

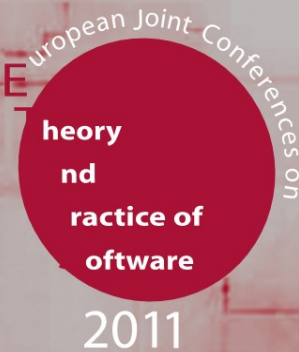
Jens Knoop (Ed.)

ARCoSS

LNCS 6601

Compiler Construction

20th International Conference, CC 2011
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2011
Saarbrücken, Germany, March/April 2011, Proceedings



 Springer

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison, UK

Josef Kittler, UK

Alfred Kobsa, USA

John C. Mitchell, USA

Oscar Nierstrasz, Switzerland

Bernhard Steffen, Germany

Demetri Terzopoulos, USA

Gerhard Weikum, Germany

Takeo Kanade, USA

Jon M. Kleinberg, USA

Friedemann Mattern, Switzerland

Moni Naor, Israel

C. Pandu Rangan, India

Madhu Sudan, USA

Doug Tygar, USA

Advanced Research in Computing and Software Science

Subline of Lectures Notes in Computer Science

Subline Series Editors

Giorgio Ausiello, *University of Rome 'La Sapienza', Italy*

Vladimiro Sassone, *University of Southampton, UK*

Subline Advisory Board

Susanne Albers, *University of Freiburg, Germany*

Benjamin C. Pierce, *University of Pennsylvania, USA*

Bernhard Steffen, *University of Dortmund, Germany*

Madhu Sudan, *Microsoft Research, Cambridge, MA, USA*

Deng Xiaotie, *City University of Hong Kong*

Jeannette M. Wing, *Carnegie Mellon University, Pittsburgh, PA, USA*

Jens Knoop (Ed.)

Compiler Construction

20th International Conference, CC 2011
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2011
Saarbrücken, Germany, March 26–April 3, 2011
Proceedings

Volume Editor

Jens Knoop
TU Vienna
Faculty of Informatics
Institute of Computer Languages
Argentinierstr. 8 / E185.1, 1040 Vienna, Austria
E-mail: knoop@complang.tuwien.ac.at

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-19860-1 e-ISBN 978-3-642-19861-8
DOI 10.1007/978-3-642-19861-8
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011922832

CR Subject Classification (1998): D.2, D.3, D.2.4, C.2, D.4, D.1

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

© Springer-Verlag Berlin Heidelberg 2011

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

The use of general descriptive names, registered names, trademarks, 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.

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)

Foreword

ETAPS 2011 was the 14th instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference that was established in 1998 by combining a number of existing and new conferences. This year it comprised the usual five sister conferences (CC, ESOP, FASE, FOS-SACS, TACAS), 16 satellite workshops (ACCAT, BYTECODE, COCV, DICE, FESCA, GaLoP, GT-VMT, HAS, IWIGP, LDTA, PLACES, QAPL, ROCKS, SVARM, TERMGRAPH, and WGT), one associated event (TOSCA), and seven invited lectures (excluding those specific to the satellite events).

The five main conferences received 463 submissions this year (including 26 tool demonstration papers), 130 of which were accepted (2 tool demos), giving an overall acceptance rate of 28%. Congratulations therefore to all the authors who made it to the final programme! I hope that most of the other authors will still have found a way of participating in this exciting event, and that you will all continue submitting to ETAPS and contributing to make of it the best conference on software science and engineering.

The events that comprise ETAPS address various aspects of the system development process, including specification, design, implementation, analysis and improvement. The languages, methodologies and tools which support these activities are all well within its scope. Different blends of theory and practice are represented, with an inclination towards theory with a practical motivation on the one hand and soundly based practice on the other. Many of the issues involved in software design apply to systems in general, including hardware systems, and the emphasis on software is not intended to be exclusive.

ETAPS is a confederation in which each event retains its own identity, with a separate Programme Committee and proceedings. Its format is open-ended, allowing it to grow and evolve as time goes by. Contributed talks and system demonstrations are in synchronised parallel sessions, with invited lectures in plenary sessions. Two of the invited lectures are reserved for ‘unifying’ talks on topics of interest to the whole range of ETAPS attendees. The aim of cramming all this activity into a single one-week meeting is to create a strong magnet for academic and industrial researchers working on topics within its scope, giving them the opportunity to learn about research in related areas, and thereby to foster new and existing links between work in areas that were formerly addressed in separate meetings.

ETAPS 2011 was organised by the *Universität des Saarlandes* in cooperation with:

- ▷ European Association for Theoretical Computer Science (EATCS)
- ▷ European Association for Programming Languages and Systems (EAPLS)
- ▷ European Association of Software Science and Technology (EASST)

It also had support from the following sponsors, which we gratefully thank: DFG DEUTSCHE FORSCHUNGSGEMEINSCHAFT; ABSINT ANGEWANDTE INFORMATIK GMBH; MICROSOFT RESEARCH; ROBERT BOSCH GMBH; IDS SCHEER AG / SOFTWARE AG; T-SYSTEMS ENTERPRISE SERVICES GMBH; IBM RESEARCH; GWSAAR GESELLSCHAFT FÜR WIRTSCHAFTSFÖRDERUNG SAAR MBH; SPRINGER-VERLAG GMBH; and ELSEVIER B.V.

The organising team comprised:

General Chair: *Reinhard Wilhelm*
 Organising Committee: *Bernd Finkbeiner, Holger Hermanns* (chair),
Reinhard Wilhelm, Stefanie Hauptert-Betz,
Christa Schäfer
 Satellite Events: *Bernd Finkbeiner*
 Website: *Hernán Baró Graf*

Overall planning for ETAPS conferences is the responsibility of its Steering Committee, whose current membership is:

Vladimiro Sassone (Southampton, Chair), Parosh Abdulla (Uppsala), Gilles Barthe (IMDEA-Software), Lars Birkedal (Copenhagen), Michael O’Boyle (Edinburgh), Giuseppe Castagna (CNRS Paris), Marsha Chechik (Toronto), Sophia Drossopoulou (Imperial College London), Bernd Finkbeiner (Saarbrücken) Cormac Flanagan (Santa Cruz), Dimitra Giannakopoulou (CMU/NASA Ames), Andrew D. Gordon (MSR Cambridge), Rajiv Gupta (UC Riverside), Chris Hankin (Imperial College London), Holger Hermanns (Saarbrücken), Mike Hinchey (Lero, the Irish Software Engineering Research Centre), Martin Hofmann (LMU Munich), Joost-Pieter Katoen (Aachen), Paul Klint (Amsterdam), Jens Knoop (Vienna), Barbara König (Duisburg), Shriram Krishnamurthi (Brown), Juan de Lara (Madrid), Kim Larsen (Aalborg), Rustan Leino (MSR Redmond), Gerald Luetzgen (Bamberg), Rupak Majumdar (Los Angeles), Tiziana Margaria (Potsdam), Ugo Montanari (Pisa), Luke Ong (Oxford), Fernando Orejas (Barcelona), Catuscia Palamidessi (INRIA Paris), George Papadopoulos (Cyprus), David Rosenblum (UCL), Don Sannella (Edinburgh), João Saraiva (Minho), Helmut Seidl (TU Munich), Tarmo Uustalu (Tallinn), and Andrea Zisman (London).

I would like to express my sincere gratitude to all of these people and organisations, the Programme Committee Chairs and members of the ETAPS conferences, the organisers of the satellite events, the speakers themselves, the many reviewers, all the participants, and Springer for agreeing to publish the ETAPS proceedings in the ARCoSS subline.

Finally, I would like to thank the Organising Chair of ETAPS 2011, Holger Hermanns and his Organising Committee, for arranging for us to have ETAPS in the most beautiful surroundings of Saarbrücken.

Preface

This volume contains the papers presented at CC 2011, the 20th International Conference on Compiler Construction (CC). This jubilee gives reason to briefly look back at the origins of the CC conference series. CC originated as a series of workshops on compiler compiler that has been organized since 1986 by Günter Riedewald in the German Democratic Republic. In 1992 the series was relaunched by Uwe Kastens in Paderborn. The extension of the scope of CC that was connected with this relaunch was reflected in the new name of the series: Compiler Construction. In 1994 the workshop on compiler construction became the conference on compiler construction. It continued being held every two years until 1998. In 1998 the CC conference federated to ETAPS. Since then CC is one of the ETAPS member conferences and is held every year. These days, CC is a forum for the presentation and discussion of recent developments in processing programs in the most general sense: analyzing and transforming or executing input that describes how a system operates, including traditional compiler construction as a special case.

CC 2011, the 20th instance of the International Conference on Compiler Construction, was held during March 28-29 in Saarbrücken, Germany, as part of the 14th instance of the European Joint Conferences on Theory and Practice of Software (ETAPS 2011). The 15 papers in this volume were selected from 52 submissions giving an overall acceptance rate of 28.85%. The Program Committee (PC) carried out the reviewing and paper selection process completely electronically, in several rounds. Initially, each paper was assigned to three committee members for review. Additional reviews were obtained for papers that were identified during the discussion at the virtual PC meeting in order to help the committee to decide which papers to finally accept. This way every paper was reviewed by at least three reviewers, both by committee members and external experts. Moreover, the reviews of most papers were extended by notes that were taken during the virtual PC meeting. These notes should provide the authors of submitted papers with additional feedback on their papers. They were an anonymous digest of the discussion of the papers and could thus contain the views of several reviewers and committee members. The virtual PC meeting took place from November 29 to December 12, 2010. It was extended by two days in order to allow for a detailed and thorough consideration of every submission. By the end of the meeting, a consensus emerged to accept the 15 papers presented in this volume.

The invited speaker at CC 2011 was Martin Odersky, whose talk was entitled “Future-Proofing Collections: From Mutable to Persistent to Parallel”. The abstract of the invited talk opens these proceedings.

The continued success of the CC conference series for now more than two decades would not be possible without the help of the CC community. First of all, I would like to greatly acknowledge and thank all the authors who submitted a paper. Even if your paper was not accepted this time, I would like to express my appreciation for the time and effort you invested and hope to be able to welcome you to CC 2011 in Saarbrücken all the same. My gratitude also goes to the PC members and the many external reviewers, who wrote reviews, for their hard work and their knowledgeable and substantial reports. My special thanks also go to the developers and supporters of the EasyChair conference management system, whose professional and free service was crucial for handling the submission of papers and camera-ready copies and for running the virtual PC meeting. Finally, I would like to thank the entire ETAPS Steering Committee and the local Organizing Committee of ETAPS 2011, who made this year's 20th jubilee instance of the CC conference series as part of ETAPS 2011 possible.

I hope you will enjoy the papers in these proceedings and that they will be useful for your future work.

January 2011

Jens Knoop

Conference Organization

Program Chair

Jens Knoop TU Vienna, Austria

Program Committee

Alex Aiken	Stanford University, USA
Koen De Bosschere	Ghent University, Belgium
Alain Darte	CNRS, Laboratoire de l'Informatique du Parallélisme, Lyon, France
Evelyn Duesterwald	IBM T.J. Watson Research Center, Hawthorne, USA
Sabine Glesner	TU Berlin, Germany
Robert Glück	University of Copenhagen, Denmark
David Gregg	Trinity College Dublin, Ireland
Sebastian Hack	Saarland University, Saarbrücken, Germany
Matthias Hauswirth	University of Lugano, Switzerland
Christoph Kessler	Linköping University, Sweden
Jens Knoop, Chair	TU Vienna, Austria
Jens Krinke	University College London, UK
Xavier Leroy	INRIA, Paris-Rocquencourt, France
Yanhong Annie Liu	State University of New York at Stony Brook, USA
Kathryn McKinley	University of Texas at Austin, USA
Peter Müller	ETH Zurich, Switzerland
Alan Mycroft	University of Cambridge, UK
Jens Palsberg	University of California, Los Angeles, USA
Markus Schordan	UAS Technikum Wien, Austria
Helmut Seidl	TU Munich, Germany
Jingling Xue	The University of New South Wales, Sydney, Australia

Reviewers

Denis Barthou	Jon Brandvein
Mike Bauer	Matthias Braun
Michael Beyer	Peter Calvert
Klaas Boesche	Poul J. Clementsen
Maximilian Bolingbroke	Joel Denny
Florian Brandner	Peng Di

Tom Dillig
Mattias Eriksson
Paul Feautrier
Morten Fjord-Larsen
Thomas Göthel
Michael Gorbovitski
Bernhard Gramlich
Daniel Grund
Armin Größlinger
Philipp Haller
Nigel Horspool
François Irigoin
Ralf Karrenberg
Lennart C. L. Kats
Uday Khedker
Jörg Kreiker
Roland Leißa
Bo Lin
Helena Loose
Anil Madhavapeddy
Avinash Malik
Christoph Mallon
Torben Mogensen
Mayur Naik

Dominic Orchard
Gabriela Ospina
Scott Owens
Matthew Parkinson
Fernando Magno Quintão Pereira
Michael Petter
Marcel Pockrandt
J. Ramanujam
Charlie Reams
Robert Reicherdt
Sergei A. Romanenko
Tom Rothamel
Claus Rørbech
Elke Salecker
Lei Shang
Axel Simon
Per Stenström
Scott Stoller
Yulei Sui
Tuncay Tekle
Dirk Tetzlaff
Michael Kirkedal Thomsen
Qing Wan

Table of Contents

Invited Talk

Future-Proofing Collections: From Mutable to Persistent to Parallel <i>Martin Odersky</i>	1
--	---

JIT Compilation and Code Generation

Dynamic Elimination of Overflow Tests in a Trace Compiler <i>Rodrigo Sol, Christophe Guillon, Fernando Magno Quintão Pereira, and Mariza A.S. Bigonha</i>	2
Staged Static Techniques to Efficiently Implement Array Copy Semantics in a MATLAB JIT Compiler <i>Nurudeen Lameed and Laurie Hendren</i>	22
SSA-Based Register Allocation with PBQP <i>Sebastian Buchwald, Andreas Zwinkau, and Thomas Bersch</i>	42

Program Analysis

Probabilistic Points-to Analysis for Java <i>Qiang Sun, Jianjun Zhao, and Yuting Chen</i>	62
Faster Alias Set Analysis Using Summaries <i>Nomair A. Naeem and Ondřej Lhoták</i>	82
JPure: A Modular Purity System for Java <i>David J. Pearce</i>	104
Tainted Flow Analysis on e-SSA-Form Programs <i>Andrei Rimsa, Marcelo d'Amorim, and Fernando Magno Quintão Pereira</i>	124

Reversible Computing and Interpreters

Clean Translation of an Imperative Reversible Programming Language <i>Holger Bock Axelsen</i>	144
Interpreter Instruction Scheduling <i>Stefan Brunthaler</i>	164

Parallelism and High-Performance Computing

Actor-Based Parallel Dataflow Analysis	179
<i>Jonathan Rodriguez and Ondřej Lhoták</i>	
Using Disjoint Reachability for Parallelization	198
<i>James Jenista, Yong hun Eom, and Brian Demsky</i>	
Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures	225
<i>Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan</i>	
Subregion Analysis and Bounds Check Elimination for High Level Arrays	246
<i>Mackale Joyner, Zoran Budimlić, and Vivek Sarkar</i>	

Task and Data Distribution

Practical Loop Transformations for Tensor Contraction Expressions on Multi-level Memory Hierarchies	266
<i>Wenjing Ma, Sriram Krishnamoorthy, and Gagan Agrawal</i>	
A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL	286
<i>Dominik Grewe and Michael F.P. O’Boyle</i>	

Author Index	307
-------------------------------	------------

Future-Proofing Collections: From Mutable to Persistent to Parallel

Martin Odersky

Programming Methods Group (LAMP)
School of Computer and Communication Sciences (IC)
École Polytechnique Fédérale de Lausanne (EPFL)
Lausanne, Switzerland
`martin.odersky@epfl.ch`

Abstract. Multicore processors are on every desk now. How are we going to make use of the extra power they provide? Some think that actors, or transactional memory, or some other new concurrency construct will save the day by making concurrent programming easier and safer. Even though these are welcome, I am skeptical about their ultimate success. Concurrency is fundamentally hard and no dressing up will be able to hide that fact completely.

A safer and for the programmer much simpler alternative is to treat parallel execution as essentially an optimization. A promising application area are collections. Programming by transforming and aggregating collections is simple, powerful, and can be optimized by executing bulk operations in parallel. To be able to do this in practice, any side effects of parallel operations need to be carefully controlled. This means that immutable, persistent collections are more suitable than mutable ones.

In this talk I will describe the new Scala collections framework, and show how it allows a seamless migration from traditional mutable collections to persistent collections, and from there to parallel collections. I show how the same vocabulary of methods can be used for either type of collection, and how one can have parallel as well as sequential views on the same underlying collection.

The design of this framework is guided by the “uniform return type principle”: every collection transformer should return the same kind of collection it applies to. Simple as this sounds, it is surprisingly hard to achieve in a statically typed language with a rich type hierarchy (in fact, I know of no other collections framework that achieved it). In the talk I will explain the type-systematic constructions required by the principle. I will also present some thoughts on how we might develop type-explanation capabilities of compilers to effectively support these techniques in a user-friendly way.

Dynamic Elimination of Overflow Tests in a Trace Compiler

Rodrigo Sol¹, Christophe Guillon², Fernando Magno Quintão Pereira¹,
and Mariza A.S. Bigonha¹

¹ UFMG – 6627 Antônio Carlos Av, 31.270-010, Belo Horizonte, Brazil
{rsol,fpereira,mariza}@dcc.ufmg.br

² STMicroelectronics – 12 Jules Horowitz St, B.P. 217, 38019, Grenoble, France
christophe.guillon@st.com

Abstract. Trace compilation is a technique used by just-in-time (JIT) compilers such as TraceMonkey, the JavaScript engine in the Mozilla Firefox browser. Contrary to traditional JIT machines, a trace compiler works on only part of the source program, normally a linear path inside a heavily executed loop. Because the trace is compiled during the interpretation of the source program the JIT compiler has access to runtime values. This observation gives the compiler the possibility of producing binary code specialized to these values. In this paper we explore such opportunity to provide an analysis that removes unnecessary overflow tests from JavaScript programs. Our optimization uses range analysis to show that some operations cannot produce overflows. The analysis is linear in size and space on the number of instructions present in the input trace, and it is more effective than traditional range analyses, because we have access to values known only at execution time. We have implemented our analysis on top of Firefox’s TraceMonkey, and have tested it on over 1000 scripts from several industrial strength benchmarks, including the scripts present in the top 100 most visited webpages in the Alexa index. We generate binaries to either x86 or the embedded microprocessor ST40-300. On the average, we eliminate 91.82% of the overflows in the programs present in the TraceMonkey test suite. This optimization provides an average code size reduction of 8.83% on ST40 and 6.63% on x86. Our optimization increases TraceMonkey’s runtime by 2.53%.

