit

Nan Wu, Yuran Qiao, Mei Wen, and Chunyuan Zhang

Computer School, National University of Defense Technology, P.R. of China
{nanwu,qry,meiwen,cyzhang}@nudt.edu.cn

Abstract. The key idea of progressive deadlock recovery scheme is providing an escape path outside the deadlock cycle. State-of-the-art schemes, such as Virtual Channel Reallocation (VCR) and DISHA, set up escape paths by dedicated deadlock-handling channels. These channels use additional data paths, central buffers and bypass logic. This paper proposed a novel deadlock recovery scheme for NoC, using Accurate on-Cycle Forwarding (ACF) path inside deadlock cycle to drain blocked packets. ACF does not decouple deadlock-handling channels from normal routing channels, but enhanced credit flow control on each channels to allow accurate deadlock detection and recovery. ACF method is constructed by three tightly combined components: adaptive routing, run-time accurate deadlock detection, and deadlock removal. We implemented ACF algorithm by O(n) time complexity and by distributed modules cooperating with NoC. The rigorous valuation on multiple traffic patterns shows that our scheme achieves significant performance improvement. ACF detected 10-20 times less fake deadlock alarms than approximation and heuristic time-out approaches. In high packet injection rates interval (40%-75%) where network is more frequently troubled by deadlock, ACF provides 67% communication latency improvement comparing to dimension order routing, and 14%-45% to VCR and DISHA. Moreover, the power consumption and hardware overhead of ACF are light.

Keywords: NoC, Adaptive Routing, Deadlock Detection, Deadlock Removal, Topology Ordering.

1 Introduction

Full adaptive routing is a very attractive goal for all networks. Full adaptive routing allows maximum routing adaptability that is most important for congestion controlling. These kinds of techniques could also be applied to fault-tolerant NoC where more flexible routing schemes are necessary when links break down and topological connectivity is reduced. Deadlock is the most catastrophic issue to full adaptive routing network, which may causes whole network failed. Up to now, there are two main strategies for deadlock handling: deadlock avoidance and deadlock recovery.

In deadlock avoidance, resources are granted to packets in a way that the whole network is deadlock free. Deadlock avoidance is the most popular strategy, including

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 319–333, 2013.
© Springer-Verlag Berlin Heidelberg 2013

schemes of turn prohibiting models such as [4][18][19], strict using of buffer such as injection limitation[20] and strict ordering of virtual channels such as [7]. In general, avoidance techniques require restricted routing functions or additional resources to prevent deadlocks [13], as well as restricted fault tolerance degree [14].

Deadlock recovery techniques do not make restriction on routing and fault tolerance. Deadlock recovery techniques can be divided into two kinds by progressive and regressive. Regressive strategy such as abort and retry mechanism [1], require storing lots of routing information and using very large buffers for retransmission. Progressive strategy, such as a variety of Virtual Channel Reallocation (VCR) [2][3][4] and DISHA[5][6][7] methods, utilizes additional escape path to drain suspected blocked packets. For an example, for a 2D torus network, VCR needs at least 3 VCs, whose cost of VC buffers and VC switchers increased by 3x. But only one VC can perform true full adaptive routing, while other two VCs still be restricted to a kind of dateline routing[4]. For another example, DISHA architecture provides one or more shared central buffers and bypass channels to construct the escape path. Like VCR, DISHA's routing in the escape path has to be deadlock free [6]. Furthermore, accurately detecting deadlock cycles in DISHA and VCR is challenging, simply because of the distributed nature of deadlocks. Heuristic approaches, such as various time-out mechanisms [6][8][9][10], are often employed to monitor the activities at each channel for deadlock speculations.

In our opinions, deadlock recovery on NoC owns more opportunity than traditional interconnect networks. First, link's round trip time is relative smaller, so the channels dependency and buffer space information could be transmitted in time. Second, at the cost of moderate hardware resources, a fast deadlock cycle detection approaches with more accuracy could be considered to replace heuristic approaches. Our new deadlock recovery technique is called Accurate on-Cycle Forwarding (ACF). ACF supports full adaptive routing with deadlock configuration. When deadlock occurs, it is able to locate all the true deadlock cycles in channel wait graph without fake. Finally, based on an elastic credit flow control mechanism, ACF constructs the forwarding path on blocked channels to drain packets simultaneously. As a result, it will be unnecessary to decouple deadlock-handling channels from normal routing channels, providing more routing flexibility, and higher performance and hardware cost efficiency.

2 ACF Methodology

2.1 Network Model

NoC: NoC consists of a set of processer tiles interconnected by point-to-point channels. No restriction is imposed on the topology. Each tile has a router. There are first-in first-out (FIFO) channel buffers for storing packets in each channel. When a channel buffer is full, the state of the channel will set as unavailable to follow-up flits. The paper doesn't distinguish channel and channel buffer in the following discussion.

Switching: Virtual Cut-Through (VCT) switching is used for presentation. Our method is also valid for wormhole switching after introducing some changes in credit control.

Routing: fully adaptive routing algorithm, when a packet reached the arbitration unit but cannot be routed because all the valid output channels are unavailable, it waits on the head of corresponding input buffer until its next turn. By doing so, the packet gets the first valid channel that becomes available when it is routed again.

Deadlock cycle: The general cycle rely conditions of deadlock are known as wait-for and circular wait conditions [4]. For a fully adaptive routing network, it is deadlock free if and only if there is at least one path that can be used to deliver blocked packets out of deadlock cycles.

Channel Wait Graph (CWG): Resource's wait-for dependencies of a network at any particular time can be expressed as a directed Channel Wait Graph [4],G= (V, E). V= {v0, v1, …, vn} is a set of vertices, where each vertex represents a channel of networks. E is a set of edges, where an ordered pairs represents dependence between two channels: $< v_i, v_j > \in E$ when there is flit in the head of channel v_i requesting channel v_j.

Selected Channel Wait Graph (SCWG): SCWG is a directed graph produced by routing selection step, and will keep stable until the end of deadlock detection. Each vertex of SCWG may have multiple in-edges, but none or only one out-edges.

Simple Cycle Graph (SCG): The graph that is only constructed by disjoint simple cycles. A *simple cycle* is a connected directed sub-graph, each of whose vertex has and only has one out-edge and one in-edge [7].

Suspected deadlock condition: "**On any one of router**, **for all** $< v_i, v_j > \in E$**, if buffer(vi)==full and buffer(vj)== full lasted for *N* cycles**". When the suspected deadlock condition is triggered, the NoC starts a deadlock detection operation and then deadlock recovery operation. On the other side, the negation of this condition is used to determine if the deadlock is removed from NoC. Note that we introduce *N* cycles' latency in the condition. This makes accurate detection combined with time-out mechanism. Waiting several cycles to determine whether make a real trigger action will reduce the rate of "fake alarms" than triggering the deadlock detection operation immediately. This is more power consumption efficient for NoC. Comparing to general time-out detection with time threshold from 64 up to 1024[13], we set rather smaller waiting time *N* here. According to our experiment, 4~6 cycles are enough to filter out more than half of fake alarms while impose slight impact on performance.

2.2 Routing Selection

Topology Ordering algorithm could be used to find out cycles in directed graph in O(n) time complexity. However, the algorithm produces correct results only if edges (channel request dependencies) are static throughout its operation. For adaptive routing network, requested channels are changeable in a period of time. It means that vertices in the CWG may have multiple dynamic out-edges and in-edges during topology ordering iteration. To solve this problem, the essential processing flow of ACF deadlock recovery is composed by three steps as shown in Fig. 1.

First of all, if suspected deadlock condition is triggered, a ***routing selection*** step selects a certain blocked channel from all adaptive requested channels for each requesting channel. The ID number of the selected channel will be reported to detection

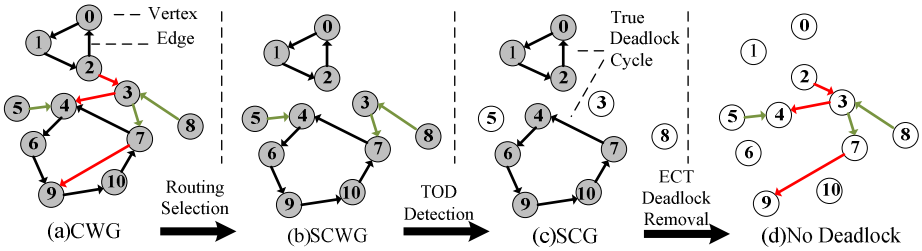


Fig. 1. Three steps of ACF method and their in/output channel wait graphs

module and then be stored in a register, until the deadlock detection finish. No matter how requested channel changes after selection step, the detection only depends on those selected channels. Anyway, the input of selection step is CWG while the output is SCWG as illustrated in Fig.1-b, where vertices with multiple out-edges (red edges in Fig.1-a) will be cut off to only one out-edge. This approach ensures the static of channel wait graph during the follow-up detection operation.

Another problem is, a typical detection iteration requires about 10-20 cycles as discussed in section 3 table 1, during this iteration, if any channels becomes available (not full) again, blocked packets could move out through that channel. So when the detection result returns to the router, there will be three possibilities: First, blocked packets has already moved out, so the router does nothing. Second, the selected channel is not in any of true deadlock cycles, thus the router continues waiting until a requested channel is available. Third, the selected channel is in a true deadlock cycle, then the router starts deadlock removal.

2.3 Deadlock Detection

The second step is ***deadlock detection***, where topology ordering algorithm is implemented to accurately locate all true deadlock cycles in the SCWG. The algorithm is presented as follow:

(1) Selecting a vertex with none of out-edge or none of in-edge from the SCWG.
(2) Removing this vertex and all of its in-edges and out-edges from the SCWG.
(3) Repeating above two steps until all of vertices have been removed, or rests of vertices have and only have one in-edge and one out-edge.

The output of the algorithm is SCG, as shown in Fig.1-c(removing all green edges and unconnected vertices). SCG graph consists and only consists of simple cycles, each of which represents a *true deadlock cycle*.

A coupled Topology Ordering Detection (TOD) network is added to NoC to implement topology ordering operations. As shown in the left of Fig.2, TOD network is composed of distributed Channel Dependence Tag (CDT) units in each router, an interconnection among CDT units and a global synchronization net. The CDT unit is used to represent the channel vertex in the SCWG. For a 2D torus network instance, there are four CDTs in each router corresponding to four input channels. A CDT unit contains a 4-bit in-edge tag, each of which connects to the corresponding out-edge tag of neighbor

CDT units as shown in the right of Fig.2. Besides, a tiny synchronization net is coupled with CDT units that is illustrated as red wires in Fig.2. The synchronization net transmits the detection *start/end* signals and computation result (*stable* signal) among CDT units. Although global synchronization operation may result in some system clocks delay, it is acceptable when the deadlock is infrequency event in most scenario [7][11] (the detailed evaluation of detection delay is described in section 3.2). Furthermore, the whole TOD network and synchronization net could be implemented as an asynchronous circuit or clocked using faster clock than router clock.

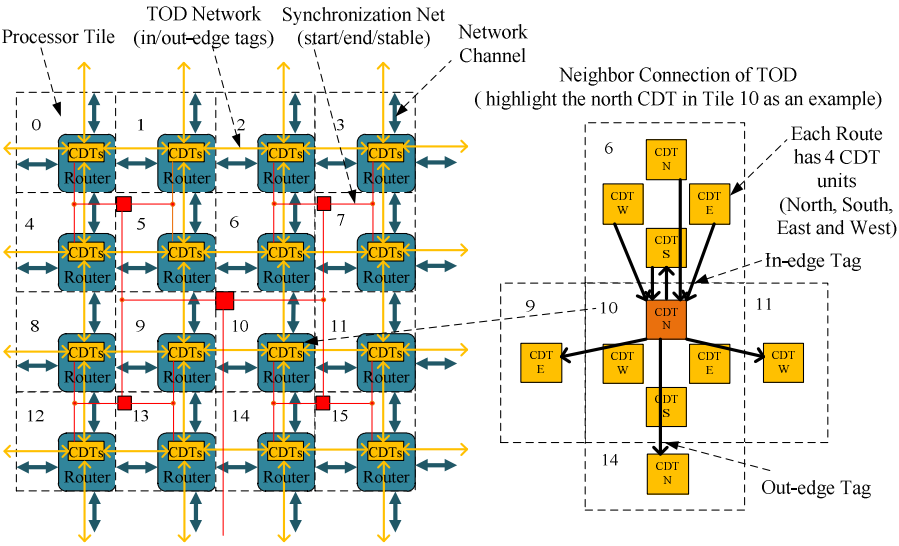


Fig. 2. A 2D NOC coupled with TOD networks

Algorithm 1: Pseudo of TOD network

```
Function for each CDT:
1  Inputs:
     in_tag[4]:  4-bit  in-edge tag wires from neighbor CDTs
     start: detection start signal from synchronization net
     end:  detection end signal from synchronization net
2  Outputs:
     out_tag [4]: 4-bit  out-edge tag wires to neighbor CDTs
     in_deadlock_cycle: if CDT  in true deadlock cycle
     stable: if CDT state changed at current clock cycle
3  Definitions:
     out_tag_reg [4]: store out-edge tag value at last clock cycle
     detecting: signal  indicates if detection is running
4  At each clock cycles:
5    if (start) :
6      set on out_tag[i] if channel[i] triggered suspected deadlock
7      detecting=1;
8    if (detecting):
9        in_tag[0:3 = out_tag[0:3] from neighbor CDTs;
10       if (!or(in_tag[0:3])):
11           out_tag[0:3] = (0,0,0,0);
12       if (out_tag_reg == out_tag):
13           stable = 1;
14       out_tag_reg[0:3] = out_tag[0:3];
15   if (end):
16       in_true_deadlock = or(out_tag[0:3]);
17       detecting= 0;

Funciton for synchronization net:
18 Inputs:
     deadlock_condition[n]:
                 if suspected deadlock condition triggered
     stable[n]:  the stable signal from each CDT
19 Outputs:
     start: the start signal to all CDTs
     end:  the end signal to all CDTs
20 At each clock cycles:
21 if (or(deadlock_condition[0:n-1])):
22     Send start signal to all CDTs by synchronization net;
23 elseif (&stable[0:n-1]):
24     Send end signal to all CDTs by synchronization net;
```

When suspected deadlock condition is triggered, the dependence information of channels will be reported to CDT units by setting on the corresponding edge tag. Then, each CDT unit keeps checking its in-edge tag. If none bits of the in-edge tag is

set on, CDT unit will set down all of it's out-edge tags. Meanwhile, CDT unit continues comparing values of out-edge tag at recent two clock cycles, and then output a *stable* signal to indicate whether the value is changed. When the whole system reaches a stable state (*stable* signals are set on by all CDT units), the detection operation finishes off. At last, the CDT unit whose out-edge tags are not set down all is in the true deadlock cycle, while the channel whose out-edge tag is not all set down is the blocked channel. The pseudo code of TOD network function is listed in algorithm 1.

The time overhead of TOD detection mainly depends on the size of NoC. Given a network with n channels, when the detection starts, CDT units will set down at least an out-edge tag every clock cycle. Thereby, the detection iteration will spend no more than O(n) clock cycles.

2.4 Deadlock Removal

The third step is applying ***deadlock removal*** to those true deadlock cycles. This section presents the idea and implementation of an on-cycle forwarding deadlock removal approach called Elastic Credit Transfer (ECT), which is compatible with the widely used credit-based flow control mechanism [4].

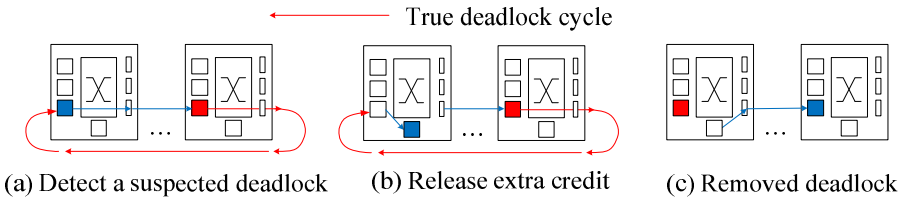


Fig. 3. Successful case of credit-based on-cycle forwarding

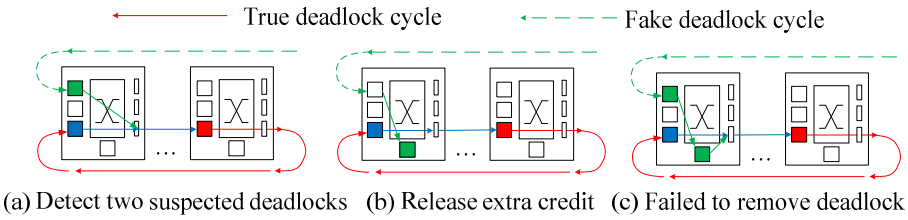


Fig. 4. Failed case of credit-based on-cycle forwarding

According to the detection result, routers release an extra deadlock-handling buffer for those channels on the true deadlock cycle, and forward at least a packet towards destination. This procedure will repeat until the network recovery from deadlock (no suspected deadlock condition is triggered). A simple credit-based deadlock removal approach is illustrated in Fig.3. Fig.3-a shows a deadlock scenario. Then, the packet blocked in the channel is moved to a deadlock-handling buffer as shown in Fig.3-b. It guarantees that at least one blocked packet could be forwarded along the deadlock cycle. In this way the deadlock cycle will be broken quickly.

But, it must be guaranteed that packets outside the deadlock cycle will not inject in, otherwise the deadlock-handling buffer may be occupied by "fake" packet, while "true" deadlock packet still blocked in channels (Fig.4-a, b). That is why the routing selection and the TOD detection steps have to be used to change CWG to SCG. Otherwise, if there is more than one packet requesting the deadlock channel (means multiple in-edges in CWG), it is very possible to choose the wrong one to occupy the deadlock-handling buffer as Fig.4-c shows. This will make credit-based deadlock removal approach failed.

Traditionally, the maximum credit of channel is constant. The credit controller of current channel is located in the previous router. As state machine shown in Fig.5-a, when a flit passes to next router, the credit is reduced by one (means that an entry of next router's channel buffer is occupied). When the next router pushes out a flit, it will reply a release signal to add one credit. When the credit value is smaller than a threshold (normally, set to the maximum packet length in VCT or one flit length in wormhole, pulsing a round trip latency), the channel is unavailable and flits which requesting this channel should be blocked.

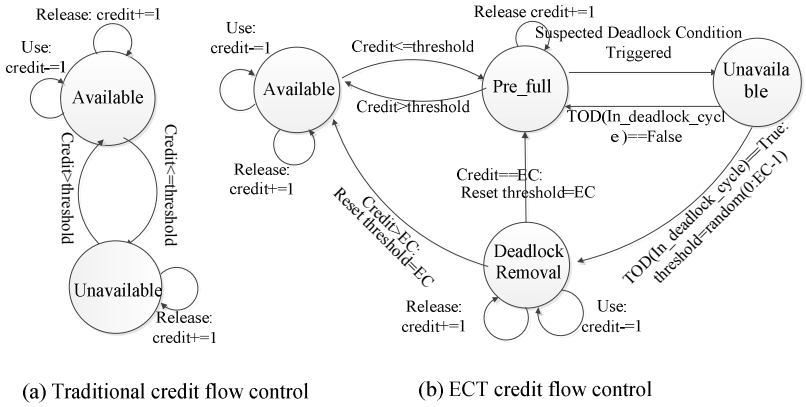


Fig. 5. Stat machines of traditional credit flow control and ECT credit flow control. To simplify of the description, the figure assumes that one packet constructed by one flit.

On-cycle forwarding could be implemented in more efficient way as our ECT method. We distribute the space of central deadlock-handling buffer to each channel and then control it by credit, whose controlling state machine is shown in Fig 5-b. The ECT method set the normal credit threshold as Elastic Credit (EC) initially. When the credit is less than threshold, we regard the channel as *pre_full*. As the result, the suspected deadlock condition presented in section 2.1 is changed to: "**On any one of router, for all** $< v_i, v_j > \in E$**, if buffer(vi)==pre_full and buffer(vj)== pre_full lasted for *N* cycles**". Then, if suspected deadlock condition triggered and the channel is detected in a true deadlock cycle (this kind of channel is called *blocked channel*) , the router changes its credit threshold to a random value between 0 and EC-1, and generates a mask to prohibit the packets in unblocked channel requesting any blocked

channel[1]. Note that blocked channel requesting blocked channel or unblocked channel requesting unblocked channel are not forbidden. As the result, at least one extra deadlock-handling credit is available to and only to the packet waiting in the blocked channel. Then, the blocked packet could pass though to the next channel without any "disturbing". This procedure repeats until current deadlock cycle removed, and then the credit threshold can and will be reset to EC again.

A detailed instance for ECT is shown in Fig.6. In this case every channel reserves an extra deadlock-handling buffer entry as elastic credit (marked as grey color). Generally, this buffer entry is closed unless deadlock occurs. First of all, we got a deadlock configuration (Fig.6-a), 4 groups of packets occupies a channel and request the next channel, which causes two circular dependencies. The green packet is so called "fake" deadlock packet that stays out of deadlock cycle but requests a blocked channel. It will be excluded by TOD detection (as shown in Fig.6-b). In Fig.6-c,d, channels on true dead lock cycle release the elastic credit to ensure all blocked packet moving at least one step. The packets could continuously using the elastic credit and

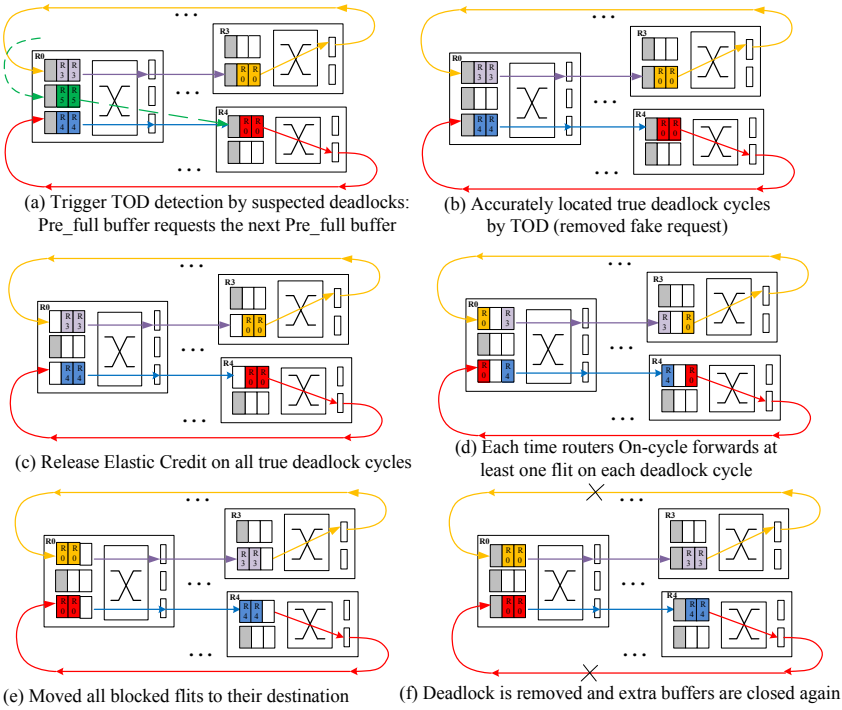


Fig. 6. ACF examples for remove tow deadlocks simultaneously

[1] This description is just for VCT, for wormhole, the packet in unblocked channel which already passed the head flit is allowed requesting blocked channel temporally until the tail flit.

moving on. Finally, when packets arrived at the destination, or left the deadlock cycle, as illustrated in Fig.6-e, the deadlock has been removed and the deadlock-handling buffer entry for elastic credit is closed again.

Note that our on-cycle forwarding path is constructed on true deadlock cycle, instead of using an escape path outside deadlock cycle. It makes ECT able to forward blocked packets in multiple deadlock circles simultaneously and fast (no need to switch to escape path). Meanwhile, ECT did not add any additional switch resources, central bypass buffers nor channels, and no packet need to be aborted to free up resources. ECT only requires an extra buffer spaces for each channel and an accurate detection mechanism like TOD.

3 Evaluation

A typical 8x8 torus network is used for the evaluation. Since VCR requires at least 3 VCs for torus NoC, for fairness, all mechanisms are configured with three VC channels as well. Each VC has an independent 8-entry input buffer, while ECT introduces one additional entry for each buffer and DISHA brings a 32-entry central buffer for each router. The router uses a 5-stage pipeline with a round robin arbitration scheme. Every node generates 20000 packets per VC per destination and injects into the NoC by preset Packet Injection Rates (PIR). Four different traffic patterns are tested: random, bit-complement, bit-reverse [7] and fault-channel. The traffic of fault-channel pattern is the same with random one, but four channels are randomly set to be broken. For each PIR, 20 networks were simulated, each with a different randomly chosen fault set. The curvy in the figure give the ensemble average latencies. In contrast, in a network with deterministic routing, a single faulty channel renders the network inoperable, so DOR is not tested in fault channel pattern.

3.1 Detection Accuracy and Latency

The accuracy of detection, calculated as the number of packets detected in suspected deadlock cycles, is shown in Fig.7. TOD and Time-Out with thresholds of 16, 32 and 64 system clocks are tested. It shows that the majority deadlocks detected by the time-out schemes are fakes. For instance, about 14% of the packets injected in the network are detected as part of deadlock under the time-out threshold of 16 when network near saturating (Fig.7-c), while TOD detected that less than 1% of packets are in true deadlock cycles. For all four patterns, TOD detected about 10-20x less deadlock alarms than time-out schemes. It can be also noticed that under some injection rates, TOD detects more deadlocks than Time-Out 16 and 32(e.g., in PIR of 0.74 in Fig.7-a). It does not mean Time-Out is more accurate. This is because sometimes Time-Out schemes may release heavy load channels as a congestion monitor, fewer deadlocks may occur if some flit draining actions made earlier.

As an important factor for the earlier removal action, Table 1 list the average TOD delay under different NoC sizes, assuming using a separate fast clock for synchronization net. The detection delay of TOD is determined primarily by three factors: the size

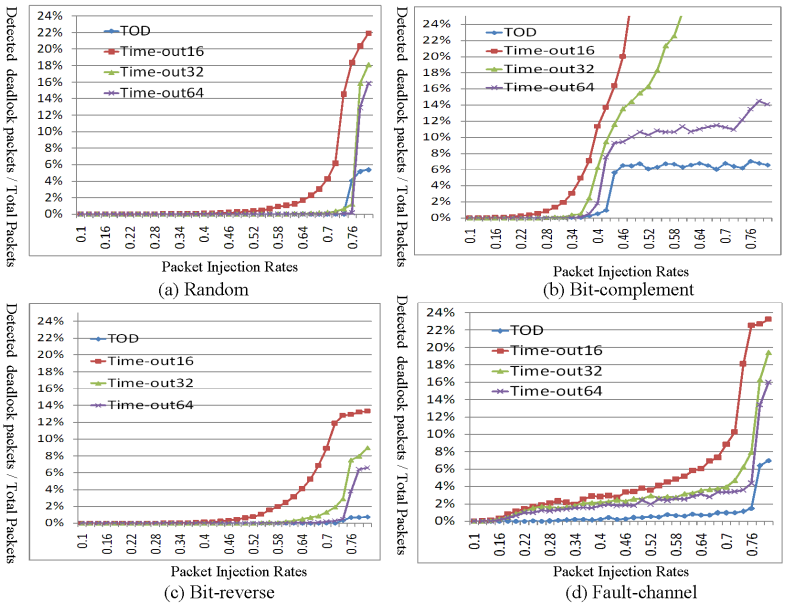


Fig. 7. The ratio of deadlock packets detected by different schemes

Table 1. TOD detection delay at PIR of 0.5 on different NoC size

NOC size	Theoretic Max delay	Tested Max delay	Tested Average delay	Equal to system clock cycle (1GHz)
4x4	58.54 ns	8.1 ns	7.588 ns	<8
6x6	113.1 ns	10.0 ns	9.258 ns	<10
8x8	222.2 ns	13.4 ns	12.434 ns	<13
16x16	863.3 ns	18.7 ns	17.401 ns	<18

of networks, physical transfer delay of wire connections and the size of detection. Note that the average practical detection latency is much less than the maximum theoretic value, because most deadlock cycles are constructed by a small vertices set in practice.

3.2 Communication Latency and Throughput

Different deadlock-free mechanisms are measured to make a comparison as follow: (1) DOR (Dimension Order Routing); (2) VCR; (3) A concurrent DISHA; (4) The ACF. Heuristic Time-Out scheme is used for VCR and DISHA to detecting deadlock.