1 Introduction

JavaScript is the most popular programming language used in the client-side of web applications [12]. Supporting this statement is the fact that JavaScript is used in 97 out of the 100 most popular websites in the alexa 2010 report [1]. Thus, it is very important that JavaScript programs benefit from efficient execution environments. Web browsers normally interpret programs written in this language. However, to achieve execution efficiency, JavaScript programs can be

¹ <http://www.alexa.com>

compiled during interpretation – a process called just-in-time (JIT) compilation. There are many ways to perform JIT compilation. *TraceMonkey* [14], the Mozilla Firefox 3.1’s JIT compiler, translates the most executed *program traces* into machine code. A program trace is a linear sequence of code representing a path inside the program’s control flow graph.

The Firefox JIT compiler, among many optimizations, does type specialization on JavaScript programs. For instance, JavaScript sees numbers as double precision 64-bit values in the IEEE 754 format [8, p.29]; however, TraceMonkey tries to manipulate them as integer values every time it is possible. Dealing with integers is much faster than handling floating-point arithmetics, but it is necessary to ensure that this optimization does not change the semantics of the program. So, every time an operation produces a result that might exceed the precision of the integer type, it is necessary to perform an overflow test. In case the test fails, float-point numbers must replace the original integer operands.

Overflow tests are pervasive in the machine code produced by TraceMonkey, a performance nuisance already acknowledged by the Mozilla community². Overflow tests impose two major problems. First, because each test may force an early exit from tracing mode, these tests complicate optimizations that require code motion, such as partial redundancy elimination and instruction scheduling. Second, instructions that handle overflows increase code size: about 12.3% of the x86 code produced by TraceMonkey per script in this compiler’s test suite implement overflow tests. This extra code is a complication on small devices that run JavaScript, such as the ST family of microcontrollers³. We have attacked this issue via a flow sensitive range analysis that proves that some overflow tests are redundant, and we present the results of this work in this paper. Our analysis runs in linear time on the number of instructions in the input trace. This analysis is implemented on top of TraceMonkey; however, it does not depend on a particular compiler. On the contrary, it works on any JIT engine that uses the trace paradigm to do code generation.

Our algorithm does range analysis [17,20], i.e, it tries to estimate the lowest and greatest values that can be assigned to any variable. However, our approach differs from previous works because we use values known only at runtime in order to put bounds on the range of values that integer variables might assume during the program execution. This is a form of *partial evaluation* [18], yet done at runtime [4], because our analysis is invoked by a just-in-time compiler while the target application is being interpreted. By relying on such values we are able to perform much more aggressive range inferences than traditional static analyses. In terms of implementation, our analysis is similar to the ABCD algorithm that eliminates array bound-checks [3]; however, there are important differences. First, our algorithm is simpler, for it runs on a program trace, which is straight-line code. Because it runs on a trace, our algorithm is linear on the number of program variables, and not quadratic, as other analyses that keep def-use chains of variables. Second, we perform the whole analysis at once, in a

² https://bugzilla.mozilla.org/show_bug.cgi?id=536641

³ <http://www.st.com/mcu/familiesdocs-51.html>

single traversal of the input trace, whereas ABCD runs on demand. In addition to these differences, the fact that we use the runtime value of variables lets us be less conservative.

We have implemented our analysis on top of TraceMonkey (release from 2010-10-3), the JIT compiler used by the Firefox web browser, and we have targeted two different processors, x86 and ST40. We have correctly compiled and run over one million lines of JavaScript code taken from a vast collection of benchmarks, including the TraceMonkey’s test suite, and the top 100 webpages according to the Alexa index. Currently our implementation can only remove overflow tests from arithmetic operations performed on variables declared locally, i.e, we do not handle global variables yet. Nevertheless, we recognize 59% of the overflow tests in the TraceMonkey test suite, which contains over 800 scripts. On average, our algorithm eliminates 91.82% of the overflow tests that we recognize in these scripts, providing an average code size reduction of 6.63% on x86, and 8.83% on ST40. Our research-quality implementation adds 2.53% of time overhead on the core TraceMonkey implementation (time to compile and run the script) without other optimizations enabled. However, we speculate that an industrial-strength implementation of our algorithm will be able to obtain runtime gains. We have instrumented the Firefox browser, to check the behavior of our algorithm in actual webpages: on the average we remove 53.5% of the overflow tests per webpage. This number is lower when compared to the TraceMonkey’s test suite because there are more global variables in actual webpages.

The remainder of this paper is organized as follows. Section 2 describes the TraceMonkey JIT compiler. In that section we show, by means of a simple example, how a trace is produced and represented. We describe our analysis in Section 3. Section 4 provides some experimental data that shows that our analysis is effective. We discuss related work in Section 5. Finally, Section 6 concludes this paper.

2 TraceMonkey in a Nutshell

There exists a number of recent works describing the implementation of trace-based JIT compilers, such as Tamarim-trace [6], HotpathVM [15], Yeti [27] and TraceMonkey [14]. In order to explain this new compilation paradigm, in this section we describe the TraceMonkey implementation. TraceMonkey has been built on top of *SpiderMonkey*, the original JavaScript interpreter used by the Firefox Browser. To produce x86 machine code from JavaScript sources, TraceMonkey uses the *Nanojit*⁴ compiler. The whole compilation process goes through three intermediate representations, a path that we reproduce in Figure 1.

1. **AST**: the abstract syntax tree that the parser produces from a script file.
2. **Bytecodes**: a stack based instruction set that SpiderMonkey interprets.
3. **LIR**: the low-level three-address code instruction representation that Nanojit receives as input.

⁴ <https://developer.mozilla.org/en/Nanojit>

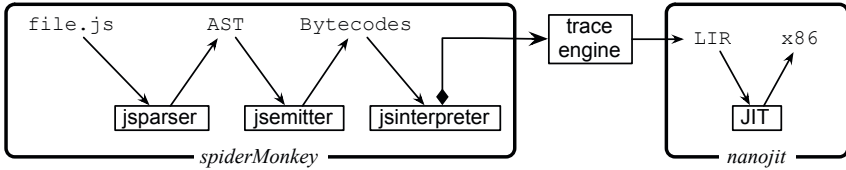


Fig. 1. The TraceMonkey JavaScript JIT compiler

SpiderMonkey has not been originally conceived as a just-in-time compiler, a fact that explains the seemingly excessive number of intermediate steps between the source program and the machine code. Segments of LIR instructions – a *trace* in TraceMonkey’s jargon – are produced according to a very simple algorithm [14]:

1. each conditional branch is associated to a counter initially set to zero.
2. If the interpreter finds a conditional branch during program interpretation, then it increments the counter. The process of checking and incrementing counters is called, in TraceMonkey’s jargon, the *monitoring phase*.
3. If the counter is two or more, and no trace exists for that counter, then the trace engine starts translating the bytecodes to a segment of LIR instructions, while they are interpreted. Overflow tests are inserted into this LIR segment. The process of building the trace is called *recording phase*.
4. Once the trace engine finds the original branch that started the recording process, the current segment is passed to Nanojit.
5. The Nanojit compiler translates the LIR segment, including the overflow tests, into machine code, which is dumped into main memory. The program flow is diverted to this code, and direct machine execution starts.
6. After the machine code runs, or in case an exceptional condition happens, e.g, an overflow test fails or a branch leaves the trace, the flow of execution goes back to the interpreter.

From bytecodes to LIR: we use the program in Figure 2 (a) to illustrate the process of trace compilation, and also to show how our analysis works. This is an artificial program, clearly too naive to find use in the real world; however, it contains the subtleties necessary to put some strain on the cheap analysis that must be used in the context of a just-in-time compiler. This program would yield the bytecode representation illustrated in Figure 2 (b). Notice that we took the liberty of simplifying the bytecode intermediate language used by TraceMonkey.

A key motivation behind the design of the analysis that we present in Section 3 is the fact that TraceMonkey might produce traces for program paths while these paths are visited for the first time. This fact is a consequence of the algorithm that TraceMonkey uses to identify traces. At the beginning of the interpretation

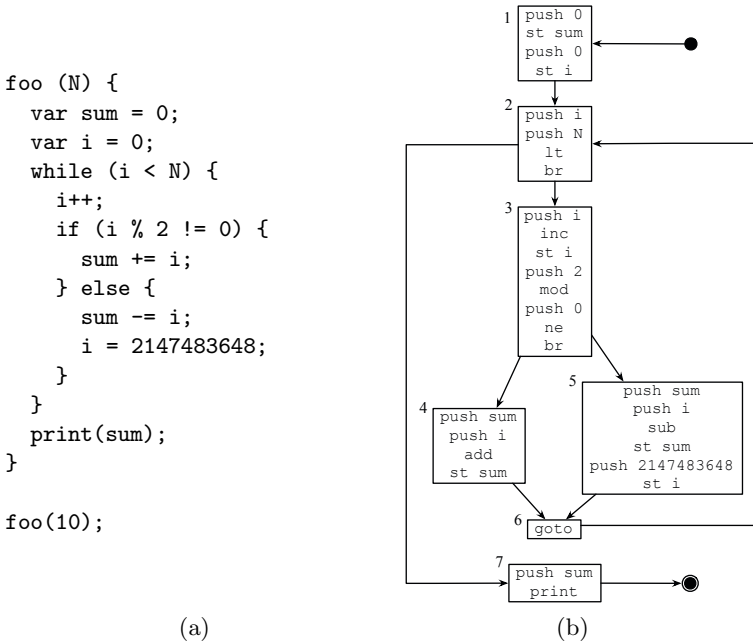


Fig. 2. Example of a small JavaScript program and its bytecode representation

process TraceMonkey finds the branch at Basic Block 2 in Figure 2 (b), and the trace engine increments the counter associated to that branch. The next branch will be found at the end of Basic Block 3, and this branch’s counter will be also incremented. The interpreter then will find the `goto` instruction at the end of Basic Block 6, which will take the program flow back to Block 2. At this moment the trace engine will increment again the counter of the branch in that basic block. Once the trace engine finds a counter holding the value two, it knows that it is inside a loop, and starts the recording phase, which produces a LIR segment. However, this segment does not correspond to the first part of the program visited: in the second iteration of the loop, Basic Block 5 is visited instead of Basic Block 4. In this case, the segment that is recorded is formed by Basic Blocks 2, 3, 5 and 6.

Because the trace that is monitored by the trace engine is not necessarily the trace that is recorded into a LIR segment, it is difficult to remove overflow tests during the recording phase. A conditional test, such as $a < N$, where N is constant, helps us to put bounds on the range of values that a might assume. However, in order to know that N is constant, we must ensure that the entire recorded segment does not contain commands that change N ’s value. Although it is not possible to remove overflow tests directly during the recording phase, it is possible to collect constraints on the ranges of the variables in this step. These constraints will be subsequently used to remove overflow tests at the

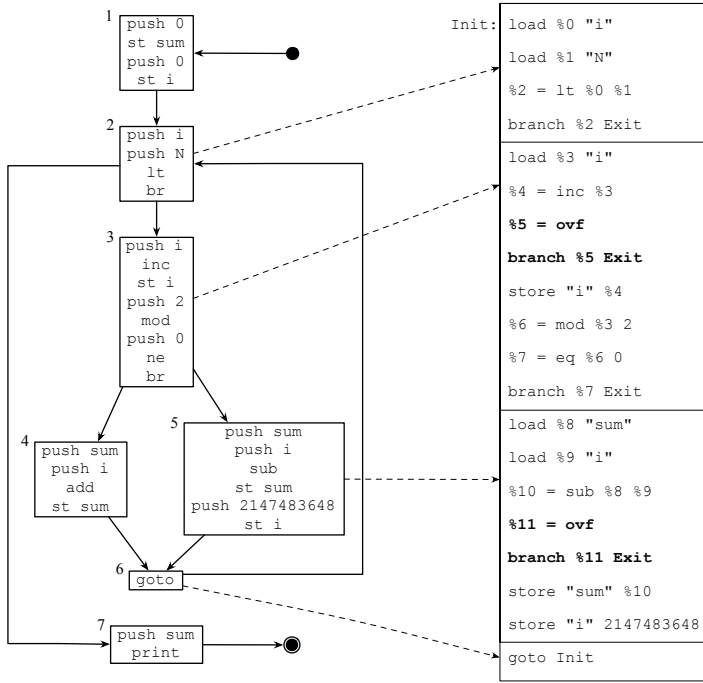


Fig. 3. This figure illustrates the match between the recorded trace and the LIR segment that the trace engine produces for it

Nanojit level. Thus, we perform the elimination of overflow tests once the whole LIR segment has been produced, right before it is passed to Nanojit. Continuing with our example, Figure 3 shows the match between the recorded trace and the LIR segment that the trace engine produces for it.

From LIR to x86: Before passing the LIR segment to Nanojit, the interpreter augments this segment with two sequences of instructions: the prologue and the epilogue. The prologue contains code to map the values from the interpreter’s execution environment to the execution environment where the compiled binary will run. The epilogue contains code to perform the inverse mapping. Nanojit produces machine code, e.g x86, ARM, ST40, from the LIR segment that it receives from the interpreter. The overflow tests are implemented as *side-exit* code: a sequence of instructions that has two functions: (i) it recovers the state of the program right before the overflow took place, and (ii) it jumps to the exit of the trace, returning control back to the interpreter. Figure 4 shows a simplified version of the x86 code that would be produced for our example trace. Notice that our analysis has no role at this point of the compilation process – we include it in this discussion only to give the reader a full picture of the trace compiler.

(LIR)		(x86)	
Prologue:	Init:	Prologue:	RS1:
load %11 ITP[i]	load %0 "i"	pushl ...	movl ...
store "i" %11	load %1 "N"	movl ...	movl ...
load %12 ITP[N]	%2 = lt %0 %1	movl ...	movl ...
store "N" %12	branch %2 Exit	movl ...	movl ...
load %13 ITP[sum]	load %3 "i"	jmp Init	jmp Exit
store "sum" %13	%4 = inc %3	Init:	RS2:
goto Init	%5 = ovf	leal ...	movl ...
	branch %5 Exit	incl ...	movl ...
Exit:	store "i" %4	jvf RS1	movl ...
load %14 "i"	%6 = mod %3 2	movl ...	movl ...
store ITP[i] %11	%7 = eq %6 0	andl ...	movl ...
load %15 "N"	branch %7 Exit	testb ...	jmp Exit
store ITP[N] %12	load %8 "sum"	je Exit	
load %16 "sum"	load %9 "i"	movl ...	
store ITP[sum] %13	%10 = sub %8 %9	leal ...	
return	%11 = ovf	subl ...	
	branch %11 Exit	jvf RS2	
	store "sum" %10	movl ...	
	store "i" 2147483648	movl ...	
	goto Init	cmpl ...	
		jb Init	
		Exit:	
		movl ...	
		movl ...	
		movl ...	
		popl ...	
		leave	

Fig. 4. The LIR code, after the insertion of a prologue and an epilogue, and the schematic x86 trace produced by Nanojit

3 Flow Sensitive Range Analysis

The flow sensitive range analysis that we use to remove overflow tests relies on a directed acyclic graph to determine the ranges of the variables. This graph, henceforth called *constraint graph*, has four types of nodes:

Name: represent program variables. Each node contains a counter, which represents a particular definition of a variable. These nodes are bound to an integer interval, which represents the range of values that the definition they encode might assume.

Assignment: denoted by `mov`, represent the copy of a value to a variable.

Relational: represent comparison operations, which are used to put bounds on the ranges of the variables. The comparison operations considered are: equals (`eq`), less than (`lt`), greater than (`gt`), less than or equals (`le`), and greater than or equals (`ge`).

Arithmetic: represent operations that might require an overflow test. We have two types of arithmetic nodes: binary and unary. The binary operations are addition (`add`), subtraction (`sub`), and multiplication (`mul`). The unary operations are increment (`inc`) and decrement (`dec`). Division is not handled because it might legitimately produce floating-point results.

The analysis proceeds in two phases: construction of the constraint graph and range propagation. The remaining of this section describes these phases.

3.1 Construction of the Constraint Graph

We build the constraint graph during the trace recording phase of TraceMonkey, that is, while the instructions in the trace are being visited and a LIR segment is being produced. In order to associate range constraints to each variable in the source program we use a program representation called *Extended Static Single Assignment* (e-SSA) [3] form, which is a superset of the well known SSA form [9]. In the e-SSA representation, a variable is renamed after it is assigned a value, or after it is used in a conditional.

Converting a program to e-SSA form requires a global view of the program, a requirement that a trace compiler cannot fulfill. However, given that we are compiling a program trace, that is, a straight line segment of code, the conversion is very easy, and happens at the same time that the constraint graph is built, e.g, during the trace recording step. The conversion works as follows: counters are maintained for every variable. Whenever we find a use of a variable v we rename it to v_n , where n is the current value of the counter associated to v . Whenever we find a definition of a variable we increment its counter. Considering that the variables are named after their counters, incrementing the counter of a variable effectively creates a new name definition in our representation. So far our renaming is just converting the source program into Static Single Assignment form [9]. The e-SSA property comes from the way that we handle conditionals. Whenever we find a conditional, e.g, $a < b$, we learn new information about the ranges of a and b . Thus, we redefine a and b , by incrementing their counters.

There are two events that change the bounds of a variable: *simple assignments* and *conditional tests*. The first event determines a unique value for the variable. The second puts a bound in one of the variable's limits, lower or upper. These are the events that cause us to increment the counters associated to variables. Thus, it is possible to assign unique range constraints to each new definition of a variable. These range constraints take into consideration the current value of the variables at the time the variable is found by the trace engine. We determine these values by inspecting the interpreter's stack. We have designed the following algorithm to build the constraint graph:

1. initialize counters for every variable in the trace. We do this initialization on the fly: the first time a variable name is seen we set its counter to zero, otherwise we increment its current counter. If we see a variable for the first time, then we mark it as *input*, otherwise we mark it as *auxiliary*. If a variable is marked as *input*, then we set its upper and lower limits to the value that the interpreter currently holds for it. Otherwise, we set the variable boundaries to undefined values, e.g, $]-\infty, +\infty[$.
2. For each instruction i that we visit:
 - (a) if i is a relational operation, say $v < u$, we build a relational node that has two predecessors: variable nodes v_x and u_y , where x and y are counters. This node has two successors, v_{x+1} and u_{y+1} .

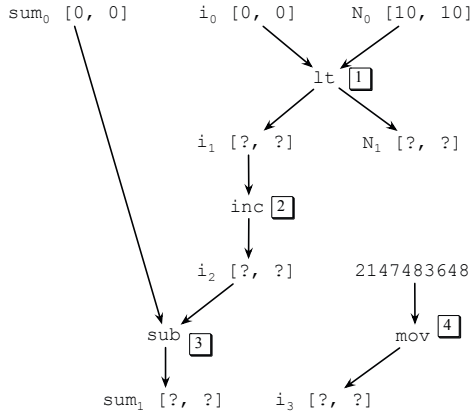


Fig. 5. Constraint graph for the trace in Figure 3. The numbers in boxes denote the order in which the nodes were created.

- (b) For each binary arithmetic operation, e.g, $v = t + u$, we build an arithmetic node n . Let the nodes related to variables t_x and u_y be the predecessors of n , and let v_z be its successor.
- (c) For each unary operation, say $u++$, we build a node n , with one predecessor u_x , and one successor u_{x+1} .
- (d) For each copy assignment, e.g, $v = u$, we build an assignment node, which has predecessor u_x , and successor v_{y+1} , assuming y is v 's counter.

Figure 5 shows the constraint graph to our running example, assuming that the function `foo` was called with the parameter $N = 10$. Notice that we bound the input variables to intervals: $sum_0 \subseteq [0, 0]$, $i_0 \subseteq [0, 0]$ and $N_0 \subseteq [10, 10]$. When constructing the constraint graph, it is important to maintain a list with the order in which each node was created. This list, which we represent by the numbers in boxes, will be later used to guide the range propagation phase.

3.2 Range Propagation

During the range propagation phase we find which overflow tests are necessary, and which ones can be safely removed from the target code. The propagation of ranges is preceded by a trivial initialization step, when we replace the constraints of the input variables with $]-\infty, +\infty[$ if those variables have been updated inside the trace. This is the case, for example, of variables `sum0` and `i0` in the example from Figure 5. In order to do the propagation of range intervals, we visit all the arithmetic and relational nodes, in topological order. This ordering is given by the “age” of the node. Nodes that have been created earlier, during the construction of the constraint graph are visited first. Notice that we get this ordering for free, simply storing the arithmetic and relational nodes in a queue while we create them, during the construction of the constraint graph. If every variable is defined before being used, then by following the node creation order,

Arithmetics	
$x + (+\infty) = +\infty + x = +\infty, x \neq -\infty$	$x + (-\infty) = -\infty + x = -\infty, x \neq +\infty$
$x \times (\pm\infty) = \pm\infty \times x = \pm\infty, x > 0$	$x \times (\pm\infty) = \pm\infty \times x = \mp\infty, x < 0$
Increment: $x_1 = x_0 + 1$	
$\frac{x_0.l + 1 > \text{MAX_INT}}{x_1.l = +\infty}$	$\frac{x_0.u + 1 > \text{MAX_INT}}{x_1.u = +\infty}$
$\frac{x_0.l + 1 \leq \text{MAX_INT}}{x_1.l = x_0.l + 1}$	$\frac{x_0.u + 1 \leq \text{MAX_INT}}{x_1.u = x_0.u + 1}$
Addition: $x = a + b$	
$\frac{a.l + b.l < \text{MIN_INT}}{x.l = -\infty}$	$\frac{a.u + b.u < \text{MIN_INT}}{x.u = -\infty}$
$\frac{a.l + b.l > \text{MAX_INT}}{x.l = +\infty}$	$\frac{a.u + b.u > \text{MAX_INT}}{x.u = +\infty}$
$\frac{\text{MIN_INT} \leq a.l + b.l \leq \text{MAX_INT}}{x.l = a.l + b.l}$	$\frac{\text{MIN_INT} \leq a.u + b.u \leq \text{MAX_INT}}{x.u = a.u + b.u}$
Multiplications: $x = a \times b$	
$\frac{a.l \times b.l < \text{MIN_INT}}{x.l = -\infty}$	$\frac{a.u \times b.u < \text{MIN_INT}}{x.u = -\infty}$
$\frac{a.l \times b.l > \text{MAX_INT}}{x.l = +\infty}$	$\frac{a.u \times b.u > \text{MAX_INT}}{x.u = +\infty}$
$\frac{\text{MIN_INT} \leq a.l \times b.l \leq \text{MAX_INT}}{x.l = \text{MIN}(a.\{l, u\} \times b.\{l, u\})}$	$\frac{\text{MIN_INT} \leq a.u \times b.u \leq \text{MAX_INT}}{x.u = \text{MAX}(a.\{l, u\} \times b.\{l, u\})}$

Fig. 6. Range propagation for arithmetic nodes. Decrements and subtractions are similar to increments and additions. We use $a.\{l, u\} \times b.\{l, u\}$ as a short form for $(a.l \times b.l, a.l \times b.u, a.u \times b.l, a.u \times b.u)$.

the propagation of ranges guarantees that whenever we reach an arithmetic, relational or assignment node, all the name nodes that point to it have been visited before.

Each arithmetic and relational node causes the propagation of ranges in a particular way, always preserving the invariant that the lower bound of an interval is less than or equal its upper bound. Figure 6 shows the updating rules that we use for some arithmetic nodes. We use standard IEEE extended arithmetics [16], except that, to improve our analysis, we assume that the result of multiplying infinity and zero is zero. Furthermore, we let the sum of $]-\infty, x[$ plus $] +\infty, +\infty[$ to be $] -\infty, +\infty[$, and vice-versa. We denote the interval $[l, u]$ associated to a

Less than with constant: $(a_1, N_1) \leftarrow (a < N)?$, when $a.l \leq N.l = N.u$	
$a_1.u = \text{MIN}(a.u, N.u - 1)$	$N_1.l = N.l$
$a_1.l = a.l$	$N_1.u = N.u$
General less than: $(a_1, b_1) \leftarrow (a < b)?$, when $[a.l, a.u] \cap [b.l, b.u] \neq \emptyset$	
$a_1.u = \text{MIN}(a.u, b.u - 1)$	$b_1.l = \text{MAX}(b.l, a.l + 1)$
$a_1.l = a.l$	$b_1.u = b.u$
Equal to constant: $(a_1, N_1) \leftarrow (a = N)?$, when $a.l \leq N.l = N.u \leq a.u$	
$a_1.l = N.l$	$N_1.l = N.l$
$a_1.u = N.u$	$N_1.u = N.u$
General equals: $(a_1, b_1) \leftarrow (a = b)?$, when $[a.l, a.u] \cap [b.l, b.u] \neq \emptyset$	
$a_1.l = \text{MAX}(a.l, b.l)$	$b_1.l = \text{MAX}(a.l, b.l)$
$a_1.u = \text{MIN}(a.u, b.u)$	$b_1.u = \text{MIN}(a.u, b.u)$

Fig. 7. Range propagation for two relational nodes. The value of variable N is not updated inside the trace.

node a by $a.l$ and $a.u$. Although we state the range propagation rules using infinity precision arithmetics, our actual implementation uses wrapping semantics, i.e: if $x + 1 > x$, then $x = +\infty$. Only arithmetic nodes might cause overflows; and each of our five types of arithmetic nodes might produce overflows in different ways. For instance, if we find an increment of $[-\infty, v.u]$, then we keep the overflow associated to this node, as long as $v.u + 1 \leq \text{MAX_INT}$. Figure 7 shows the updating rules for some relational nodes. These nodes are not associated to any overflow test, but they help us to constrain the range of intervals bound to program variables. As an optimization, we do not update the ranges of variables that are not defined inside the trace. This is the case, for instance, of Variable N in the program of Figure 2 (a). Notice that we do not perform range updates that break the invariant $v.l \leq v.u$.

After range propagation we have a conservative estimate of the intervals that each integer variable might assume during program execution. This information allows us to go over the LIR segment, before it is passed to Nanojit, removing the overflow tests that our analysis has deemed unnecessary. Figure 8 shows this step: we have removed the overflow test from the `inc` operation, because it receives as input a node bound to the interval $]-\infty, 9]$; hence, its result will never be greater than 10. However, our analysis cannot prove that the test in the `sub` operation is also unnecessary, although that is the case.

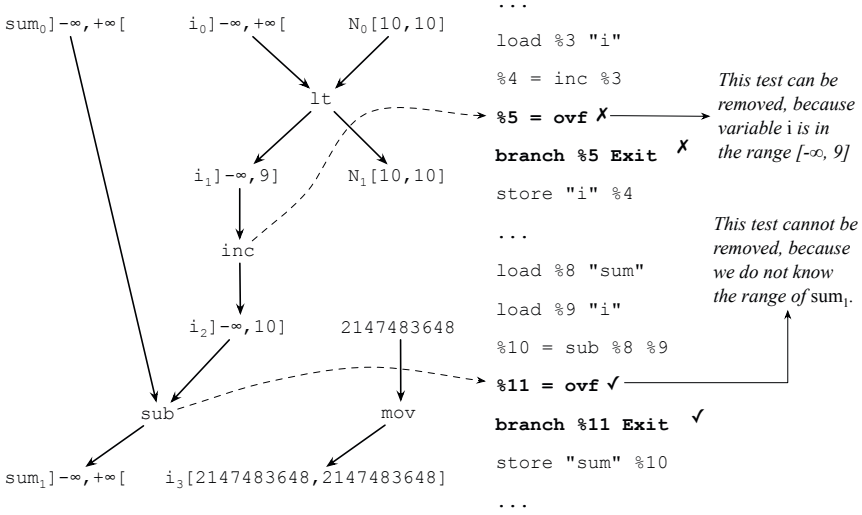


Fig. 8. Once we know the ranges of each variable involved in an arithmetic operation we can remove the associated overflow test, if it is redundant

3.3 Complexity Analysis

The proposed algorithm has running time linear on the number of instructions of the source trace. To see this fact, notice that the constraint graph has a number of conditional and arithmetic nodes proportional to the number of instructions in the trace. During our analysis we traverse the constraint graph one time, at the range propagation phase, visiting each node only once. We do not have to preprocess the graph beforehand, in order to sort it topologically, because we get this ordering from the sequence in which instructions are visited in the source trace. Our algorithm is also linear in terms of space, because each type of arithmetic node has a constant number of predecessors and successors. This low complexity is in contrast to the complexity of many graph traversal algorithms, which are $O(E)$, where E is the number of edges in the graph. These algorithms have a worst case quadratic complexity on the number of vertices, because $E = O(V^2)$. We do not suffer this drawback, for in our case $E = O(V)$.

4 Experimental Results

We have implemented our algorithm on top of TraceMonkey, revision 2010-10-01. Our implementation handles the five arithmetic operations described in Section 3.1, namely additions, subtractions, increments, decrements and multiplications. We perform the range analysis described in Section 3 during the recording phase of TraceMonkey, that is, while a segment of JavaScript bytecodes is translated to a segment of LIR. We have also modified Nanojit to remove the overflow tests, given the results of our analysis. Our current implementation

has some shortcomings, which are all due to our limited understanding of TraceMonkey’s implementation, and that we are in the process of overcoming:

- we cannot read values stored in global variables, a fact that hinders us from removing overflows related to operations that manipulate these values.
- We cannot recognize when TraceMonkey starts tracing constructs such as `foreach{...}`, `while(true){...}` and loops that range on iterators.

The benchmarks. We have correctly compiled and executed over one million lines of JavaScript code that we took from three different benchmark suites:

Alexa top 100: the 100 most visited webpages, according to the Alexa index⁵. This list includes **Google**, **Facebook**, **Youtube**, etc. We tried to follow Richards *et al.*’s methodology [12], manually visiting each of these pages with our instrumented Firefox. Notice that we present results for only 80 of these benchmarks, because we did not find overflow tests in 20 webpages.

Trace-Test: the test suite that is used in TraceMonkey⁶. This collection of scripts includes popular benchmarks, such as Webkit’s Sunspider and Google’s V8. Many scripts do not contain arithmetic operations; thus, we show results only for the 224 scripts that contains at least one overflow test.

PeaceKeeper: an industrial strength benchmark used to test browsers⁷.

The hardware. We have used our modified TraceMonkey compiler in two different hardware: (i) x86: 2GHz Intel Core 2 Duo, with 2GB of RAM, featuring Mac OS 10.5.8. (ii) ST40-300: a real-time processor manufactured by STMicroelectronics. The ST40 runs STLinux 2.3 (kernel 2.6.32), has a 450MHz clock, provides 200MB of physical memory and 512MB of virtual memory. This target is a two level cache architecture with separated 32KB instruction and 32KB data L1 caches on top of a 256KB unified L2 cache.

How effective is our algorithm? Figure 9 shows the effectiveness of our algorithm when we run it on the Alexa webpages and the Trace-Test scripts. We have ordered the scripts on the x-axis according to the effectiveness of our algorithm. We present static numbers, in the sense that we have not instrumented the binary traces to see how many times each overflow test is executed. On the average, we remove 91.82% of the overflows tests from the scripts in Trace-Test, and 53.50% of the overflow tests from the Alexa webpages. This is the geometric mean; that is, we are very effective in removing overflow tests from most of the scripts. In particular, we remove most of the overflow tests used in counters that index simple loops. However, the arithmetic mean is much lower, because of the outliers. Three of these scripts, which are part of SunSpider (included in Trace-Test) deal with cryptography and manipulate big numbers. They account for 8,756 tests, out of which we removed only 347. In absolute terms we remove 961

⁵ <http://www.alexa.com/>

⁶ <http://hg.mozilla.org/tracemonkey/file/c3bd2594777a/js/src/trace-test/tests>

⁷ <http://service.futuremark.com/peacekeeper/index.action>

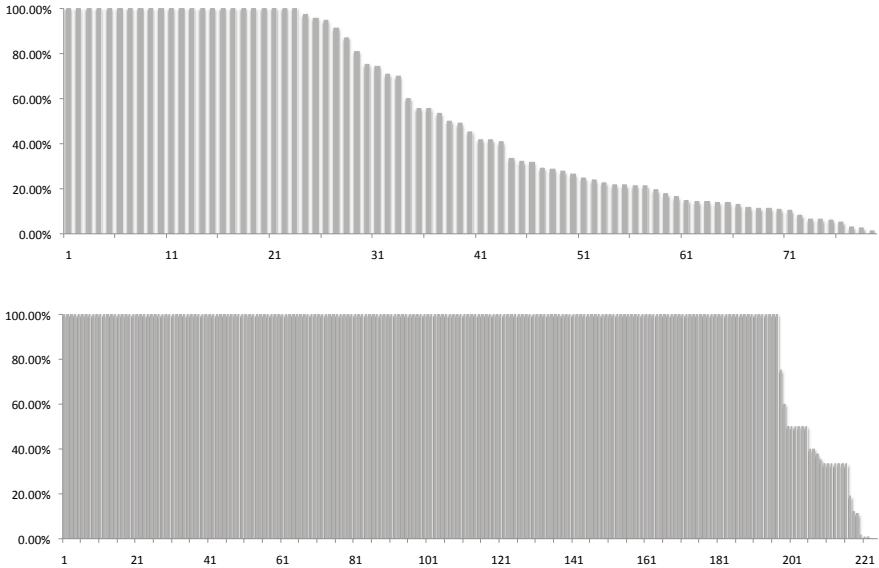


Fig. 9. The effectiveness of our algorithm in terms of percentage of overflow tests removed per script. (Top) Alexa; geo mean: 53.50%. (Bottom) Trace-test; geo mean: 91.82%. Hardware: same results for ST40 and x86. The 224 scripts are sorted by average effectiveness. We removed 700/859 overflow tests for PeaceKeeper (one script).

out of 11,052 tests in the Trace-Test benchmark, and 2,680 out of 11,126 tests in the Alexa benchmark. The arithmetic mean, in this case, is 8.7% for Trace-Test, and 24.08% for Alexa. Figure 9 does not contain a chart for PeaceKeeper, because this benchmark contains only one script. We removed 700 tests out of the 859 tests that we found in this script.

What is the code size reduction due to the elimination of overflow tests? On the x86 target, TraceMonkey uses eight instructions to implement each overflow test, which includes the branch-if-overflow instruction, load instructions to reconstruct the interpreter state and an unconditional jump back to interpreter mode. On the ST40 microprocessor, TraceMonkey uses 10 instructions. We eliminate all these instructions after removing an overflow test. Figure 10 shows the average size reduction that our algorithm obtains on each target. On the average, we shrink each script by 8.83% on the ST40, and 6.63% on the x86. On most of the scripts the size reduction is modest; however, there are cases in which our algorithm decreases the binaries by over 60.0%. Notice that by using integers instead of floating-point numbers TraceMonkey already reduces the size of the binaries that it produces. For instance, on the ST40, TraceMonkey obtains an average 31.47% of size reduction with the floating-point to integer optimization.

What is the effect of our algorithm on TraceMonkey’s runtime? Figure 11 shows that, on the average, our algorithm has increased the runtime of

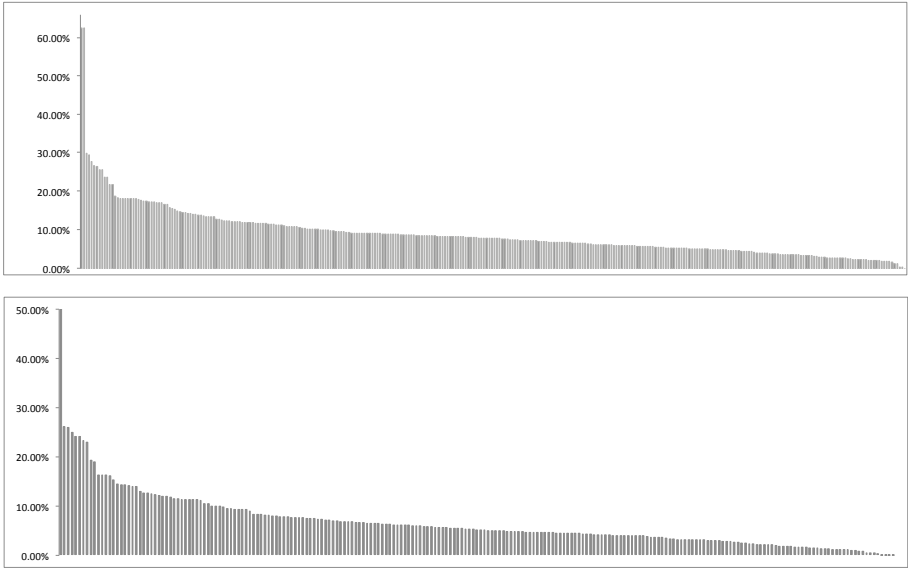


Fig. 10. Size reduction per script in two different architectures. (Top) ST40; geo mean: 8.83%. (Bottom) x86; geo mean: 6.63% Benchmark: Trace-Test. The 224 scripts are sorted by average size reduction.