The result of packet transfer latency is shown in Fig.8. First, DOR shows the poorest performance. Taking random pattern for instance (Fig.8-a), it saturates at about 0.5 PIR, while ACF saturating at 0.74 PIR achieves 1.5x performance speedup. In adaptive routing schemes, ACF have the best performance on average. The DISHA has comparable communication latency of ACF on low PIR, but its latency becomes worse when PIR being closer to saturation point. DISHA, has much more fake deadlock packets to drain, and only a central structure can be used to forward one packet

once a time. On the contrary, ACF processing true deadlock packets only and allows forwarding multiple deadlock packets directly and simultaneously on all paths. However, both of their latencies surpass the DOR and VCR after saturation. It is because in both of ACF and DISHA, the deadlock packet has higher priority on all VC channels than normal packet. Thus if deadlock is no longer an very infrequent event, more normal packets may be blocked in buffers to produce new deadlock cycles, makes a positive feedback phenomenon [7]. VCR is worse than others because two of VCs are restricted on routing, but in a sense, these restricted VCs could not be deadlock, making the latency curvy smoother after network saturation.

In a word, at high PIR where deadlock occurs more, ACF outperforms DISHA in most instances and always outperforms VCR before saturation. Among 0.4-0.75 PIR interval, the average communication latencies are list as following: DOR: 233; 32T+VCR: 91; 32T+DISHA:142; ACF:79. It shows that ACF provides 67% improvement of the latency comparing to DOR, 14%-45% to VCR and DISHA.

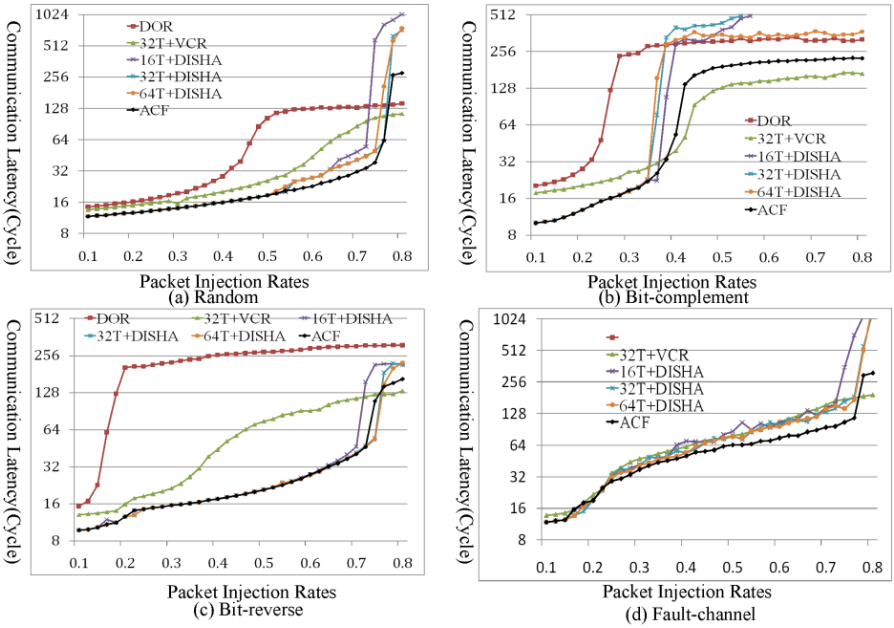


Fig. 8. Performance of communication latency on four traffic patterns

The other important performance metric is throughput, illustrated in Fig. 9. The figure shows throughput (accepted traffic) as an average function of offered traffic as flits per cycle per node. In traffic patterns such as random and fault-channel, ACF works the best. In others, ACF performs the second best. It does make sense, because of impossibility of finding out absolutely the best time threshold and escape path configuration for VCR and DISHA, which are sensitive to too many factors and depend on traffic pattern as well. However, it shows that ACF always works quite well on all traffic patterns. The average maximum throughput of ACF is about 1.5x higher than DOR, and about 1.1x higher than VCR and DISHA.

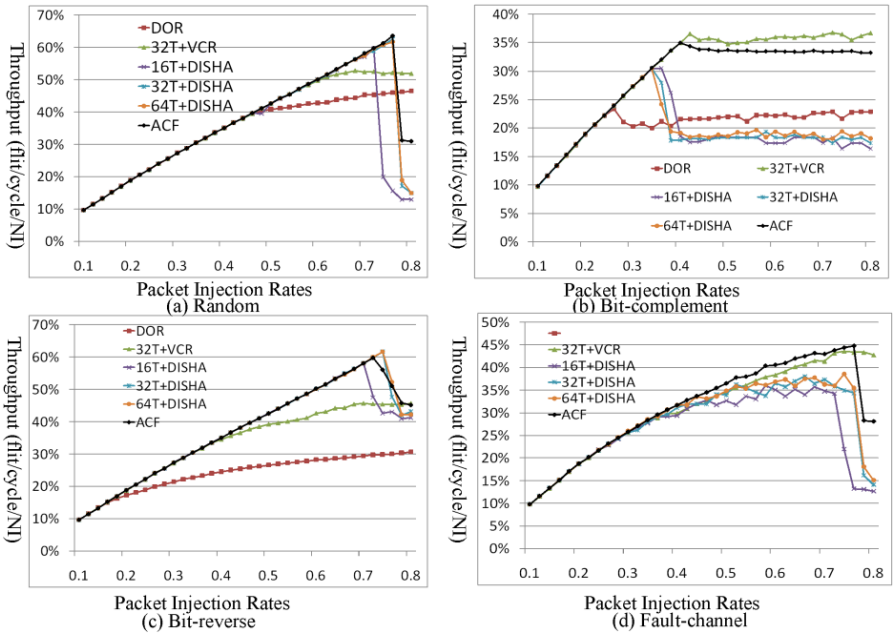


Fig. 9. Performance of throughput under four traffic patterns

3.3 Area and Power Cost

Fig.10 shows the micro-architecture of the router used for hardware cost evaluation. The components added to baseline NoC for different deadlock recovery mechanism are rendered in different colors.

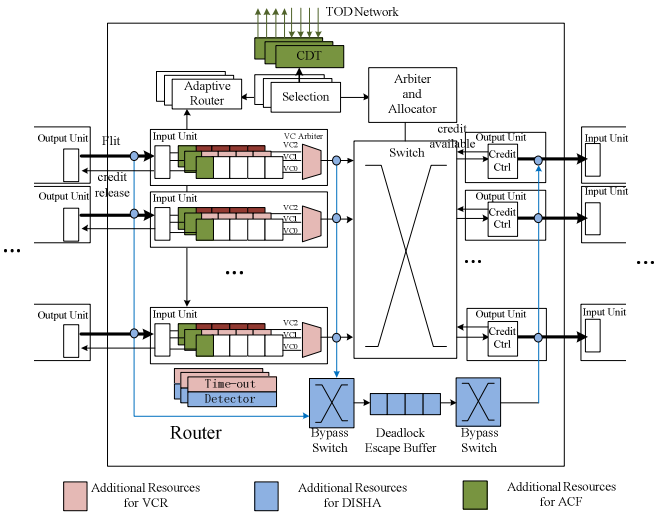


Fig. 10. Micro-architecture of the evaluation router

Table 2. Area overhead for different deadlock recovery tech.(8x8 torus NoC, 3VC 48-bit flit width, 1GHz)

	Combination (mm^2)	SRAM (mm^2)	Num.of cross-router wires	Total (mm^2)
Baseline NOC	49.4	15.4	2816	69.2
Time-Out+VCR	52.5	15.4	2816	73.6
Time-Out+DISHA	72.0	27.2	3328	108.1
ACF(TOD+ECT)	51.2	20.1	3645	78.4

The area evaluation was carried out using a modified version of ORION 2.0[15] on the TMSC 65nm standard cell library. The operating frequency is 1G MHz and the supply voltage is 1.0V with temperature at 110 C. Table 2 lists the area overhead of four kinds of NoC. We found that the TOD detection circuit uses 55% less combination area than the Time-out circuit with 10-bit counter and will add only 2.6% area overhead of the total area compared to 5.6% for the time-out implementation. Moreover, the hardware overhead of the ACF is light that is 11% of a baseline network without deadlock recovery mechanism. Compared with other deadlock recovery mechanisms, ACF uses 73% area of DISHA and 107% area of VCR.

We also found that although TOD detection spend about 10% more power on synchronization net, ACF deadlock removal circuit consumes 31% less power than the DISA circuit and 22% less than VCR, because of less actions triggered by fake alarms and fewer central logic. ACF is worthwhile. Furthermore, in fact, three kinds of deadlock-recovery-able NoC only add less than 10% of the total power overhead compared to NoC without deadlock recovery in most situations, where the deadlock is infrequency event [11]. Note that, we evaluated three VCs here. The ACF is able to implement true adaptive routing with only one VC, which is flexible

4 Related Works

DISHA and VCR are typical deadlock recovery techniques, there are many related studies and improvements, such as [2][3][4][5][6][7][8][9][10][12]. Comparing with our paper, there are two main different: the first is that they all use inaccurate heuristic detection approach, but we presents an accurate detection method. Second, they all drain deadlock packets through an additional escape path, but we use on-cycle forwarding method.

For accurate detection, [13] presented an accurate method that utilizes run-time Transitive Closure (TC) computation to discover the existence of deadlock-equivalence sets, which imply cycles of requests in NoC. However, The time complexity of its TC algorithm is $O(n^3)$ that may result in higher detection delay, while our algorithm is O(n). And it also need considerate extra $O(4n^2)$ global wires by not taking advantage of the locality of channels, but our method use much less global wires that are only in synchronization net. Finally, a simple token-ring protocol is necessary in TC algorithm, which causes detection operation delay increasing as well.

For credit-based deadlock removal, [14] introduced a deadlock removing mechanism based on adding a central deadlock-handling buffer to every routers. When the network looks like to be deadlock, one extra credit from the deadlock-handling buffer is added to each blocked channels to recovery deadlock. But there are two problems. On one side, since accurate detection mechanism was missed, their method actually works incorrectly as discussed in Fig.2. On other side, the central buffer cannot deal with more than one wait-for packets in different deadlock cycles simultaneously.

5 Conclusion

This work studies deadlock detection and recovery in NoCs supporting full adaptive routing. We proposed a novel architecture coupled with network to realize deadlock recovery. An accurate run-time deadlock detection method based on topology ordering algorithm is presented. Furthermore, based on accurate location of true deadlock cycles, a new on-cycle forwarding mechanism cooperating with elastic credit control is used to remove deadlock. We investigated the proposed methodology in a RTL level NoC simulator, and evaluated it rigorously on latency, throughput, hardware area and power consumption. The result is encouraging that demonstrates the effectiveness of ACF method compared to other state-of-the-art methods.

Acknowledgement. The authors gratefully acknowledge supports from National Nature Science Foundation of China under NSFC No.61033008, 61272145, and 61103080, Research Fund for the Doctoral Program of Higher Education of China under SRFDP No. 20104307110002, 20124307130004; Hunan Provincial Innovation Foundation For Postgraduate under No.CX2010B028, Fund of Innovation in Graduate School of NUDT under No.B100603, No. B120605.

References

1. Kim, J.H., Ziqiang, L., Chien, A.A.: Compression less routing: a framework for adaptive and fault-tolerant routing. IEEE Trans. on Parallel and Distrib. Syst. 8(3), 229–244 (1997)
2. Duato, J.: Improving the efficiency of virtual channels with time-dependent selection functions. In: Etiemble, D., Syre, J.-C. (eds.) PARLE 1992. LNCS, vol. 605, pp. 635–650. Springer, Heidelberg (1992)
3. Petrini, F., Vanneschi, M.: Performance analysis of minimal adaptive wormhole routing with time-dependent deadlock recovery. In: Proceedings of the 11th International Parallel Processing Symposium, pp. 589–595 (April 1997)
4. Dally, W.J., Towles, B.: Principles and Practices of Interconnection Networks. Morgan Kaufmann Publishers (2004)
5. Khonsari, A., Shahrabi, A., et al.: A performance model of disha routing in k-ary n-cube networks. Parallel Process. Lett. 17, P213 (2007)
6. Anjan, K.V., Pinkston, T.M.: DISHA: A deadlock recovery scheme for fully adaptive routing. In: Proceedings of the 9th International Parallel Processing Symposium, pp. 537–543 (April 1995)

7. Duato, J., Sudhakar, Y., Ni, L.: Interconnection Networks An engineering Approach. Elsevier Science, USA (2004)
8. Lopez, P., Martinez-Rubio, J.M., Duato, J.: A very efficient distributed deadlock detection mechanism for wormhole networks. In: HPCA 1998, pp. 80–86. IEEE Computer Society (1998)
9. Martinez-Rubio, J.M., Lopez, P., Duato, J.: FC3D: Flow control based distributed deadlock detection mechanism for true fully adaptive routing in wormhole networks. IEEE Trans. on Parallel Distributed System 14(8), 765–779 (2003)
10. Soojung, L.: A deadlock detection mechanism for true fully adaptive routing in regular wormhole networks. Computer Communications 30(8), 1826–1840 (2007)
11. Warnakulasuriya, S., Pinkston, T.: Characterization of deadlocks in interconnection networks. In: IPPS 1997. IEEE Computer Society (1997)
12. Anjan, K.V., Pinkston, T.M., Duato, J.: Generalized theory for deadlock-free adaptive wormhole routing and its application to DISHA concurrent. In: Proc. 10th ACM/IEEE Int. Parallel Processing Symposium, pp. 815–821 (1996)
13. Al, D.R., Mak, T., Xia, F.: A run-time deadlock detection in networks-on-chip using coupled transitive closure networks. In: Design, Automation and Test in Europe Conference and Exhibition, pp. 1–6. IEEE, Grenoble (2011)
14. Zarza, G., Lugones, et al.: Deadlock Avoidance for Interconnection Networks with Multiple Dynamic Faults. In: 18th EuroMicro (2010)
15. Kahng, A.B.: ORION 2.0:A power-Area Simulator for Interconnection Networks. IEEE Transactions on VLSI Systems 1(1), 191–196 (2012)
16. Balfour, J., Dally, W.J.: Design Tradeoffs for Tiled CMP On-Chip Networks. In: ICS 2006, Cairns, Queensland, Australia (June 2006)
17. Hoskote, Y., Vangal, S., Singh, A., Borkar, N., Borkar, S.: A 5-GHz Mesh Interconnect for a Teraflops Processor. IEEE Micro, 51–61 (2007)
18. Chiu, G.-M.: The odd-even turn model for adaptive routing. IEEE Transactions on Parallel and Distributed Systems 11(7), 729–738 (2000)
19. Christopher, J.G., Lionel, M.N.: The turn model for adaptive routing. In: Proceedings of 9th International Parallel Processing Symposium, vol. 41(5), pp. 874–902 (1994)
20. Puente, V., Izu, C., Beivide, R., Gregorio, J.A., Vallejo, F., Prellezo, J.M.: The adaptive bubble router. J. Parallel Distrib. Comput. 61(9), 1180–1208 (2001)

An Auction and League Championship Algorithm Based Resource Allocation Mechanism for Distributed Cloud

Jiajia Sun[1], Xingwei Wang[1], Keqin Li[2], Chuan Wu[3], Min Huang[1], and Xueyi Wang[1]

[1] College of Information Science and Engineering, Northeastern University
[2] Department of Computer Science, State University of New York
[3] Department of Computer Science, The University of Hong Kong
sunplusplus@163.com, {wangxw,mhuang,xywang}@mail.neu.edu.cn, lik@newpaltz.edu, cwu@cs.hku.hk

Abstract. In cloud computing, all kinds of idle resources can be pooled to establish a resource pool, and different kinds of resources combined as a service is provided to users through virtualization. Therefore, an effective mechanism is necessary for managing and allocating the resources. In this paper, we propose a double combinatorial auction based allocation mechanism based on the characteristics of cloud resources and inspired by the flexibility and effectiveness of microeconomic methods. The feedback evaluation based reputation system with attenuation coefficient of time and the hierarchy of users introduced is implemented to avoid malicious behavior. In order to make decisions scientifically, we propose a price decision mechanism based on a BP (back propagation) neural network, in which various factors are taken into account, so the bidding/asking prices can adapt to the changing supply-demand relation in the market. Since the winner determination is an NP hard problem, a league championship algorithm is introduced to achieve optimal allocation with the optimization goals being market surplus and total reputation. We also conduct empirical studies to demonstrate the feasibility and effectiveness of the proposed mechanism.

Keywords: cloud computing, double combinatorial auction, reputation, BP neural network, league championship algorithm.

1 Introduction

First proposed by Google and commercialized by Amazon in IT industry, cloud computing [1-3] has been developed for several years and has already become the mainstream in the research community.

Currently, most IT companies sell cloud resources with the fixed-pricing model. However, this pricing scheme has a lot of disadvantages, i.e., low efficiency, inflexibility, low profit, and difficulty in achieving an equilibrium price according to changing supply-demand relation in the market, etc. The fixed-pricing scheme is designed for static allocation and disregards the dynamic nature of a cloud environment.

Therefore, dynamic resource allocation that is more suitable for a cloud is urgent to be addressed. There have been some research results on auction mechanisms introduced in cloud computing. Tan et al. [4] proposed a novel stable continuous double

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 334–346, 2013.
© Springer-Verlag Berlin Heidelberg 2013

auction (SCDA) to alleviate the unnecessarily volatile behavior of the continuous double auction and to ensure economic efficiency for grid resource allocation. Danak et al. [5] modeled users' preference relations, and presented a repeated auction-based allocation protocol and a utility-maximizing bidding algorithm for sharing computational grid resources. Tsai et al. [6] used a bid-proportional auction model, which is capable of adaptively adjusting resource price to improve revenues and resource utilization of providers in capacity-constrained cloud environment. Prodan et al. [7] proposed a negotiation-based approach for scheduling scientific applications on heterogeneous computing infrastructures, and presented a negotiation protocol between the scheduler and resource manager using a market-based continuous double auction model to manage the access to resources in an open market. Shang et al. [8] first presented a framework for constructing global cloud resource markets and then proposed a knowledge-based continuous double auction model to determine the price using a learning algorithm based on historical trading information. Song et al. [9] presented a novel combinatorial auction that allows the group of service providers to publish their bids collaboratively as a single bid to the auctioneer as the bidding mechanism. Its goal is to reduce conflicts among providers and the negotiation time.

However, most of the methods above have the following drawbacks. First, while making use of the effectiveness and flexibility of market mechanisms, they ignore the high probability of malicious behavior, and that leads to lack of corresponding mechanisms to ensure credibility. Second, most methods ignore the importance of history information when deciding the bidding/asking price, which makes the bidding/asking price fixed and not adaptive to the changing supply-demand relation in the market. Third, for methods using a single-item auction, they can only allocate one kind of resource, and they do not display the important characteristic of cloud computing, i.e., the combination of different resources is usually used to offer a variety of services.

According to the above, we propose the intelligent agent based double combinatorial auction mechanism, using feedback evaluation based reputation system to prevent malicious behavior and BP (back propagation) neural network based bidding/asking price decision mechanism to make prices that can adapt to the changing supply-demand relation in the market. Finally, we introduce market surplus and overall reputation as the optimization goals and use league championship algorithm to determine the winner in the auction, which realizes dynamic, efficient and combinatorial allocation of resources in cloud computing.

2 The Auction Mechanism

2.1 Framework

The framework of the resource allocation system is shown in Fig. 1. It involves five roles: cloud service provider (CSP), provider agent (PA), cloud service consumer (CSC), consumer agent (CA) and auction intermediary (AI). The basic process of the double combinatorial auction proposed in this paper is described as follows.

(1) CSC submits the related information to CA when he/she requires service, and CA makes the initial tender according to the requirements. In a similar fashion, CSP submits the related information to PA when he/she has redundant resources, and then PA makes the initial tender.

(2) CA and PA respectively use the bidding/asking price decision algorithm to get the bidding/asking price, and submit the complete tender to AI.

(3) AI runs the winner determination algorithm to get the optimal allocation at regular intervals or when the number of tenders received has outnumbered the threshold. If the auction does not reach the maximum round, AI sends a message to the failed participants to let them modify their prices in order to perform the allocation process again; otherwise, the allocation ends.

(4) AI informs every participant the result of the auction, and the winning CSPs provide services to the winning CSCs, while CSCs need to pay service fee to the corresponding CSPs.

2.2 Double Combinatorial Auction Protocol

Description of Provider's Tender. A CSP's tender includes the following attributes: the CSP's ID number, CPU ($/(MIPS*hour)), memory ($/(GB*hour)), storage ($/(100GB*hour)), bandwidth ($/(Mbps*hour)), and the total capacity of the above resources with time changes, the set of platforms and software that he/she can support, the lowest requirement about CSC's reputation. So, a CSP's tender can be described as the following 7-tuple: {*CSP_ID*, {*price_c, quantity(t)*}, {*price_m, quantity(t)*}, {*price_s, quantity(t)*}, {*price_b, quantity(t)*}, *support_set, r_reputation*}

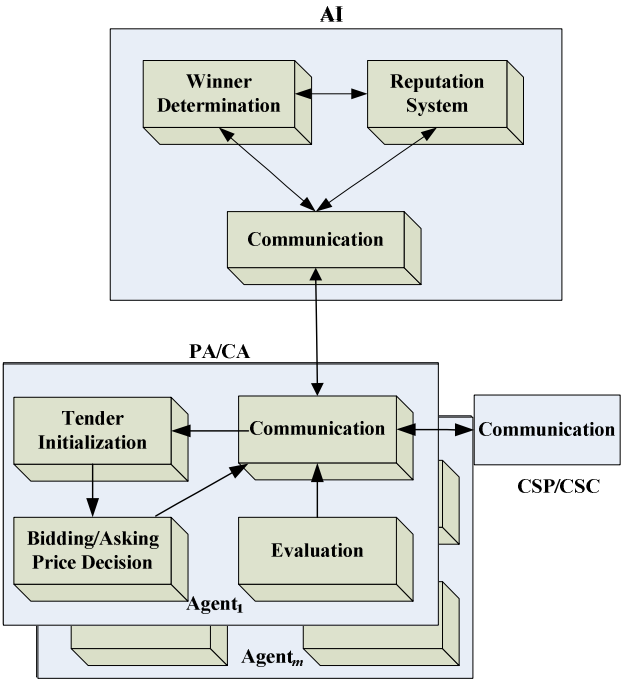


Fig. 1. Framework of Allocation System

Description of Consumer's Tender. A CSC's tender includes the following attributes: the CSC's ID number, bidding price, start time, end time, speed of CPU (MIPS), capacity of memory (GB), capacity of hard disk (GB), network bandwidth (Mbps), task size (MI), data size (GB), granularity, the demands of platform and software, the lowest requirement about CSP's reputation. For certain kind of service, the description of CSC's tender is a subset of the whole set of all the above attributes.

For four kinds of services, i.e., virtual machine service, computation-intensive service, IO-intensive service and storage service, the CSC's tender can be respectively described as {*CSC_ID, bid_price, start_time, end_time, CPU, memory, storage, bandwidth, software&platform, r_reputation*}, {*CSC_ID, bid_price, start_time, deadline, task_size, partition, r_reputation*}, {*CSC_ID, bid_price, start_time, deadline, task_size, data_size, memory, storage, partition, r_reputation*} and {*CSC_ID, bid_price, start_time, end_time, data_size, partition, r_reputiton*}.

Winner Determination Rule. The winner determination problem can be divided into two parts, i.e., optimization goals and constraints. The optimization goals can be described in Eqn. (1)-Eqn. (3), this winner determination problem becomes optimizing an allocation matrix (two-dimensional variable-length array). To be specific, a row of the matrix represents one CSC, and a column represents one CSP, so the element R_{ij} represents the proportion of demanded service that CSC_i allocates to CSP_j.

$$\max total_surplus = \sum_{i=1}^{m} bid_price_i \times \left(\sum_{j=1}^{n} R_{ij}\right) - \sum_{j=1}^{n}\sum_{i=1}^{m} R_{ij} \times ask_price_{ij} \tag{1}$$

$$\max unit_surplus = \sum_{i=1}^{m}\left(bid_price_i / length_i\right) \times \left(\sum_{j=1}^{n} R_{ij}\right) + \sum_{j=1}^{n}\sum_{i=1}^{m} R_{ij} \times \left(ask_price_{ij} / length_i\right) \tag{2}$$

$$\max total_reputation = \sum_{i-1}^{m}\sum_{j-1}^{n} R_{ij} \times reputation_i + \sum_{j-1}^{n}\sum_{i-1}^{m} R_{ij} \times reputation_j \tag{3}$$

3 Winner Determination Method

The winner determination of the double combinatorial auction has been proved to be a very complicated NP-hard problem [10, 11]. Up to now, no method can effectively solve this NP problem, while the simple and easy-to-understand swarm intelligent algorithm can be applied to approximately solve such large-scale problems. In this paper, we chose one of the swarm intelligent algorithms, i.e., the league championship algorithm [12], to solve the winner determination problem.

3.1 Problem Simplification

Process of Multi-objective Problem. In this paper, the three optimization goals are of different magnitudes and units, and are also mutually constrained. Hence, it is not appropriate to convert the problem to a single objective optimization problem using the method of weighted summation. Therefore, the following method is utilized to better solve the problem.

Eqn. (4) is used to compare the quality of any two solutions, i.e., X_a and X_b.

$$\Delta = \alpha \cdot \frac{TS(X_b) - TS(X_a)}{\min(|TS(X_a)|, |TS(X_b)|)} + \beta \cdot \frac{US(X_b) - US(X_a)}{\min(|US(X_a)|, |US(X_b)|)} + \gamma \cdot \frac{TR(X_b) - TR(X_a)}{\min(|TR(X_a)|, |TR(X_b)|)} \quad (4)$$

Here, TS, US and TR represent $total_surplus$, $unit_surplus$, $total_reputation$; $\alpha + \beta + \gamma = 1$, this represents the weight of three objectives.

After the above comparison is done, Eqn. (5) can be used to calculate the fitness.

$$fitness(X_k) = \begin{cases} 1 & k = 1 \\ fitness(X_{k-1}) + \Phi(\Delta_{k-1,k}) & k > 1 \end{cases} \quad (5)$$

Here, $\Delta_{k-1,k} \geq 0$ means that X_k is better than X_{k-1} $\Delta_{k-1,k} \geq 0$ times; $\Phi(\Delta_{k-1,k}) = ae^{\Delta_{k-1,k}} - 1 (a > 1)$ is the increasing function of fitness, and it grows exponentially to make the solution with larger $\Delta_{k-1,k} \geq 0$ has higher fitness.

Process of Constraint Optimization. In this paper, the method of comparison difference is adopted. Its main idea is that when comparing the fitness of solutions, the degree that a solution violates the constraints should also be taken into consideration. Deb [13] proposed a league selection operator, which uses the following rules to determine which solution is better.

(1) When one solution is feasible and the other is not feasible, then choose the feasible solution;

(2) When both the two solutions are feasible, then choose the one with higher fitness;

(3) When the two solutions are not feasible, then choose the one that violates the constraints to a less extent.

3.2 Constraints Adjustment

The final allocation matrix will be of practical significance only if it satisfies all the constraints. It is difficult to make the solution satisfy all constraints only by the swarm intelligent algorithm and comparison difference, so adjusting the matrix in each iteration of the algorithm is needed.

(1) Adjustment of price constraint. If the bidding price is lower than the asking price for a point in the matrix, and the value of this element does not equal 0, then set this value to be 0.

(2) Adjustment for reputation constraint. If the reputation cannot satisfy each other's requirements for a point in the matrix, and the value of this element does not equal 0, then set this value to be 0.

(3) Adjustment for each row. The sum of all elements' values on each row must be 0 or 1. The specific steps are as follows,

Step 1: Record the maximum value $R[i,large]$ and second maximum value $R[i,secondLarge]$ of the a row that does not satisfy the above constraint;

Step 2: Set $R[i,large]-=\left(sum(i)-1\right)$, if $R[i,large]\geq 0$, adjustment of this ends; otherwise, set $R[i,secondLarge]+=R[i,large]$, $R[i,large]$.

(4) Adjustment for granularity constraint. For the line whose granularity is too small, combine the minimum value and second minimum value, repeat this operation until it satisfies the constraint.

(5) Adjustment for capacity constraint. The resources of a CSP must satisfy the requirements of CSC. This adjustment should be done at last. The main idea of this operation is recording the point that violates the capacity constraint, and moving the value of this point to other points where it can be allocated service.

3.3 League Championship Algorithm

The league championship algorithm [12] is inspired and proposed by the phenomenon that teams are continuously adjusting their formation to improve their strength. In the algorithm, every two teams compete, and the result is decided by the strength (fitness). In the remaining period, every team will design the best formation (new solution) for the next week based on the results of the last week and the upcoming rival.

Primary Steps. In the algorithm, roulette is adopted to determine the game results, and the probability of winning the game is related to fitness. The probability of team *i* winning can be calculated as shown in Eqn. (6).

$$p_i^t = \left[f(X_j^t)-f^*\right] \Big/ \left[f(X_j^t)+f(X_i^t)-2f^*\right] \tag{6}$$

Here, X_i^t and X_j^t are the formation of team *i* and team *j* respectively; f^* represents the optimal fitness.