TraceMonkey by 2.56% on the x86. This time includes parsing, interpreting, JIT compiling and executing the scripts. Our algorithm increases the time of JIT compilation, but decreases the time of script execution. So far the cost, in time, is negative for two reasons: a prototype implementation and a low benefit optimization. Figure 11 shows that running our analysis without disabling overflows tests increases TraceMonkey’s runtime by 4.12%. Furthermore, the effects of avoiding the overflows tests are negligible. Most of the time, the impact of an overflow test is the cost of executing the x86 instruction *branch-if-overflow*; however, this instruction is normally predicted as not taken. If we disable every overflow test without running the analysis, then we improve the runtime by 1.56% on the scripts that finish correctly. The noise in the chart is due to short runtimes – the slowest script executes for 0.73s, and the fastest for 0.01s.

Why sometimes we fail to remove overflows? We have performed a manual study on the 42 smallest programs from Trace-Test in which we did not remove overflow tests. Each of these tests contains a loop indexed by an induction variable. Figure 12 explains our findings. In 69% of the traces, we failed to remove the test because the induction variable was bounded by a global variable. As we explained before, our implementation is not able to identify the values of global variables; hence, we cannot use them in the estimation of ranges. In 17% of the remaining cases the trace is produced after a **foreach** control structure, which our current implementation fails to recognize. Once we fix these omissions, we will be able to remove at least one more overflow test in 36 out of these 42

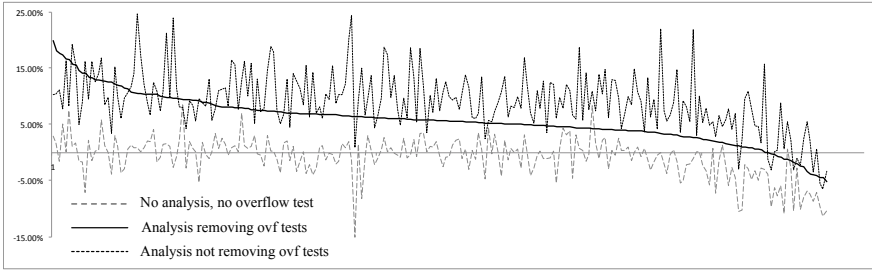


Fig. 11. Impact of our algorithm on TraceMonkey’s runtime. Benchmark: Trace-Test. Hardware: x86. The 224 scripts are sorted by average runtime increase. We run each test 8 times, ignoring smallest and largest outliers. Zero is TraceMonkey’s runtime without our analysis and with overflow tests enabled.

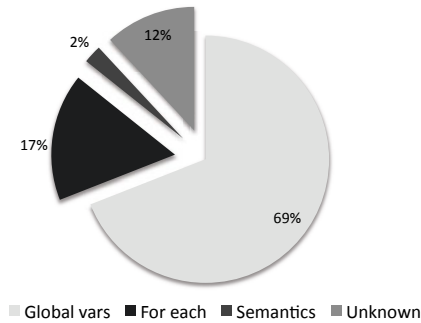


Fig. 12. The main reasons that prevent us from removing overflow tests

scripts analyzed. There was only one script in which our algorithm legitimately failed to remove a test: in this case the semantics of the program might lead to a situation in which an overflow, indeed, happens.

What we gain by knowing the runtime value of variables? We perform more aggressive range estimations than previous range analyses described in the literature, because we are running alongside a JIT compiler, a fact that gives us the runtime value of variables, including loop limits. Statically we can only rely on constants to start placing bounds in variable ranges. Non-surprisingly, a static implementation of our algorithm removes only 472 overflow tests from the Trace-Test, which corresponds to 37% of the tests that we remove dynamically.

Why are effectiveness results so different for Trace-Test and Alexa? Figure 9 shows that our algorithm is substantially more effective when applied on the programs in Trace-Test than on the programs in the Alexa top 100 pages. This happens because most of the scripts in the Trace-Test benchmark suite are small, and contain only simple loops controlled by locally declared variables, which our algorithm can recognize. In order to compare the two test suites, we have used our instrumented Firefox to produce the histogram of assembly

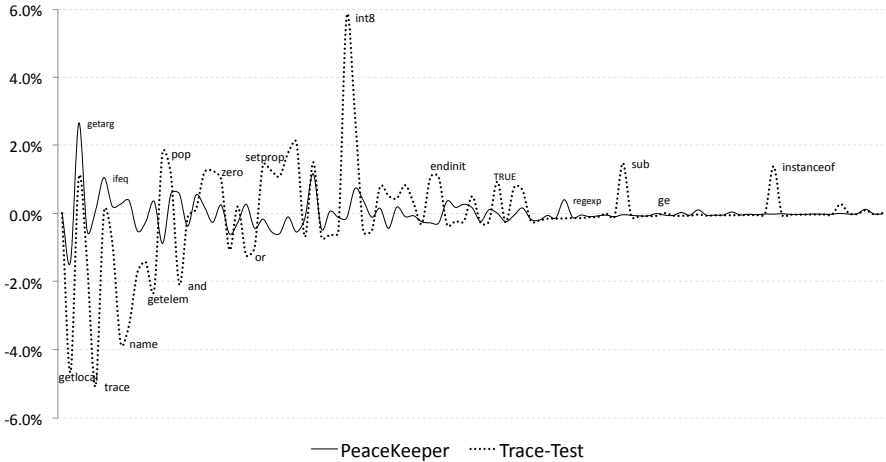


Fig. 13. A histogram of the bytecodes present in Trace-Test and PeaceKeeper, compared to the bytecodes present in the Alexa collection. X-axis: instructions i ordered by their frequency in the Alexa collection.

instructions that TraceMonkey generates for each benchmark. Figure 13 shows the results of our findings. We have produced this chart by plotting pairs $(i, f(i))$, which we define as follows:

1. let I be the list formed by the 100 most common SpiderMonkey bytecodes found in the Alexa benchmark. We order the bytecodes i according to $a(i)$, the frequency of occurrences of i in the Alexa collection.
2. For each bytecode $i \in I$ and benchmark $B \in \{\text{Trace-Test}, \text{PeaceKeeper}\}$, we let $b(i)$ be i 's frequency in B , and we let $f(i) = a(i) - b(i)$.

The closer to zero the values of $f(i)$, more similar is the frequency of i in the Alexa collection and in the other benchmark. Using this rough criterion, we see that PeaceKeeper is closer to Alexa than Trace-Test. Summing up the absolute values of $f(i)$, we find 0.24 for PeaceKeeper and 0.75 for Trace-Test. Coincidentally, our analysis is more effective on Trace-Test than on PeaceKeeper.

5 Related Work

Just-in-time compilers are old allies of programming language interpreters. Since the seminal work of John McCarthy [19], the father of Lisp, a multitude of JIT compilers have been designed and implemented. A comprehensive survey on just-in-time compilation is given by John Aycock [1].

The optimization that we propose in this paper is a type of partial evaluation at runtime. Partial evaluation is a technique in which a compiler optimizes a program, given a partial knowledge of its input [18]. Variations of partial evaluation have been used to perform general code optimization [22, 23]. This type of partial evaluation in which the JIT compiler uses runtime values to produce better

binaries is sometimes called *specialization by need* [21]. Implementations that use this kind of technique include Python’s Pyco JIT compiler [21], Matlab [7,10] and Maple [4]. Partial evaluation has been used in the context of just-in-time compilation mostly as a form of type specialization [7,5,14]. That is, once the compiler proves that a value belongs into a certain type, it uses this type directly, instead of resorting to costly boxing and unboxing techniques.

The competition between popular browsers has brought renewed attention to trace compilation. However, the idea of focusing the compilation effort on code traces, instead of whole methods, has been known at least since the work of Bala *et al.* [2], or even before if we consider trace scheduling [11]. Traces have been independently integrated on JIT-compilers by Gal [13] and Zaleski [27]. Currently, there are many trace-based JIT-compilers [6,14,15,27]. For a formal overview of trace compilation, we recommend the work of Guo and Palsberg [26].

Our algorithm to remove overflow tests is a type of *range analysis* [17,20]. Range analysis tries to infer lower and upper bounds to the values that a variable might assume during the program execution. In general these algorithms rely on theorem provers, an approach deemed too slow to a JIT compiler. Bodik *et al.* [3] have described a specialization of range analysis that removes array bound checks, the ABCD algorithm, that targets JIT compilation. Zhendong and Wagner [25] have described a type of range analysis that can be solved in polynomial time. Stephenson *et al.* [24] have used a polynomial time analysis to infer the bitwidth of each integer variable used in the source program. Contrary to our approach, all these previous algorithms work on the static representation of the source program. Such fact severely constraints the amount of information that these analysis can rely on. By knowing the runtime value of program variables we can perform a very extensive, yet fast, range analysis.

6 Conclusion

This paper has presented a new algorithm to remove redundant overflow tests during the JIT compilation of JavaScript programs. The proposed algorithm works in the context of a trace compiler. Our algorithm is able to find very precise ranges for program variable by inspecting their runtime values. We have implemented our analysis on top of TraceMonkey, the JIT compiler used by the Mozilla Firefox browser to speed up the execution of JavaScript programs. We have submitted our implementation to Mozilla, as a Firefox patch, available at <http://github.com/rodrigoso/Dynamic-Elimination-Overflow-Test>. In terms of future work, we would like to improve the performance and effectiveness of our implementation. For instance, currently we can only read the runtime values of local variables, a limitation that we are working to overcome.

Acknowledgments. This project has been made possible by the cooperation FAPEMIG-INRIA, grant 11/2009. Rodrigo Sol is supported by the Brazilian Ministry of Education under CAPES and CNPq. We thank David Mandelin from the Mozilla Foundation for helping us with TraceMonkey.

References

1. Aycock, J.: A brief history of just-in-time. *ACM Computing Surveys* 35(2), 97–113 (2003)
2. Bala, V., Duesterwald, E., Banerjia, S.: Dynamo: a transparent dynamic optimization system. In: *PLDI*, pp. 1–12. ACM, New York (2000)
3. Bodik, R., Gupta, R., Sarkar, V.: ABCD: eliminating array bounds checks on demand. In: *PLDI*, pp. 321–333. ACM, New York (2000)
4. Carette, J., Kucera, M.: Partial evaluation of maple. In: *PEPM*, pp. 41–50. ACM, New York (2007)
5. Chambers, C., Ungar, D.: Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language. *SIGPLAN Not.* 24(7), 146–160 (1989)
6. Chang, M., Smith, E., Reitmaier, R., Bebenita, M., Gal, A., Wimmer, C., Eich, B., Franz, M.: Tracing for web 3.0: trace compilation for the next generation web applications. In: *VEE*, pp. 71–80. ACM, New York (2009)
7. Chevalier-Boisvert, M., Hendren, L., Verbrugge, C.: Optimizing MATLAB through just-in-time specialization. In: Gupta, R. (ed.) *CC 2010*. LNCS, vol. 6011, pp. 46–65. Springer, Heidelberg (2010)
8. ECMA Committe. *ECMAScript Language Specification*. ECMA, 5th edn. (2009)
9. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. *TOPLAS* 13(4), 451–490 (1991)
10. Elphick, D., Leuschel, M., Cox, S.: Partial evaluation of MATLAB. In: Pfenning, F., Macko, M. (eds.) *GPCE 2003*. LNCS, vol. 2830, pp. 344–363. Springer, Heidelberg (2003)
11. Fisher, J.A.: Trace scheduling: A technique for global microcode compaction. *Trans. Comput.* 30, 478–490 (1981)
12. Richards, G., Lebesne, S., Burg, B., Vitek, J.: An analysis of the dynamic behavior of javascript programs. In: *PLDI*, pp. 1–12 (2010)
13. Gal, A.: *Efficient Bytecode Verification and Compilation in a Virtual Machine*. PhD thesis, University of California, Irvine (2006)
14. Gal, A., Eich, B., Shaver, M., Anderson, D., Kaplan, B., Hoare, G., Mandelin, D., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E., Reitmaier, R., Haghighat, M.R., Bebenita, M., Change, M., Franz, M.: Trace-based just-in-time type specialization for dynamic languages. In: *PLDI*, pp. 465–478. ACM, New York (2009)
15. Gal, A., Probst, C.W., Franz, M.: Hotpathvm: an effective jit compiler for resource-constrained devices. In: *VEE*, pp. 144–153 (2006)
16. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. *Comput. Surv.* 23, 5–48 (1991)
17. Harrison, W.H.: Compiler analysis of the value ranges for variables. *IEEE Trans. Softw. Eng.* 3(3), 243–250 (1977)
18. Jones, N.D., Gomard, C.K., Sestoft, P.: *Partial Evaluation and Automatic Program Generation*, 1st edn. Prentice Hall, Englewood Cliffs (1993)
19. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of ACM* 3(4), 184–195 (1960)
20. Patterson, J.R.C.: Accurate static branch prediction by value range propagation. In: *PLDI*, pp. 67–78. ACM, New York (1995)
21. Rigo, A.: Representation-based just-in-time specialization and the psyco prototype for python. In: *PEPM*, pp. 15–26. ACM, New York (2004)

22. Schultz, U.P., Lawall, J.L., Consel, C.: Automatic program specialization for java. *TOPLAS* 25(4), 452–499 (2003)
23. Shankar, A., Sastry, S.S., Bodík, R., Smith, J.E.: Runtime specialization with optimistic heap analysis. *SIG. Not.* 40(10), 327–343 (2005)
24. Stephenson, M., Babb, J., Amarasinghe, S.: Bidwidth analysis with application to silicon compilation. In: *PLDI*, pp. 108–120. ACM, New York (2000)
25. Su, Z., Wagner, D.: A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computer Science* 345(1), 122–138 (2005)
26. Guo, S.y., Palsberg, J.: The essence of compiling with traces. In: *POPL*. ACM, New York (2011) (page to appear)
27. Zaleski, M.: YETI: a gradually extensible trace interpreter. PhD thesis, University of Toronto (2007)

Staged Static Techniques to Efficiently Implement Array Copy Semantics in a MATLAB JIT Compiler*

Nurudeen Lameed and Laurie Hendren


School of Computer Science, McGill University, Montréal, QC, Canada
{nlamee,hendren}@cs.mcgill.ca

Abstract. MATLAB has gained widespread acceptance among scientists. Several dynamic aspects of the language contribute to its appeal, but also provide many challenges. One such problem is caused by the copy semantics of MATLAB. Existing MATLAB systems rely on reference-counting schemes to create copies only when a shared array representation is updated. This reduces array copies, but requires runtime checks.

We present a staged static analysis approach to determine when copies are not required. The first stage uses two simple, intraprocedural analyses, while the second stage combines a forward *necessary copy analysis* with a backward *copy placement analysis*. Our approach eliminates unneeded array copies without requiring reference counting or frequent runtime checks.

We have implemented our approach in the McVM JIT. Our results demonstrate that, for our benchmark set, there are significant overheads for both existing reference-counted and naive copy-insertion approaches, and that our staged approach is effective in avoiding unnecessary copies.

1 Introduction

MATLABTM is a popular programming language for scientists and engineers. It was designed for sophisticated matrix and vector operations, which are common in scientific applications. It is also a dynamic language with a simple syntax that is familiar to most engineers and scientists. However, being a dynamic language, MATLAB presents significant compilation challenges. The problem addressed in this paper is the efficient compilation of the array copy semantics defined by the MATLAB language. The basic semantics and types in MATLAB are very simple. Every variable is assumed to be an array (scalars are defined as 1x1 arrays) and copy semantics is used for assignments of one array to another array, parameter passing and for returning values from a function. Thus a statement of the form $\mathbf{a} = \mathbf{b}$ semantically means that a copy of \mathbf{b} is made and that copy is assigned to \mathbf{a} . Similarly, for a call of the form $\mathbf{a} = \text{foo}(\mathbf{c})$, a copy of \mathbf{c} is made and assigned

* This work was supported, in part, by NSERC and FQRNT.

¹ <http://www.mathworks.com/products/pfo/>

to the parameter of the function `foo`, and the return value of `foo` is copied to `a`. Naive implementations take exactly this approach.

However, in the current implementations of `MATLAB` the copy semantics is implemented lazily using a reference-count approach. The copies are not made at the time of the assignment, rather an array is shared until an update to one of the shared arrays occurs. At update time (for example a statement of the form `b(i) = x`), if the array being updated (in this case `b`) is shared, a copy is generated, and then the update is performed on that copy. We have verified that this is the approach that Octave open-source system [1] takes (by examining and instrumenting the source code). We have inferred that this approach (or a small variation) is what the Mathworks' closed-source implementation does based on the user-level documentation [19, p. 9-2].

Although the reference-counting approach reduces unneeded copies at runtime, it introduces many redundant checks, requires space for the reference counts, and requires extra code to update the reference counts. This is clearly costly in a garbage-collected VM, such as the recently developed `McVMM`, a specializing JIT [8,9]. Furthermore, the reference-counting approach may generate a redundant copy during an update of a variable, if the updated variable shares an array only with dead variables.

Thus, our challenge was to develop a static analysis approach, suitable for a JIT compiler, which could determine which copies were required, without requiring reference counts and without the expense of dynamic checks. Since we are in the context of a JIT compiler, we developed a staged approach. The first phase applies very simple and inexpensive analyses to determine the obvious cases where copies can be avoided. The second phase tackles the harder cases, using a pair of more sophisticated static analyses: a forward analysis to locate all places where an array update requires a copy (*necessary copy analysis*) and then a backward analysis that moves the copies to the best location and which may eliminate redundant copies (*copy placement analysis*). We have implemented our analyses in the `McJIT` compiler as structured flow analyses on the low-level AST intermediate representation used by `McJIT`.

To demonstrate the applicability of our approach, we have performed several experiments to: (1) demonstrate the behaviour of the reference-counting approaches, (2) to measure the overhead associated with the dynamic checks in the reference-counting approach, and (3) demonstrate the effectiveness of our static analysis approach. Our results show that actual needed copies are infrequent even though the number of dynamic checks can be quite large. We also show that these redundant checks do contribute significant overheads. Finally, we show that for our benchmark set, our static approach finds the needed number of copies, without introducing any dynamic checks.