Every team *i* will adjust the formation of the last week $X_i^t=(x_{i1}^t,x_{i2}^t,...,x_{in}^t)$ when preparing the game of the next week *(t+1)*. We assume the scene that in the *t*th week, team *i* competed with team *j*, team *l* competed with team k, and in the *(t+1)*th week, team *i* will compete with team *l*. The general idea of the readjustment is that adjusting the formation to move closer with the winner, and meanwhile, to move oppositely from the looser. In more detail, the specific description is as follows.

a) If both team *i* and team *l* are winners in *t*th week, then the new formation can be obtained from Eqn. (7).

$$x_{id}^{t+1}=b_{id}^t+y_{id}^t(c_2r_1(x_{id}^t-x_{kd}^t)+c_2r_2(x_{id}^t-x_{jd}^t)) \qquad \forall d=1,...,n \tag{7}$$

b) If team *i* is the winner and team *l* is the looser in the *t*th week, then the new formation can be derived from Eqn. (8).

$$x_{id}^{t+1} = b_{id}^{t} + y_{id}^{t}(c_1 r_1(x_{kd}^{t} - x_{id}^{t}) + c_2 r_2(x_{id}^{t} - x_{jd}^{t})) \qquad \forall d = 1,...,n \tag{8}$$

c) If team i is the looser and team l is the winner in the tth week, then the new formation can be derived from Eqn. (9).

$$x_{id}^{t+1} = b_{id}^{t} + y_{id}^{t}(c_2 r_1(x_{id}^{t} - x_{kd}^{t}) + c_1 r_2(x_{jd}^{t} - x_{id}^{t})) \qquad \forall d = 1,...,n \tag{9}$$

d) If team i and team l are all losers in the tth week, then the new formation can be derived from Eqn. (10).

$$x_{id}^{t+1} = b_{id}^{t} + y_{id}^{t}(c_1 r_1(x_{kd}^{t} - x_{id}^{t}) + c_1 r_2(x_{jd}^{t} - x_{id}^{t})) \qquad \forall d = 1,...,n \tag{10}$$

Here, b_{id}^{t} is the dth component in the best formation of team i; r_1 and r_2 conform to the uniform distribution in $[0,1]$; c_1 and c_2 are fixed parameters, respectively representing the weights of superiority and inferiority; $y_{id}^{t} \in \{0,1\}$ represents whether the component needs to be changed, and the probability of changing is simulated through the truncated geometry distribution.

Algorithm Process. The allocation matrix with m rows and n columns represents a solution of the winner determination problem: $\{r_{11}, r_{12}, \cdots, r_{mn}\}$ with $r_{ij} \in \{0, 0.1, 0.2, \cdots, 1\}$ using discrete encoding. The steps of the algorithm are as follows.

Step 1: Initialize L (number of teams) and $Seasons$ (number of seasons); set $t = 1$, $s = 1$; generate the schedule;

Step 2: Process the multi-objective problem and constraints to simplify the problem based on Sec. 3.1 and Sec. 3.2.

Step 3: Calculate the strength (fitness) of each team, set the current formation to be the best formation of the teams;

Step 4: If $s \le Seasons$, go to step 5; otherwise, the algorithm ends;

Step 5: Teams compete according to the schedule of the tth week, and determine the game results based on Eqn. (6);

Step 6: Set $t = t + 1$; every team adjusts formation according to Eqn. (7)-Eqn. (10);

Step 7: Conduct constraint adjustments based on the steps in Sec. 3.2; if the adjusted formation is better, the current formation is replaced by it;

Step 8: If $t = L - 1$, set $t = 1$, $s++$, generate new schedule, and go to step 4.

4 The Reputation System

The feedback evaluation based reputation system is meant to reduce malicious behavior and improve the satisfaction in the auction. The reputation is decided by the evaluation from feedback, and the evaluation is based on the performance. $evaluation_{i,j} \in [-1,1]$ is the evaluation of i on j.

Firstly, the reputation is based on the evaluation and the amount of transaction. Secondly, the reputation is time-sensitive, and it declines over time. Finally, to ensure correct operation, the reputation system must prevent the defaming action and speculation of the reputation. So in this paper, a hierarchy of users is introduced to reduce the impact of malicious behavior.

If an evaluation that a participant received is of great difference with his/her current reputation, then this evaluation is regarded to be abnormal, and there may be the defaming action or speculation. We set $deviation = \left| evaluation_i - reputation_j \right|$, and divide the malicious behavior into n levels based on the value of $deviation$, where each level corresponds to a coefficient of suspicious degree $co_k \in (0,1]$. Based on the above, the credibility of a participant can be calculated from Eqn. (11).

$$CR_i = \left(N_i - co_k \times N_{i*}\right) \Big/ N_i \tag{11}$$

Here, N_i is the total number of times that participant *i* evaluated others; N_{i*} is the number of times that his/her evaluation was thought to be malicious.

To summarize, the reputation can be calculated from Eqn. (12).

$$\begin{aligned} reputation_{j,k} = {} & de(\Delta t) \times \left(1 - CR_i\right) \times \frac{transaction_{j,k-1}}{transaction_{j,k}} \times reputation_{j,k-1} + \\ & \left(1 - de(\Delta t)\right) \times CR_i \times \frac{R_{i,j} \times price_{i,j,k}}{transaction_{j,k}} \times evaluation_{i,j} \end{aligned} \tag{12}$$

Here, $transaction_{j,k}$ represents the transaction amount of CSC_j or CSP_j after the *k*th transaction; $R_{i,j} \times price_{i,j,k}$ is the amount of the *k*th transaction; $de(\Delta t)$ is the attenuation coefficient of time, as shown in Eqn. (13).

$$de(\Delta t) = \begin{cases} 1 & \Delta t < t_0 \\ \left(\dfrac{t_1 - \Delta t}{t_1 - t_0}\right)^k & t_0 \le \Delta t \le t_1 \\ 0 & t_1 < \Delta t \end{cases} \tag{13}$$

Here, $\Delta t = t_k - t_{k-1}$ is the interval between *k*th transaction and the *(k+1)*th transaction.

5 Price Decision

Introducing the mechanism of bidding/asking price decision is to let PA and CA intelligently decide the bidding/asking price, and increase their income, also cut the loss of incomplete information. Factors influencing the decision are varying and complicated, and there does not exist a formula that can take all these factors into consideration while making decisions. Therefore, BP neutral network [14] is adopted in this paper to

address this problem. The fact that every function can be simulated by a three-layer neutral network in arbitrary precision has been proved. The main idea is that historical data are used as samples to train the neural network, to get the matrix of weights, and then decide to the bidding/asking price.

5.1 Prediction of Supply-Demand Relation

Supply-demand relation in the market is one factor that can influence the decision, so it is needed to predict the supply-demand relation in the *t*th transaction for PA and CA to make decisions through the relationship in previous $t-1$ transactions.

In this paper, a single-index moving method is utilized to predict the supply-demand relation, and Eqn. (14) shows the calculation of SA_t (the prediction of the relationship in the *t*th transaction).

$$SA_t = \alpha sd_{t-1} + (1-\alpha)SA_{t-1} \tag{14}$$

According to the recursive rule, we can derive Eqn. (15).

$$SA_t = \alpha \sum_{j=0}^{t-1} (1-\alpha)^j sd_{t-j-1} + (1-\alpha)^t SA_0 \tag{15}$$

Here, $\alpha(0 \le \alpha \le 1)$ is the smoothing factor; sd_i is the real supply-demand relation in the *i*th transaction; SA_0 is the initial value of prediction.

5.2 Bidding/Asking Price Decision

The factors that a CA considers when deciding the bidding price are as follows.

(1) Supply-demand relation: If oversupply occurs, decrease lower the bidding price; otherwise, increase the price.

(2) Latest start time: If the interval between this time and the current time is short, which means the demand is urgent, bidding price should be increased.

(3)Budget: This is the upper bound of the bidding price.

(4) Current period: The bidding price should be higher in the peak time.

(5) Reputation: A participant win with high reputation is more likely to win. Those with high reputation can lower their bidding prices.

(6) Risk preference: Risk-seeking participants can bid lower than risk-averse ones.

Instead of the *latest_start_time* and the *budget*, a PA should consider the *current_load* and *cost* when deciding the asking price. The other four factors are the same as those that a PA considers.

The specific description of the algorithm is as follows.

Step 1: Initialize samples; set the minimum number of samples that are needed to train the neural network;

Step 2: Test the number of samples, and if it is insufficient, randomly generated bidding/asking prices as used samples, then go to Step 4; otherwise, go to Step 3;

Step 3: Normalize the samples, train the neural network, and save the matrix of coefficients; use the neural network to determine the bidding/asking price, and train the neural network at intervals;

Step 4: After the auction ends, save it in the database.

6 Simulation and Evaluation

The double combinatorial auction mechanism is implemented and evaluated based on Simjava2.0 toolkit [15] on the Eclipse platform. The three types of services are set according to the cloud computing platform of Amazon [16].

The values of the different supply-demand relations are as follows: short supply (S_S) (0.4~0.6), balance (Ba) (0.9~1.1), oversupply (O_S) (1.4~1.6), and sufficient resource (S_R) (≥4). The market size can be divided into six types: smaller size (Sm) (8 services, 4 resources), small size (S_er) (16 services, 4 resources), medium size (Me) (32 services, 4 resources), large size (La) (64 services, 4 resources), larger size (L_er) (128 services, 4 resources), superior (Su) size (128 services, 8 resources).

Our simulation results demonstrate the superiority of the proposed algorithm over its counterparts in Ref. [10], which applies a stable continuous double auction. Each datum in the following figures is the average of 20 trials under the corresponding conditions. The parameters of the league championship algorithm (LCA) are detailed in Table. 1.

Table 1. Parameters of LCA

Parameter	Valuation	Meaning
iterations	20	Number of iteration
teamNum	20	Number of team
c_1	0.5	Weight of superiority
c_2	0.5	Weight of inferiority
p_c	0.01	Parameter of probability

The comparisons on the total market surplus and the unit market surplus with different supply-demand relations between the double combinatorial auction with league championship algorithm (DCA_LCA) proposed in this paper and the stable continuous double auction (SCDA) proposed in [4] are presented in Fig. 2, respectively. The value of the best market surplus is set to be 1.

As shown in Fig. 2, in most cases, the performance of DCA_LCA is superior to that of SCDA in both the total market surplus and the unit market surplus; but in the condition that resources are sufficient, SCDA is slightly better. It can be explained as follows. The service that one CSC demands will not be allocated to multiple CSPs in SCDA. A CSC can always get the cheapest resources when resources are sufficient. While the randomness of LCA cannot guarantee no divided allocation, and this leads to slightly poorer performance. However, in the cases with capacity constraints, DCA_LCA is much better than SCDA. In addition, the tighter the constraints are, the more obvious the superiority is.

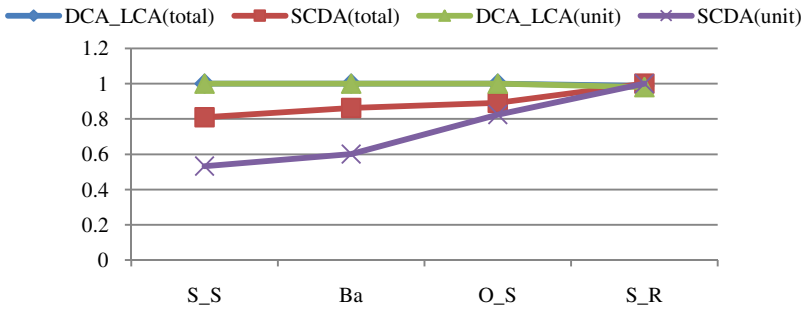


Fig. 2. Comparison on Total/Unit Market Surplus with Different Supply-demand Relation

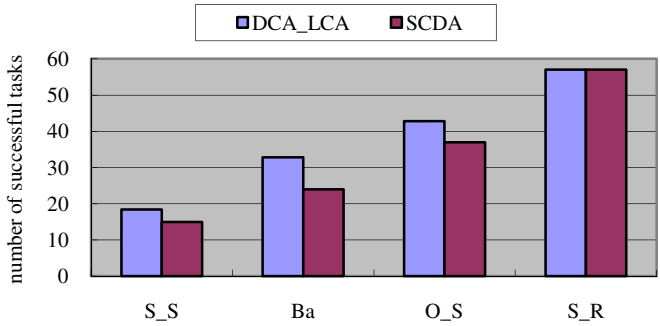


Fig. 3. Comparison on Number of Successful Tasks with Different Supply-demand Relation

We can see from Fig. 3 that, the number of successful tasks in DCA_LCA is the same with that in SCDA when resources are sufficient, and the number achieves the maximum. In cases of other supply-demand relation, the number of successful tasks is more in DCA_LCA. This is because DCA_LCA can divide the service and allocate each part to several different CSPs, and then more demands can be satisfied together.

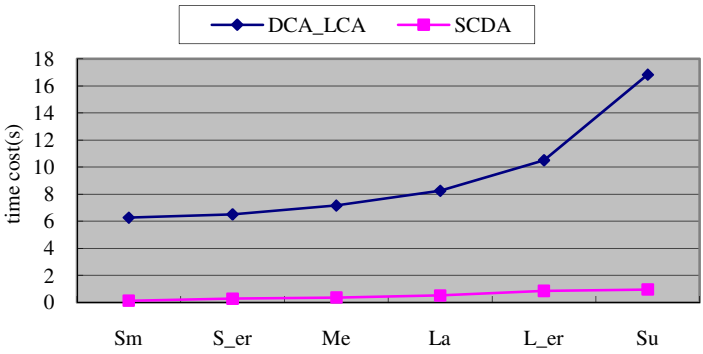


Fig. 4. Comparison on Time Cost with Different Market Size

From Fig. 4, we can conclude that DCA_LCA spends more time than SCDA does. As the market size expands, the time cost of DCA_LCA increases noticeably, and the maximum is 20s, which can also be accepted. If the conditions of the hardware improve, the time cost will become more acceptable.

7 Conclusion

In this paper, methods in economics and intelligent algorithms are introduced to propose an effective resource allocation mechanism in cloud computing. A feedback evaluation based reputation system is implemented to avoid malicious behavior, and BP neural network based price decision mechanism is proposed to determine prices scientifically. Finally, the winner determination problem is optimized by league championship algorithm with optimization goals being maximizing market surplus and total reputation. Simulation results validate the feasibility and demonstrate the superiority of the proposed mechanism on improving both market surplus and success ratio.

Acknowledgements. This work is supported by the National Science Foundation for Distinguished Young Scholars of China under Grant No. 61225012; the National Natural Science Foundation of China under Grant No. 61070162, No. 71071028 and No. 70931001; the Specialized Research Fund of the Doctoral Program of Higher Education for the Priority Development Areas under Grant No. 20120042130003; the Specialized Research Fund for the Doctoral Program of Higher Education under Grant No. 20100042110025 and No. 20110042110024; the Specialized Development Fund for the Internet of Things from the ministry of industry and information technology of the P.R. China; the Fundamental Research Funds for the Central Universities under Grant No. N110204003 and No. N120104001.

References

1. Vasan, R.: A venture perspective on cloud computing. IEEE Computer 44(3), 60–62 (2011)
2. Papazoglou, M.P., van den Heuvel, W.: Blueprinting the cloud. IEEE Internet Computing 15(6), 74–79 (2011)
3. Mell, P., Grance, T.: Definition of cloud computing. Technical report, National Institute of Standard and Technology (NIST) (July 2009)
4. Tan, Z., Gurd, J.R.: Market-based grid resource allocation using a stable continuous double auction. In: IEEE/ACM International Conference on Grid Computing, pp. 283–290 (September 2007)
5. Danak, A., Mannor, S.: Efficient bidding in dynamic grid markets. IEEE Transactions on Parallel and Distributed Systems 22(9), 1483–1496 (2011)
6. Tsai, C.W., Tsai, Z.: Bid-proportional auction for resource allocation in capacity-constrained clouds. In: International Conference on Advanced Information Networking and Applications Workshops (WAINA), pp. 1178–1183 (March 2012)
7. Prodan, R., Wieczorek, M., Frad, M.: Double auction-based scheduling of scientific applications in distributed grid and cloud environment. Journal of Grid Computing 9(4), 531–548 (2011)

8. Shang, S., Jiang, J., Wu, Y., Yang, G., Zheng, W.: A knowledge-based continuous double auction model for cloud market. In: International Conference on Semantics Knowledge and Grid (SKG), pp. 129–134 (November 2010)
9. Song, B., Hassan, M.M., Huh, E.N.: A novel cloud market infrastructure for trading service. In: International Conference on Computational Science and Its Applications (ICCSA), pp. 44–50 (June 2009)
10. Rassenti, S.J., Smith, V.L., Bulfin, R.L.: A combinatorial auction mechanism for airport time slot allocation. The Bell Journal of Economics 13(2), 402–417 (1982)
11. Xia, M., Stallaert, J., Andrew, B.: Solving the combinatorial double auction problem. European Journal of Operational Research 164(1), 239–251 (2005)
12. Kashan, A.H.: League Championship Algorithm: A new algorithm for numerical function optimization. In: International Conference of Soft Computing and Pattern Recognition, pp. 43–48 (2009)
13. Deb, K.: An efficient constraint handling method for genetic algorithms. Computation Methods in Applied Mechanics and Engineering 86(2-4), 311–338 (2000)
14. Hecht-Nielsen, R.: Theory of the backpropagation neural network. In: International Joint Conference on Neural Networks, pp. 593–605 (1989)
15. Howell, F., McNab, R.: Simjava: a discrete event simulation package for Java with applications in computer systems modelling. In: First International Conference on Web-based Modelling and Simulation (1998)
16. Amazon Web Services, `http://aws.amazon.com/`

Accelerating Software Model Checking Based on Program Backbone

Kuanjiu Zhou, Jiawei Yong, Xiaolong Wang*, Longtao Ren, Gang Hou, and Junwang Chang

School of Software, Dalian University of Technology (DUT), Dalian 116620, China
wxl_dut@163.com

Abstract. Model checking technique has been gradually applied to verify the reliability of software systems. However, as to software with large scale and complex structure, the verification procedure suffers from the state space explosion, thus leading to a low efficiency or resource exhaustion. In this paper, we propose a method of accelerating software model checking based on program backbone to verify the properties in ANSI-C source codes. We prune the program with respect to the assertion property, and compress the circular paths by maximal strongly connected components to extract the program backbone. Subsequently, the Hoare theory is used to generate an invariant from the compressed circular nodes, which reduces the length of path encoding. The assertion property is finally translated into a quantifier-free formula and checked for satisfiability by the SMT solver Z3. The experiments show that this method substantially improves the efficiency of program verification.

Keywords: Model Checking, Program Backbone, Path Compression, Program Verification.

1 Introduction

With the rapid development of computer technology, the application fields of software are growing wider. However, the exponential growth on software scale easily leads to some unpredictable crashes. Therefore, many researchers pay close attention to find out the loopholes in the early stage of software design and development. The main methods of software quality assurance include software testing, model checking [1], and theorem proving [2]. But it is difficult for software testing to detect all execution paths, and theorem proving cannot be accomplished automatically limited by complex data structures and low performances. Along with the increasing demands on software performance, the verification of software source code combined with model checking has been a significant method.

Some valuable researches focusing on model checking of software source code have been carried out in recent years. The Turing Award winner, Prof. E.M. Clarke has made a research in CBMC [3-4] and SATABS [5], which support full or rich subset of ANSI-C programs. CBMC and SATABS use predicate abstraction

* Corresponding author.

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 347–358, 2013.
© Springer-Verlag Berlin Heidelberg 2013

techniques to check properties including pointer safety, array bounds, and user-provided assertions, but these model checkers have difficulty in verifying more complex software systems since they need to unwind the loops in the programs at a certain bound. Some other existing tools such as SLAM [6], BLAST [7], MAGIC [8], or NuSMV [9], a new symbolic model verification tool, also achieve good results in software verification. However, most of them accept proprietary input languages that are not used for programming. F-SOFT [10-11] is developed on the basis of BLAST, using the lazy abstraction approach and localization techniques. It firstly takes program blocks as the primary elements to build an LTG (Labeled Transition Graph), then translates the control logic and data logic into Boolean representation respectively, and finally uses a SAT (Satisfiability Problems) solver to verify the program. In order to deal with more theory domains, SMT-BMC [12] adopts the more powerful SMT (Satisfiability Modulo Theories) [13] solvers instead of SAT solvers to verify software programs, but it still needs to unwind the loops with the BMC (bounded model checking) approach. Furthermore, Lucas [14] also leads some valuable research focusing on bounded model checking of embedded software.

When verifying the properties in source code, it would cause the space explosion problem [15-16] if we traverse all the program paths. That problem severely restricts the scale of verified software. The key remission approach is simplifying program verification process as far as possible. In this work, we extract program backbone from software source code to accelerate software model checking. Firstly we treat the assertion property as guidance, pruning the program parts irrelevant to the property; then we compress the circular paths by maximal strongly connected components. Finally we encode the property and program backbone into a quantifier-free formula to check for satisfiability of the assertion property automatically.

The outline of the rest of this paper is as follows. In Section 2, we systematically describe our method of extracting program backbone, including pruning with respect to property and compressing circular paths. In Section 3, we present the path encoding for Z3 [17]. Section 4 shows the results and evaluation of our experiments. Section 5 makes conclusions and describes our future work.

2 Extracting Program Backbone

The program backbone is a core framework of software source code obtained through the procedure of pruning and compression. By extracting program backbone, we can remove the data variables, branch conditions, and assignment statements which are irrelevant to the assertion properties. Meanwhile, the complex structures such as loops and recursions will be hided, thus simplifying the complexity of the final predicate logic formula and improving the efficiency of program verification.

2.1 Pruning with Respect to Property

Pruning with respect to property is a useful way of reducing program size and formula complexity to allow more efficient model checking. Some existing software model checkers also adopt a similar method named program slicing [18-19] to remove the irrelevant program parts, but most of them are explicit state model checkers [20-21].

As to the assertion property to be checked, some existed statements of the source program are insignificant or have no influence on the verification result. In the model checking of program properties, it is unnecessary to encode all the statements. Therefore, we should make best efforts to remove those irrelevant program parts, and we call it pruning in this work. Before processing pruning, we present the definition of the dependency between properties and variables.

Definition 1 - Property Dependency. We use the letter P to denote the program, V_p to denote the variable set of P, $Expr(v)$ to denote the expressions including variable v. For $v \in V_p$, $u \in V_\varphi$, if one of the following conditions is satisfied:

(1) $v \in V_\varphi$;

(2) P contains assignment statements with the format of $u=Expr(v)$;

(3) P contains guards with the format of $Expr(u) \otimes Expr(v)$, where $\otimes \in \{<, <=, ==, !=, >=, >\}$;

(4) P contains assignment statements $u=Expr(v)$ or guards $Expr(u) \otimes Expr(v)$, and a series of assignments $v_i=Expr(v_{i-1})$, … , $v_2=Expr(v_1)$, $v_1=Expr(v)$, where $v_i \in V_p$, $i \in N^+$;

Then we call that the variable v is *relevant* to the property φ; otherwise, the variable v is *irrelevant* to the property φ.

Note that, we cannot consider the variable v to be relevant to the property φ even if P contains assignment statements with the format of $v=Expr(u)$.

Definition 2 - Pruning Rule. If the variable v is irrelevant to the property φ, prune the statements containing v in the program P.

```
0:  #define SIZE 6
1:  void example() {
2:  int m, n, i, sum;
3:  int a[] = {7,3,8,4,15,6};
4:  m=n=sum=0;
5:  i=0;
6:  while(i<SIZE){
7:    if(m<a[i]){
8:      m=a[i];
9:    }
10:   if(n>a[i]){
11:     n=a[i];
12:   }
13:   sum=sum+a[i];
14:   i++;
15:  }
16: ASSERT(sum<=SIZE*m);
17: }
```

(a)

```
0:  #define SIZE 6
1:  void example() {
2:  int m, i, sum;
3:  int a[] = {7,3,8,4,15,6};
4:  m=sum=0;
5:  i=0;
6:  while(i<SIZE){
7:    if(m<a[i]){
8:      m=a[i];
9:    }
10:   sum=sum+a[i];
11:   i++;
12:  }
13: ASSERT(sum<=SIZE*m);
14: }
```

(b)

Fig. 1. An example of an ANSI-C program and its minimal pruning program

Definition 3 - Minimal Pruning Program. For a pruned program P_{min} based on the property φ, if no other pruned programs based on the same property contain less statements than P_{min}, then we call P_{min} is the *minimal pruning program* with the property φ.

Fig. 1 shows an example of (a) an ANSI-C program and (b) its minimal pruning program. In the example, the variable *n* is irrelevant to the assertion *sum<=SIZE*m*, so we can prune the statements containing *n* in the program.

Note that even though the pruning procedure above has removed the statements irrelevant to the assertion property, the pruned program can still execute without influence. Besides, the final status and property will not be changed either when the program is executed to the assertion.

2.2 Compressing Circular Paths

After pruning with respect to property, we have accomplished the primary treatment for the program. Now we will continue to abstract the pruned program by compressing circular path in order to obtain the program backbone. The program backbone is dynamically extracted in accordance with the property. A common method of ensuring properties in programs is the form of assertions.

In Definition 4, we depict the program including assertions with control flow graph (CFG).

Definition 4 - Control Flow Graph. The control flow graph of a program is a directed graph composed of finite nodes and edges. It can be described by a tuple $G^{cfg} :< N^{cfg}, op, E^{cfg}, N_{in}^{cfg}, N_{out}^{cfg} >$.

N^{cfg} denotes the finite nodes set of CFG, the same as the finite set of code lines.

op denotes the operation set of the program. It consists of assignment statements and branch statements.

E^{cfg} denotes the edges set of CFG, $E^{cfg} \subseteq N^{cfg} \times op \times N^{cfg}$.

N_{in}^{cfg} denotes the root node of CFG, $N_{in}^{cfg} \in N^{cfg}$.

N_{out}^{cfg} denotes the node to be checked in CFG, $N_{out}^{cfg} \in N^{cfg}$.

Therefore, the state transition relation can be intuitively expressed by the CFG. On the basis of that, we regard the maximal strongly connected components [22] as compressed objects to obtain the compressed control flow graph.

The maximal strongly connected component means that any two nodes have a path reaching each other in connected nodes set. Here we use only one abstract node instead of each maximal strongly connected component to compress the CFG. The algorithms searching for the maximal strongly connected component in CFG refer to Kosaraju algorithm [23], Tarjan algorithm [24], or Gabow algorithm [25]. Definition 5 reveals the principle of compression.

Definition 5 - Compression Principle

(1) Find out the maximal strongly connected components in CFG, and merge the connected nodes into one node with the entry edge and exit edge unchanged.

(2) If there are some assignment statements, such as *u=Expr(v)*, next to others, merge them into one node with the entry edge and exit edge unchanged.

After the compression operation, the CFG will be translated into a compressed CFG. Here we give the definition of compressed control flow graph(CCFG) as follows.

Definition 6 - Compressed Control Flow Graph

Assume a directed graph $G^{ccfg}:(N^{ccfg}, op, E^{ccfg}, N_{in}^{ccfg}, N_{out}^{ccfg})$, we define it as a CCFG if the following conditions are satisfied.

N^{ccfg} denotes the nodes set of CCFG, the nodes represent the maximal strongly connected components.

E^{ccfg} denotes the edges set of CCFG, the edges represent the connection between two nodes.

op, N_{in}^{cfg}, N_{out}^{cfg} are similar with Definition 4.

The example program in Fig. 1 can be graphically shown in the Fig. 2. The CFG of the raw program is displayed in (a), the CFG of the pruned program is displayed in (b) and the CCFG of the program backbone according to the compression principle is in (c).

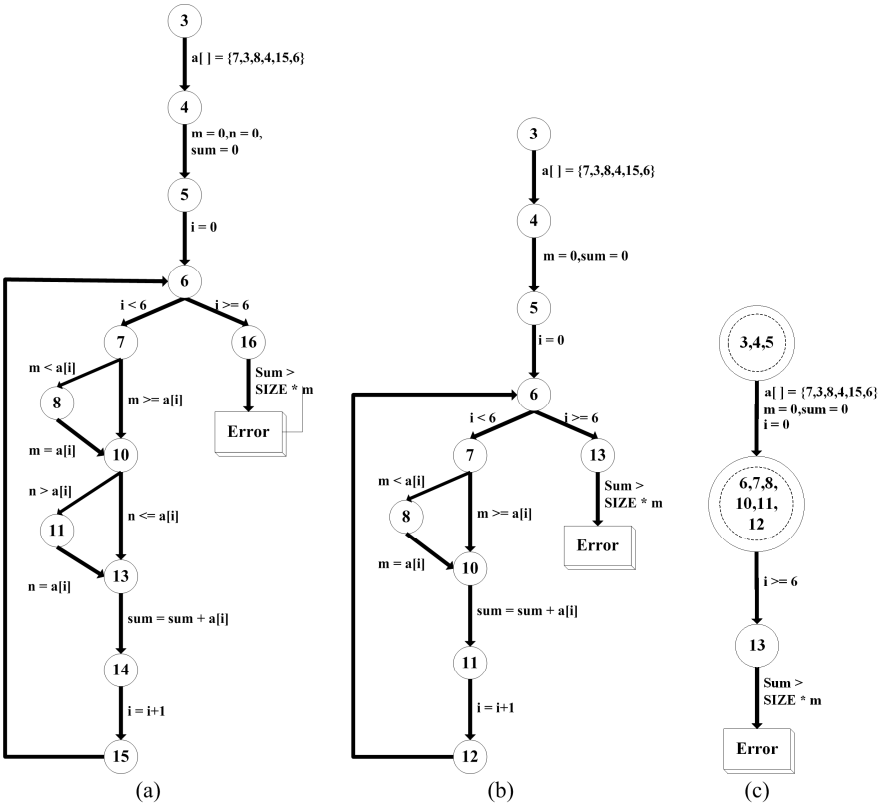


Fig. 2. The CFGs of the raw program, pruned program, and the program backbone

The CCFG is a directed acyclic graph which is able to avoid analysis of complex structures, as we consider the compressed maximal strongly connected component as a whole node. Thus we have accomplished the extracting procedure of program backbone.

3 Path Encoding

The program backbone has been extracted from the source codes after pruning with respect to property and compressing the circular path. In this section, we present the detailed methods of path encoding to Boolean form for software model checking. The path encoding will firstly use Hoare theory to find out an invariant for each compressed circular node of the program, and then translate the program backbone into the single assignment form. Finally, we make a conjunction of the whole program conditions and the negation of the assertion property to generate a quantifier-free first order predicate logic formulas, which can be easily compatible with an SMT solver for verification.

3.1 Invariants of Compressed Circular Nodes

In the previous section, we have abstracted the circular paths of the program into compressed circular nodes, hereinafter called abstract nodes for simplicity. The abstract nodes ignore the details in the loop structures, making the program backbone much easier to SMT solvers. To this end, we need to obtain an invariant for each abstract node to replace a long encoding of the complicated loop blocks in the program. The following paragraphs will give the detailed approach to find out the invariants of abstract nodes.

As a start, we introduce Hoare theory [26] into software model checking. The basic formula of Hoare's logic can be denoted by a triple $\{\varphi\}S\{\psi\}$, where φ and ψ are first order formulas representing assertion properties. S denotes a segment or statement of a program, including the assignment statements, skip statements, if-then-else branch statements, and while-loop statements. The Hoare triple $\{\varphi\}S\{\psi\}$ means that φ holds before the execution of S, and then ψ holds after the execution of S terminates.

Based on Hoare theory, the while-loop of a program satisfies an invariant φ, which can be hold before and after each loop iteration, that is the while rule as follows.

$$\frac{\{\varphi \wedge p\}S\{\varphi\}}{\{\varphi\}\mathit{while}\ p\ \mathit{do}\ S\ \mathit{end}\{\varphi \wedge \neg p\}} \quad . \tag{1}$$

Here, p is the quantifier-free first order assertion condition of the while-loop. Since p shall hold before each execution of the loop body S and certainly do not hold when the program exit the loop, so we use $\varphi \wedge p$ as the precondition of the loop body S, and $\varphi \wedge \neg p$ as the postcondition. Therefore, the while rule means that φ holds after

each execution of loop body *S*, including the last execution, from any statements that satisfy $\varphi \wedge p$. The φ above is rightly the invariant of the compressed circular node we need.

On the basis of the theory described above, we can further find out the invariants of abstract nodes through the inductive method. Here we take the program backbone in Fig. 2(c) as an example in order to show how to figure out the invariants and proof procedure.

Initial condition: $i = 0 \wedge m = 0 \wedge sum = 0$;

After the 1st execution of the loop body *S*, it holds $i = 1 \wedge m = \max\{a[0]\} \wedge sum = a[0]$;

After the 2nd execution of the loop body *S*, it holds
$i = 2 \wedge m = \max\{a[0], a[1]\} \wedge sum = a[0] + a[1]$;

And so forth, the invariant after the kth execution of the loop body *S* can be inducted as follows.

$$(i = k) \geq 0 \wedge m = \max_{j=0}^{k-1} a[j] \wedge sum = \sum_{j=0}^{k-1} a[j]. \tag{2}$$

Proof. Based on the *if- then-else* rule [27],

$$\{m = \max_{j=0}^{i-1} a[j] \wedge m < a[i]\}\ if\ \left(m < a[i]\right)\ m = a[i]\ \{m = \max_{j=0}^{i} a[j]\}. \tag{3}$$

By the assignment axiom [27],

$$\{sum = \sum_{j=0}^{i-1} a[j] \wedge sum + a[i] = \sum_{j=0}^{i} a[j]\}\ sum = sum + a[i]\ \{sum = \sum_{j=0}^{i} a[j]\}; \tag{4}$$

$$\{i + 1 \geq 0\}\ i{+}{+}\ \{i \geq 0\}. \tag{5}$$

Now

$$m{=}0 \rightarrow m{=}\max_{j=0}^{i-1} a[j] \wedge m{<}a[i]; \tag{6}$$

$$sum = 0 \rightarrow sum = \sum_{j=0}^{i-1} a[j] \wedge sum + a[i] = \sum_{j=0}^{i} a[j]; \tag{7}$$

$$i = 0 \rightarrow i + 1 \geq 0. \tag{8}$$

holds; so (6) (7) (8) and (3) (4) (5) imply

$$\begin{aligned} &\{m = 0 \wedge sum = 0 \wedge i = 0\}\ if\ \left(m < a[i]\right) m = a[i];\ sum = sum + a[i]; i{+}{+} \\ &\{i \geq 0 \wedge m = \max_{j=0}^{i} a[j] \wedge sum = \sum_{j=0}^{i} a[j]\}. \end{aligned} \tag{9}$$

So the loop body *S* satisfies the while rule

$$\{i \geq 0 \wedge m = \max_{j=0}^{i} a[j] \wedge sum = \sum_{j=0}^{i} a[j] \wedge i < 6\}\ S\text{-}loop\{i \geq 0 \wedge m = \max_{j=0}^{i} a[j] \wedge sum = \sum_{j=0}^{i} a[j]\}. \tag{10}$$

Obviously, as to the circular node in the example program, its invariant is

$$i \geq 0 \wedge m = \max_{j=0}^{i} a[j] \wedge sum = \sum_{j=0}^{i} a[j]; \tag{11}$$

After the loop exits, it holds

$$i \geq 0 \wedge m = \max_{j=0}^{i} a[j] \wedge sum = \sum_{j=0}^{i} a[j] \wedge i \geq 6. \tag{12}$$

3.2 Translating into SSA Form

In this subsection, we interpret how to translate statements and variables of the program into static single assignment (SSA) form.

Assignment Statements. The assignment statement is the basic part of the program after pruning and compressing. At first, the SSA form needs to record the number of the variables' assignment operations (the number of variables which appear on the left of assignments). Let i denote the lines of codes, and N_i denote the statement corresponding to Line i. Before the statement N_i (not including the statement i), the number of any variable v's assignment operations is denoted by $ass(v, N_i)$:

(1) When $i = 1$, $ass\ (v, N_i) = 0$;

(2) When $i \geq 2$, and the N_{i-1} statement assigns a value to the variable v, $ass(v, N_i)= ass(v, N_{i-1})+1$;

(3) When $i \geq 2$, and the N_{i-1} statement does not assign a value to the variable v, $ass(v, N_i)= ass(v, N_{i-1})$;

Using the rules above, we are able to record the number of any variables' assignment operations at any code line, then rename the variable v appearing on the left side as $v_{ass(v,N_i)+1}$, the variable v on the right side as $v_{ass(v,N_i)}$. Thus we could accomplish the translation of SSA form on assignment statement by rename procedure.

Array Variables. The array variable is a special storage structure in the program. Assume that some values of array are updated, it is necessary for us to update the whole array. Here we introduce the *with*-operator to achieve it. Let e be a new value with the type of the ith element in array a, and the updated array can be expressed as $a' == (a\, with[i := e])$ by with-operator. It means

(1) The ith element $a'[i]$ of the updated array a' can be assigned to $a'[i] = e$;

(2) The rest elements of the updated array a' will be assigned to the value of the same elements, $a'[j] = a[j]$, $(0 \leq j < sizeof(a), j \neq i)$.

Branch Statements. Branch statement is also an important program part besides loops. Let the ternary operator $i\ ?\ t : e$ be an expression of the variables o, p, q, …, and o', p', q', …, denote the variables with updated values. Let i denote *if* conditions,

then i_o represents the conditions related to *o*. Let *t* denote *then* operations, then t_o represents the operations related to *o*. Let *e* denote *else* operations, then e_o represents the *else* operations related to *o*. So the SSA form of branch statements combined with the assignment statements can be defined as follows.

(1) When the branch statement *if-then-else* assigns values for some variables *o*, *p*, *q*, …, we should rewrite all these variables in SSA form. For example, if we give a new value to variable *o* in the *then* or *else* statement, the new value in the left side is $o_{ass(o,N_i)+1} = (i_o ? t_o : e_o)$. The variable *o* in the right side (i_o, t_o, e_o) stays unchanged.

(2) List all variables which have been assigned in branch statement. For example, *o*, *p*, *q* are variables assigned new values. We can rewrite the SSA forms as

$$o_{ass(o,N_i)+1} = (i_o ? t_o : e_o); \quad p_{ass(a,N_i)+1} = (i_p ? t_p : e_p); \quad q_{ass(a,N_i)+1} = (i_q ? t_q : e_q).$$

Consequently, we are able to obtain the whole program backbone's SSA form of the example program in Fig. 1.

3.3 Encoding SMT Formula

Now we insert the invariant of compressed circular nodes into the SSA form of program backbone based on Formula (11) in order to encode the whole program an SMT formula based on context. The constraints *C* of the example program Fig. 1 is

$$C := \begin{bmatrix} a_1 = store(store(store(store(store(store(a_0,0,7), \\ 1,3),2,8),3,4),4,15),5,6) \\ \wedge m_1 = 0 \\ \wedge sum_1 = 0 \\ \wedge i_1 = 0 \\ \wedge sum_2 = select(a_1,0) + select(a_1,1) + select(a_1,2) + \\ select(a_1,3) + select(a_1,4) + select(a_1,5) \\ \wedge m_2 = \max(select(a_1,0), select(a_1,1), select(a_1,2), \\ select(a_1,3), select(a_1,4), select(a_1,5)) \end{bmatrix}.$$

The properties *P* and its negation $\neg P$ are

$$P := \begin{bmatrix} i_2 \geq 0 \wedge i_2 < 6 \\ \wedge sum_2 \leq 6 * m_2 \end{bmatrix}, \quad \neg P := \begin{bmatrix} i_2 < 0 \vee i_2 \geq 6 \\ \vee sum_2 > 6 * m_2 \end{bmatrix}.$$

Finally, the SMT formula $C \wedge \neg P$ would be passed to an SMT solver to check for satisfiability.

4 Experiments and Evaluations

After getting the SMT formula through pruning, compressing, and encoding by the method above, we could further check for its satisfiability by an SMT solver. In this work, we choose Z3 [17] as our experimental tool. Z3 is a powerful SMT solver which integrates several decision procedures. It has a wide range of solution domains including linear real and integer arithmetic, uninterpreted functions, extensional arrays, and several input formats. Compared with other SMT solvers, Z3 also supports model generation, thus having an advantage in complex program.

To demonstrate the correctness and effectiveness of our method, we choose some representative programs. The example program in Fig. 1 is verified as the first experimental case. It consists of the array summation algorithm SumArray and the maxima and minima algorithm MinMax. We select the Division algorithm by circular subtraction, intuitively showing the result of our method. The BubbleSort is chosen as another case because of its circular node. Meanwhile, we select the BellmanFord algorithm searching for the monophyletic minimum cost path. The famous Eight Queens problem and Prime Number algorithm are also checked in the experiments. To illustrate the situation for large scale program, we add two more large programs the Red Black Tree and the C-implemented HTTP protocol [28] to show the robustness of our method.

In order to illustrate the validity of our program backbone-based accelerating method, the regular method based on the raw source code without pruning and compressing is designed to verify the same program assertions as a comparison. Both of the two methods encode the program constraints C and assertion properties P into SMT formulas $C \wedge \neg P$ and check satisfiability with Z3.

The experiments are performed on the Windows 7 operation system, with a 2.4GHz CPU (AMD Athlon II X4 610E) and 2GB RAM. The experiments adopt the Windows-based SMT Z3 in Version 4.3.0. Table 1 shows the time cost of the verification experiments and the number of failures which are caused by memory overflow or time-out.

Table 1. Experimental results

Test Cases	Lines of Codes	Assertions	Total Time (sec)		Failures Number	
			Regular	Our	Regular	Our
Example	17	1	0.30	0.17	0	0
Division	12	5	1.48	1.21	0	0
BubbleSort	43	17	1.54	0.82	0	0
BellmanFord	49	33	0.75	0.24	3	1
EightQueens	83	65	15.93	16.70	8	3
PrimeNumber	20	12	10.26	9.13	2	0
Red-Black tree	291	163	134.76	80.15	11	4
HTTP protocol	242	151	121.94	102.78	17	5

The results in Table 1 indicate that the total time is substantially declined and the number of failures is reduced in our method. Some inevitable failures occur in the verification process of the latter two large scale cases, but most of them attribute to memory overflow and time-out. We can conclude that the software model checking based on program backbone improves the verification efficiency than the common method directly based on source code.

5 Conclusions

In this work, we propose a backbone-based software model checking approach to accelerate the verification of properties in ANSI-C programs. The program backbone is obtained on the basis of pruning with respect to property, and compressing the loops into abstract nodes instead of unwinding them limited to a certain bound like CBMC or SMT-CBMC. We further make use of Hoare theory to generate an invariant from the compressed circular node to replace the complicated logic of the loop body, which reduces the length of path encoding to a great extent. The experimental results show that our method based on program backbone accelerates the efficiency of software model checking compared with the common method directly based on source code. However, some details can still be improved in this work. We will continue to extend our method to verify multi-threaded concurrent programs and try to implement an integrated software verification tool in the future.

Acknowledgments. This work is supported by the National Nature Science Foundation of China under Gant No. 61272174.

References

1. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys (CSUR) 41(4), 21 (2009)
2. Schumann, J.M.: Automated theorem proving in software engineering. Springer (2001)
3. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
4. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. Formal Methods in System Design 25(2-3), 105–127 (2004)
5. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
6. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 1–20. Springer, Heidelberg (2004)
7. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. ACM SIGPLAN Notices 39(1), 232–244 (2004)
8. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE Transactions on Software Engineering 30(6), 388–402 (2004)
9. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: A new symbolic model verifier. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 495–499. Springer, Heidelberg (1999)

10. Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P.: F-SOFT: Software verification platform. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 301–306. Springer, Heidelberg (2005)
11. Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Ashar, P.: Efficient SAT-based bounded model checking for software verification. Theoretical Computer Science 404(3), 256–274 (2008)
12. Armando, A., Mantovani, J., Platania, L.: Bounded model checking of software using SMT solvers instead of SAT solvers. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 146–162. Springer, Heidelberg (2006)
13. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. Handbook of Satisfiability 185, 825–885 (2009)
14. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. IEEE Transactions on Software Engineering 38(4), 957–974 (2012)
15. Valmari, A.: The state explosion problem. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998)
16. Lin, F.J., Chu, P.M., Liu, M.T.: Protocol verification using reachability analysis: the state space explosion problem and relief strategies. ACM SIGCOMM Computer Communication Review 17(5), 126–135 (1987)
17. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
18. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages 3(3), 121–189 (1995)
19. Li, B.X., Zheng, G.L., Wang, Y.F., Li, X.D.: An Approach to Analyzing and Understanding Program - Program Slicing. Journal of Computer Research & Development 37(3), 284–291 (2000) (in Chinese)
20. Visser, W., Havelund, K., Brat, G., Park, S.: Model checking programs. In: Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering, ASE 2000, pp. 3–11. IEEE (2000)
21. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Zheng, H.: Bandera: Extracting finite-state models from Java source code. In: Proceedings of the 2000 International Conference on Software Engineering, pp. 439–448. IEEE (2000)
22. He, Y.X., Wu, W., Chen, Y., Xu, C.: Path sensitive program verification based on SMT solvers. Journal of Software 23(10), 2655–2664 (2012) (in Chinese)
23. Callahan, P.B., Kosaraju, S.R.: A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. Journal of the ACM (JACM) 42(1), 67–90 (1995)
24. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)
25. Gabow, H.N.: Path-Based Depth-first Search for Strong and Biconnected Components. Information Processing Letters (2000)
26. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
27. Apt, K.R.: Ten years of Hoare's logic: A survey—part I. ACM Transactions on Programming Languages and Systems (TOPLAS) 3(4), 431–483 (1981)
28. Del Val, D., Klemets, A.E.: Method and apparatus for communication media commands and media data using the HTTP protocol: U.S. Patent 6,128,653 (2000)

A Cloud Computing System for Snore Signals Processing

Jian Guo[1], Kun Qian[2], Zhaomeng Zhu[1], Gongxuan Zhang[1,*], and Huijie Xu[3,*]

[1] School of Computer Science and Engineering,
Nanjing University of Science and Technology, Nanjing, China
{johnkuo83,zhaomeng.zhu}@gmail.com,
gongxuan@njust.edu.cn
[2] School of Electronic and Optical Engineering,
Nanjing University of Science and Technology, Nanjing, China
isonar@sina.cn
[3] Beijing Hospital, Beijing, China
xhj0531@163.com

Abstract. Recently, snore signals (SS) have been demonstrated carrying significant information about the obstruction site and degree in the upper airway of Obstructive Sleep Apnea-Hypopnea Syndrome (OSAHS) suffers. To make this acoustic based method more accurate and robust, big SS data processing and analysis are necessary. Cloud computing has the potential to enhance decision agility and productivity while enabling greater efficiencies and reducing costs. We look to cloud computing as the structure to support processing big SS data. In this paper, we focused on the aspects of a Cloud environment that processing big SS data using software services hosted in the Cloud. Finally, we set up a group of comparable experiments to evaluate the performance of our proposed system with different system scales.

Keywords: cloud computing, big data, signal processing, snore signals (SS), Obstructive Sleep Apnea-Hypopnea Syndrome (OSAHS).

1 Introduction

Obstructive Sleep Apnea-Hypopnea Syndrome (OSAHS) is a prevalent disorder among community, which is estimated to affect 4% of the adult male population and 2% of the adult female population [1]. This chronic and easy-overlooked disorder has a high risk to trigger high blood pressure, coronary heart disease, pulmonary heart failure and even dangerous nocturnal death [2]. In medical practice, Polysomnogram (PSG) is regarded as the gold standard. However, the uncomfortable experience of patients and the expensive manufacture of equipment restrict its further developing and widely use. Sound snoring is a typical symptom of OSAHS therefore in the past 10-20 years numerous researchers and scholars focused on acoustic features analysis of snore signals (SS) generated by patients [3]-[6], which leads a relatively cheap and non-intrusive method of PSG. Nevertheless, there is a more demanding requirement

* Corresponding author.

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 359–366, 2013.
© Springer-Verlag Berlin Heidelberg 2013

from doctors to know the variations of the upper airway (UA), which is significant to help them to find the collapse site of the UA and adopt an accurate plan for surgery [2]. Long-time analysis of SS data is a good method to understand the relationship between acoustic analysis and anatomical theory. P. D. Hill et al. calculated crest factor of OSAHS patients and inferred that the substantial changes of this value during the night indicate the mechanism variations of snores [3]. W. D. Duckitt et al. studied all night SS data and built a system based on Hidden Markov Models (HMM) to detect, segment and assess the snores and acoustic signals [7].

All the scholars have taken no attention on the ability of their systems to process big data. Especially when the scales and samples of SS data become tremendously large, the processing and analysis will be a time-consuming and difficult task. Therefore, we need a powerful computing system to handle the huge amount of would-be big SS data. Cloud computing technology has the ability to afford a platform contains software services that are made available to consumers in a pay-as-you-go model [8]. In industry area the services are referred as Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS), respectively. Big data has three characteristics: volume, variety and velocity [8]. We can predict that with a lager and deeper collaboration of medical service all over the world, medical data such as SS data should be big data with no doubt. Some researchers have studied how to implement big medical data processing based on distributed and cloud computing technology [9]. Cloud computing, an emerging concept and technology, has motivated numerous researchers to exploit its super power in computing and storage by developing applications in biomedical treatment [15]. Encouraged by the high performance achieved by cloud system, we utilize it to deal with the processing of big SS data mentioned above. In this paper, we designed a cloud system includes IaaS, PaaS and SaaS to cope with big SS data, which is an original idea to conduct the medical cloud computing research for further work.

2 Analysis Methods and System Design

2.1 Signal Processing of SS Data

The original SS data are recorded by a high-quality microphone array positioned at a sleep laboratory and stored as audio files. Initially, the SS data will be divided into frames as samples based on Short-Time-Analysis (STA) approach. The frames can be regarded as samples for further processing and analysis. Subsequently, these samples will be pre-processed to eliminate the background noise and interference (which needs to utilize beam forming technology based on array signal processing). Then we extract frequently-used acoustic features from samples. Features extraction is the most essential step for machine learning, pattern recognition and establishment of SS database.

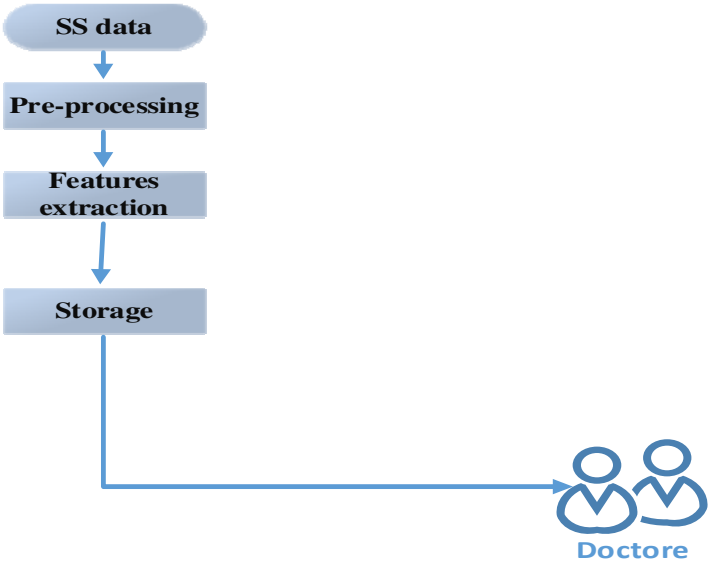


Fig. 1. The signal processing for SS data in the SaaS

Features Extraction. Snoring is the result of the vibration of the tissues when the air flow passes through the narrowing UA. It is widely accepted that the UA acts as an acoustic filter during the production of SS, hence, vital information of the structure of UA are carried by SS data.

Frequency domain features are also popular in medical researches [4]. The center point, peak point and mean point of the spectrum, f_{center}, f_{peak}, f_{mean} is defined as follows respectively:

$$\text{s.t.} \quad \sum_{f_i=0}^{f_{center}} S_{f_i} = \sum_{f_i=f_{center}}^{f_c} S_{f_i} \tag{1}$$

$$\text{s.t.} \quad S_{f_{peak}} = \max\left\{S_{f_i}, f_i = 0,\ldots,f_c\right\} \tag{2}$$

$$f_{mean} = \frac{\sum_{f_i=0}^{f_c} f_i * S_{f_i}}{\sum_{f_i=0}^{f_c} S_{f_i}} \tag{3}$$

Where, S_{fi} is the absolute amplitude spectra of SS at frequency of f_i Hz calculated by Fast Fourier Transform (FFT). And f_c is the cut-off frequency of the SS spectrum. Relative studies have indicated that these three features extracted from SS of OSAHS

patients are normally much higher than simple snorers [4]-[5]. The physical explanation of the phenomena is that the narrowing extent of the UA is more severe, the frequency of SS generated by air flow is higher [2].

Power ratio at the frequency of 800 Hz is capable to classify SRS generated by different obstruction site in UA []. We defined this feature as:

$$PR800 = \lg\left(\frac{\sum_{f_i=0}^{800} S_{f_i}^2}{\sum_{f_i=800}^{f_c} S_{f_i}^2}\right) \tag{5}$$

Totally, we extracted 4 acoustic features for establishment of a preliminary database of individual OSAHS patients.

2.2 Cloud Computing System Design

The overall functionality infrastructure of a big SS data analysis cloud computing system involves the following steps as Fig. 2 illustrates:

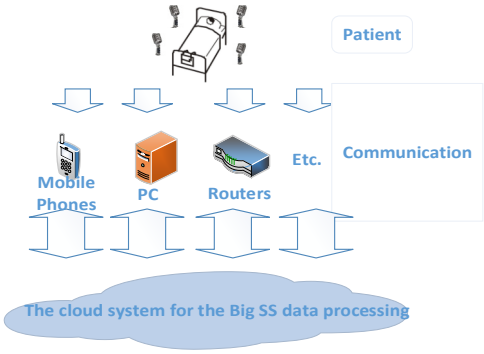


Fig. 2. The overall functionality infrastructure of a big SS data analysis cloud computing system

The architecture of cloud computing system is revealed in Fig. 3, which affords us a method to address the bid SS data in collection, storage and analysis. From bottom to top, each box represents IaaS, PaaS and SaaS layer, respectively. The software service is hosted as signal processing in order to make any client-side implementation simply call the underlying functions (e.g., Pre-Processing, Analysis, etc.) without going through the complexities of application.

IaaS Layer. To meet the needs of our study, we deployed a private cloud computing system with the help of OpenStack, a global collaboration of developers and cloud computing technologists on producing the ubiquitous open source cloud computing platform for public and private clouds. We utilize it to create and manage the instances which run big SS data processing programs.

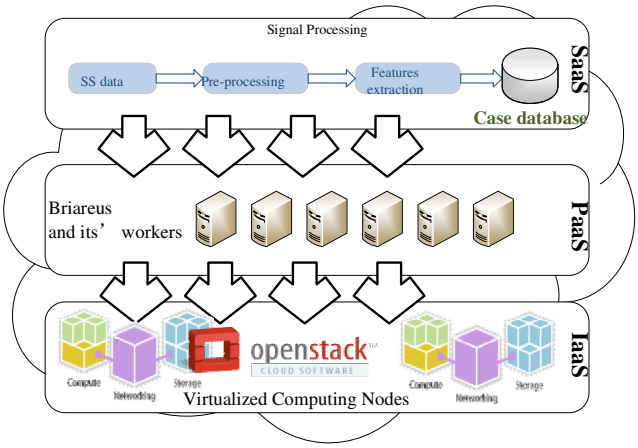


Fig. 3. The architecture of cloud computing system for big SS data processing

PaaS Layer. In PaaS layer, we adopted a framework called Briareus [16], which provides convenient tools to make use of computing resources provided by cloud to accelerate Python applications. MapReduce has motivated us to design the distributed computing methodology of Briareus. Briareus offers accelerating techniques into a Python program, which brings us a set of convenient to design this cloud computing system. All signal processing computation tasks can be migrated into cloud computing system, which will significantly enhance the efficiency of running programs.

SaaS Layer. Big SS data processing belongs to the SaaS layer, which should be developed due to the specific applications. We selected Python as our programing language to exploit the SS processing applications. This language is an open source scripting language and it will be efficient and simple to develop programs in Briareus. Meanwhile, thanks for the open-source fundamental packages for scientific computing with Python such as NumPy, SciPy, Matplotlib [17]-[19] etc. which makes development of scientific computation with Python as easy as Matlab.

3 Experimental Setup and Evaluation

We setup comparable experiments of the same computation tasks taken down by a server in the Matlab and a cloud computing system we proposed (see Fig.3). The SS data, a whole 1-hour audio recording of an OSAHS patient, was offered by the Department of Otolaryngology, a hospital, Beijing, China (People's Republic of). We utilize the SS processing applications all by Python for the platforms to run the programs. The hardware of the experiment environments are illustrated in Table 1. In the cloud, we set 1, 2, 4, 8, 16 instances to run the computation tasks relatively and record the time costs. Then, we also run the same tasks in the Matlab and do comparison with above results which in the cloud.

Table 1. Hardware of each computation equipment

	CPU	Memory
Server	Intel Xeon E7420 (2.13 GHz)×4	32.0 GB
Cloud (instances)	Intel Xeon E5504 (1 VCPU 2.0 GHz)	512 MB

The OpenStack IaaS platform has been deployed in a cluster of IBM BladeCenter servers as the private cloud. For each server, there are two 2.0 GHz Intel Xeon quad-core CPUs, 24GB memory and 1000M wired Ethernet. In the private cloud, we set up at most 16 instances (each with 1 VCPU, 512MB) in the OpenStack as Briareus' workers. In the PaaS, the time costs of the communications of each instance and the activation of Briareus are relatively low (about 1 second). Meanwhile this system is a private cloud computing system, which needs no transferring data in Internet compared with the public cloud computing system.

Table 2. Time costs of different scale in the cloud and the Matlab running at server (Unit: sec.)