The paper is organized as follows. Sec. 2 describes the `McLab` project and how this work fits into the project. Sec. 3 describes the simple first-stage analyses, and Sec. 4 and Sec. 5 describe the second-stage forward and the backward analyses, with examples. Sec. 6 discusses the experimental results of our approach; we review some related work in Sec. 7, and Sec. 8 concludes the paper.

2 Background

The work presented in this paper is a key component of our McLab system [2]. McLab provides an extensible set of compilation, analysis and execution tools built around the core MATLAB language. One goal of the McLab project is to provide an open-source set of tools for the programming language and compiler community so that researchers (including our group) can develop new domain-specific language extensions and new compiler optimizations. A second goal is to provide these new languages and compilers to scientists and engineers to both provide languages more tailored to their needs and also better performance.

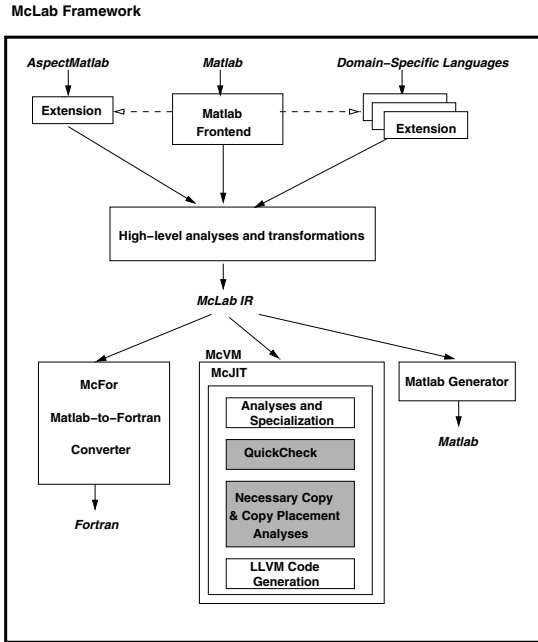


Fig. 1. Overview of McLab (shaded boxes correspond to analyses in this paper)

The McLab framework is outlined in Fig. 1, with the shaded boxes indicating the components presented in this paper. The framework is comprised of an extensible front-end, a high-level analysis and transformation engine and three backends. Currently there is support for the core MATLAB language and also a complete extension supporting ASPECTMATLAB [5] [2]. The front-end and the extensions are built using our group’s extensible lexer, Metalexer [7], and JastAdd [11]. There are three backends: McFor, a FORTRAN code generator [18]; a MATLAB generator (to use McLab as a source-to-source compiler); and McVM,

² We use ASPECTMATLAB for some dynamic measurements in Sec. 6.

a virtual machine that includes a simple interpreter and a sophisticated type-specialization-based JIT compiler, which generates LLVM [17] code.

The techniques presented in this paper are part of McJIT, the JIT compiler for McVM. McJIT is built upon LLVM, the Boehm garbage collector [6], and several numerical libraries [4, 28]. For the purposes of this paper, it is important to realize that McJIT specializes code based on the function argument types that occur at runtime. When a function is called the VM checks to see if it already has a compiled version corresponding to the current argument types. If it does not, it applies a sequence of analyses including live variable analysis and type inference. Finally, it generates LLVM code for this version.

When generating code McJIT assumes reference semantics, and not copy semantics, for assignments between arrays and parameter passing. That is, arrays are dealt with as pointers and only the pointers are copied. Clearly this does not match the copy semantics specified for MATLAB and thus the need for the two shaded boxes in Fig. 1 in order to determine where copies are required and the best location for the copies. These two analysis stages are the core of the techniques presented in this paper. It is also important to note that the specialization and type inference in McJIT means that variables that certainly have scalar types will be stored in LLVM registers and thus the copy analyses only need to consider the remaining variables. The type-inference analysis is used to disambiguate between function calls and array accesses since MATLAB uses the same syntax for both.

In the next section we introduce the first stage of our approach which is the *QuickCheck*. Following that we introduce the second stage — the *necessary copy* and *copy placement* analyses.

3 Quick Check

The *QuickCheck* phase (*QC*) is a combination of two simple and fast analyses. The first, *written parameters analysis*, is a forward analysis which determines the parameters that *may* be modified by a function. The intuition is that during a call of the function, the arguments passed to it from the caller need to be copied to the corresponding formal parameters of the function only if the function may modify the parameters. Read-only arguments do not need to be copied.

The analysis computes a set of pairs, where each pair represents a parameter and the assignment statement that last defines the parameter. For example, the entry (p_1, d_1) indicates that the last definition point for the parameter p_1 is the statement d_1 . The analysis begins with a set of initial definition pairs, one pair for each parameter declaration. The analysis also builds a *copy list*, a list of parameters which must be copied, which is initialized to the empty list. The analysis is a forward flow analysis, using union as the merge operator. The key flow equations are for assignment statements of two forms:

$p = rhs$: If the left-hand side (*lhs*) of the statement is a parameter p , then this statement is redefining p , so all other definitions of p are killed and this

new definition of p is generated. Note that according to the MATLAB copy semantics, such a statement is not creating an alias between p and rhs , but rather p is a new copy; subsequent writes to p will write to this new copy.

$p(i) = rhs$: If the lhs is an array index expression (i.e., the assignment statement is writing to an element of p), and the array symbol p is a parameter, it checks if the initial definition of the parameter reaches the current assignment statement and if so, it inserts the parameter into the copy list.

At the end of the analysis, the copy list contains all the parameters that must be copied before executing the body of the function.

The second analysis is *copy replacement*, a standard sort of copy propagation/elimination algorithm that is similar to the approach used by an APL compiler [27]. It determines when a copy variable can be replaced by the original variable (copy propagation). If all the uses of the copy variable can be replaced by the original variable then the copy statement defining the copy can be removed after replacing all the uses of the copy with the original (copy elimination).

If the analysed function does not return an array and all the remaining copy statements have been made redundant by the QC transformation, then there is no need to apply a more sophisticated analysis. However, if copies do remain, then phase 2 is applied, as outlined in the next two sections.

4 Necessary Copy Analysis

The *necessary copy analysis* is a forward analysis that collects information that is used to determine whether a copy should be generated before an array is modified. To simplify our description of the analysis, we consider only simple assignment statements of the form $lhs = rhs$. It is straightforward to show that our analysis works for both single (one lhs variable) and multiple assignment statements (multiple lhs variables). We describe the analysis by defining the following components.

Domain: the domain of the analysis' flow facts is the set of pairs comprising of an array reference variable and the ID of the statement that allocates the memory for the array; henceforth called *allocators*. We write (a, s) if a may reference the array allocated at statement s .

Problem Definition: at a program point p , a variable references a shared array if the number of variables that reference the array is greater than one. An array update via an array reference variable requires a copy if the variable *may* reference a shared array at p and at least one of the other variables that reference the same array is *live* after p .

Flow Function: $out(S_i) = gen(S_i) \cup (in(S_i) - kill(S_i))$.

Given the assignment statements of the forms:

$$S_i : a = \text{alloc} \quad (1)$$

$$S_i : a = b \quad (2)$$

$$S_i : a(j) = x \quad (3)$$

$$S_i : a = f(\text{arg}_1, \text{arg}_2, \dots, \text{arg}_n) \quad (4)$$

where S_i denotes a statement ID; `alloc` is a new memory allocation performed by statement S_i ; a, b are array reference variables; x is a *rvalue*; f is a function, $\text{arg}_1, \text{arg}_2, \dots, \text{arg}_n$ denote the arguments passed to the function and the corresponding formal parameters are denoted with p_1, p_2, \dots, p_n .

We partition $\text{in}(S_i)$ using allocators. The partition, $Q_i(m)$, containing flow entries for allocator m is:

$$Q_i(m) = \{(x, y) \mid (x, y) \in \text{in}(S_i) \wedge y = m\} \quad (5)$$

Now consider statements of type 2 above; if the variable b has a reaching definition at S_i then there must exist some $(b, m) \in \text{in}(S_i)$ and there exists a non-empty $Q_i(m) \ ((b, m) \in Q_i(m))$.

In addition, if b may reference a shared array at S_i then $|Q_i(m)| > 1$. Let us call the set of all such $Q_i(m)$ s, P_i . We write $P_i(a)$ for the set of Q_i s obtained by partitioning $\text{in}(S_i)$ using the allocators of the variable a .

Considering statements of the form 3, $P_i(a) \neq \emptyset$ implies that a copy of a must be generated before executing S_i and in that case, S_i is a *copy generator*. This means that after this statement a will point to a new copy and no other variable will refer to this copy.

We are now ready to construct a table of *gen* and *kill* sets for the four assignment statement kinds above. To simplify the table, we define

$$\begin{aligned} \text{Kill}_{\text{define}}(a) &= \{(x, s) \mid (x, s) \in \text{in}(S_i) \wedge x = a\} \\ \text{Kill}_{\text{dead}} &= \{(x, s) \mid (x, s) \in \text{in}(S_i) \wedge \text{not live}(S_i, x)\} \\ \text{Kill}_{\text{update}}(a) &= \{(x, s) \mid (x, s) \in \text{in}(S_i) \wedge x = a \wedge P_i(a) \neq \emptyset\} \end{aligned}$$

Stmt	Gen set	Kill set
1	$\{(x, s) \mid x = a \wedge s = S_i \wedge \text{live}(S_i, x)\}$	$\text{Kill}_{\text{define}}(a) \cup \text{Kill}_{\text{dead}}$
2	$\{(x, s) \mid x = a \wedge (y, s) \in \text{in}(S_i) \wedge y = b \wedge \text{live}(S_i, x)\}$	$\text{Kill}_{\text{define}}(a) \cup \text{Kill}_{\text{dead}}$
3	$\{(x, s) \mid x = a \wedge s = S_i \wedge P_i(x) \neq \emptyset\}$	$\text{Kill}_{\text{update}}(a) \cup \text{Kill}_{\text{dead}}$
4	see $\text{gen}(f)$ below	$\text{Kill}_{\text{define}}(a) \cup \text{Kill}_{\text{dead}}$

Computing the *gen* set for a function call is not straightforward. Certain built-in functions allocate memory blocks for arrays; such functions are categorized as *alloc functions*. A question that arises is: does the return value of the called function reference the same shared array as a parameter of the function? If the return value references the same array as a parameter of the function then this sharing must be made explicit in the caller, after the function call statement. Therefore, the *gen* set for a function call is defined as:

³ Functions such as *zeros*, *ones*, *rand* and *magic* are memory allocators in MATLAB.

$$gen(f) = \left\{ \begin{array}{l} \{(a, S_i)\}, \text{ if } \mathit{live}(S_i, a) \text{ and } \mathit{isAllocFunction}(f) \\ \{(x, s) \mid x = a \wedge (arg_j, s) \in \mathit{in}(S_i) \wedge \mathit{live}(S_i, x)\}, \\ \text{if } \mathit{ret}(f) \text{ aliases } param_j(f), 0 < j \leq \mathit{size}(params(f)), \\ \{(a, S_i)\}, \text{ if } \forall (p \in params(f)), \text{ not } (\mathit{ret}(f) \text{ aliases } p) \\ \{(x, s) \mid x = a \wedge arg \in args(f) \wedge (arg, s) \in \mathit{in}(S_i) \wedge \mathit{live}(S_i, x)\}, \\ \text{otherwise (e.g., if } f \text{ is recursive)} \end{array} \right.$$

The first alternative generates a flow entry (a, S_i) if the *rhs* is an *alloc* function and the *lhs* (a) is live after statement S_i ; this makes statement S_i an allocator. In the second alternative, the analysis requests for the result of the necessary copy analysis on f from an analysis manager⁴. The manager caches the result of the previous analysis on a given function. From the result of the analysis on f , we determine the return variables of f that are aliases to the parameters of f and hence aliases to the arguments of f . This is explained in detail under the next section on Initialization. The return variable of f corresponds to the *lhs* (a) in statement type [4](#). Therefore we generate new flow entries from those of the arguments that the return variable may reference according to the summary information of f and provided that a is also *live* after S_i . The third alternative generates $\{(a, S_i)\}$, if the return variable aliases no parameters of f . The fourth alternative is conservative: new flow entries are generated from those of *all* the arguments to f . This can happen if the call of f is recursive or f cannot be analyzed because it is neither a user-defined function nor an *alloc* function.

We chose a simple strategy for recursion because recursive functions occur rarely in MATLAB. In a separate study by our group, we found that out of 15966 functions in 625 projects examined, only 48 functions (0.3%) are directly recursive. None of the programs in our benchmarks had recursive functions.

Therefore, we expect that the conservative option in the definition of $gen(f)$ above will be rarely taken in practice.

Initialization: The input set for a function is initialized with a flow entry for each parameter and an additional flow entry (a shadow entry) for each parameter is also inserted. This is necessary in order to determine which of the parameters (if any) return variable references. We use a shadow entry to detect when a parameter that has not been assigned to any other variable is updated. At the entry to a function, the input set is given as

$$\mathit{in}(entry) = \{(p, s) \mid p \in params(f) \wedge s = S_p\} \cup \{(p', s) \mid p' \in params(f) \wedge s = S_p\}.$$

We illustrate this scheme with an example. Given a function f , defined as:

```
function u = f(x, y)
    u = x;
end
```

the *in* set at the entry of f is $\{(x, S_x), (x', S_x), (y, S_y), (y', S_y)\}$ and at the end of the function, the *out* set is $\{(u, S_x), (x, S_x), (x', S_x), (y, S_y), (y', S_y)\}$.

⁴ This uses the same analysis machinery as the type estimation in McJIT.

We now know that u is an alias for x and encode this information as a set of integers. An element of the set is an integer representing the input parameter that the output parameter may reference in the function. In this example, the set is $\{1\}$ since x is the first (1) parameter of f . This is useful during a call of f . For instance, in $c = f(a, b)$; we can determine that c is an alias for the argument a by inspecting the summary information generated for f .

4.1 if-else Statement

So far we have been considering sequences of statements. As our analysis is done directly on a simplified AST, analyzing an `if-else` statement simply requires that we analyze all the alternative blocks and merge the result at the end of the `if-else` statement using the merge operator (\cup).

4.2 Loops

We compute the input set reaching a loop and the output set exiting a loop using standard flow analysis techniques, that is, we merge the input flow set from the loop's entry with the output set from the loop back-edge until a fixed point is reached.

To analyse a loop more precisely, we implemented a context-sensitive loop analysis [16], but found that real MATLAB programs did not require the context-sensitivity to achieve good results. The standard approach is sufficient for typical MATLAB programs.

5 Copy Placement Analysis

In the previous section, we described the forward analysis which determines whether a copy should be generated before an array is updated. One could use this analysis alone to insert the copy statements, but this may not lead to the best placement of the copies and may lead to redundant copies. The backward *copy placement analysis* determines a better placement of the copies, while at the same time ensuring safe updates of a shared array. Examples of moving copies include hoisting copies out of if-then constructs and out of loops.

The intuition behind this analysis is that often it is better to perform the array copy close to the statement which created the sharing (i.e. statements of the form $a = b$) rather than just before the array update statements (i.e. statements of the form $a(i) = b$) that require the copy. In particular, if the update statement is inside a loop, but the statement that created the sharing is outside the loop, then it is much better to create the copy outside of the loop. Thus, the *copy placement analysis* is a backward analysis that pushes the necessary copies upwards, possibly as far as the statement that created the sharing.

5.1 Copy Placement Analysis Details

A copy entry is a three-tuple:

$$e = \langle copy_loc, var, alloc_site \rangle \quad (6)$$

where $copy_loc$ denotes the ID of the node that generates the copy, var denotes the variable containing a reference to the array that should be copied, and $alloc_site$ is the allocation site where the array referenced by var was allocated. We refer to the three components of the three-tuple as $e.copy_loc$, $e.var$, and $e.alloc_site$.

Let C denote the set of all copies generated by a function.

Given a function, the analysis begins by traversing the block of statements of the function backward. The domain of the analysis' flow entries is the set of copy objects and the merge operator is intersection (\cap).

Define C_{out} as the set of copy objects at the exit of a block and C_{in} as the set of copy objects at the entrance of a block. Since the analysis begins at the end of a function, C_{out} is initialized to \emptyset . The rules for generating and placing copies are described here.

Statement Sequence. Given a sequence of statements, we are given a C_{out} for this block and the analysis traverses backwards through the block computing a C_{in} for the block. As each statement is traversed the following rules are applied for the different kinds of the assignment statements in the sequence. The sets $in(S_i)$, $Q_i(m)$, $P_i(a)$ are defined in Section [4](#).

Rule 1: array updates, $S_i : a(y) = x$: Given that the array variable of the *lhs* of statement S_i is a , when a statement of this form is reached, we add a copy for each partition for a shared array to the current copy set. Thus

$$C_{in} := C_{in} \cup \begin{cases} \emptyset & \text{if } P_i(a) = \emptyset \\ \{ \langle s, x, y \rangle \mid s = S_i \wedge x = a \wedge Q_i(y) \in P_i(x) \} & \text{otherwise} \end{cases}$$

Rule 2: array assignments, $S_j : a = b$: If $\forall e \in C_{in}(e.var \neq a \text{ and } e.var \neq b)$, and $\forall e \in C_{out}(e.var \neq a \text{ and } e.var \neq b)$, we skip the current statement. However, if in the current block, $\exists e \in C_{in}(e.var = a \text{ or } e.var = b)$, we remove e from the current copy flow set C_{in} . This means that the copy has been placed at its current location — the location specified in the copy entry e . Otherwise, if $\exists e \in C_{out}(e.var = a \text{ or } e.var = b)$, we perform the following:

if $P_j(a) = \emptyset$, this is usually the case, we move the copy from the statement $e.copy_loc$ to S_j and remove e from the flow set. The copy e has now been finally placed.

if $P_j(a) \neq \emptyset$, $\forall(Q_i(m) \in P_j(a))$, we add a runtime equality test for a against the variable x ($x \neq a$) of each member of $Q_i(m)$ at the statement $e.copy_loc$.

Since $P_j(a) \neq \emptyset$, there is at least a definition of a that reaches this statement and for which a references a shared array. In addition, because the copy e was generated after the current block there are at least two different paths to the statement $e.copy_loc$, the current location of e . We place a copy of e at the current statement S_j and remove e from the flow set. Note that two copies of e have been placed; one at $e.copy_loc$ and another at S_j . However, runtime guards have also been placed at $e.copy_loc$, ensuring that only one of these two copies materializes at runtime.