	1-hour data	1-hour data	1-hour data
1 instance	44.115	43.883	43.830
2 instances	23.110	22.119	22.295
4 instances	12.875	11.923	11.967
8 instances	6.836	6.930	6.742
16 instances	3.631	3.973	3.526
Matlab	32.094	31.171	31.440

In the table 2. We compare the time costs for 3 independent experiments of the same tasks with the different scale in the cloud. Meanwhile, the results of Matlab which running at server with same tasks are shown in the bottom of table 2.

The cloud with only one instance takes the maximum time costs for accomplishing the computation tasks. Meanwhile, the Matlab tasks running in the Server has a good stable performance for the experiments. However, its computation ability is not much better than one instance. When we setup more instances in the cloud, we can see that the time costs reduce significantly. Specifically, the more instances we have, the low time costs we get, and the cloud will bring a breakthrough in boosting the efficiency of big SS data processing, if we have enough instances in the cloud. We can infer that once the scale of SS data becomes large enough, the processing tasks would be time-consuming and even difficult for one instance or server. The transferring of acquired SS data by microphones from subjects in Internet would also take much of time for a public cloud computing system. This private cloud computing system we proposed would get rid of the problem.

4 Relation to Prio Work

Hill et al. studied the changes of *crest factor* value of SS data for OSAHS patients during a whole night and they indicated that the obstructive site or mechanism of snoring varied during long-time analysis [3]. Inspired by the long-time analysis of SS data, we adopted the method to monitor patients' long-time SS data for further study. Azarbarzin et al. and Duckitt et al. utilized features clustering and HMM method to group all night SS data into different classifications, respectively [6], [7]. We integrate the two methods in our analysis system to achieve more clear results of variations of the UA of OSAHS patients. Good acoustic features which can reveal the changes of the UA structure in medical practice were extracted [3], [4], [5]. However, most existing scholars ignored the would-be intensive big SS data scale, which is our topic in this work. Jones et al. proposed an architecture design for mobile management of chronic conditions and medical emergencies, which focuses on defining a generic mobile solution [11]. Our system shares many good characteristics of this architecture. In addition, we took a stronger attention on the part of processing and storage, which takes scalability, economy and QoS issues into account. Analysis of heartbeat waveforms can be time-consuming hence automated computer-based processing of ECG data serves as a useful clinical tool. One of the major tasks to be provided is the accurate determination of the QRS complex [12]. Deelman et al. carried out a study to assess the cost of doing science in the cloud by renting computing and storage resources from Amazon Web Services to run a scientific workflow [13]. They concluded that costs could be reduced with little impact on performance. Technologies, such as MapReduce and Dryad have also been evaluated in the scientific context to support data analysis problems that traditionally relied on MPI-style parallel programming [14]. We adopted advanced ideas and knowledge from above researches and applied them into analysis of SS data.

5 Conclusions

This cloud system we proposed could significantly boost the efficiency in big SS data processing and research on analysis of big SS data for large scale of subjects. This study is a preliminary study in our group and the methodology and system design will be implemented and improved by large scales of practical experiments, which is a promising method for biomedical signal processing in the cloud.

Acknowledgment. This study is supported by the National Natural Science Foundation of China (People's Republic of) under grant No.61271410 and the 973 Project of Jiangsu Province under grant No. BK2011022.

References

1. Otero, A., Félix, P., Presedo, J., Zamarrón, C.: Evaluation of an Alternative Denition for the Apnea- Hypopnea Index. In: Engineering in Medicine and Biology Society (EMBC) 2010 Annual International Conference of the IEEE Proceedings, pp. 4654–4657. IEEE, Buenos Aires (2010)

2. He, Q., Chen, B.: Sleep Disordered Breathing. People's Medical Publishing House, Beijing (2009)
3. Hill, P.D., Osman, E.Z., Osborne, J.E., Lee, B.W.V.: Changes in snoring during natural sleep identified by acoustic crest factor analysis at different times of night. In: Clinical Otolaryngology and Allied Sciences, pp. 507–510. Blackwell Science Ltd., USA (2000)
4. Xu, H., Huang, W., Yu, L., Chen, L.: Spectral Analysis of Snoring Sound and Site of Obstruction in Obstructive Sleep Apnea/Hypopnea Syndrome. Journal of Audiology and Pathology, 28–32 (2011)
5. Karunajeewa, A.S., Abeyratne, U.R., Hukins, C.: Multi-feature snore sound analysis in obstructive sleep apnea-hypopnea syndrome. Physiological Measurement, 83–97 (2010)
6. Azarbarzin, A., Moussavi, Z.M.K.: Automatic and Unsupervised Snore Sound Extraction From Respiratory Sound Signals. IEEE Transactions on Biomedical Engineering, 1156–1162 (2011)
7. Duckitt, W.D., Tuomi, S.K., Niesler, T.R.: Automatic detection, segmentation and assessment of snoring from ambient acoustic data. Physiological Measurement, 1047–1056 (2006)
8. Hilbert, M., López, P.: The World's Technological Capacity to Store, Communicate, and Compute Information. Science, 60–65 (2011)
9. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., et al.: A view of cloud computing. Communications of the ACM, 50–58 (2010)
10. Sergios, Koutroumbas, K.: Pattern Recognition, 4th edn. Elsevier, Canada (2009)
11. Jones, V., Halteren, A.V., Widya, I., Dokovsky, N., Bults, R., Konstantas, D., Herzog, R.: Mobihealth: Mobile Health Services Based on Body Area Networks. In: Topics in Biomedical Engineering, pp. 219–236. Springer, Boston (2006)
12. Kohler, B.U., Hennig, C., Orglmeister, R.: The principles of software QRS detection. IEEE Engineering in Medicine and Biology Magazine, 42–57 (2002)
13. Deelman, E., Singh, G., Livny, M., Berriman, B., Good, J.: The cost of doing science on the cloud: the montage example. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, pp. 1–12. IEEE (2008)
14. Qiu, X., Ekanayake, J., Beason, S., Gunarathne, T., Fox, G., Barga, R., Gannon, D.: Cloud technologies for bioinformatics applications. In: Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers, pp. 1–10. ACM (2009)
15. Rosenthal, A., Mork, P., Li, H.M., Stanford, J., Koester, D., Reynolds, P.: Cloud computing: A new business paradigm for biomedical information sharing. Journal of Biomedical Informatics 43(2), 342–353 (2009)
16. Zhu, Z., Zhang, G., Zhang, Y., Guo, J., Xiong, N.: Briareus: Accelerating Python Applications with Cloud. In: 27th IEEE International Symposium on Parallel & Distributed Processing Workshops and PhD Forum, pp. 1449–1456. IEEE Press, Boston (2013)
17. NumPy, `http://www.numpy.org/`
18. SciPy, `http://www.scipy.org/`
19. Matplotlib, `http://matplotlib.org/`

Research on Optimum Checkpoint Interval for Hybrid Fault Tolerance

Lei Zhu, Jianhua Gu, Yunlan Wang, and Tianhai Zhao

School of Computer, Northwestern Polytechnical University, Xi'an, China
zeiier@126.com,
{gujh,wangyl,zhaoth}@nwpu.com

Abstract. With the rapid growth of the high performance computer system size and complexity, passive fault tolerance can no longer effectively provide reliability of the system because of the high overhead and poor scalability of these methods. Hybrid fault tolerant method which is the combination of passive and active fault tolerant approaches has the potential to be widely used in fault tolerance of exascale system. However, there are still many issues of this method need to be ironed out. This paper focuses on the issues of checkpointing of hybrid fault tolerant method. A common question surrounding checkpointing is the optimization of the checkpoint interval. This paper proposes two models to model the systems which adopt hybrid fault tolerance. By comparing their results with the simulation, this paper evaluates the effectiveness of these two models. Experimental result shows that the modified model can not only predict the total work time excellently, but also can predict the optimum checkpoint interval precisely.

Keywords: optimum checkpoint interval, fault tolerance, model.

1 Introduction

Nowadays, high performance computer systems (HPCs) are growing more and more complex [1]. It makes the system mean time to failure (SMTTF) of HPCs extremely shortened. According to the current technical routes, the latest data shows that SMTTF of the high-end systems will be less than 10 hours. If the scale of HPCs continuous increasing and MTTF of single hardware remain stable, the scale of exascale system will much more than the petascale system and the SMTTF will as low as 1 hour[2].

Passive fault tolerance which is commonly known as Checkpoint/Restart (C/R) will not satisfy the fault tolerant requirements of exascale systems. It is because these methods has two main drawbacks: (1) its executing overhead is so high that contributes 15% to 50% of the overall processing cost [3]; (2) the scalability is poor. If the SMTTF is lower than the period checkpoint–restart operation takes, the whole system will dump because the time interval between two interrupts are too short to write a checkpoint image.

To deal with these issues, researchers begin to consider active fault tolerant methods. These methods base on fault alert mechanism and predict possible failures in the near

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 367–380, 2013.
© Springer-Verlag Berlin Heidelberg 2013

future. Systems can keep processing application by implementing fault avoid operation before interrupt. Comparing with passive fault tolerance, active fault tolerance can effectively reduce the overhead and improve the scalability of fault tolerant system.

However, the predictor can't forecast all failures of the system in practice. Which means the system will dump if the predictor fails to dig out the failures in the near future. Hence, hybrid fault tolerant approaches are put forward to address this issue. By combining active and passive fault tolerant methods, these two methods can overcome the drawbacks for each other. Thus, such solution will be an efficient fault tolerant method of next-generation extreme scale systems. However, there are still many issues of this method need to be ironed out. For example, how to avoid the interrupt occurs during writing a checkpoint image?

This paper focuses on the issues of checkpointing of combinative fault tolerant method. The matter in question which surrounds checkpointing is the definition of the frequency in which checkpoints should be taken to minimize the overhead introduced by this technique, which is better known as the optimum checkpoint interval [4]. If the checkpoint interval is chosen inadequately, the system overhead will increase rapidly. Because checkpoint method is a widely used solution, there are many studies regarding the definition of its optimum interval [5][6][7]. But these studies are all focus on modeling the systems which only adopt passive fault tolerant method. For the systems which adopt hybrid fault tolerant method, these models are not accurate enough. The behavior of these systems is quite different from the systems only adopt single fault tolerant method. For example, the SMTTF can't be used to model the system directly, because some of fault will not interrupt the system because they are avoided by active fault tolerance.

In this paper, the authors model the systems which adopt hybrid fault tolerant method. The purpose of defining the optimum checkpoint interval is to minimize the total work time. Thus, the cost function for these systems should be

$$T(\epsilon) = operation\ time + checkpointing\ time + rework\ time + restart\ time + prediction\ time + fault\ avoid\ time$$

ϵ is the interval between checkpionting operation. Operation time is defined as time spent on effective operation (solving applications). If there is no interrupts of the system, the total work time $T(\epsilon)$ is equal to operation time. Checkpointing time is the overhead of writing checkpoints. Rework time is defined as time spent re-operating the lost work when an application is interrupt by a fault. It is equal to the amount of time elapsed since the last checkpoint. Restart time is the time required before an application is able to resume. Prediction time is the overhead of failure predictor. Fault avoid time is the time spent on handling the possible failures which are forecasted by predictor.

Base on this model, this paper quantifies the optimum checkpoint interval that minimizes the total work time. The experimental results show that our model is effective to calculate the optimum checkpoint interval for hybrid fault tolerance.

Table 1 lists the notations used in the model.

Table 1. Description of variables

Variable	Description
$\boldsymbol{T(\epsilon)}$	total work time
$\boldsymbol{\epsilon}$	checkpoint inverval
$\boldsymbol{T_s}$	operation time
$\boldsymbol{N}$	the number of time segment required to complete the application
$\boldsymbol{M}$	SMTTF
$\boldsymbol{M'}$	The approximation of the effective SMTTF
$\boldsymbol{\sigma}$	the overhead to write a checkpoint image
$\boldsymbol{n(\epsilon)}$	the total number of interrupts
$\boldsymbol{n'(\epsilon)}$	the total number of interrupt attempts
$\boldsymbol{R}$	average restart time
$\boldsymbol{k}$	The number alter generate by predictor
$\boldsymbol{F_i(\epsilon)}$	the overhead spent on handling ith possible failures
$\boldsymbol{F}$	the average overhead spent on handling each possible failure
$\boldsymbol{P}$	the overhead spent on predicting

The content of this paper is organized as follows. Section 2 introduces the related works. The system model is proposed and the methodology used to define variables of the model is presented in section 3. Evaluation results and discussion are shown in section 4. Finally, conclusions and future work are stated in section 5.

2 Related Work

Because checkpointing is a widely used method, there are many studies regarding the definition of its interval. As early as 1974, Young [5] introduced an analytical model to define the checkpoint interval for serial applications. The optimum checkpoint interval can be calculated once some variables such as checkpointing overhead and fault probability are available. W. Gropp [6] presented a simpler model and uses a different approach to deduce his model. The result achieved by Gropp's model is similar to Young's model.

In 2003, Daly [7] presented more deeply study to determine optimum checkpoint interval. Daly starts with analyzing first order approximation of the checkpoint interval, and generalizes to higher order estimation of it. He solved the model by using simple bisection method and the precision of approximation is not very high. In order to figure out an excellent approximation, Daly solved the model by using Lambert's function $W(z)$ in [8]. The experimental results show that Daly's model is more precise than Young's model.

All models introduced above can only precisely calculate the optimum checkpoint interval for coordinated checkpoint protocols. Leonardo F. et al. [4] explored the relationship existent between processes of parallel applications and propose a new

checkpoint interval model which includes a factor representing the parallel application inter-process relationship. The model they proposed can calculate the checkpoint interval which minimizes the fault tolerance overhead for uncoordinated checkpoint protocols.

There are many studies on improving the performance of C/R. In[14], Moody's team designed a scalable C/R library which is a multi-level checkpointing system that writes checkpoints to RAM, Flash, or disk on the compute nodes in addition to the parallel file system. Their multi-level checkpointing method can reduce the overhead of C/R significantly . They also presented a probabilistic Markov model that predicts the performance of their method on current and future systems. In 2013, Jangjaimon I.et al.[13] improved Moody's research by proposing an adaptive incremental checkpointing method (AIC). The experimental results show that the AIC can reduce the checkpointing file size considerably.

There are a number of researches on active fault tolerance. In 2006, Avritzer [9] presented three algorithms for detecting the need for software rejuvenation by monitoring the changing values of a customer-affecting performance metric. Gujrati's team [10] presents a failure predictor to automatically process RAS events and further discover failure patterns for prediction in Blue Gene/L systems. The Experiments show the effectiveness of the three-phase failure predictor. X. Gu [11] and his colleagues explored light-weight stream-based classification methods to perform online failure prediction. Experiment results show that their method can manage system failure efficiently.

In [12], C. Wang's team designed a hybrid fault tolerant method. First, they propose an efficient process-level live migration mechanism. Then they combine the mechanism with checkpointing method. Experiment indicates their method can effectively provide reliability of the system.

Hybrid fault tolerance has the potential to be widely used in fault tolerance of next-generation extreme scale systems. Despite the effectiveness of hybrid fault tolerant method, as far as we know, there is no model to calculate the optimum checkpoint interval for this method.

3 The Checkpoint Interval Model for Hybrid Fault Tolerance

3.1 The First Order Model

The operation time is defined as

$$T_s = N\epsilon \tag{1}$$

The total checkpointing time will be $(N-1)\sigma$. The rework time is some fractions of time segments which are interrupted by fault. This paper assumes that the proportion of fraction is $f(\epsilon+\sigma)$ on average. Then the rework time can be defined as $f(\epsilon+\sigma)[\epsilon+\sigma]n(\epsilon)$. The restart time is $Rn(\epsilon)$. The prediction time is simply assumed as P which is fixed proportion of operation time. The fault avoid time is defined as follow.

$$fault\ avoid\ time = \sum_{i=1}^{k} F_i(\epsilon) \tag{2}$$

Hence, the cost function will be

$$T(\epsilon) = T_s + \left(\frac{T_s}{\epsilon} - 1\right)\sigma + f(\epsilon + \sigma)[\epsilon + \sigma]n(\epsilon) + Rn(\epsilon) + P + \sum_{i=1}^{n} F_i(\epsilon) \quad (3)$$

3.2 The Assumptions and the Number of Interrupts

The first assumption is

$$f(\epsilon + \sigma) = \frac{1}{2} \quad (4)$$

Which means the interrupts will occur halfway through the checkpoint interval on the average. To address the issues of fault avoid time, this paper denotes the number of correct warning of active fault tolerance as TP (true positive) and denotes the number of false warning as FP (false positive). Meanwhile the number of correct missing is defined as TN (true negative) and the number of missing warning is defined as FN (false negative). Obviously, $k = TP + FP$. The recall and precision of the predictor are defined as follows.

$$recall = \frac{TP}{TP+FN} \in [0,1] \quad (5)$$

$$precision = \frac{TP}{TP+FP} \in [0,1] \quad (6)$$

Thus

$$n'(\epsilon) = \frac{n(\epsilon)}{1-\text{recall}} \quad (7)$$

This paper assumes that the system only adopts single fault avoid method. The fault avoid time will be

$$fault\ avoid\ time = Fn'(\epsilon)\frac{recall}{precison} \quad (8)$$

where F is the average overhead spent on handling each possible failure. $n'(\epsilon)$ is the total number of interrupt attempts of system fault. Some of faults will not interrupt the system because they are avoided by active fault tolerance.

$n(\epsilon)$ is defined as follow (see [7]).

$$n(\epsilon) = \frac{T_s}{\epsilon}\left(e^{(\epsilon+\sigma)/M} - 1\right) \quad (9)$$

If $\epsilon + \sigma \ll M$, the exponential term can be approximated by linear term.

$$n(\epsilon) \cong \frac{T_s}{\epsilon}\left(\frac{\epsilon+\sigma}{M}\right) \text{ for } \frac{\epsilon+\sigma}{M} \ll 1 \quad (10)$$

Eq. (9) and (10) can't be used directly in the model. The first reason is that some of fault will not interrupt the system because they are avoided by active fault tolerant method. The second reason is that the assumption $\epsilon + \sigma \ll M$ is erroneous for hyper scale

systems which require adopting hybrid fault tolerance. Thus, this paper uses M' to approximate the effective SMTTF. M' is defined as follow.

$$M' = \frac{M}{1-\text{recall}} \tag{11}$$

$$n(\epsilon) = \frac{T_s}{\epsilon}\left(e^{(\epsilon+\sigma)/M'} - 1\right) \tag{12}$$

$$n(\epsilon) \cong \frac{T_s}{\epsilon}\left(\frac{\epsilon+\sigma}{M'}\right) \text{ for} \frac{\epsilon+\sigma}{M'} \ll 1 \tag{13}$$

Combining all of these terms, the cost function will be

$$T(\epsilon) = T_s + \left(\frac{T_s}{\epsilon} - 1\right)\sigma + \frac{1}{2}(\epsilon+\sigma)\frac{T_s}{\epsilon}\left(\frac{\epsilon+\sigma}{M'}\right) + R\frac{T_s}{\epsilon}\left(\frac{\epsilon+\sigma}{M'}\right) + P$$
$$+F\frac{recall}{precison} \times \frac{T_s}{\epsilon(1-\text{recall})}\left(\frac{\epsilon+\sigma}{M'}\right) \tag{14}$$

3.3 Solving the Model

To solve the model this paper considers solutions of the first and second derivative with respect to ϵ. First, this paper considers the second derivative with respect to ϵ.

$$\frac{d^2T(\epsilon)}{d\epsilon^2} = \frac{2T_s}{\epsilon^3}\sigma + \frac{T_s}{M'\epsilon^3}\sigma + \frac{2RT_s}{M'\epsilon^3}\sigma + \frac{2FT_s \times recall}{M'\epsilon^3(1-recall)\times precision}\sigma > 0 \tag{15}$$

Then the minimum point of first derivative is unique.

$$\frac{dT(\epsilon)}{d\epsilon} = -\frac{T_s}{\epsilon^2}\sigma + \frac{T_s}{2M'} - \frac{T_s}{2M'\epsilon^2}\sigma^2 - \frac{T_sR}{M'\epsilon^2}\sigma - \frac{T_sF \times recall}{M'\epsilon^2(1-recall)\times precision}\sigma$$
$$= -\frac{1}{\epsilon^2}\left[2M'\sigma + \sigma^2 + 2R\sigma + 2F\sigma\frac{recall}{(1-recall)\times precision}\right] + 1 = 0 \tag{16}$$

The optimum checkpoint interval ϵ_{opt} is the solution of (16).

$$\hat{\epsilon}_{opt} = \sqrt{\sigma\left[2M' + 2R + \sigma + 2F\frac{recall}{(1-recall)\times precision}\right]} \tag{17}$$

3.4 Relaxing the Assumption

The problematic assumption associated with the original model is $\epsilon + \sigma \ll M'$. $M'$is determined by the recall and the scale of the system. For exascale systems, the assumption will be valid only if the active fault tolerant method can achieve very high recall (over 99 percent). For generalization purpose, Eq. (14) should be rewritten as

$$T(\epsilon) = T_s + \left(\frac{T_s}{\epsilon} - 1\right)\sigma + \frac{1}{2}(\epsilon+\sigma)\frac{T_s}{\epsilon}\left(e^{\frac{\epsilon+\sigma}{M'}} - 1\right) + R\frac{T_s}{\epsilon}\left(e^{\frac{\epsilon+\sigma}{M'}} - 1\right) + P$$
$$+F\frac{recall}{precison} \times \frac{T_s}{\epsilon(1-recall)}\left(e^{\frac{\epsilon+\sigma}{M'}} - 1\right) \tag{18}$$

3.5 Solving the Modified Model

The second derivative with respect to ϵ of Eq. (18) is complex. Thus, it is trivial to prove the convexity of Eq. (18) directly. However, as shown in the next section, the minimum point of Eq. (18) is unique. To be brief, this paper defines two symbols:

$$\tau = e^{\frac{\epsilon+\sigma}{M'}} \tag{19}$$

$$q = F\frac{recall}{(1-recall)\times precision} \tag{20}$$

Then, the first derivative with respect to ϵ is

$$\frac{dT(\epsilon)}{d\epsilon} = -\frac{T_S\sigma}{\epsilon^2} + \frac{T_S}{2M'}\tau + \frac{T_S\sigma}{2M'\epsilon}\tau - \frac{T_S\sigma}{2\epsilon^2}\tau + \frac{T_S\sigma}{2\epsilon^2} + \frac{T_SR}{M'\epsilon}\tau - \frac{T_SR}{\epsilon^2}\tau + \frac{T_SR}{\epsilon^2} + qT_S\left(\frac{1}{M'\epsilon}\tau - \frac{1}{\epsilon^2}\tau + \frac{1}{\epsilon^2}\right) \tag{21}$$

Setting it to zero, we get

$$\tau[\epsilon^2 + (\sigma + 2R + 2q)\epsilon - (\sigma + 2R + 2q)M'] = (\sigma - 2R - 2q)M' \tag{22}$$

$$\Rightarrow e^{\frac{\epsilon+\sigma}{M'}} = \frac{(\sigma-2R-2q)M'}{\epsilon^2+(\sigma+2R+2q)\epsilon-(\sigma+2R+2q)M'} \tag{23}$$

Adopting logarithm on both sides of Eq. (23), then

$$\frac{\epsilon+\sigma}{M'} = ln\frac{(\sigma-2R-2q)M'}{\epsilon^2+(\sigma+2R+2q)\epsilon-(\sigma+2R+2q)M'} \tag{24}$$

To simplify the function, we set

$$\varphi(\epsilon) = \frac{(\sigma-2R-2q)M'}{\epsilon^2+(\sigma+2R+2q)\epsilon-(\sigma+2R+2q)M'} \tag{25}$$

Obviously, the left side of Eq. (24) is non-negative. Thus $\varphi(\epsilon) > 1$ for all ϵ. If we use Euler-Maclaurin expansion for natural logarithm directly, the series will diverge. However, Eq. (24) can be rewritten as

$$\frac{\epsilon+\sigma}{M'} = -ln\left[1 + \left(\frac{1}{\varphi(\epsilon)} - 1\right)\right] = -\sum_{n=0}^{\infty}(-1)^n\frac{\left(\frac{1}{\varphi(\epsilon)}-1\right)^{n+1}}{n+1} \tag{26}$$

This series expansion is expected to converge. Ignoring higher order terms (the effect will be discussed in next section), we get

$$\frac{\epsilon+\sigma}{M'} = 1 - \frac{1}{\varphi(\epsilon)} = 1 - \frac{\epsilon^2+(\sigma+2R+2q)\epsilon-(\sigma+2R+2q)M'}{(\sigma-2R-2q)M'} \tag{27}$$

$$\Rightarrow \epsilon^2 + 2\sigma\epsilon + \sigma^2 = 2\sigma(M' + R + q) \tag{28}$$

Therefore, the optimum checkpoint interval ϵ_{opt} is approximately

$$\hat{\epsilon}_{opt} = \sqrt{2\sigma\left(M' + R + F\frac{recall}{(1-recall)\times precision}\right)} - \sigma \tag{29}$$

4 Evaluation

This paper develops a simulation to valid the effectiveness of the models. The simulation system include three modules: interrupting attempts module, fault tolerant module and operation module. The interrupting attempts module generates random interrupting attempts when the operation module executing the application. The distribution of interrupting attempts is described by an exponential model. This model is the simplest useful life distribution model for mechanical and electrical equipment [7]. Then, the probability of an interrupt occurring before time t is given by the distribution function

$$P(t < \Delta t) = \int_0^{\Delta t} \frac{1}{M} e^{-t/M} dt = 1 - e^{-\Delta t/M} \tag{30}$$

The interrupting attempt will suspend the operating module. The fault tolerant module catches each interrupting attempt and calculates whether there is a real interrupt according to recall parameter. If the result is FALSE (the interrupt is avoid by active fault tolerance), the fault tolerant module will calculate the overhead of fault avoid. If the result is TURE (the active fault tolerance is fall to avoid the interrupt), the fault tolerant module will calculate the overhead of recovery. Then the operating module will remain standstill for corresponding time. The system simulates the checkpointing (write checkpoint image) as a delay operation. The prediction P is ignored since it only correlate with the operation time. The simulation is run multiple times for each parameter setting and the results will be averaged.

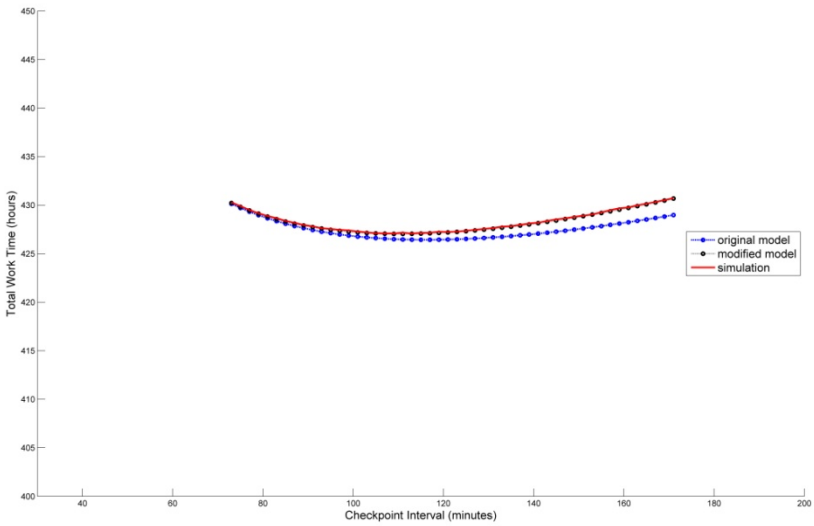


Fig. 1. Simulation results for T_s=387 hours, M=200min, σ=5 min, recall=0.85, precision=0.9, R=1min, F=2 min. ϵ_{opt} = 111.02min. The optimum interval $\hat{\epsilon}_{opt}$ predicted by modified model is 111.05min.

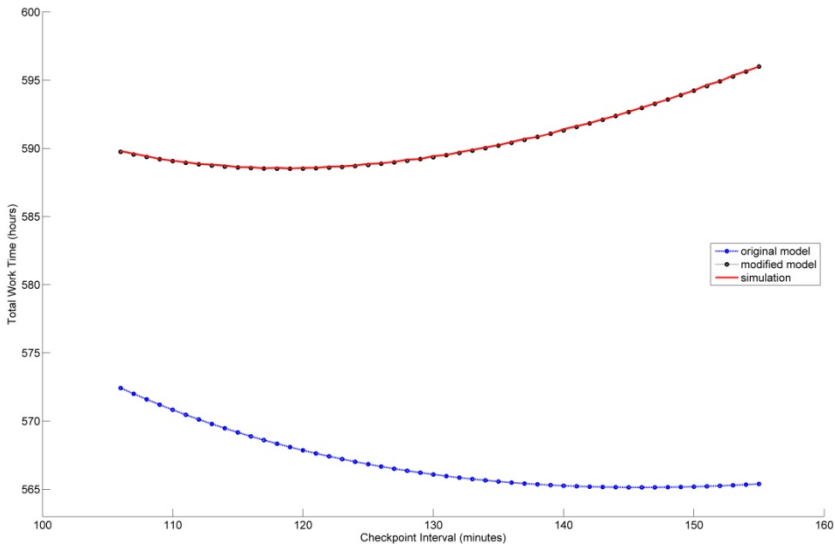


Fig. 2. Simulation results for T_s=387 hours, M=60min, σ=25 min, recall=0.85, precision=0.9 R=1min, F=2 min. ϵ_{opt} = 118.69min. The optimum interval $\hat{\epsilon}_{opt}$ predicted by modified model is 118.8min.