We expect that such guards will not usually be needed, and in fact none of our benchmarks required any guards.

if-else Statements. Let C_{if} and C_{else} denote the set of copies generated in an **if** and an **else** block respectively. First we compute

$$C' := (C_{out} \cap C_{else} \cap C_{if})$$

Then we compute the differences

$$C'_{out} := C_{out} \setminus C'; \quad C'_{else} := C_{else} \setminus C'; \quad C'_{if} := C_{if} \setminus C'$$

to separate those copies that do not intersect with those in other blocks but should nevertheless be propagated upward. Since the copies in the intersection will be relocated, they are removed from their current locations.

And finally,

$$C_{in} := C'_{out} \cup C'_{else} \cup C'_{if} \cup \{ \langle s, e.var, e.alloc_site \rangle \mid s = S_{IF} \wedge e \in C' \}$$

Note that a copy object e with its first component set to S_{IF} is attached to the *if-else* statement S_{IF} . That means if these copies remain at this location, the copies should be generated before the *if-else* statement.

Loops. The main goal here is to identify copies that could be moved out of a loop. To place copies generated in a loop, we apply the rules for statement sequence and the **if-else** statement. The analysis propagates copies upward from the inner-most loop to the outer-most loop and to the main sequence until either loop dependencies exist in the current loop or it is no longer possible to move the copy according to Rule 2 in Section 5.1.

A disadvantage of propagating the copy outside of the loop is that if none of the loops that require copies is executed then we would have generated a useless copy. However, the execution is still correct. For this reason, we assume that a loop will *always* be executed and generate copies outside loops, wherever possible. This is a reasonable assumption because a loop is typically programmed to execute. With this assumption, there is no need to compute the intersection of C_{loop} and C_{out} . Hence

$$C_{in} := C_{out} \cup \{ \langle s, e.var, e.alloc_site \rangle \mid s = S_{loop} \wedge e \in C_{loop} \}$$

5.2 Using the Analyses

This section illustrates how the combination of the forward and the backward analyses is used to determine the actual copies that should be generated. First consider the following program, *test3*. Fig. 2(a) shows the result of the forward analysis.

```

1 function test3()
2   a = [1:5];
3   b = a;
4   i = 1;
5   if (i > 2) % I
6     a(1) = 100;
7   else
8     a(1) = 700;
9   end
10  a(1) = 200;
11  disp(a); disp(b);
12 end

```

#	Gen set	In	Out
2	$\{(a, S_2)\}$	\emptyset	$\{(a, S_2)\}$
3	$\{(b, S_2)\}$	$\{(a, S_2)\}$	$\{(a, S_2)(b, S_2)\}$
6	$\{(a, S_6)\}$	$\{(a, S_2), (b, S_2)\}$	$\{(b, S_2)(a, S_6)\}$
8	$\{(a, S_8)\}$	$\{(a, S_2), (b, S_2)\}$	$\{(b, S_2), (a, S_8)\}$
10	\emptyset	$\{(b, S_2), (a, S_6), (a, S_8)\}$	$\{(b, S_2), (a, S_6), (a, S_8)\}$

(a) Necessary Copy Analysis Result

#	C_{out}	C_{in}	Current Result
10	\emptyset	\emptyset	\emptyset
8	\emptyset	$\{< S_8, a, S_2 >\}$	$\{(a, S_8)\}$
6	\emptyset	$\{< S_6, a, S_2 >\}$	$\{(a, S_6)\}$
I	\emptyset	$\{< S_I, a, S_2 >\}$	$\{(a, S_I)\}$
3	$\{< S_I, a, S_2 >\}$	\emptyset	$\{(a, S_I)\}$
2	\emptyset	\emptyset	$\{(a, S_I)\}$

(b) Copy Placement Analysis Result

Fig. 2. Introductory example for Copy Placement Analysis

Fig. 2(b) gives the result of the backward analysis. The *I* used in Fig. 2 stands for the *if-else* statement in *test3*. The analysis begins from line 12 of *test3*. The out set C_{out} is initially empty. At line 10, C_{out} is still empty. When the *if-else* statement is reached, a copy of C_{out} (\emptyset) is passed to the *Else* block and another copy is passed to the *If* block. The copy $\{< S_8, a, S_2 >\}$ is generated in the *Else* block because $|Q(S_2) = \{(a, S_2), (b, S_2)\}| = 2$, hence $P_i(a) \neq \emptyset$. Similarly $\{< S_6, a, S_2 >\}$ is generated in the *If* block.

By applying the rule for *if-else* statement described in Section 5.1, the outputs of the *If* and the *Else* blocks are merged to obtain the result at S_I (the *if-else* statement). Applying Rule 2 for statement sequence (Section 5.1) in S_3 , $\{< S_I, a, S_2 >\}$ is removed from C_{in} and the analysis terminates at S_2 . The final result is that a copy must be generated before the *if-else* statement instead of generating two copies, one in each block of the *if-else* statement. This example illustrates how common copies generated in the alternative blocks of an *if-else* statement could be combined and propagated upward to reduce code size.

The second example, *tridisolve* is a MATLAB function from [10]. The forward analysis information is shown in Fig. 3(a). The table shows the *gen* and *in* sets at each relevant assignment statement of *tridisolve*. The results in different loop iterations are shown using a subscript to represent loop iteration. For example, the row number 25_2 refers to the result at the statement labelled S_{25} in the second iteration. The analysis reached a fixed point after the third iteration.

At the function's entry, the *in* set is initialized with two flow entries for each parameter of the function as outlined in Sec. 4. The analysis continues by generating the *gen*, *in* and *out* sets according to the rules specified in Section 4. Notice that statement S_{25} is an allocator because $P_{25}(b) \neq \emptyset$ since

```

function x = tridisolve(a,b,c,d)
% TRIDISOLVE Solve tridiagonal system of equations.
20: x = d;
21: n = length(x);
    for j = 1:n-1           %F_1
        mu = a(j)/b(j);
25:   b(j+1) = b(j+1) - mu*c(j);
26:   x(j+1) = x(j+1) - mu*x(j);
    end
29: x(n) = x(n)/b(n);
    for j = n-1:-1:1       %F_2
31:   x(j) = (x(j)-c(j)*x(j+1))/b(j);
    end

```

#	Gen	In
20	{{(x, S _d , 0)}}	{{(a, S _a , 0), (a', S _a , 0), (b, S _b , 0), (b', S _b , 0), (c, S _c , 0), (c', S _c , 0), (d, S _d , 0), (d', S _d , 0)}}
25 ₁	{{(b, S ₂₅ , 1)}}	{{(a, S _a , 0), (a', S _a , 0), (b, S _b , 0), (b', S _b , 0), (c, S _c , 0), (c', S _c , 0), (d', S _d , 0), (x, S _d , 0)}}
26 ₁	{{(x, S ₂₆ , 1)}}	{{(a, S _a , 0), (a', S _a , 0), (b', S _b , 0), (c, S _c , 0), (c', S _c , 0), (d', S _d , 0), (x, S _d , 0), (b, S ₂₅ , 1)}}
25 ₂	{{(b, S ₂₅ , 2)}}	{{(a, S _a , 0), (a', S _a , 0), (b, S _b , 0), (b', S _b , 0), (c, S _c , 0), (c', S _c , 0), (d', S _d , 0), (x, S _d , 0), (b, S ₂₅ , 1), (x, S ₂₆ , 1)}}
26 ₂	{{(x, S ₂₆ , 2)}}	{{(a, S _a , 0), (a', S _a , 0), (b', S _b , 0), (c, S _c , 0), (c', S _c , 0), (d', S _d , 0), (x, S _d , 0), (b, S ₂₅ , 2), (x, S ₂₆ , 1)}}
25 ₃	{{(b, S ₂₅ , 3)}}	{{(a, S _a , 0), (a', S _a , 0), (b, S _b , 0), (b', S _b , 0), (c, S _c , 0), (c', S _c , 0), (d', S _d , 0), (x, S _d , 0), (b, S ₂₅ , 2), (x, S ₂₆ , 2)}}
26 ₃	{{(x, S ₂₆ , 3)}}	{{(a, S _a , 0), (a', S _a , 0), (b', S _b , 0), (c, S _c , 0), (c', S _c , 0), (d', S _d , 0), (x, S _d , 0), (b, S ₂₅ , 3), (x, S ₂₆ , 2)}}
29	{{(x, S ₂₉ , 0)}}	{{(a', S _a , 0), (b, S _b , 0), (b', S _b , 0), (c, S _c , 0), (c', S _c , 0), (d', S _d , 0), (x, S _d , 0), (b, S ₂₅ , 3), (x, S ₂₆ , 3)}}
31 ₁	∅	{{(a', S _a , 0), (b, S _b , 0), (b', S _b , 0), (c, S _c , 0), (c', S _c , 0), (d', S _d , 0), (b, S ₂₅ , 3), (x, S ₂₉ , 0)}}
31 ₂	∅	{{(a', S _a , 0), (b, S _b , 0), (b', S _b , 0), (c, S _c , 0), (c', S _c , 0), (d', S _d , 0), (b, S ₂₅ , 3), (x, S ₂₉ , 0)}}

(a) Necessary Copy Analysis Result

#	C _{out}	C _{in}	Current Result
31	∅	∅	∅
F ₂	∅	∅	∅
29	∅	{{(S ₂₉ , a, S _d)}}	{{(x, S ₂₉)}}
26	{{(S ₂₉ , x, S _d)}}	{{(S ₂₆ , x, S _d)}}	{{(x, S ₂₉), (x, S ₂₆)}}
25	{{(S ₂₉ , x, S _d)}}	{{(S ₂₅ , b, S _b), (S ₂₆ , x, S _d)}}	{{(x, S ₂₉), (x, S ₂₆), (b, S ₂₅)}}
F ₁	{{(S ₂₉ , x, S _d)}}	{{(S _{F1} , x, S _d), (S ₂₅ , b, S _b)}}	{{(x, S _{F1}), (b, S ₂₅)}}
20	∅	{{(S ₂₅ , b, S _b)}}	{{(x, S _{F1}), (b, S ₂₅)}}
0	∅	∅	{{(x, S _{F1}), (b, S ₀)}}

(b) Copy Placement Analysis Result

Fig. 3. Example for *tridisolve*

$|Q_{25}(S_b)| = |\{(b, S_b, 0), (b', S_b, 0)\}| > 1$. Similarly, S_{26} and S_{29} are also allocators. This means that generating a copy of the array referenced by the variable b just before executing the statement S_{25} ensures a safe update of the array. The same is true of the array referenced by the variable x in lines 26 and 29. However, are these the best points in the program to generate those copies? Could the number of copies be reduced? We provide the answers to these questions when we examine the results of the backward analysis.

Fig. 3(b) shows the copy placement analysis information at each relevant statement of *tridisolve*. Recall that the placement analysis works by traversing the statements in each block of a function backward. In the case of *tridisolve*, the analysis begins in line 31 in the second *for* loop of the function. The set C_{out} is passed to the loop body and is initially empty. The set C_{in} stores all the copies generated in the block of the *for* statement. Line 31 is neither a definition nor an allocator, therefore no changes are recorded at this stage of the analysis.

At the beginning of loop F_2 , the analysis merges with the main path and the result at this point is shown in row F_2 . Statement S_{29} generated a copy as indicated by the forward analysis, therefore C_{in} is updated and the result set is also updated. The analysis then branches off to the first loop and the current C_{in} is passed to the loop’s body as C_{out} . The copies generated in loop F_1 are stored in C_{in} , which is then merged with C_{out} at the beginning of the loop to arrive at the result in row F_1 . The result set is also updated accordingly; at this stage, the number of copies has been reduced by 1 as shown in the column labelled *Current Result* of Fig. 3(b). The copy flow set that reaches the beginning of the function is non-empty. This suggests that the definition or the allocator of the array variables of the remaining entries could not be reached. Therefore, the array variables of the flow entries *must* be the parameters of the function and the necessary copy should be generated at the function’s entry. Hence, a copy of the array referenced by b must be generated at the entry of *tridisolve*.

6 Experimental Results

To evaluate the effectiveness of our approach, we set up experiments using benchmarks collected from disparate sources, including those from [10,22,23]. Table 1 gives a short description of the benchmarks together with a summary of the results of our analyses, which we discuss in more detail in the following subsections. For all our experiments, we ran the benchmarks with their smallest input size on an AMD Athlon™ 64 X2 Dual Core Processor 3800+, 4GB RAM computer running Linux operating system; GNU Octave, version 3.2.4; MATLAB, version 7.9.0.529 (R2009b) and McVM/McJIT, version 0.5.

The purpose of our experiments was three-fold. First, we wanted to measure the number of array updates and copies performed by the benchmarks at runtime using existing systems (Sec. 6.1). Knowing the number of updates gives an idea of how many dynamic checks a reference-counting-based (RC) scheme for lazy copying, such as used by Octave and Mathworks’ MATLAB, need to perform. Recall that our approach does not usually require any dynamic checks. Knowing the number of copies generated by such systems allows us to verify that our approach does not increase the number of copies as compared to the RC approaches. Secondly, we would like to measure the amount of overhead generated in RC systems (Sec. 6.2). Finally, we would like to assess the impact of our static analyses in terms of their ability to minimize the number of copies (Sec. 6.3).

6.1 Dynamic Counts of Array Updates and Copies

Our first measurements were designed to measure the number of array updates and array copies that are required by existing RC systems, Octave and Mathworks’ MATLAB. Since we had access to the open-source Octave system we were able to instrument the interpreter and make the measurements directly. However, the Mathworks’ implementation of MATLAB is a proprietary system and thus we were unable to instrument it to make direct measurements. Instead, we developed an alternative approach by instrumenting the benchmark programs themselves via aspects using our ASPECTMATLAB compiler *amc* [5]. Our aspect [6] defines all the patterns for the relevant points in a MATLAB program including all array definitions, array updates, and function calls. It also specifies the actions that should be taken at these points in the source program. In effect, the aspect computes all of the information that a RC scheme would have, and thus can determine, at runtime, when an array update triggers a copy because the number of references to the array is greater than one. The aspect thus counts all array updates and all copies that would be required by a RC system.

Table 1. Benchmarks and the results of the copy analysis [6]

Benchmark	# Array Updates	# Copies					
		Lower Bound		With Analyses			
		Aspect	Octave	Naive	QC	CA	
adpt	adaptive quadrature using Simpson’s rule	19624	0	0	12223	12223	0
capr	capacitance of a transmission line using finite difference and Gauss-Seidel iteration	9790800	10000	10000	40000	20000	10000
clos	transitive closure of a directed graph	2954	0	0	2	2	0
crni	Crank-Nicholson solution to the one-dimensional heat equation	21143907	4598	6898	11495	6897	4598
dich	Dirichlet solution to Laplace’s equation	6935292	0	0	0	0	0
fdtd	3D FDTD of a hexahedral cavity with conducting walls	803	0	0	5400	5400	0
fft	fast fourier transform	44038144	1	1	2	2	1
fiff	finite-difference solution to the wave equation	12243000	0	0	0	0	0
mbrt	mandelbrot set	5929	0	0	0	0	0
nb1d	N-body problem coded using 1d arrays for the displacement vectors.	55020	0	0	10984	10980	0
nb3d	N-body problem coded using 3d arrays for the displacement vectors.	4878	0	0	5860	5858	0
nfrc	computes a newton fractal in the complex plane $-2..2, -2i..2i$	12800	0	0	6400	6400	0
trid	Solve tridiagonal system of equations	2998	2	2	5	2	2

In Table [1] the column labelled **# Array Updates** gives the total number of array updates executed. The column **# Copies** shows the number of copies generated by the benchmarks under Octave (reported as **Octave** in the table) and MATLAB (column labelled **Aspect**). The column **# Copies** is split into two: **Lower Bound** and **With Analyses**. The number of copies generated by Octave and MATLAB (Aspect) are considered the expected lower bounds

⁵ This aspect is available at: www.sable.mcgill.ca/mclab/mcvm_mcjit.html

⁶ The benchmarks are also available at: www.sable.mcgill.ca/mclab/mcvm_mcjit.html

(since they perform copies lazily, and only when required) and are therefore grouped under *Lower Bound* in the table [7](#)

At a high-level, the results in Table [1](#) show that our benchmarks often perform a significant number of array updates, but very few updates trigger copies. We observed that no copies were generated in ten out of the thirteen benchmarks. This low rate for array copies is not surprising because MATLAB programmers tend to avoid copying large objects and often only read from function parameters. *With Analyses* comprises of three columns, **Naive**, **QC**, and **CA** representing respectively, the number of copies generated in our naive system, with the QC phase, and with the copy analysis phase. We return to these results in Sec. [6.3](#)

6.2 The Overhead of Dynamic Checks

With RC approaches a dynamic check is needed for each array update, in order to test if a copy is needed. Our counts indicated that several of our benchmarks had a high number of updates, but no copies were required. We wanted to measure the overhead for all of these redundant dynamic checks. The ideal measurement would have been to time the redundant checks in a JIT-based system that used reference-counting, such as Mathworks’ MATLAB. Unfortunately we do not have access to such a system. Instead we performed two similar experiments, as reported in Table [2](#), for three benchmarks with a high number of updates and no required copies (`dich`, `fiff` and `mbrt`).

Table 2. Overhead of Dynamic Checks

Bmark	McVM						Octave(O)		
	McJIT		McJIT(+RC)		Overhead(%)		Time(s)		Overhead
	time(s)	# LLVM	time(s)	# LLVM	time	size	O(+RC)	O(-RC)	(%)
dich	0.18	546	0.27	625	47.37	14.47	425.05	408.08	4.16
fiff	0.39	388	0.52	415	33.72	6.96	468.64	438.69	6.83
mbrt	5.06	262	5.65	271	11.69	3.44	34.91	31.95	9.29

We first created a version of Octave that does not insert dynamic checks before array update statements. In general this is not safe, but for these three benchmarks we knew no copies were needed, and thus removing the checks allowed us to measure the overhead without breaking the benchmarks. The column labelled **O(+RC)** gives the execution time with dynamic checks and the column labelled **O(-RC)** gives the times when we artificially removed the checks. The difference gives us the overhead, which is between 4% and 9% for these benchmarks. Although this is not a huge percentage, it is not negligible. Furthermore, we felt that the absolute time for the checks was significant and would be even more significant in a JIT system which has many fewer other overheads.