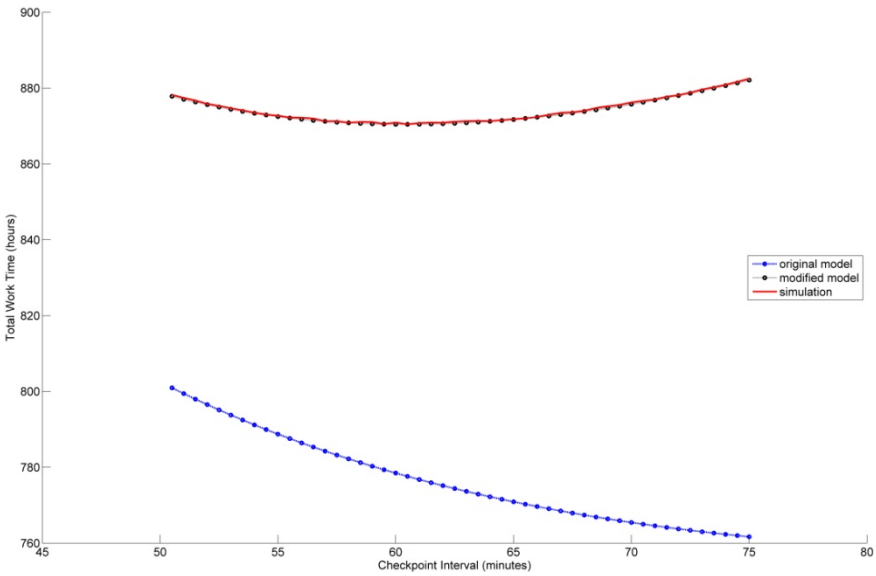


Fig. 3. Simulation results for T_s=387 hours, M=20min, σ=25 min, recall=0.85, precision=0.9 R=1min, F=2 min. ϵ_{opt} = 60.4min. The optimum interval $\hat{\epsilon}_{opt}$ predicted by modified model is 60.7min.

Figure 1 shows that there is agreement between two models and simulation when $M' \gg \epsilon + \sigma (M' \approx 22hours)$. It means that the original model is appropriate to model the system if $M' \gg \epsilon + \sigma$.

As shown in figure 2-3, the SMTTF decreases and the total work time increases rapidly. The original model can't model the system accurately. This is because the assumption that $M' \gg \epsilon + \sigma$ is no longer valid. With the decreasing of M', the agreement between the original model and simulation gets worse. However, the modified model is perfectly consistent with simulation results. Figure 4-5 show the results of the modified model for extreme SMTTF. Because the huge deviation between the original model and simulation, the results of original model are not shown in these plots.

There is little deviation between the optimum checkpoint interval $\hat{\epsilon}_{opt}$ calculated by Eq. (29) and ϵ_{opt}. This is because the higher order terms of Eq. (26) are ignored.

Figure 6-7 show the simulation results of higher recall and lower precision. The effectiveness of the modified model is still good.

Figure 8-9 show the deviation between ϵ_{opt} and the optimum interval $\hat{\epsilon}_{opt}$ predicted by modified model for different σ. As the M increasing, the deviation between $\hat{\epsilon}_{opt}$ and ϵ_{opt} will be reduced.

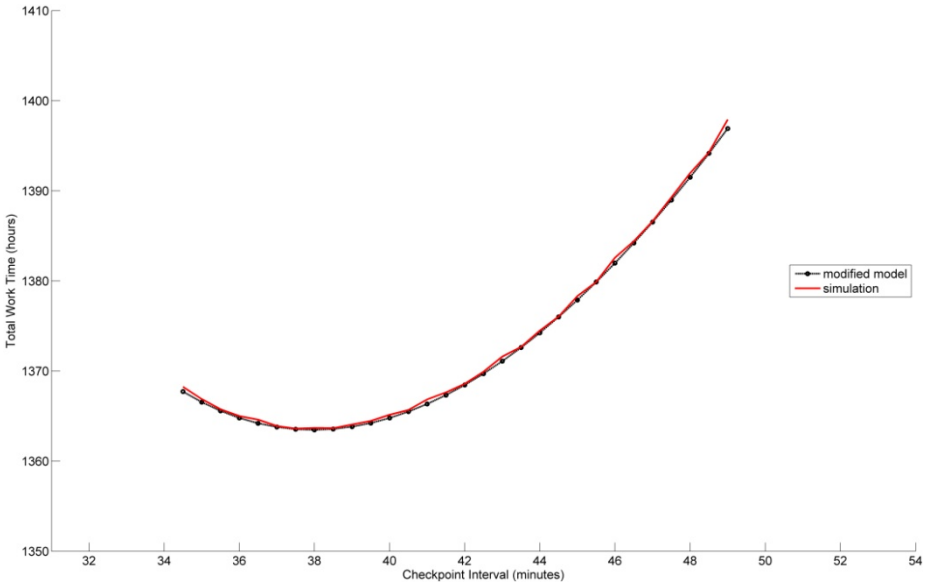


Fig. 4. Simulation results for T_s=387 hours, M=10min, σ=25 min, recall=0.9, precision=0.9 R=1min, F=2 min. ϵ_{opt} = 38min. The optimum interval $\hat{\epsilon}_{opt}$ predicted by modified model is 38.3min.

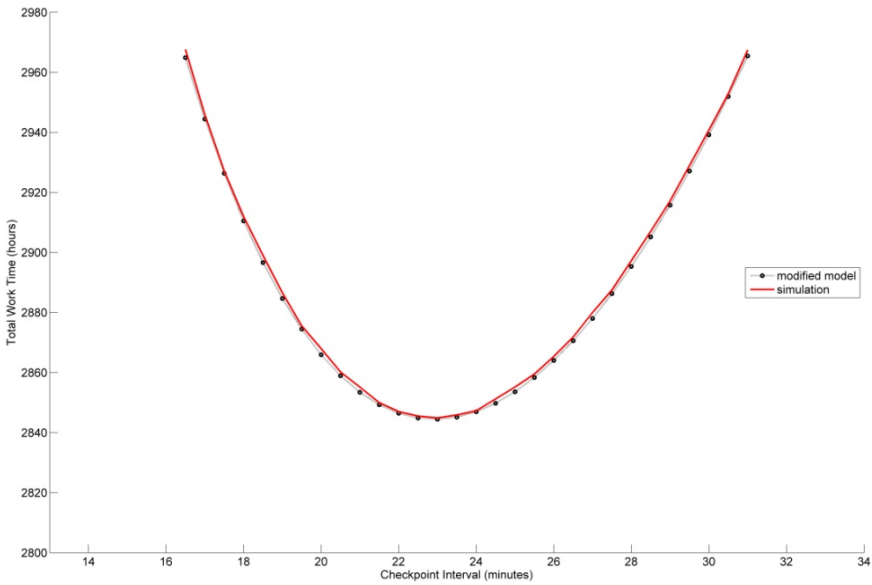


Fig. 5. Simulation results for T_s=387 hours, M=5min, σ=25 min, recall=0.85, precision=0.9 R=1min, F=2 min. ϵ_{opt} = 22.9min. The optimum interval $\hat{\epsilon}_{opt}$ predicted by modified model is 23.4min.

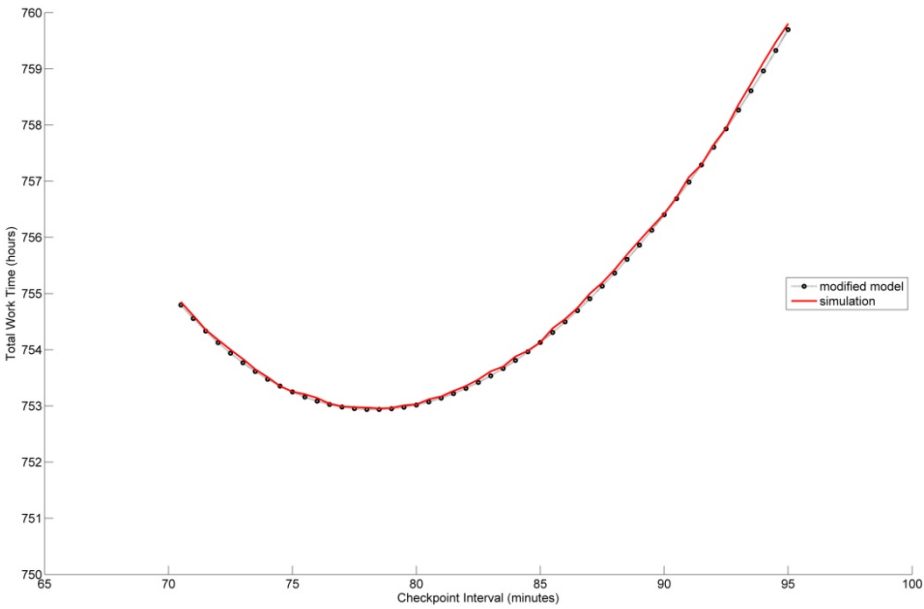


Fig. 6. Simulation results for T_s=387 hours, M=20min, σ=25 min, recall=0.9, precision=0.9 R=1min, F=2 min. ϵ_{opt} = 78.3min. The optimum interval $\hat{\epsilon}_{opt}$ predicted by modified model is 80.1min.

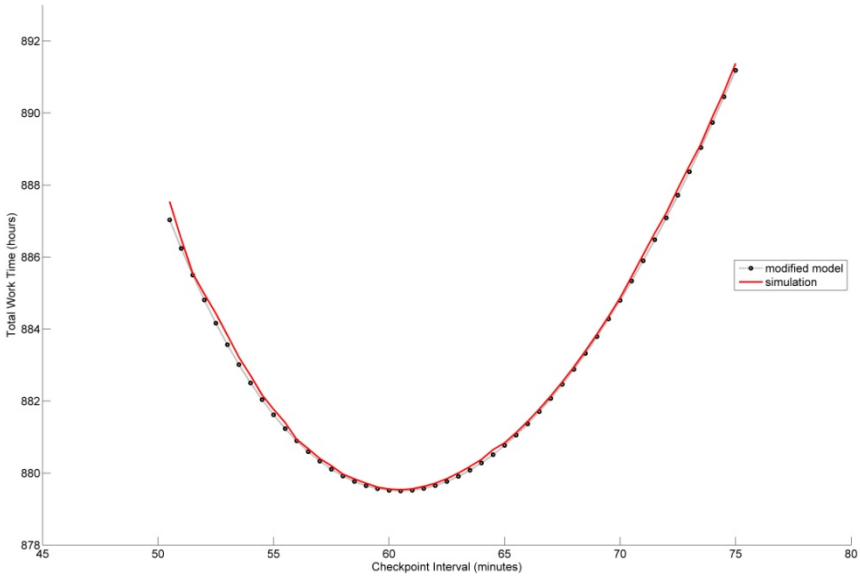


Fig. 7. Simulation results for T_s=387 hours, M=20min, σ=25 min, recall=0.85, precision=0.8 R=1min, F=2 min. ϵ_{opt} = 60.5min. The optimum interval $\hat{\epsilon}_{opt}$ predicted by modified model is 61.1min.

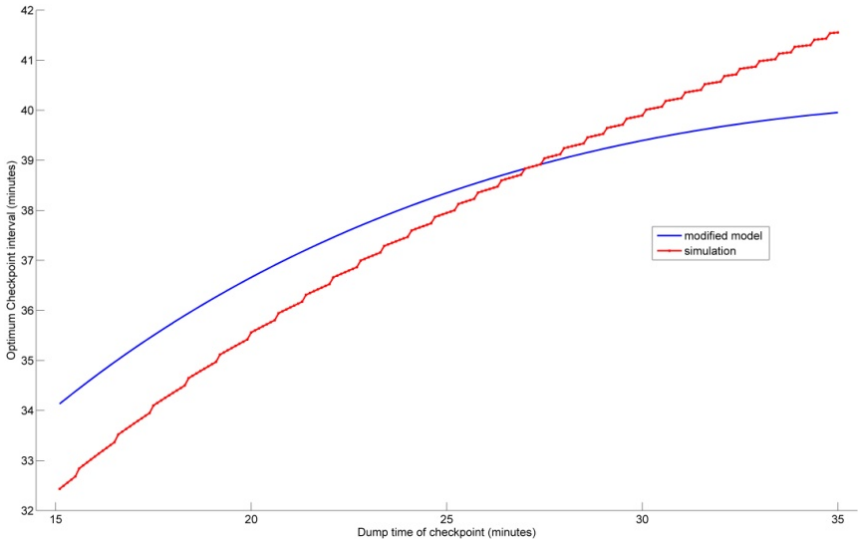


Fig. 8. The deviation between ϵ_{opt} and the optimum interval $\hat{\epsilon}_{opt}$ predicted by modified model for different σ. T_s=387 hours, M=10min, σ=20 min, recall=0.85, precision=0.9 R=1min, F=2 min.

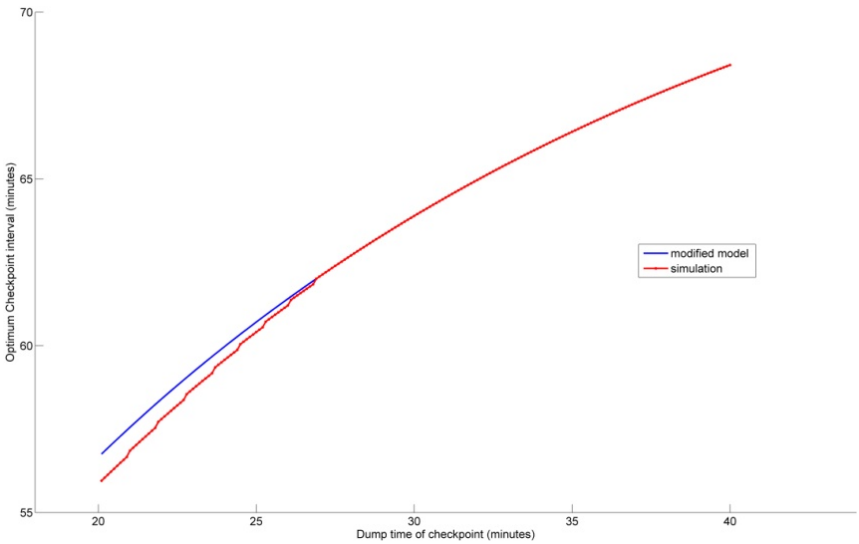


Fig. 9. The deviation between ϵ_{opt} and the optimum interval $\hat{\epsilon}_{opt}$ predicted by modified model for different σ. T_s=387 hours, M=20min, σ=20 min, recall=0.85, precision=0.9 R=1min, F=2 min.

Normally, the deviation can be ignored. However, if the value of M is very small, the relative error is relatively large. In this case, ϵ_{opt} can be easily calculated by $\hat{\epsilon}_{opt}$ and the modified model by using simple search method. Because the deviation between ϵ_{opt} and $\hat{\epsilon}_{opt}$ is very little, ϵ_{opt} is located in the neighborhood of $\hat{\epsilon}_{opt}$. First, we set step length μ, and calculate $t_l = T(\hat{\epsilon}_{opt} - \mu)$ and $t_r = T(\hat{\epsilon}_{opt} + \mu)$. If $t_l < t_r$, we set $t_r = t_l$, $\hat{\epsilon}_{opt} = \hat{\epsilon}_{opt} - \mu$. Then we calculate$t_l = T(\hat{\epsilon}_{opt} - \mu)$. Repeating this step until $t_l > t_r$. $\mathrm{t_r}$ is approximate of $\hat{\epsilon}_{opt}$. For $t_l > t_r$, the process is similar.

Figure 1-5 show the simulation results of two models. Then we find out that the original model fails to represent the behavior of the system for small M'. However, the main target of experiments is to compare the modified model with simulation results. We find out the modified model can effectively model the system.

5 Conclusions and Further Work

This paper presents two models to model the systems which adopt hybrid fault tolerance and predict the optimum checkpoint interval. Their results are compared with the simulation. The experimental result shows that the modified model agrees with simulation in predicting total work time. Furthermore, this paper discusses the deviation between the optimum checkpoint interval predicted by the modified model and the optimum checkpoint interval obtained by simulation. Normally, the deviation can be ignored. In other words, $\hat{\epsilon}_{opt}$ is a good approximation of the optimum checkpoint interval. Then this paper proposes a simple search method to deal with the deviation for small M.

However, the modified model introduced in this paper could be further improved. First of all, this paper assumes that, on the average, the interrupts will occur halfway through the compute interval. This assumption is problematic. In practice, on the average, $f(\epsilon + \sigma) < 1/2$. What' more, this paper assumes that the system only adopts single fault avoid method. The model will fail to model the systems which adopt multiple fault avoid methods. We will exploit these aspects in future.

References

1. Felix, S., Maren, L., Miroslaw, M.: A Survey of Online Failure Prediction Methods. ACM Computing Surveys 42, Article No. 10 (2010)
2. Cappello, F.: fault tolerance in petascale/exascale systems: current knowledge, challenges and research opportunities. The International Journal of High Performance Computing Applications 23, 212–226 (2009)
3. Varela, M.R., Ferreira, K.B., Riesen, R.: Fault-Tolerance for Exascale Systems. In: 2010 IEEE International Conference on Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), pp. 1–4 (2010)
4. Leonardo, F., Dolores, R., Emilio, L.: What Is Missing in Current Checkpoint Interval Models? In: 2011 International Conference on Distributed Computing Systems, pp. 322–332 (2011)
5. Young, J.W.: A first order approximation to the optimum checkpoint interval. Commun. ACM 17, 530–531 (1974)
6. Gropp, W., Lusk, E.: Fault Tolerance in Message Passing Interface Programs. International Journal of High Performance Computing Applications 18(3), 363–372 (2004)
7. Daly, J.: A model for predicting the optimum checkpoint interval for restart dumps. In: Sloot, P.M.A., Abramson, D., Bogdanov, A.V., Gorbachev, Y.E., Dongarra, J., Zomaya, A.Y. (eds.) ICCS 2003, Part IV. LNCS, vol. 2660, pp. 3–12. Springer, Heidelberg (2003)
8. Daly, J.: A higher order estimate of the optimum checkpoint interval for restart dumps. Future Generation Computer Systems 22, 303–312 (2006)
9. Avritzer, A., Bondi, A., Grottke, M., Trivedi, K.S., et al.: Performance assurance via software rejuvenation: Monitoring, statistics and algorithms. In: Proc. International Conference on Dependable Systems and Networks, pp. 435–444 (2006)
10. Gujrati, P., Li, Y., Lan, Z., Thakur, R., et al.: A meta-learning failure predictor for BlueGene/L systems. In: The 2007 International Conference on Parallel Processing, p. 40 (2007)
11. Gu, X., Papadimitriou, S., Yu, P.S., Chang, S.P.: Toward predictive failure management for distributed stream processing systems. In: The 28th International Conference on Distributed Computing Systems, pp. 825–832 (2008)
12. Wang, C., Mueller, F., Engelmann, C., Scott, S.L.: Proactive process-level live migration and back migration in HPC environments. Journal of Parallel and Distributed Computing 72, 254–267 (2012)
13. Jangjaimon, I., Tzeng, N.-F.: Adaptive Incremental Checkpointing via Delta Compression for Networked Multicore Systems. In: The 27th IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2013), pp. 7–18 (2013)
14. Moody, A., Bronevetsky, G., Mohror, K., de Supinski, B.R.: Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In: The 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2010), pp. 1–11 (2010)

Programming Real-Time Image Processing for Manycores in a High-Level Language

Essayas Gebrewahid[1], Zain-ul-Abdin[1], Bertil Svensson[1], Veronica Gaspes[1], Bruno Jego[2], Bruno Lavigueur[2], and Mathieu Robart[3]

[1] Center for Research on Embedded Systems, Halmstad University, Halmstad, Sweden
[2] STMicroelectronics – Advanced System Technology, Grenoble, France
[3] STMicroelectronics – Advanced System Technology, Bristol, United Kingdom

Abstract. Manycore architectures are gaining attention as a means to meet the performance and power demands of high-performance embedded systems. However, their widespread adoption is sometimes constrained by the need for mastering proprietary programming languages that are low-level and hinder portability.

We propose the use of the concurrent programming language `occam-pi` as a high-level language for programming an emerging class of manycore architectures. We show how to map `occam-pi` programs to the manycore architecture Platform 2012 (P2012). We describe the techniques used to translate the salient features of the language to the native programming model of the P2012. We present the results from a case study on a representative algorithm in the domain of real-time image processing: a complex algorithm for corner detection called Features from Accelerated Segment Test (FAST). Our results show that the `occam-pi` program is much shorter, is easier to adapt and has a competitive performance when compared to versions programmed in the native programming model of P2012 and in OpenCL.

Keywords: Parallel programming, `Occam-pi`, Manycore architectures, Real-time image processing.

1 Introduction

The design of high-performance embedded systems for signal processing applications is facing the challenge of increased computational demands. Moore's Law still gives us more transistors per chip but, since increased processor clock speed is no longer an option, current hardware designs are shifting to manycore architectures to cope with the computational demand of DSP applications. However, developing applications that employ such architectures poses several other challenging tasks. The challenges include learning multiple proprietary low-level languages for describing the communication structure of the application and the computational kernels, as well as partitioning and decomposing the application into several sub-tasks that can execute concurrently. Sequential programming languages (like C, C++, Java …), which were

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 381–395, 2013.
© Springer-Verlag Berlin Heidelberg 2013

originally designed for sequential computers with unified memory systems and rely on sequential control flow, procedures, and recursion, are difficult to adapt for many-core architectures with distributed memories. Usually, as a partial solution, these languages provide annotations that the programmer can use to direct the compiler how to adapt the implementation to the target architecture.

We propose to use the concurrent programming model of `occam-pi` [1] that combines Communicating Sequential Processes (CSP) [2] with the pi-calculus [3]. This model allows the programmer to express concurrent computations in a productive manner, matching them to the target hardware using high-level constructs. The features of `occam-pi` that make it suitable for mapping applications to a wide class of embedded parallel architectures are: a) constructs for expressing concurrent computations, b) computations that reside in different memory spaces, c) dynamic parallelism, d) dynamic process invocation, and e) support for placement attributes.

The feasibility of using the `occam-pi` language to program an emerging class of massively parallel reconfigurable architectures has been demonstrated in earlier work [4]. The applicability of the approach was also previously demonstrated on a more fine-grained reconfigurable architecture, viz., PACT XPP [5]. This paper is focused on using `occam-pi` to map applications to an embedded manycore architecture, the Platform 2012 (P2012) [6], which is currently under joint development by STMicroelectronics and CEA. P2012 is a scalable manycore computing fabric based on multiple processor clusters with independent power and clock domains. Clusters are connected via a high-performance fully asynchronous network-on-chip (NoC). The independent power domain for each cluster allows switching-off power to a cluster, and the independent clock domain enables frequency/voltage scaling in order to achieve energy-efficient solutions.

The paper describes the different translation steps involved in the code generation phase of the compiler. The paper also presents as a case study the implementation of the FAST (Features from Accelerated Segment Test) algorithm [7] for corner detection. The case study aims at verifying that programming is actually simplified, and at evaluating the competitiveness in performance of our compilation based approach compared to the use of the native programming model of the P2012 architecture. We have used a parameterized approach in the form of replicated parallel processes in the `occam-pi` language to control the degree of parallelism.

In previous papers we have demonstrated the suitability of `occam-pi` for expressing task parallelism in applications like FIR (finite impulse response) filter, DCT (discrete cosine transform) and Autofocus in image forming radar systems [4, 5, 18]; here we show the applicability of the approach also for truly data parallel computations.

In the following three sections we present some related work, review the `occam-pi` language basics, and give an overview of the P2012 architecture and its native programming model. We then describe the compiler framework and the various translation steps involved to generate code for P2012. The approach is experimentally evaluated through a case study implementation of the FAST algorithm, and conclusions are drawn.

2 Related Work

There have been a number of initiatives in both industry and academia to address the requirement of raising the abstraction level in the form of high-level parallel programming languages. Recently developed parallel programming languages include `Chapel` [8], `Fortress` [9], and `X10` [10]. These mainly rely on implicit parallelism based on data-parallel operations on parallel collections and are primarily targeting high-performance large-scale computers.

Apart from the above-mentioned parallel programming languages, there are some recently introduced domain specific languages (DSLs) intended for the domain of digital signal processing (DSP). The `Feldspar` language [11], being developed at Chalmers University of Technology, is one such DSL where the domain expert expresses the DSP algorithms by using constructs like filters, vectors, and bit manipulation operations. The functional basis of the `Feldspar` core language facilitates performing different source code transformations such as fusion techniques and graph transformations. `CAL` [12] is another domain-specific language, developed at UC Berkeley, for dataflow programming and is based on the actor's model of computation. By describing the application as a dataflow network of actors, the available parallelism is explicitly exposed. `CAL` has been chosen as a specification language for the ISO/IEC 23001-4 MPEG standard. The `Spiral` project at CMU [13] deals with the domain of linear signal transforms in the broad field of DSP algorithms. `Spiral` makes use of the mathematical knowledge expressed in a particular algorithm in order to transform it into a concise declarative framework that is suitable for computer representation, exploration, and optimization. These high-level domain-specific languages are best suited for application programming because of their productivity and expressiveness. However, they are not well suited for compiling directly down to the manycore architectures; rather, they require transformations via a parallel intermediate representation.

Since we are interested in both the signal processing domain and mapping to the manycore architectures, we have proposed the use of `occam-pi` because it provides explicit control of concurrency in terms of processes that communicate by message passing (however, this is not demonstrated in the present paper) [22]. This closely matches the underlying architecture and it supports both task and data-level parallelism, thereby allowing the programmer to exploit the available parallelism more effectively. Based on this property, `occam-pi` is a candidate for the parallel intermediate representation mentioned above.

3 `Occam-pi` Language Overview

`Occam-pi` [1] is a programming language based on the concurrency model of CSP [2] and the pi-calculus [3]. It offers a minimal run-time overhead and comes with constructs for expressing parallelism and reconfigurations. It has a built in semantics for concurrency and inter-process communication. `Occam-pi` can be regarded as an extension of classical `occam` [14] to include the mobility feature of the pi-calculus.

It is this property of occam-pi that is useful when creating a network of processes in which the functionality of processes and their communication network changes at runtime.

The primitive processes of occam include assignment, input (?) and output (!). In addition to these there are constructs for sequential processes (SEQ), parallel processes (PAR), iteration (WHILE) selection (IF/ELSE, CASE) and replication [2]. In occam-pi the SEQ and PAR constructs can be replicated. A replicated SEQ is similar to a for-loop. A replicated PAR can be used to instantiate a number of processes in parallel and helps managing the multitude of parallel resources in a given hardware architecture.

PAR *i* = *start* FOR *Number of Replications*
<process i>

Finally, a procedure is a named process that can take parameters. In occam the data a process can access is strictly local and can be observed and modified by the owner process only. The communication between processes uses channels and message passing, which helps to avoid interference problems. In occam-pi data can be declared to be MOBILE, which means that the ownership of the data, including communication channels, can be passed between different processes. Moreover, channel type definitions have been extended to include the direction specifiers input (?) and output (!). Thus, a variable of a channel type refers only to one end of the channel. Channels in occam-pi are first-class citizens. Channel direction specifiers are added to the type of a channel definition and not to its name. Based on the direction specification, the compiler can do static checks of the usage of the channel both in the body of the process and the processes that communicate with it. Channel direction specifiers are also used when referring to channel variables as parameters of a process call.

Mobile data and channels, together with dynamic process invocation and the process placement attributes of occam-pi, are used to express the different configurations of hardware resources as well as run-time reconfiguration.

Mobile Data and Channels: Assignment and communication in classical occam follow the copy semantics, i.e., for transferring data from the sender process to the receiver both the sender and the receiver maintain separate copies of the communicated data. The mobility concept of the pi-calculus enables the movement semantics during assignment and communication, which means that the respective data has moved from the source to the target and afterwards the source has lost the possession of the data. In case the source and the target reside in the same memory space, the movement is realized by swapping of pointers, which is secure and does not introduce aliasing.

In order to incorporate mobile semantics into the language, the keyword MOBILE has been introduced as a qualifier for data types [5]. The definition of the MOBILE types is consistent with the ordinary types when considered in the context of defining expressions, procedures and functions. However, the mobility concept of MOBILE types is applied in assignment and communication. The modeling of mobile channels is independent of the data types and the structures of the messages that they carry.