To measure overheads in a JIT context, we modified McJIT to include enough reference-counting machinery to measure the overhead of the checks (remember

⁷ Note that for the benchmark `crni` Octave performs 6898 copies, whereas the lower bound according to the Aspect is 4598. We verified that Octave is doing some spurious copies in this case, and that the Aspect number is the true lower bound.

that McVM is garbage-collected and does not normally have reference counts). For the modified McVM we added a field to the array object representation to store reference counts (which is set to zero for the purposes of this experiment) and we generated LLVM code for a runtime check before each array update statement. Table 2 shows, in time and code size, the amount of overhead generated by redundant checks. The column labelled **McJIT** is the original McJIT and the column labelled **McJIT(+RC)** is the modified version with the added checks. We measured code size using the number of LLVM instructions (**# LLVM**) and execution time overhead in seconds. For these benchmarks the code size overhead was 3% to 14% and the running time overhead ranged from 12% to 47%.

Our conclusion is that the dynamic checks for a RC scheme can be quite significant in both execution time and code size, especially in the context of a JIT. Thus, although the original motivation of our work was to enable a garbage-collected VM that did not require reference counts, we think that our analyses could also be useful to eliminate unneeded checks in RC systems.

6.3 Impact of Our Analyses

Let us now return to the number of copies required by our analyses, which are given in the last three columns of Table 1. As a reminder, our goal was to achieve the same number of copies as the lower bound.

The column labelled **Naive** gives the number of copies required with a naive implementation of MATLAB’s copy semantics, where a copy is inserted for each parameter, each return value and each copy statement, where the *lhs* is an array. Clearly this approach leads to many more copies than the lower bound.

The column labelled **CA** gives the number of copies when both phases of our static analyses are enabled. We were very pleased to see that for our benchmarks, the static analyses achieved the same number of copies as the lower bound, without requiring any dynamic checks. The column labelled **QC** shows the number of copies when only the QuickCheck phase is enabled. Although the QuickCheck does eliminate many unneeded copies, it does not achieve the lower bound. Thus, the second stage is really required in many cases.

To show the impact copies have on execution performance, we measured the total bytes of array data copied by a benchmark together with its corresponding execution time. These are shown in Fig. 4 and Table 3 for *Naive*, *QC* and *CA*. The columns $\frac{Naive}{QC}$ and $\frac{Naive}{CA}$ of Table 3 show respectively how many times QC and CA perform better than *Naive*. The table shows that *CA* generally outperforms *QC* and *Naive*. Copying large arrays affects execution performance and the results in Table 3 validate this claim. Where a significant number of bytes were copied by the naive implementation, for example, *capr*, *crni* and *fdtd*, *CA* performs better than both *Naive* and *QC*. In the three benchmarks that do not generate copies, the performance of *CA* is comparable to *Naive* and *QC*. This shows that the overhead of *CA* is low. It is therefore clear from the results of our experiments that the naive implementation generates significant overhead and is therefore unsuitable for an high-performance system.

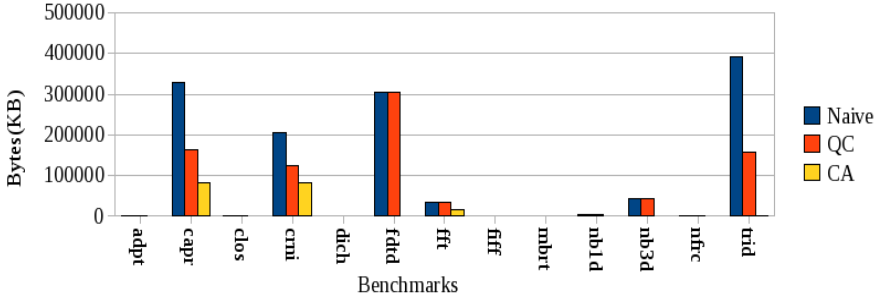


Fig. 4. The total bytes of array data copied by the benchmarks under the three options

Table 3. Benchmarks against the total execution times in seconds

Bmark	Naive	QC	CA	$\frac{Naive}{QC}$	$\frac{Naive}{CA}$	Bmark	Naive	QC	CA	$\frac{Naive}{QC}$	$\frac{Naive}{CA}$
adpt	1.57	1.57	1.61	1.00	0.98	fiff	0.39	0.39	0.39	0.99	0.99
capr	1.54	0.91	0.58	1.70	2.66	mbrt	5.06	5.12	5.04	0.99	1.00
clos	0.49	0.49	0.48	0.99	1.01	nb1d	0.48	0.48	0.45	1.00	1.07
crni	135.09	140.35	131.62	0.96	1.03	nb3d	0.48	0.48	0.36	1.00	1.35
dich	0.18	0.18	0.18	1.00	1.00	nfrc	3.23	3.23	3.25	1.00	0.99
fdtd	3.79	3.78	2.80	1.00	1.35	trid	1.57	1.04	1.02	1.51	1.53
fft	1.50	1.50	1.47	1.00	1.02						

Impact of the First Phase. We measured the number of functions that are completely resolved by the first phase of our approach — in terms of finding all the necessary copies required to guarantee copy semantics. We found that out of the 23 functions in the benchmark set, the first stage (i.e., QuickCheck) was only able to resolve about 17% of the functions. None of the benchmarks was resolved completely by QC. The main reason for this poor performance is that the first phase cannot resolve functions that return arrays to their callers. And like most MATLAB programs, most of the functions in the benchmarks return arrays. This really shows that the second stage is actually required to completely determine the needed copies by typical MATLAB programs.

So, the bottom line is that a very low fraction of array updates result in copies, and frequently no copies are necessary. For our benchmark set, our static analysis determined the needed number of copies, while at the same time avoiding all the overhead of dynamic checks. Furthermore, our approach does not require reference counting and thus enables an efficient implementation of array copy semantics in garbage-collected systems like McVM.

7 Related Work

Redundant copy elimination is a hard problem and implementations of languages such as Python [3] are able to avoid copy elimination optimizations by providing

multiple data structures: some with copy semantics and others with reference semantics. Programmers decide when to use mutable data structures. However, efficient implementations of languages like the MATLAB programming language that use copy semantics require copy elimination optimization. The problem is similar to the aggregate update problem in functional languages [12, 14, 21, 24, 26]. To modify an aggregate in a strict functional language, a copy of the aggregate must be made. This is in contrast with the imperative programming languages where an aggregate may be modified multiple times.

APL [15] is one of the oldest array-based languages. Weigang [27] describes a range of optimizations for APL compiler, including a copy optimization that finds uses of a copy of a variable and replaces the copy with the original variable wherever possible. We implemented this optimization as part of our QuickCheck phase. We found the optimization effective at enabling the elimination of redundant copy statements by the dead-code optimizer. However, this optimization is unable to eliminate redundant copies of arguments and return values. Hudak and Bloss [14] use an approach based on abstract interpretation and conventional flow analysis to detect cases where an aggregate may be modified in place. Their method combines static analysis and dynamic techniques. It involves a rearrangement of the execution order or an optimized version of reference counting, where the static analysis fails. Our approach is based on flow analysis but we do not change the execution order of a program.

Interprocedural aliasing and the side-effect problem [20] is related to the copy elimination problem. By using call by reference semantics, when an argument is passed to a function during a call, the parameter becomes an alias for the argument in the caller and if the argument contains an array reference, the referenced array becomes a shared array; any updates via the parameter in the callee updates the same array referenced by the corresponding argument in the caller. Without performing a separate and expensive flow analysis, our approach easily detects aliasing and side effects in functions. Wand and Clinger present [26] interprocedural flow analyses for aliasing and liveness based on set constraints. They present two operational semantics: the first one permits destructive updates of arrays while the other does not. They also define a transformation from a strict functional language to a language that allows destructive updates. Like Wand and Clinger, our approach combines liveness analysis with flow analysis. However, unlike Wand and Clinger, our analyses are intraprocedural and have been implemented in a JIT compiler for an imperative language.

The work of Goyal and Paige [13] on copy optimization for SETL [25] is particularly interesting. Their approach combines a RC scheme with static analysis. A combination of must-alias and live-variable analyses is used to identify dead variables and the program points where a statement that redefines a dead variable can be inserted to facilitate destructive updates. Like our approach, this technique is capable of eliminating the redundant copying of a shared location that can occur during an update of the location; however, it is different from our approach. In particular, it generates dynamic checks to detect when to make copies. As mentioned in Sec. 6, our approach rarely generates dynamic checks.

8 Conclusions and Future Work

In this paper we have presented an approach for using static analysis to determine where to insert array copies in order to implement the array copy semantics in MATLAB. Unlike previous approaches, which used a reference-counting scheme and dynamic checks, our approach is implemented as a pair of static analysis phases in the McJIT compiler. The first phase implements simple analyses for detecting read-only parameters and standard copy elimination, whereas the second phase consists of a forward *necessary copy analysis* that determines which array update statements trigger copies, and a backward *copy placement analysis* that determines good places to insert the array copies. All of these analyses have been implemented as structured-based analyses on the McJIT intermediate representation.

Our approach does not require frequent dynamic checks, nor do we need the space and time overheads to maintain the reference counts. Our approach is particularly appealing in the context of a garbage-collected VM, such as the one we are working with. However, similar techniques could be used in a reference-counting-based system to remove redundant checks. Our experimental results validate that, on our benchmark set, we do not introduce any more copies than the reference-counting approach, and we eliminate all dynamic checks.

References

1. GNU Octave, <http://www.gnu.org/software/octave/index.html>
2. McLab, <http://www.sable.mcgill.ca/mclab/>
3. Python, <http://www.python.org>
4. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide, 3rd edn. SIAM, Philadelphia (1999)
5. Aslam, T., Doherty, J., Dubrau, A., Hendren, L.: AspectMatlab: An Aspect-Oriented Scientific Programming Language. In: Proceedings of the 9th International Conference on Aspect-Oriented Software Development, pp. 181–192 (March 2010)
6. Boehm, H., Spertus, M.: N2310: Transparent Programmer-Directed Garbage Collection for C++ (June 2007), <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2310.pdf>
7. Casey, A., Hendren, L.: MetaLexer: A Modular Lexical Specification Language. In: Proceedings of the 10th International Conference on Aspect-Oriented Software Development (March 2011)
8. Chevalier-Boisvert, M.: McVM: An Optimizing Virtual Machine for the MATLAB Programming Language. Master's thesis, McGill University (August 2009)
9. Chevalier-Boisvert, M., Hendren, L., Verbrugge, C.: Optimizing MATLAB through Just-In-Time Specialization. In: International Conference on Compiler Construction, pp. 46–65 (March 2010)
10. Moler, C.: Numerical Computing with MATLAB. SIAM, Philadelphia (2004)
11. Ekman, T., Hedin, G.: The Jastadd Extensible Java Compiler. In: OOPSLA 2007: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, pp. 1–18. ACM, New York (2007)

12. Gopinath, K., Hennessy, J.L.: Copy Elimination in Functional Languages. In: POPL 1989: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 303–314. ACM, New York (1989)
13. Goyal, D., Paige, R.: A New Solution to the Hidden Copy Problem. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 327–348. Springer, Heidelberg (1998)
14. Hudak, P., Bloss, A.: The Aggregate Update Problem in Functional Programming Systems. In: POPL 1985: Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 300–314. ACM, New York (1985)
15. Iverson, K.E.: A Programming Language. John Wiley and Sons, Inc., Chichester (1962)
16. Lameed, N., Hendren, L.: Staged Static Techniques to Efficiently Implement Array Copy Semantics in a MATLAB JIT Compiler. Technical Report SABLE-TR-2010-5, School of Computer Science, McGill University, Montréal, Canada (July 2010)
17. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: CGO 2004: Proceedings of the International Symposium on Code Generation and Optimization, p. 75. IEEE Computer Society, Washington, DC, USA (2004)
18. Li, J.: McFor: A MATLAB to FORTRAN 95 Compiler. Master’s thesis, McGill University (August 2009)
19. MathWorks. MATLAB Programming Fundamentals. The MathWorks, Inc. (2009)
20. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco (1997)
21. Odersky, M.: How to Make Destructive Updates Less Destructive. In: POPL 1991: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 25–36. ACM, New York (1991)
22. Press, H.W., Teukolsky, A.S., Vetterling, T.W., Flannery, P.B.: Numerical Recipes: the Art of Scientific Computing. Cambridge University Press, Cambridge (1986)
23. Rose, L.D., Gallivan, K., Gallopoulos, E., Marsolf, B.A., Padua, D.A.: FALCON: A MATLAB Interactive Restructuring Compiler. In: Huang, C.-H., Sadayappan, P., Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) LCPC 1995. LNCS, vol. 1033, pp. 269–288. Springer, Heidelberg (1996)
24. Sastry, A.V.S.: Efficient Array Update Analysis of Strict Functional Languages. PhD thesis, University of Oregon, Eugene, USA (1994)
25. Schwartz, J.T., Dewar, R.B., Schonberg, E., Dubinsky, E.: Programming with Sets; an Introduction to SETL. Springer, New York (1986)
26. Wand, M., Clinger, W.D.: Set Constraints for Destructive Array Update Optimization. *Journal of Functional Programming* 11(3), 319–346 (2001)
27. Weigang, J.: An Introduction to STSC’s APL Compiler. *SIGAPL APL Quote Quad* 15(4), 231–238 (1985)
28. Whaley, R.C., Petitot, A., Dongarra, J.J.: Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing* 27(1-2), 3–35 (2001)

SSA-Based Register Allocation with PBQP

Sebastian Buchwald, Andreas Zwinkau, and Thomas Bersch

Karlsruhe Institute of Technology (KIT)

{buchwald,zwinkau}@kit.edu,
thomas.bersch@student.kit.edu

Abstract. Recent research shows that maintaining SSA form allows to split register allocation into separate phases: spilling, register assignment and copy coalescing. After spilling, register assignment can be done in polynomial time, but copy coalescing is NP-complete. In this paper we present an assignment approach with integrated copy coalescing, which maps the problem to the Partitioned Boolean Quadratic Problem (PBQP). Compared to the state-of-the-art recoloring approach, this reduces the relative number of swap and copy instructions for the SPEC CINT2000 benchmark to 99.6% and 95.2%, respectively, while taking 19% less time for assignment and coalescing.

Keywords: register allocation, copy coalescing, PBQP.

1 Introduction

Register allocation is an essential phase in any compiler generating native code. The goal is to map the program variables to the registers of the target architecture. Since there are only a limited number of registers, some variables may have to be spilled to memory and reloaded again when needed. Common register allocation algorithms like Graph Coloring [3,6] spill on demand during the allocation. This can result in an increased number of spills and reloads [11]. In contrast, register allocation based on static single assignment (SSA) form allows to completely decouple the spilling phase. This is due to the live-range splits induced by the ϕ -functions, which render the interference graph chordal and thus ensure that the interference graph is k -colorable, where k is the maximum register pressure.

The live-range splits may result in *shuffle code* that permutes the values (as variables are usually called in SSA form) on register level with copy or swap instructions. To model this circumstance, *affinities* indicate which values should be *coalesced*, i.e. assigned to the same register. If an affinity is fulfilled, inserting shuffle code for the corresponding values is not necessary anymore. Fulfilling such affinities is the challenge of copy coalescing and a central problem in SSA-based register allocation.

In this work we aim for a register assignment algorithm that is aware of affinities. In order to model affinities, we employ the Partitioned Boolean Quadratic Problem (PBQP), which is a generalization of the graph coloring problem. In the following, we

- employ the chordal nature of the interference graph of a program in SSA form to obtain a linear PBQP solving algorithm, which does not spill registers, but uses a decoupled phase instead.
- integrate copy coalescing into the PBQP modelling, which makes a separate phase afterwards unnecessary.
- develop a new PBQP reduction to improve the solution quality by merging two nodes, if coloring one node implies a coloring of the other one.
- introduce an advanced technique to handle a wide class of register constraints during register assignment, which enlarges the solution space for the PBQP.
- show the competitiveness of our approach by evaluating our implementation for quality and speed. Additionally, some insight into our adaptations is gained by interpreting our measurements.

In [Section 2](#) we describe related register allocation approaches using either SSA form or a PBQP-based algorithm, before we combine both ideas in [Section 3](#), which shows the PBQP algorithm and our adaptations in detail. [Section 4](#) presents our technique to handle register constraints. Afterwards, an evaluation of our implementation is given in [Section 5](#). Finally, [Section 6](#) describes future work and [Section 7](#) our conclusions.

2 Related Work

2.1 Register Allocation on SSA Form

Most SSA-based compilers destruct SSA form in their intermediate representation after the optimization phase and before the code generation. However, maintaining SSA form provides an advantage for register allocation: Due to the live-range splits that are induced by the ϕ -functions, the interference graph of programs in SSA form is *chordal* [\[4\]\[13\]](#), which means every induced subgraph that is a cycle, has length three. For chordal graphs the chromatic number is determined by the size of the largest clique. This means that the interference graph is k -colorable, if the spiller has reduced the register pressure to at most k at each program point. Thus, spilling can be decoupled from assignment [\[13\]](#), which means that the process is not iterated as with Graph Coloring.

To color the interference graph we employ the fact that there is a *perfect elimination order* (PEO) for each chordal graph [\[7\]](#). A PEO defines an ordering $<$ of nodes of a graph G , such that each successively removed node is *simplicial*, which means it forms a clique with all remaining neighbors in G . After spilling, assigning the registers in reverse PEO ensures that each node is simplicial and thus has a free register available. Following Hack, we obtain a PEO by a post-order walk on the dominance tree [\[11\]](#).

In addition to ϕ -functions, live-range splits may originate from constrained instructions. For instance, if a value is located in register $R1$, but the constrained instruction needs this value in register $R2$, we can split the live-range of this value. More generally, we split all live-ranges immediately before the constrained instruction to allow for unconstrained copying of values into the required registers and employ copy coalescing to remove the overhead afterwards. With this in mind, we add *affinity edges* to the interference graph, which represent that the incident nodes should be assigned to the same register. Each affinity has assigned costs, which can be weighted by the execution frequency of the potential copy instruction. The goal of *copy coalescing* is to find a coloring that minimizes the costs of unfulfilled affinities. Bouchez et al. showed that copy coalescing for chordal graphs is NP-complete [1], so one has to consider the usual tradeoff between speed and quality.

Grund and Hack [10] use integer linear programming to optimally solve the copy coalescing problem for programs in SSA form. To keep the solution time bearable some optimizations are used, but due to its high complexity the approach is too slow for practical purposes.

In contrast, the recoloring approach from Hack and Goos [12] features a heuristic solution in reasonable time. It improves an existing register assignment by recoloring the interference graph. Therefore, an initial coloring can be performed quickly without respect to quality. To achieve an improvement, the algorithm tries to assign the same color to affinity-related nodes. If changing the color of a node leads to a conflict with interfering neighbors, the algorithm tries to solve this conflict by changing the color of the conflicting neighbors. This can cause conflicts with other neighbors and recursively lead to a series of color changes. These changes are all done temporarily and will only be accepted, if a valid recoloring throughout the graph is found.