4 P2012 Architecture and Development Tools

P2012 [6] is a manycore architecture, which is aimed at replacing existing specialized hardware and software subsystems by using a single, modular, scalable, and programmable computing fabric. The architecture is based on multiple clusters with independent power and clock domains. Clusters are connected via a high-performance fully asynchronous network-on-chip (NoC). The independent power domain for each cluster allows switching-off power to a cluster and the independent clock domain enables frequency/voltage scaling in order to achieve energy-efficient execution. The P2012 fabric can support up to 32 clusters [6]. The current P2012 cluster is composed of a cluster controller, one to sixteen ENcore processors and Hardware Processing Elements (HWPEs) [6]. The cluster controller is responsible for starting/stopping the execution of ENcore processors and notifying the host system. The processing elements share an advanced DMA engine, a hardware synchronizer, level-1 shared data memories and an individual program cache [6].

The P2012 Software Development Kit (SDK) supports several programming models that can be classified into three main classes. The native programming layer is a low-level C-based API providing the most efficient use of P2012 resources at the expense of a lack of abstraction. Standards-based programming models target effective implementations of industry standards, such as OpenCL and OpenMP, on the P2012 platform. The SDK provides the GePOP platform for simulation.

4.1 P2012 Native Programming Model

The Native Programming Model (NPM) is a component-based development framework. Application components are developed based on the MIND framework [16]. A component may provide services to other components by its provided interfaces and get service from its environment by using required interfaces. The communication between two components is hidden by binding their provided and required interface [16]. An NPM application is designed by using the Architecture Description Language (ADL), Interface Description Language (IDL), and an extended C code. ADL is used to define the structure of each component, IDL to specify component interface, and the extended C language for the implementation of the code that runs on the ENcore processors and the cluster controller. After the application is designed, a host-side program also has to be developed to deploy, manage and run the application. For this purpose, the middleware Comete is used.

NPM is designed to have direct access to specific features of the P2012 hardware platform, while still providing a high level of abstraction. Since the current standards-based programming models of P2012 don't have explicit means for dynamic resource allocation, we propose to translate `occam-pi` to the P2012 native programming model. Fig. 1 shows our approach to map `occam-pi` programs to the Platform 2012 Software Development Kit stack.

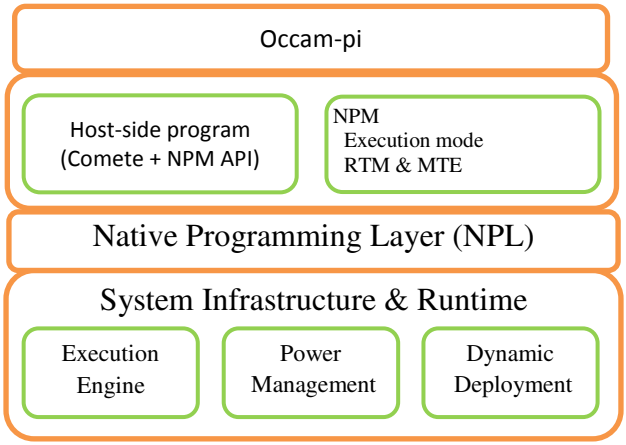


Fig. 1. Mapping of `occam-pi` to P2012 SDK

The main implementation of an NPM application will run on the ENcore processors, and the cluster controller will execute code for resource allocation and configuration. Interaction between the cluster controller and the ENcore processors can be handled by two execution engines: Reactive Task Manager (RTM) and/or Multi-Threaded Engine (MTE). RTM expresses parallelism based on forking and duplication of tasks, and MTE allows execution of synchronized parallel threads. Currently, our compilation directly uses the APIs provided by the base runtime and hardware abstraction layer (HAL), instead of using any of the two execution engines.

5 `Occam-pi` Compilation to P2012

The compiler that we have developed is based on the frontend of an existing Translator from `occam` to C from Kent (Tock) [17]. Our compiler can be divided into three main phases as shown in Fig. 2. The front end consists of phases up to machine independent optimization and the backend includes the remaining phases that are dependent upon the target machine architecture. The Ambric and the eXtreme Processing Platform (XPP) backends were developed and describer earlier [18] [5]. We have also earlier described the P2012 backend, focusing on fault recovery mechanisms using dynamic reconfiguration [22].

In the current paper we have extended the P2012 backend to support data intensive computations. Our P2012 backend targets the whole platform including its integration with the host system.

Frontend: The frontend of the Tock compiler consists of several modules, which perform operations like lexical analysis, parsing and semantic analysis. The frontend of the compiler has been extended to support mobile data and channel types, dynamic

process invocation, and process placement attributes [18][5]. We have also introduced new grammar rules corresponding to the additional constructs to create Abstract Syntax Trees (AST) from tokens generated at the lexical analysis stage. In the current work, we have revised the frontend in order to provide support for channels that communicate an entire array of data in a single transfer.

The transformation stage consists of a number of passes either to reduce complexity in the Abstract Syntax Tree (AST) for subsequent phases or to convert the input program to a form which is suitable for the backend or to implement different optimizations required by some specific backend.

P2012 Backend: The P2012 backend generates the complete structure of application components in NPM as well as the host-side program to deploy, control and run the application components on the P2012 fabric. The generated code can then be executed on the GePOP simulation environment. The P2012 backend is divided into two main passes. The first pass traverses the AST to create a list of parameters passed in procedure calls specified for processes to be executed in parallel. In addition to parameters the list also includes two integer values which store the first value and the count of replicated `PAR`.

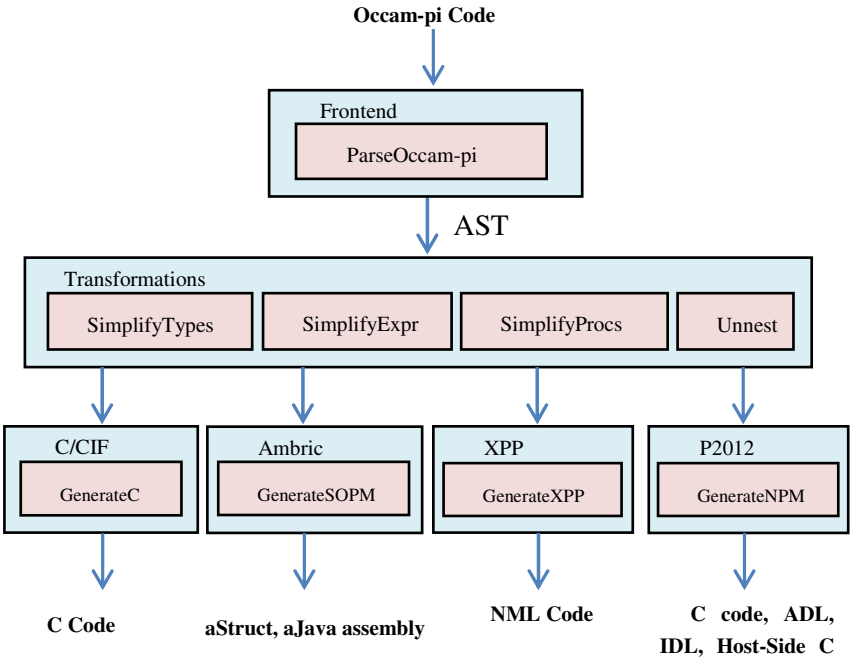


Fig. 2. Occam-pi compiler block diagram

Since a procedure can be called more than once in different places, besides name of the procedure, a counter and the name of the procedure that calls the procedure (parent procedure) is also added on the parameter list to indicate parameters of this

particular procedure call. To facilitate the code generation, if the list is composed of several parent procedures and simple procedures, it will be transformed to a list of simple procedures and one parent procedure. This list of parameters of procedure calls is used to generate the required and provided interface of each component along with its specific binding codes, i.e., the architectural description of the application using ADL and IDL. Listing 1b shows the ADL file generated for a component called 'prod', which corresponds to a process call in `occam-pi` (Listing 1a). PullBuffer and PushBuffer are services provided by the NPM communication components. The two source files, 'prod_cc.c, and 'so_prod.c', will be generated in the next pass.

```
PROC SimpleEx()
  CHAN INT e:
  CHAN INT f:
  PAR
    prod(f?,e!)
    con(e?,f!)
:
```

(a)

```
primitive SimpleEx.prod {
    requires PullBuffer as f;
    requires PushBuffer as e;
    @CC
    source prod_cc.c;
    source so_prod.c;
}
```

(b)

Listing 1. Translation of `Occam-pi` process (a) to ADL file (b)

The list of parameters of procedure calls is also used to generate deployment, instantiation and control code of an application component from the host-side. For each procedure call, binary code of the procedure is deployed on the intended cluster using the NPM_instantiateAppComponent API, then the cluster controller will execute this binary code on one of the ENCore processors. The NPM_instantiateFIFOBuffer API is used to bind the push buffer with the corresponding pull buffer. For replicated `PAR` an array of processes is created and a for-loop is used to deploy, run and stop the processes. The for-loop gets the start and count of the replicator from the information stored in the list of the procedure calls. Listing 2 shows the translation of replicated `PAR` of `occam-pi` to the corresponding host code sequences that instantiate, deploy, run and stop each process.

The second pass generates implementation code of the application components and the cluster controller. The `genProcess` function traverses the AST to generate the corresponding extended C code for different `occam-pi` primitive processes such as assignment, input process (?), output process (!), `WHILE`, `IF/ELSE`, and replicated `SEQ`. Since we are not using the execution engines, the cluster controller code uses runtime APIs to execute, control and configure the application component. Cluster controller code is differentiated from the component code by inserting the `@CC` annotation; in Listing 1b `prod_cc.c` will be executed on the cluster controller and `so_prod.c` will run on the ENCore processors.

```
PAR pr=0 FOR 16
  fastProc (idT[pr],inIm[pr], offsetX[pr],
            offsetY[pr], inF[pr]?, outF[pr]!)
```

(a)

```
fastProc_processor_bare_t   fastProc_inst_100[16];

for(pr=0;pr<(16+0);pr++)
   err = deployfastProcBare(pr, &fastProc_inst_100[pr]);

for(pr=0;pr<(16+0);pr++)
  NPM_run(&fastProc_inst_100[pr].appComp.runItf);

for(pr=0;pr<(16+0);pr++)
  CM_stop(fastProc_inst_100[pr].appComp.comp);
```

(b)

Listing 2. Translation of Replicated `PAR` (a) to corresponding C code (b)

6 Experimental Case Study

In this section, we will describe the implementation of a FAST Corner Detection algorithm, which is used to evaluate our compilation methodology. We have compared our implementation in `occam-pi` with a hand written NPM version and with an OpenCL implementation.

6.1 Features from Accelerated Segment Test (FAST) Corner Detection

FAST is an algorithm that is used to spot corners in an image [7]. In image processing, corners are detected and used to derive a lot of information that is important for computer vision systems. The FAST corner detection algorithm is a high performance detector, suitable for real-time visual tracking applications that run on limited computational resources. According to Rosten et al [19], FAST performs better than conventional algorithms in terms of execution time and repeatability (i.e., detecting the same corner in several similar images).

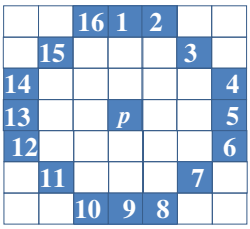


Fig. 3. Bresenham circle of radius three surrounding the pixel of interest

The FAST algorithm examines a pixel by comparing the intensity value of the pixel with the values of sixteen pixels that surround the pixel in a Bresenham circle of radius three, as shown in Fig. 3 [19]. Among the sixteen pixels, if the intensity of N pixels are either greater than or less than the intensity of the pixel by a threshold T, then that pixel is categorized as a corner. In our implementation, the values of N and T are set to 14 and 35, respectively. This step usually detects multiple neighboring pixels as a corner. To solve this problem, the score of each corner pixel is computed and corner candidates with lower score are discarded by using non-maximal suppression [19].

To speed up the computation, we have implemented a parallel version of the algorithm using `occam-pi` primitives. To control the degree of parallelism, we have used replicated `PAR` statements of `occam-pi`. As shown in Listing 3, the amount of parallelism can be varied by changing just one parameter (noP). In the implementation the host-CPU loads and splits the image vertically for the given number of processes (noP). Listing 3 shows sample `occam-pi` code that starts with converting the RGB image to a grayscale intensity image, and then splits the intensity image according to the given number of processes (noP), which are instantiated by using replicated `PAR`. In our implementation, we create a circle of 16 pixels that surround a pixel p under test and then in order to identify the pixel as a corner the intensity value of 14 neighboring pixels has to be above or below intensity of p by the threshold value of 35.

As mentioned above, to examine a pixel we have to create a Bresenham circle of radius three, which requires a 7x7 block. So, to examine boundary pixels, a process has to share three columns of pixels from both left and right processes. Since an `occam-pi` process cannot share data with any other process, the host-CPU duplicates three columns of pixels on the new borders that are created when the image is split. Therefore, each process examines the pixels of its own portion of the image and computes the scores for detected corners without sharing any data with other processes.

```
inImT[i][j]:=((im[i][j][1]+im[i][j][2])+
im[i][j][0])/3

SEQ k=0 FOR noP
SEQ jy=0 FOR (procWT+6)
input[k][jy] := inImT[i][jy+(procWT*k)]

PAR pi=0 FOR noP
inF[pi] ! input[pi]

PAR pr=0 FOR noP
fastProc (idT[pr], inIm[pr], offsetX[pr],
offsetY[pr], inF[pr]?,outF[pr]!)

  SEQ op=0 FOR noP
  outF[op] ? output[op]
```

Listing 3. Occam-pi code that splits an intensity image for the given number of processes

If a pixel is not detected as a corner, its score is –1. From its portion a process reads seven lines and examines the pixels in the middle row one by one. When it's done with the middle row, the process pushes the computed score as an output, releases the first input line, moves the remaining six image lines upward, fetches the next input line, and starts to examine the pixels in the new middle row until it has fetched the last input line. In our implementation each process (fastProc from Listing 3) is executed by one ENcore processor and we use the host CPU to select the good corners.

Just like our implementation, the FAST implementation that comes with the P2012 SDK uses ENcore processors to detect corners and to compute scores, and the host CPU for non-maximal suppression. This implementation is not modified. The implementation reads the entire line of the input image and spawns sixteen slave processes using an RTM engine which then works on a specific portion of the input image.

7 Implementation Results and Discussion

In this section, we will analyze our compilation methodology using the FAST corner detection case study. Our aim is to demonstrate the applicability of the programming model of `occam-pi`, to verify that programming is simplified when using the `occam-pi` language, and to assess the competitiveness in terms of performance. We compare our compilation based implementation with one implementation that was hand written in NPM, as well as with an other compiled implementation based on OpenCL. We implement a computation intensive application, which can benefit from the parallel compute resources of P2012 and show the simplicity of using `occam-pi` to express parallelism in an algorithm where communicating processes are natural elements of abstraction.

In Table 1 we have compared our implementation with the hand written NPM version. As a measure of implementation complexity we use the number of lines-of-code. The `occam-pi` program shows significant reduction in lines-of-code, 2x in the implementation of FAST. In Table 1, we also show the set up times and execution times for both versions. The set up time includes the configuration and deployment, and the execution time includes computation and communication time.

Both versions of the FAST implementation use 16 processes to detect corners and to compute corner scores, and they both use the host CPU to execute non-maximal suppression. As seen by the measured performance times, the `occam-pi` version outperforms the hand written version not only in simplicity in terms of lines-of-code but also in speed. The difference in time is the result of three main reasons:

1. The FAST implementation in `occam-pi` transfers data in the form of arrays. After the image is split, each process reads the entire line of its portion in a single step.
2. The hand written version has an overhead of protecting shared memory accesses. The `occam-pi` version solves this problem by duplicating the boundary pixels when splitting the image.
3. The third reason is the overhead due to dynamic allocation of resources when using RTM engine.

Both versions of FAST implantations have been tested on the same image (VGA sized input image) and they have detected 3146 corners. With non-max suppression, 772 have been selected as good corners. The (identical) outputs of the `occam-pi` version and the hand written NPM implementation are shown in Fig. 4.

Image analysis applications are usually data intensive and are suitable for programming models that can expose a high degree of data-level parallelism like OpenCL [20]. Our `occam-pi` implementation has utilized data-level parallelism by duplicating critical sections and by using channels that transfer an entire array of data. By this it has achieved better performance than the NPM version, which uses RTM engines. An implementation of FAST on P2012 using OpenCL was reported in [21], where 777 corners were detected as good corners on the same image that was used in the case of `occam-pi` and NPM implementations, resulting in an execution time of 30 milliseconds. In the case of OpenCL implementation, the threshold value is varied from 20 to 35 to get the best value of detected corners.

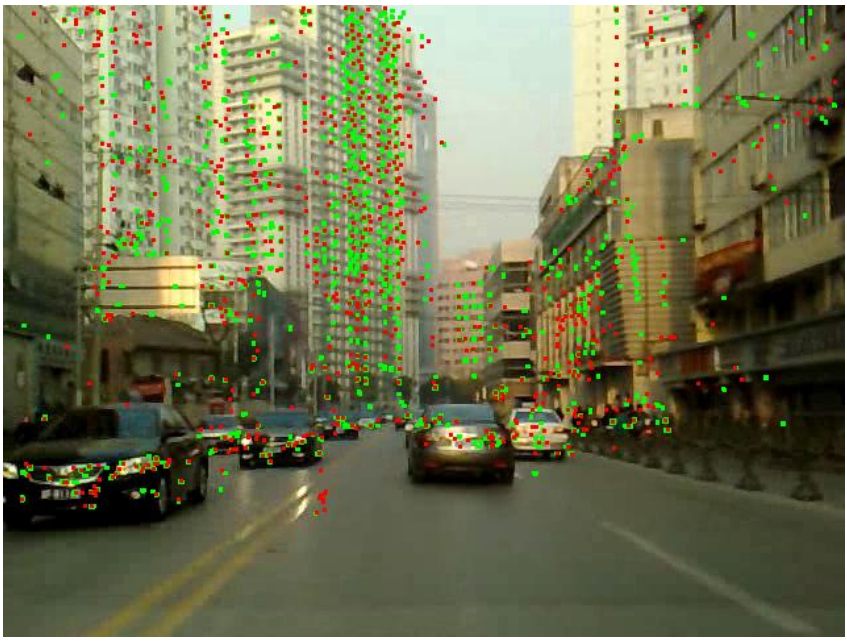


Fig. 4. An image with detected corners (red dots) and suppressed corners (green dots)

Table 1. Simulation results for the FAST Corner Detection implementations

Simulation Results/Languages	NPM	OpenCL	Occam-pi
Lines of Code	453	450	190
No of ENcore processors	16	16	16
Setup time (µs)	53,383	-	47,515
Execution time (µs)	67,115	30,000	32,549

The implementation results reveal that both the OpenCL and occam-pi implementations outperform the NPM version in terms of execution time. The occam-pi implementation is much simpler when compared to the OpenCL and hand-coded NPM versions, which is evident from the lines of code counts. From the implementation results we can see that the cost of configuration and deployment (the setup time) is significant as compared to the actual execution time. Especially, dynamic loading of tasks, as done in the RTM engine is very costly. But, if the processors are deployed at start up and used for long time, which is often the case in streaming applications, the time spent on configuration and deployment could be compensated. The OpenCL implementation was developed under that assumption, therefore the setup time was not measured. On the other hand, knowing the setup time may be important when scheduling reconfigurations.

The implementation using occam-pi is more concise than the two other versions. This is a consequence of the high level constructs of the language, which is also a feature that leads to fewer opportunities to introduce errors and to a higher likelihood of finding errors. Also, using a high-level approach like occam-pi and OpenCL makes the program easier to *scale*, in the sense that changing the number of processing elements involved in the computation is determined in one place in the program (the bound for the replication of processes). We gain also in portability given that a change of platform requires a change in one program: the compiler, instead of changes to all applications.

OpenCL implementations are based on the single instruction stream, multiple threads (SIMT) execution model, meaning that each processing element is executing the same instruction flow. On the other hand the occam-pi implementations are based on the multiple instruction streams, multiple data streams (MIMD) approach, where each processing core can execute its own instruction stream. This closely resembles the underlying manycore architecture.

8 Conclusions and Future Work

We have presented our approach to map programs in a CSP based language to a manycore architecture. We have extended our occam-pi compiler framework to generate native programming language code for Platform 2012. We have shown the simplicity of programming in occam-pi and the performance competitiveness of our compilation based approach through a case study using FAST corner detection implementations. The result of the case study demonstrates the practicality of our approach for an algorithm that is both communication intensive and compute-data intensive. It has been concluded from the results that the occam-pi implementation achieves much better execution time results with respect to the hand-coded NPM version with a relatively less development effort. The occam-pi implementation execution time results are also comparable to those of an OpenCL version, again at a reduced development effort as evident from the lines of code counts. Future work will focus on making further evaluations of the approach using complex examples.

Acknowledgment. The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement number 100230 and from the national programmes / funding authorities.

References

1. Welch, P.H., Barnes, F.R.M.: Communicating mobile processes Introducing occam-pi. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) Communicating Sequential Processes. LNCS, vol. 3525, pp. 175–210. Springer, Heidelberg (2005)
2. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
3. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes Part I. Information and Computation 100(1) (1989)
4. Zain-ul-Abdin, Svensson, B.: Using a CSP based programming model for reconfigurable processor arrays. In: International Conference on Reconfigurable Computing and FPGAs (2008)
5. Zain-ul-Abdin, Svensson, B.: Occam-pi as a high-level language for coarse-grained reconfigurable architectures. In: 18th International Reconfigurable Architectures Workshop (RAW 2011) in conjunction with International Parallel and Distributed Processing Symposium (IPDPS 2011) (May 2011)
6. STMicroelectronics and CEA.: Platform 2012: A manycore programmable accelerator for ultra-efficient embedded computing in nanometer technology (November 2010)
7. Rosten, E., Drummond, T.: Fusing points and lines for high performance tracking. In: Proceedings of 10th IEEE International Conference on Computer Vision, vol. 2, pp. 1508–1515 (2005)
8. Chamberlain, B., Callahan, D., Zima, H.: Parallel Programmability and the Chapel Language. International Journal of High Performance Computing Applications 21(3), 291–312 (2007)
9. Steele Jr., G.L.: Parallel programming and parallel abstractions in fortress. In: IEEE PACT, p. 157 (2005)
10. Charles, P.C., Grothoff, S.V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. SIGPLAN Not. 40(10), 519–538 (2005)
11. Axelsson, E., Claessen, K., Devai, G., Horvath, Z., Keijzer, K., Lyckegård, B., Persson, A., Sheeran, M., Svenningsson, J., Vajda, A.: Feldspar: A domain specific language for digital signal processing algorithms. In: 8th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE), pp. 169–178 (July 2010)
12. Eker, J., Janneck, J.W.: CAL language report. Technical Report, UCB/ERL M03/48 (2003)
13. Püschel, M., Moura, J., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: Spiral: Code generation for DSP transforms. In: EEE Special Issue on Program Generation, Optimization, and Platform Adaptation (2005)
14. Occam® 2.1 Reference Manual, SGS-Thomson Microelectronics Limited (1995)
15. Welch, P.H., Barnes, F.R.M.: Prioritised dynamic communicating processes: Part II. In: Communicating Process Architectures, pp. 353–370. IOS Press (2002)
16. The MIND Project (December 15, 2011), `http://mind.ow2.org`
17. Kent.: Tock: Translator from Occam to C (December 15, 2011), `http://projects.cs.kent.ac.uk/projects/tock/trac/`

18. Zain-ul-Abdin, Svensson, B.: Occam-pi for programming of massively parallel reconfigurable architectures. In: International Journal of Reconfigurable Computing 2012, Article ID 504815 (2012)
19. Rosten, E., Porter, R., Drummond, T.: FASTER and better: A machine learning approach to corner detection. IEEE Transactions on Pattern Analysis and Machine Intelligence 32, 105–119 (2010)
20. The Khronos Group, OpenCL 1.0 (December 21, 2012), `http://www.khronos.org/opencl`
21. Melpignano, D., Benini, L., Flamand, E., Jego, B., Lepley, T., Haugou, G., Clermidy, F., Dutoit, D.: Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications. In: 49th Annual Design Automation Conference (2012)
22. Zain-ul-Abdin, Gebrewahid, E., Svensson, B.: Managing Dynamic Reconfiguration for Fault-tolerance on a Manycore Architecture. In: 19th International Reconfigurable Architectures Workshop (RAW 2011) in conjunction with International Parallel and Distributed Processing Symposium (IPDPS 2011), pp. 312–319 (May 2012)

Self-adaptive Retransmission for Network Coding with TCP

Chunqing Wu*, Hongyun Zhang, Wanrong Yu,
Zhenqian Feng, and Xiaofeng Hu

School of Computer, The National University of Defense Technology,
410073, Changsha, Hunan, China
wuchunqing@nudt.edu.cn, {grandcloud88,wangrongyu}@gmail.com

Abstract. Incorporating network coding with TCP is a natural way to enhance the robustness and effectiveness of data transmission in lossy channels, it can mask packet loss by mixing data across time and across flows. The key of this approach is a suitable retransmission scheme which can adjust according to the changed of the lossy channel condition. However, most retransmission schemes can't compensate losses effectively. In this paper we propose a novel self-adaptive retransmission scheme combining prospection with compensation, which can dynamically adjust the number and time of coding packets's retransmission according to the channel state change. Compensatory retransmission transmit exact number of packets the receiver needs for decoding all packets based on feedback, and prospective retransmission transmit extra packet before losses happened, and the redundancy factor R is adjusted based on the channel conditions. The scheme can work well on handling not only random losses but also bursty losses. Our scheme also keeps the end-to-end philosophy of TCP that the coding operations are only performed at the end hosts. Thus it is easier to be implemented in practical systems. Simulation results show that our scheme significantly outperforms the previous coding approach in reducing size of decoding matrix and decoding delay, and and produces better TCP-throughput than the standard TCP/NC, TCP-Reno.

Keywords: network coding, retransmission, TCP, packet loss, redundancy packet.

1 Introduction

It is well known that TCP protocol has an awful performance in the lossy wireless network[2][3][4]. It is because that each loss is interpreted as a congestion signal in TCP. Network Coding allows nodes of a network to send packets that are linear combinations of previously received information, instead of delivering the

* The work described in this paper is partially supported by the project of National Science Foundation of China under grant No. 61103182; the National High Technology Research and Development Program of China (863 Program) No. 2011AA01A103.

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 396–407, 2013.
© Springer-Verlag Berlin Heidelberg 2013

information to their destination in the standard store-and-forward-manner[1][2]. Network coding has emerged as an important potential approach in the operation of communication networks [1].The main benefits of network coding are the potential throughput improvements and a high degree of robustness to packet losses[7].

In [5] Sundararajan et al. propose TCP/NC protocol that successfully implemented the network coding into TCP with minor changes to the protocol stack. The key idea was introducing a new network coding layer between the transport layer and IP layer in TCP/IP stack, which masks non-congestion packet losses from congestion control algorithm. In this layer TCP segments are encoded at the sender and decoded at the receiver. In[6]Sundararajan et al. present a real-world implementation of this protocol that addresses several important practical aspects of incorporating network coding and decoding with TCP's window management mechanism. For every packet that arrives from TCP, $R(>= 1)$linear combinations are sent to the IP layer on average, These packets are used by the receiver to counteract non-congestion losses.In TCP/NC, the redundancy factor R is constant, we need to know the loss rate of the network circumstance, and set R to the optimal number. However, when the system is under lossy networks, especially wireless network where the loss rate is hart to acquire and not constant, the constant redundancy factor R may cause problems, either sending bunches of useless redundancy packets or being not able to mask the packets loss. Both will impair the performance of the network.

FNC protocol is presented in 2009[7], which focuses on reducing the decoding delay and redundancy by adding some information in packets header. It inherits the coding approach and see packets notion presented by the TCP/NC scheme [9]. In the receiver, the FNC brings in a new factor loss, which indicates how many combinations the sender needs to retransmit enable the receiver decode all the combinations it has received. The loss factor will be sent back to the sender, and the sender uses this factor to decide how many redundant packets should be sent and how many original packets should be coded. By doing this, this new scheme can avoid the retransmission of the useless redundant packets, and due to sending redundancy packets coded by the appropriate number of original packets, it significantly reduces the decoding delay and improves the performance of the networks. However, FNC protocol use a feedback-driven retransmission scheme, it always compensate the loss of past passively. TCP-throughputs is limited by the RTT(Round Trip Time).

The work by Sicong Song et al.[8] propose a new scheme named SANC-TCP. It adds some feedback information in the ACK header to indicate the current network state, thus enable the sender to dynamically change the R according to the real system. SANC-TCP aim to better the utility of the networks and decrease the retransmission of the useless redundant packets. In [7] Hamlet Medina Ruiz et al. propose a loss differentiation scheme to adjust R, based on the Vegas Loss Predictor and the collective feedback information of ACKs and duplicates ACKs. However, when the channel state change rapidly or some bursty losses happened, both this two scheme may lead to TCP time outs because R can't

fast converge to the optimal number. What more, redundancy rate R considered only predictive aspects of packet losses, it can not rapidly compensate for the losses of past.

To overcome the disadvantages in existing approaches, we propose a new Self-adaptive retransmission scheme named TCP-CPNC, which mainly optimizes the retransmission scheme based on TCP/NC. Our scheme combines compensatory retransmission with prospective retransmission. Compensatory retransmission can rapidly counteract the lossy before the retransmission, and prospective retransmission can initiatively transmit suitable redundant coding packets before losses happened. Our scheme can dynamically adjust the number and time of coding packets's retransmission according to the channel state change, and hence significantly improve the performance of throughput and decoding delay under both random losses and bursty losses. Simulation results show that our scheme significantly outperforms the previous coding approach in increasing throughput while reducing decoding delay.

The remainder of the paper is organized as follows. In section 2, we describe basic ideas of network coding in TCP with the *seen* scheme for background and TCP/NC protocol. Details about the self-adaptive retransmission scheme are given in Section 3, simulation results are described in Section 4, and some conclusions and future research directions are drawn in Section 5

2 Network Coding for TCP

Network coding has emerged as an important potential to bring benefits in terms of throughput and robustness, since it can mask the packet loss via redundant packets, thus decrease the delay caused by the timeout and to raise the utilization of the channels. However, Despite this potential of network coding, we still seem far from seeing widespread implementation of network coding across networks[5][6]. And the main problem that needs to be solved is how to put it into practice in real communication network, one solution is implementing the network coding into TCP. To do so, TCP/NC protocol incorporates the *seen* scheme with congestion control and introduces a new network coding layer between the transport layer and the network layer , such practice can lets us utilize the standard TCP protocol with the minimal change.

The *seen* scheme is promoted by ANC[10] firstly. In ANC, the decision of which packets to combine relies on the concept of the *seen* packets. A packet P_k is said to be *seen* by a receiver if it has enough information to compute a linear combination in the form of $(P_k + Q)$, in which q is a linear combination of packets that are newer than P_k, i.e. $Q = \sum_{l>k} \alpha_l P_l$, with $\alpha_l \in \mathbf{F_q}$ for all $l > k$. The receiver acknowledges the oldest *unseen* packet, so the sender always transmits a packet that is a combination of the oldest *unseen* packets of each receiver. A packet can be dropped from the sender queue when it is *seen* by all receivers. The expected queue size of the scheme is reduced from the traditional length $\Omega(1-\varepsilon)^{-2}$, to $\Omega(1-\varepsilon)^{-1}$ where ε is the erasure probability.

TCP/NC's *seen* scheme borrows from ANC, but it is designed with respect to a single source that generates a stream of packet s to single sink. The heart of

TCP/NC is that the sink acknowledges every newly *seen* packet even if it does not reveal an original packet immediately. Such heart enables a TCP-compatible sliding-window approach to network coding. It more easy and efficiency add redundant packets by make use of the ability of network coding to mix data across segment. In the implementation of TCP/NC, the source side buffers packets generated by TCP in the coding buffer, and for every segment arriving from TCP, R random linear combinations of the most W recently arrived packets in the network coding buffer are sent to IP layer on average, where R is the constant redundancy factor. To convey the combination requires an additional network coding header (contain coding coefficients et al.) that is added to the coded packet. On the receiver side, upon receiving a linear combination from the sender side, it first retrieves the coding coefficients from the packet header and appends them to the basis matrix of its knowledge space. Then the Gaussian elimination method is adopted to find the newly *seen* packet and decoded packet. The newly *seen* packet can be ACKed and the newly decoded packet can be submitted to TCP layer. Fig. 1 illustrates an example of encoding and decoding. In addition, any ACK generated by the receiver TCP is suppressed and not sent to the sender. These ACKs may be used for managing the decoding buffer. An important point is that the new NC layer is invisible to the transport layer and IP layer. For more details, the reader is referred to [6].

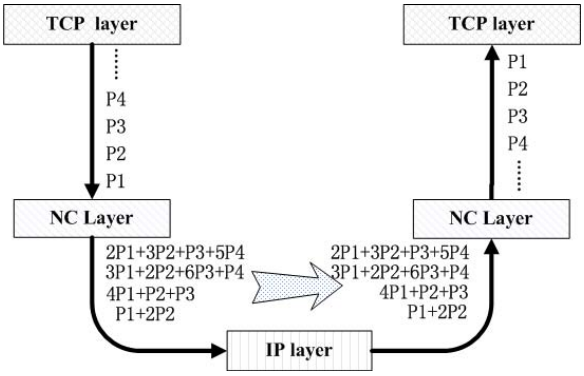


Fig. 1. TCP/NC

3 Self-adaptive Retransmission Scheme

In this section, we will describe the basic ideas of the TCP-CPNC scheme. Our scheme aims to better the utilization of channels by dynamically adjusting the number and time of retransmission in unknown lossy networks. Before introduce TCP-CPNC, we first expatiate four state of uncoded segment in the receiver side buffer, Fig. 2 shows a typical situation.

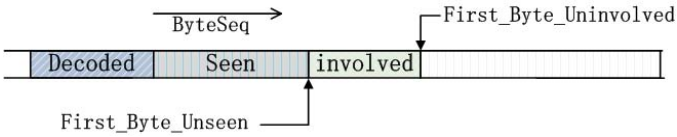


Fig. 2. Four state of segment in receiver buffer

It is easy to understand seen and decoded state, we mainly focus on involved and uninvolved state. When a packet is in involved state means some linear combinations that contain this packet has been receive, but still cant been *seen*. As for uninvolved state, a packet is in uninvolved state when none linear combinations in receiver side contain this packet.

At the receiver, the difference between $First_Byte_Uninvolved$ and $First_Byte_Unseen$ is called *Gap*. When the channel state is stability and retransmission scheme suited to loss rate, the value of *Gap* keep stable at the decoding matrix. When channel condition change or retransmission scheme is not suited to loss rate, the value of *Gap* will also vary.

To fulfill our target, TCP-CPNC combines compensatory retransmission with prospective retransmission. Compensatory retransmission can rapidly counteract the lossy before the retransmission, and prospective retransmission can initiatively transmit suitable redundant coding packets before losses happened. both compensatory and prospective retransmission dynamically adjust the number and transmit time of redundancy packets based on the feedback *Gap* in acknowledge.we adding *Gap* to the ACK header.

The sender should maintain a variable ACK_Count which indicates the number of ACK receive from sink. The sender enter a period of adjustment whenever it receive 10 ACK. For compensatory retransmission, the value of *Gap* is the exact number of packets the receiver needs for decoding all packets in decoding matrix. Each compensation / adjustment period to Source side send *Gap* linear combinations of the original packet in coding window of NC layer, these combinations can effectively accelerate the decoding speed of decoding matrix. For prospective retransmission, The values of *Gap* change with the bit error rate(BER) of channel. So we perceive the change of loss rate by comparing *Gap* on ACK with *SGap* on source side, and thereby dynamically adjust the redundancy factor R. When sender is in explorative period, we maintain two variable:$riseCount$ and $redundancyCount$, $riseCount$ record the time of *Gap*'s augment, $redundancyCount$ record the time of redundancy packet. In adjustment period, If acked *Gap* is -1 which indicates redundancy packets is too many, R is decreased until the lower limit(LL). On the other hand, if acked *Gap* is up 0, R should increase (below the upper limit(UL)) to counterbalance channel loss. Initially R is set to a value R0 that takes into account the losses in throughput due to the finiteness of the field. The improved algorithm is specied in Algorithm 1 using pseudo-code.(both $\alpha, \beta, \mu < 1$)

To make it clear, we independently describe the actions which are taken on the sender and receiver side. Provided we have introduced a network coding

Algorithm 1. adaptive alg

```
Initialization: ACK_Count = 0, SGap = 0, R0 = 1.05, R = R0 , UL = 2, LL = 1.05
Each time an ACK is received: ACK_Count++ and Pick up Gap in NC_ACK header
if ACK_Count = 10 then
    NC layer enter adjustment period
    if Gap == −1 then
        if R > LL then
            R = R − α
        end if
        SGap = 0
    else
        SGap = Gap
        if Gap > 0 then
            Send Gap linear combination to receiver
            if R < UL then
                R = R + riseCount ∗ μ + redundancyCount ∗ β
            end if
        else
            R remain unchanged
        end if
    end if
    redundancyCount = 0
    riseCount = 0
    ACKCount = 0
else
    NC layer enter explorative period
    if Gap = −1 then
        redundancyCount++
    else
        R remain unchanged
        SGap = Gap
        if Gap > SGap then
            redundancyCount++
        end if
    end if
end if
```

layer between the transport layer and the IP layer. To implement our scheme in the network coding layer, we make some minor changes to standard TCP/NC protocol. The changes algorithm is specied below:

Source Side :

ACK arrives from receiver
1) Call $Algorithm1$
2) Remove the $ACKed$ packet from the coding buffer
3) Generate a new ACK by the ACK and send it to the TCP layer

Sink Side :

Packet arrives from source side:
1) Performs Gaussian elimination to update the decoding matrix.
2) Update $First_Byte_Uninvolved$ and and $First_Byte_Unseen$
3) Generate a new ACK, consisting of the value of Gap which is the difference between $First_Byte_Uninvolved$ and $First_Byte_Unseen$

Following the approach above, the sender adjusts the retransmission scheme from time to time, thus to dynamically change the R according to the real system. The algorithm to adjust the redundancy factor R in the sender is showed in Fig. 3 And the algorithm feedback Gap in the Sink side is showed in Fig. 4

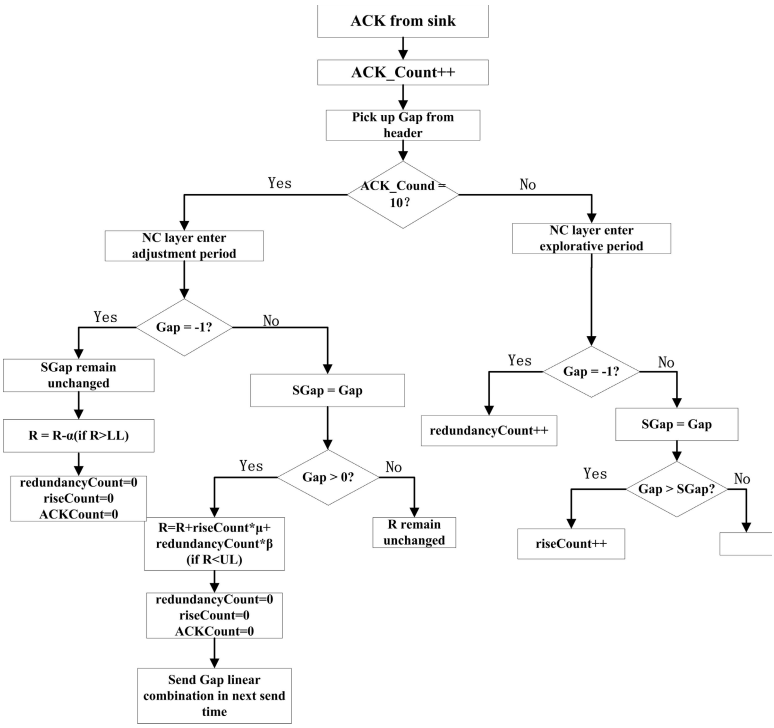


Fig. 3. The algorithm to adjust retransmission in sender

4 Simulation Results

The Implementation of TCP/NC with self-adaptive retransmission is base on discrete event simulation environment OMNET++ and the open source TCP/IP protocol framework INET. We also use OMNET++ to evaluate and compare the performance of different protocol in network. The topology for the simulations is a tandem network consisting of 7 nodes, three router and 4 host,shown in Fig. 5. The source and sink nodes are at opposite ends of the chain.

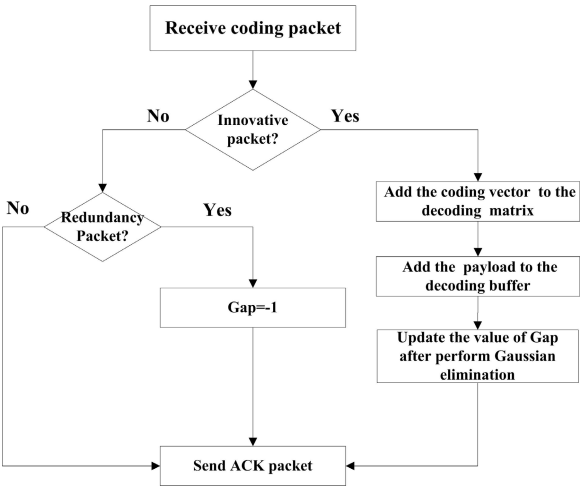


Fig. 4. The algorithm to feedback in sink

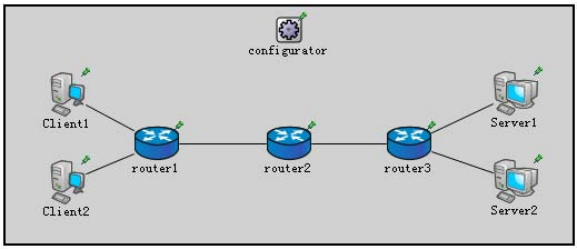


Fig. 5. Topology of network

4.1 Simulation Environment Setup

In this system, there are two flows generated by two FTP applications, the applications type of sender side is TCPSessionApp and receiver side is TCPSinkApp. One flow is from Client1 to Server1, and the other is from Client to Server2. They will compete for the intermediate channels. The queue type of wire interface is DropTailQueue which the first item stored is the first item output. The frame capacity of DropTailQueue is 150. All the channels have a bandwidth of 1 Mbps, and the propagation delay between host to router is 10ms, between routers is 50ms. The TCP receive NC layer buffer size is set to 200, and the IP packet size is 556 bytes. TCP-Reno is chosen for the transport layer protocol. The TCP-throughput is measured using outputhook(a kind of measure class in INET framework). Each point is averaged over 4 or more iterations of such session, depending on the variability.

4.2 Simulation Results

In order to evaluate the effect of our medication of our simulate protocol on fairness, we first study the fairness of the standard TCP and TCP/NC with our self-adaptive retransmission scheme. By fairness, we mean that if two similar flows compete for the same link, they must receive an approximately equal share of the link bandwidth[5]. We figure out the fairness characteristic under three different situation:

*Situation*1: a TCP-Reno flow competes with another flow running TCP/NC with self-adaptive scheme.

*Situation*2: a standard TCP/NC flow competes with another flow running TCP/NC with self-adaptive scheme.

*Situation*3: two TCP/NC flows with self-adaptive scheme compete with each other.

In three cases, the loss rate is set to 0% and the redundancy parameter is set to 1.05 for a fair comparison. The current throughput is calculated at intervals of 1s. TCP/NC with self-adaptive scheme flow start at t=0.1s, the second flow start at t=30s. In situation 1, the second flow is TCP-Reno flow, and it is standard TCP/NC flow in simulation 2, TCP/NC with self-adaptive scheme in simulation 3. Three simulation is all over in 120s. The plot for both three simulation is essentially identical to Fig. 6 (and hence is not shown each simulations respective figure). Both the three simulations show that when the second flow joins in the channel, it quickly shares an equal amount of bandwidth of the channel with the previous TCP/NC with self-adaptive scheme flows, thus proving the fairness of new scheme with TCP/NC.

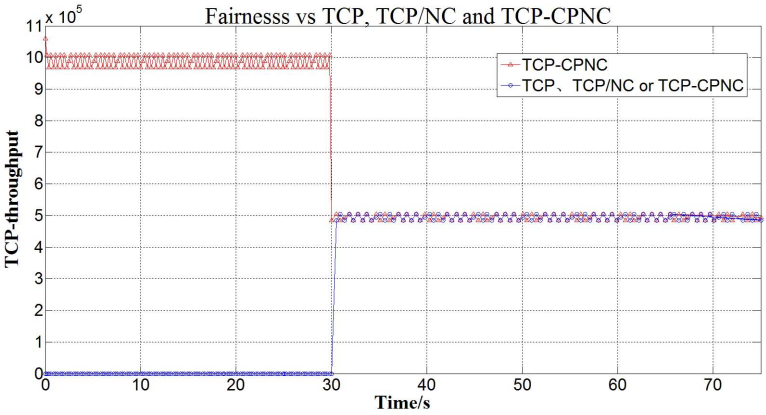


Fig. 6. A TCP/NC compete another TCP flow

Next, we try to prove that our new self-adaptive scheme has a better throughput rate and lower decoding delay under lossy channels with variational packet error rate(PER). Packets in the network are subject to these losses in the forward and the reverse direction. The PER can been calculated by a equivalence

bit error rate(BER), since the size of packet is stable. We study the variation of receive Seq with time. PER vary over time: 0-50s: 5% PER, 50-90s: 40% PER,PER is set to 20% after 90s. Only flow from client1 to server1 is choose and the size of this flow is 5MB. Fig. 7 shows the evolution of the Seqs sent by the Server1s NC layer as a function of time for different values of R, as well as coding packets is dynamically retransmit. Fig. 8 shows the value of R in Client1's NC layer as a function of time.

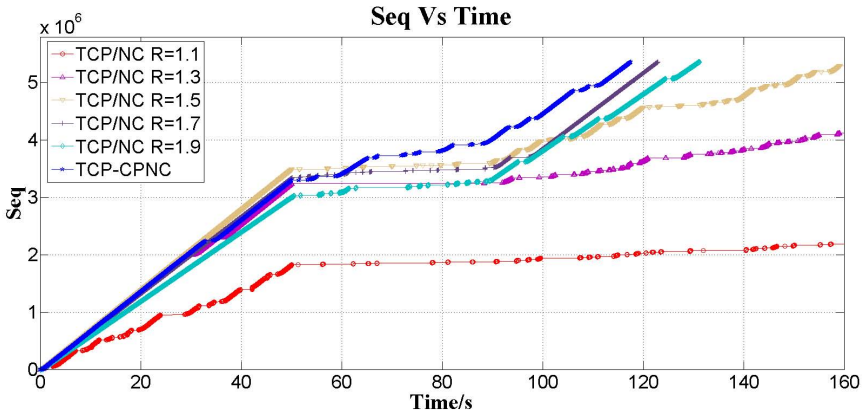


Fig. 7. Revd Seqs vs Time

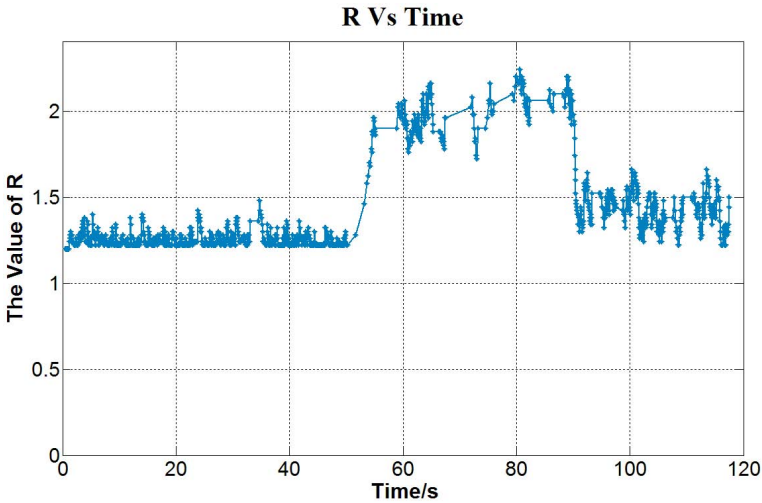


Fig. 8. R vs Time

We can observe that R plays an important role in TCP/NC. For standard TCP/NC, The peak average throughput achieved is 0.325Mbps(5MB*8/123.002) when R=1.7, but TCP/NC with our self-adaptive can achieve 0.34Mbps(5MB*8/117.59). We clearly appreciate the improvement

obtained in TCP-throughput with our scheme. We believe that when BER of channel is more protean, our scheme can archived better performance. Fig. 9 shows that the max size of decoding matrix is remain small while increasing TCP-throughput.

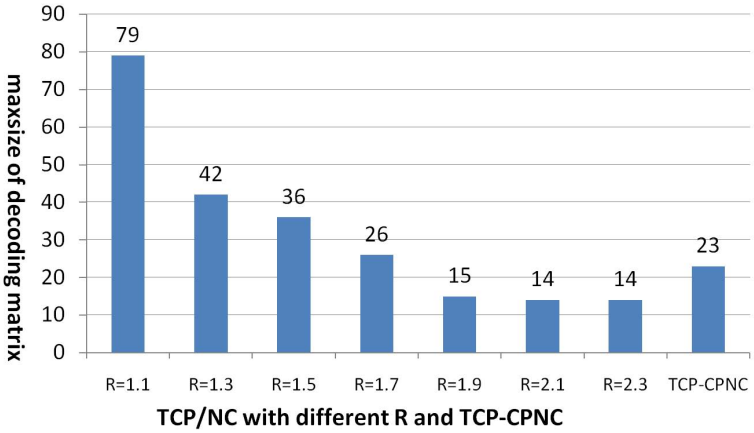


Fig. 9. Decoding Buffer vs TCP/NC

All previous simulations results are get by set coding window as 3. Fig. 10 shows that when PER is 5%, R have been chosen the optimization by trial and error, the max size of decoding matrix grows rapidly as coding window increase. In contrast, our scheme also keeps quite a small size and the value does not increase as the coding window increases. Whats more, the decoding delay mainly

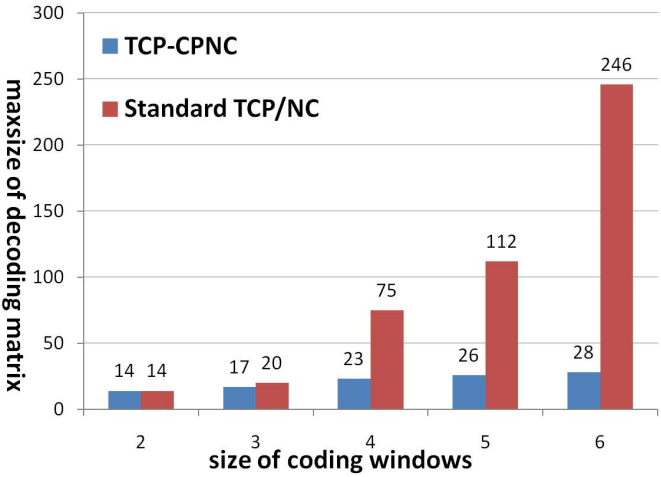


Fig. 10. Decoding Buffer vs Coding Windows

rests with the size of decoding matrix, a small decoding matrix can achieve a small decoding delay.

5 Conclusions and Future Works

Incorporating network coding with TCP works much better than TCP for loss channels. However, due to the different PER in different period of time in lossy networks, the retransmission scheme of coding packet is the key to compensate the non-congestion packets losses.

In this paper, we proposes an self-adaptive retransmission scheme in the TCP/NC protocol based on the collective feedback information of sinks decoding matrix, the transmit of redundancy packets is adjust according to the real current circumstance.

Simulation results show that our scheme significantly outperforms the standard TCP protocol and standard TPC/NC protocol in reducing size of decoding matrix and TCP-throughput. Furthermore, we intend to implement our algorithm in a Linux protocol stack to assess of relation between PER, coding window, R and available computer/memory resource,and investigate new scheme of adjust coding window,

References

1. Ahlswede, R., Cai, N., yen Robert Li, S., Yeung, R.W.: Network information flow. IEEE Transactions on Information Theory 46(4), 1204–1216 (2000)
2. Li, S.R., Yeung, R.W., Cai, N.: Linear Network Coding. IEEE Transactions on Information Theory 49, 371–381 (2003)
3. Polyzos, G.C., Xylomenos, G.: Internet Protocols over Wireless Networks. In: Gibson, J.D. (ed.) Multimedia Communications: Directions and Innovations. Academic Press (2000)
4. Lefevre, F., Vibier, G.: Understanding TCPs behavior over wireless links. In: Proc. IEEE Symposium on Computers and Communications (June 2000)
5. Sundararajan, J.K., Shah, D., Médard, M., Mitzenmacher, M., Barros, J.: Network coding meets TCP. In: Proceedings of IEEE INFOCOM, pp. 280–288 (April 2009)
6. Sundararajan, J.K., Jakubcza, S., Médard, M., Mitzenmacher, M., Barros, J.: Interfacing network coding with TCP: an implementation. In: Proceedings of IEEE INFOCOM, pp. 280–288 (April 2009)
7. Chen, J., Liu, L.X., Hu, X.H.: Effective Retransmission in Network Coding for TCP. Int. J. of Computers, Communications and Control 6(1), 53–62 (2011)
8. Song, S., Li, H., Pan, K.: Self-Adaptive TCP Protocol Combined with Network Coding Scheme. In: The Sixth International Conference on Systems and Networks Communications (2011)
9. Ruiz, H.M., Kieffer, M., Pesquet-Popescu, B.: Redundancy Adaptation Scheme for Network Coding with TCP. In: 2012 International Symposium on Network Coding (2012)
10. Ho, T.: Networking from a network coding perspective. PhD Thesis, Massachusetts Institute of Technology, Dept. of EECS (May 2004)

Author Index

Beni, Laleh Aghababaie 199
Bilas, Angelos 1

Cammarota, Rosario 199
Chang, Junwang 62, 347
Chasapis, Dimitrios 1
Chen, Chen 77
Chen, Wenzhi 32
Chen, Xiao 32
Chen, Zhigang 160
Cheng, Yuxia 32
Corporaal, Henk 184
Custers, Pieter 184

Dastgeer, Usman 170

Fan, Dongrui 241
Fang, Li 77
Feng, Shuaitao 47
Feng, Zhenqian 396

Gao, Long 277
Gaspes, Veronica 381
Gebrewahid, Essayas 381
Gu, Jianhua 367
Guo, Jian 359
Guo, Zhenhua 307

He, Yanzhang 132
Hou, Gang 62, 347
Hu, Lei 104
Hu, Xiaofeng 396
Huang, He 91
Huang, Min 334

Jego, Bruno 381
Ji, Zhenzhou 160
Jia, Zhiping 119
Jiang, Xiaohong 132
Ju, Lei 119

Kessler, Christoph 170

Lavigueur, Bruno 381
Li, Chongmin 254
Li, Guohong 254
Li, Keqin 334
Li, Lu 170
Li, Mingchu 62
Li, Qiong 146
Li, Rui 62
Li, Shanshan 91
Li, Xiang 132
Li, Yibin 119
Liao, Xiangke 91
Lin, Bin 91
Long, Xiang 47, 104
Lu, Kai 77
Lu, Pingjing 214
Luo, Yingwei 295

Ma, Ran 132

Nicolau, Alexandru 199
Nikolopoulos, Dimitrios S. 1, 17
Nugteren, Cedric 184

Pang, Zhengbin 214
Papatriantafyllou, Angelos 17
Pratikakis, Polyvios 1, 17

Qian, Kun 359
Qiao, Yuran 319

Ren, Longtao 347
Robart, Mathieu 381

Shao, Lisong 277
Shen, Xukun 47, 104
Song, Fenglong 241
Song, Zhenlong 146
Sui, Tianxiang 307
Sun, Jiajia 334
Svensson, Bertil 381

Tan, Yusong 277
Tang, Zhimin 241
Tu, Qun 269
Tzenakis, George 17

Vajdi, Ahmadreza 227
Vandierendonck, Hans 17
Veidenbaum, Alexander V. 199

Wang, Da 241
Wang, Dongsheng 254
Wang, Haixia 254
Wang, Lei 47
Wang, Shaogang 214
Wang, Tianshu 227
Wang, Xiaolin 295
Wang, Xiaolong 347
Wang, Xiaoping 77
Wang, Xingwei 334
Wang, Xinheng 269
Wang, Xueyi 334
Wang, Yufeng 91
Wang, Yunlan 367
Wang, Zhenlin 295
Wei, Dengping 146
Wei, Lifeng 277
Wen, Ling 91
Wen, Mei 319
Weng, Lingmei 295
Wu, Chuan 334
Wu, Chunqing 396
Wu, Dan 214
Wu, Fuhui 277
Wu, Hao 160
Wu, Nan 319
Wu, Qingbo 277
Wu, Yanxia 307
Xiao, Liquan 146
Xie, Shuai 119
Xie, Xuchao 146
Xu, Bin 32
Xu, Huijie 359
Xu, Weixia 214
Xu, Xiaolong 269
Xue, Yibo 254

Yang, Jingwei 47
Yang, Xichen 227
Ye, Kejiang 132
Ye, Xiaochun 241
Yong, Jiawei 347
Yu, Wanrong 396

Zain-ul-Abdin, 381
Zakkak, Foivos S. 1
Zhang, Chunyuan 319
Zhang, Gongxuan 227, 359
Zhang, Guoyin 307
Zhang, Hao 241
Zhang, Hongyun 396
Zhang, Shaoyu 32
Zhao, Tianhai 367
Zheng, Siyao 47
Zhou, Kuanjiu 62, 347
Zhou, Xu 77
Zhu, Lei 367
Zhu, Suxia 160
Zhu, Zhaomeng 359