Another heuristic approach is the preference-guided register assignment introduced by Braun et al. [2]. This approach works in two phases: in the first phase the register preferences of instructions are determined and in the second phase registers are assigned in consideration of these preferences. The preferences serve as implicit copy coalescing, such that no extra phase is needed. Furthermore, the approach does not construct an interference graph. Preference-guided register assignment is inferior to recoloring in terms of quality, but significantly faster.

2.2 PBQP-Based Register Allocation

The idea to map register allocation to PBQP was first implemented by Scholz and Eckstein [16] and features a linear heuristic. Since their implementation does not employ SSA form, they need to integrate spilling into their algorithm. Hames and Scholz [14] refine the approach by presenting a new heuristic and a branch-and-bound approach for optimal solutions. The essential advantage of the PBQP approach is its flexibility, which makes it suited for irregular architectures.

3 PBQP

3.1 PBQP in General

The PBQP is a special case of a Quadratic Assignment Problem and essentially consists of multiple interdependent choices with associated costs. While the problem can be expressed formally [8], a graph-based approach is more intuitive. Each graph node has an associated choice vector, which assigns each alternative its cost in \mathbb{R} . For each node only one alternative can be selected. Interdependencies are modeled by directed edges with an associated matrix, which assigns a cost to each combination of alternatives in $\mathbb{R} \cup \{\infty\}$. Naturally, a node with n choices and one with m choices have cost vectors of dimension n and m , respectively. An edge between them must have a corresponding matrix of size $n \times m$. If we select the i -th alternative at the source node and the j -th alternative at the target node, we implicitly select the entry at the i -th row and j -th column of the edge matrix.

A *selection* assigns each node one of its alternatives. This also implies a matrix entry for each edge. The cost of a selection is the sum of all chosen vector and matrix entries. If this cost is finite, the selection is called a *solution*. The goal of the PBQP is to find a solution with minimal cost.

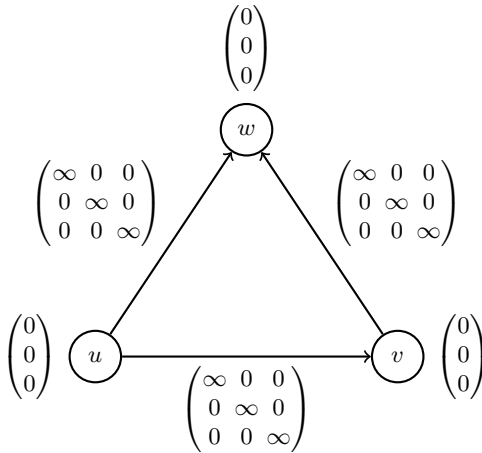


Fig. 1. PBQP instance for 3-coloring

Figure 1 shows a PBQP instance for 3-coloring. Each node u , v , and w has three alternatives that represent its possible colors. Due to the edge matrices, each solution of the PBQP instance has to select different colors for adjacent nodes and thus represents a valid 3-coloring. The reduction from 3-coloring to PBQP renders PBQP NP-complete. Additionally, it shows that finding *any* PBQP solution is NP-complete, since each solution is optimal. However, our algorithm can still solve all of our specific problems in linear time as we show in **Section 3.4**.

3.2 PBQP Construction

As mentioned above, SSA-based register allocation allows to decouple spilling and register assignment, such that register assignment is essentially a graph coloring problem. In [Figure 1](#) we show that PBQP can be considered as a generalization of graph coloring by employing color vectors and interference matrices:

Color vector. The color vector contains one zero cost entry for each possible color.

$$(0\ 0\ 0)^T$$

Interference matrix. The matrix costs are set to ∞ , if the corresponding colors are equal. Otherwise, the costs are set to zero.

$$\begin{pmatrix} \infty & 0 & 0 \\ 0 & \infty & 0 \\ 0 & 0 & \infty \end{pmatrix}$$

Register allocation can be considered as a graph coloring problem by identifying registers and colors. In order to integrate copy coalescing into register assignment, we add affinity edges for all potential copies, which should be prevented.

Affinity matrix. The matrix costs are set to zero if the two corresponding registers are equal. Otherwise, the costs are set to a positive value that represents the cost of inserted shuffle code.

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

A PBQP instance is constructed like an interference graph by inspecting the live-ranges of all values. The values become nodes and get their corresponding color vectors assigned. For each pair of interfering nodes an interference matrix is assigned to the edge between them. Likewise, the affinity cost matrix is assigned to an edge between a pair of nodes, which should get the same register assigned. If a pair of nodes fulfills both conditions, then the sum of both cost matrices is assigned. While the copy cannot be avoided in this case, there may still be cost differences between the register combinations.

3.3 Solving PBQP Instances

Solving the PBQP is done by iteratively reducing an instance to a smaller instance until all interdependencies vanish. Without edges every local optimum is also globally optimal, so finding an optimal solution is trivial (unless there is none). By backpropagating the reductions, the selection of the smaller instance can be extended to a selection of the original PBQP instance. Originally, there are four reductions [\[5,9,16\]](#):

RE: Independent edges have a cost matrix that can be decomposed into two vectors \mathbf{u} and \mathbf{v} , i.e. each matrix entry C_{ij} has costs $u_i + v_j$. Such edges can be removed after adding \mathbf{u} and \mathbf{v} to the cost vector of the source and target node, respectively. If this would produce infinite vector costs, the corresponding alternative (including matrix rows/columns) is deleted.



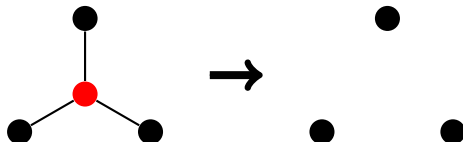
R1: Nodes of degree one can be removed, after costs are accounted in the adjacent node.



R2: Nodes of degree two can be removed, after costs are accounted in the cost matrix of the edge between the two neighbors; if necessary, the edge is created first.



RN: Nodes of degree three or higher. For a node u of maximum degree we select a locally minimal alternative, which means we only consider u and its neighbors. After the alternative is selected, all other alternatives are deleted and the incident edges are independent, so they can be removed by using RE.



RE, R1, and R2 are *optimal* in the sense that they transform the PBQP instance to a smaller one with equal minimal costs, such that each solution of the small instance can be extended to a solution of the original instance with equal costs. For sparse graphs these reductions are very powerful, since they diminish the problem to its core. If the entire graph can be reduced by these reductions, then the resulting PBQP selection is optimal. If nodes of degree three or higher remain, the heuristic RN ensures linear time behavior.

Hames et al. [14] introduced a variant of RN that removes the corresponding node from the PBQP graph without selecting an alternative. Thus, the decision is delayed until the backpropagation phase. We will refer to this approach as *late decision*. The other approach is *early decision*, which colors a node during RN. In this paper we follow both approaches and we will show which one is more suitable for SSA-based register allocation.

3.4 Adapting the PBQP Solver for SSA-Based Register Allocation

In the context of SSA-based register allocation, coloring can be guaranteed to succeed, if done in reverse PEO with respect to the interference graph. This

seems to imply that our PBQP solver must assign registers in reverse PEO. However, we show in the following that this restriction is only necessary for heuristic reductions. Therefore, choosing a node for heuristic decision must use the last node of the PEO for early decision and the first node of the PEO for late decision. This different selection of nodes is needed, because the backpropagation phase inverts the order of the reduced nodes.

Early application of optimal reductions. There is a conflict between the necessity to assign registers in reverse PEO and the PBQP strategy to favor RE, R1, and R2 until only RN is left to apply. Fortunately, we can show that the application of this reductions preserves the PEO property with [Theorem 1](#) below.

Lemma 1. *Let P be a PEO of a graph $G = (V, E)$ and $H = (V', E')$ an induced subgraph of G , then $P|_{V'}$ is a PEO of H .*

Proof. For any node $v \in H$ let $V_v = \{u \in N_G(v) \mid u > v\}$ be the neighbor nodes in G behind in P and respectively $V'_v = \{u \in N_H(v) \mid u > v\}$. By definition $V'_v \subseteq V_v$. V_v is a clique in G , therefore V'_v is a clique in H and v is simplicial, when eliminated according to $P|_{V'}$. \square

From [Lemma 1](#) we know, that R1 preserves the PEO, since the resulting graph is always an induced subgraph. However, R2 may insert a new edge into the PBQP graph.

Lemma 2. *Let P be a PEO of a graph $G = (V, E)$ and $v \in V$ a vertex of degree two with neighbors $u, w \in V$. Further, let $H = (V', E')$ be the subgraph induced by $V \setminus \{v\}$. Then $P|_{V'}$ is a PEO of $H' = (V', E' \cup \{\{u, w\}\})$.*

Proof. If $\{u, w\} \in E$ this follows directly from [Lemma 1](#), since no new edge is introduced. In the other case, we assume without loss of generality $u < w$. Since $\{u, w\} \notin E$, v is not simplicial and we get $u < v$. Therefore, the only neighbor node of u in H behind in P must be v . Within H' the node w is the only neighbor of u behind in the PEO, hence u is simplicial. For the remaining nodes v and w the lemma follows directly from [Lemma 1](#). \square

With [Lemma 2](#) the edge inserted by R2 is proven harmless, so we can derive the necessary theorem now. Remember that the PEO must be derived from the interference graph, while our PBQP graph may also include affinity edges.

Theorem 1. *Let $G = (V, E)$ be a PBQP graph, $i(E)$ the interference edges in E , i.e. edges that contain at least one infinite cost entry, P a PEO of $(V, i(E))$ and $H = (V', E')$ the PBQP graph G after exhaustive application of RE, R1 and R2. Further, let E_i be the interference edges that are reduced by RE. Then, $P|_{V'}$ is a PEO of $(V', i(E') \cup E_i)$.*

Proof. If at least one affinity edge is involved in a reduction, there is no new interference edge and we can apply [Lemma 1](#). If we consider only interference edges, [Lemma 1](#) handles R1 and [Lemma 2](#) handles R2. Applying RE for an interference edge moves the interference constraint into incident nodes. Thus, we have to consider such edges E_i for the PEO. \square

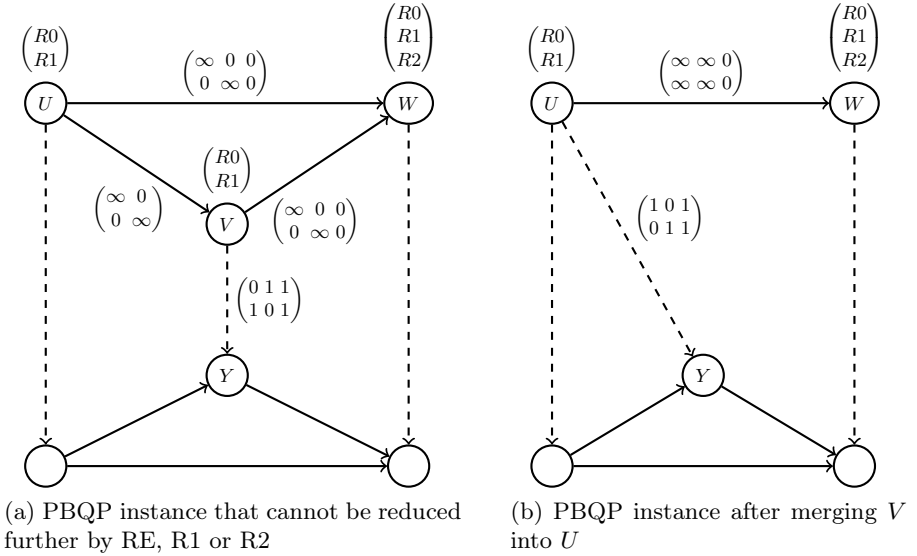


Fig. 2. Example of an RM application

Merging PBQP nodes. The PBQP instances constructed for register assignment contain many interference cliques, which cannot be reduced by optimal reductions. This implies that the quality of the PBQP solution highly depends on the RN decisions. In this section we present *RM*, a new PBQP reduction that is designed to improve the quality of such decisions.

Figure 2a shows a PBQP instance that cannot be further reduced by application of RE, R1 or R2. Hence, we have to apply a heuristic reduction. We assume that the PEO selects U to be the heuristically reduced node. The new reduction is based on the following observation: If we select an alternative at U , there is only one alternative at V that yields finite costs. Thus, a selection at U implicitly selects an alternative at V . However, the affinities of V are not considered during the reduction of U . The idea of the new reduction is to merge the neighbor V into U . After the merge, U is also aware of V 's affinities which may improve the heuristic decision.

To perform the merge, we apply the RM procedure of Algorithm 1 with arguments $v = V$ and $u = U$. In line 2 the algorithm chooses $w = Y$ as adjacent node. We now want to replace the affinity edge (v, w) by an edge (u, w) . In lines 4-9 we create a new matrix Δ for the edge (u, w) . To construct this matrix we use the fact that the selection of an alternative i at u also selects an alternative i_v at v . Thus, the i -th row of Δ is the i_v -th row of C_{vw} . For our example this means that we have to swap the rows for $R0$ and $R1$ in order to obtain Δ from C_{vw} . Since (u, w) does not exist, we create the edge with the matrix Δ . Afterwards we delete the old edge (v, w) .

Algorithm 1. RM merges the node v into the node u . The notation is adopted from [16].

Require: a selection at u implies a specific selection at v .

```

1: procedure RM( $v, u$ )
2:   for all  $w \in \text{adj}(v)$  do
3:     if  $w \neq u$  then
4:       for  $i \leftarrow 1$  to  $|c_u|$  do
5:          $c \leftarrow \mathbf{0}$ 
6:         if  $c_u(i) \neq \infty$  then
7:            $i_v \leftarrow i_{\min}(C_{uv}(i, :) + c_v)$ 
8:            $c \leftarrow C_{vw}(i_v, :)$ 
9:            $\Delta(i, :) \leftarrow c$ 
10:         $C_{uw} \leftarrow C_{uw} + \Delta$ 
11:        remove edge  $(v, w)$ 
12:   ReduceI( $v$ )

```

In the next iteration, the algorithm chooses $w = W$ as adjacent node. Similar to the previous iteration, we compute the matrix Δ . Since the edge (u, w) exists, we have to add Δ to the matrix C_{uw} . After the deletion of (v, w) , the node v has degree one and can be reduced by employing R1. Figure 2b shows the resulting PBQP instance. Due to the merge the edge (U, W) is independent. After removing the edge the PBQP instance can be solved by applying R1 and R2, leading to an optimal solution.

Although RM is an optimal reduction, we only apply it immediately before a heuristic reduction at a node U . If there is a neighbor V of U that can be merged into U we apply RM for these two nodes. This process iterates until no such neighbor is left. In some cases—like our example—the RM allows further optimal reductions that supersede a heuristic reduction at U . If U still needs a heuristic reduction, the neighbors of the merged nodes are also considered by the heuristic and thus can improve the heuristic decision.

The reason for applying RM only immediately before a heuristic decision at a node u is that in this case each edge is reassigned only once, due to the fact that the node u (and all incident edges) will be deleted after the merge. Thus, each edge is considered at most twice: once for reassignment, and once during the reduction of u . As a result, the overall RM time complexity is in $\mathcal{O}(mk^2)$ where $m = |E|$ is the number of edges in the PBQP graph and k is the maximum number of alternatives at a node. The same argument can be used to show that the overall RN time complexity is in $\mathcal{O}(mk^2)$. Together with the existing optimal reductions this leads to an overall time complexity in $\mathcal{O}(nk^3 + mk^2)$ where $n = |V|$ denotes the number of nodes in the PBQP graph [5, 16]. Since k is the constant number of registers in our case, the solving algorithm has linear time complexity.

Similar to the other PBQP reductions, RM modifies the PBQP graph and thus we have to ensure that our PEO is still valid for the resulting graph.

Theorem 2. *Let $G = (V, E)$ be a PBQP graph, $i(E)$ the interference edges in E , P a PEO of $(V, i(E))$ and $H = (V', E')$ the PBQP graph G after exhaustive application of RM and the reduction of the greatest node u with respect to P . Further, let E_i be the interference edges that are reduced by RE. Then, $P|_{V'}$ is a PEO of $(V', i(E') \cup E_i)$.*

Proof. If u is reduced by RN or R1 this follows from [Lemma 1](#). In the R2 case, let v be the last node whose merge changed u 's degree. The theorem follows from applying [Lemma 1](#) for u and all nodes that are merged before v and [Lemma 2](#) for v and all nodes that are merged after v . Independent edges will be handled as in [Theorem 1](#). \square

4 Register Constraints

For SSA-based register allocation, naïve handling of register constraints can inhibit a coloring of the interference graph. For example, [Figure 3a](#) shows an instruction *instr* with three operands and two results. The operands I_1 and I_2 are live before the instruction, but are not used afterwards, so they *die* at the instruction. In contrast, operand I_3 is used afterwards (as indicated by the dashed live-range) and interferes with both results. We assume that a, b and c are the only available registers. The values are constrained to the annotated registers, for instance, the operand I_1 is constrained to registers $\{a, b\}$. However, a previous instruction may impose the constraint $\{c\}$ on I_1 . Since both constraints are contradictory, there is no coloring of the interference graph. To prevent such situations, Hack splits all live-ranges before the constrained instruction [\[11\]](#). For our example, this allows to fulfill the constraints by inserting a copy from register c to register a or b .

The next problem is that a PEO ensures a k -coloring only for unconstrained nodes. For example, we can derive the coloring order I_1, I_2, I_3, O_1, O_2 from a PEO of [Figure 3a](#), but assigning c to I_2 inhibits a valid register assignment. To tackle this issue we employ the fact that if we obtain the PEO by a post-order walk of the dominance tree, the values live before and immediately after the constrained instruction are colored first. Thus, if we provide a coloring for these nodes, we can use our PEO to color the remaining nodes. Hack showed [\[11\]](#) how such a coloring can be found in the case of *simple constraints*, i.e. if each value is either constrained to one register or unconstrained. In case of a value with a non-simple constraint, the interference cliques before and after the statement are colored separately and values with non-simple constraints are pinned to the chosen color. This may increase the register demand, but ensures a valid register allocation.

Simple constraints can easily be integrated into the PBQP solving algorithm, since we only have to ensure that operands which are live after the constrained instruction are colored first. However, pinning the values to a single register is very restrictive. In the following, we assume that the spiller enables a coloring by inserting possibly necessary copies of operands and present an algorithm that can deal with *hierarchic constraints*.

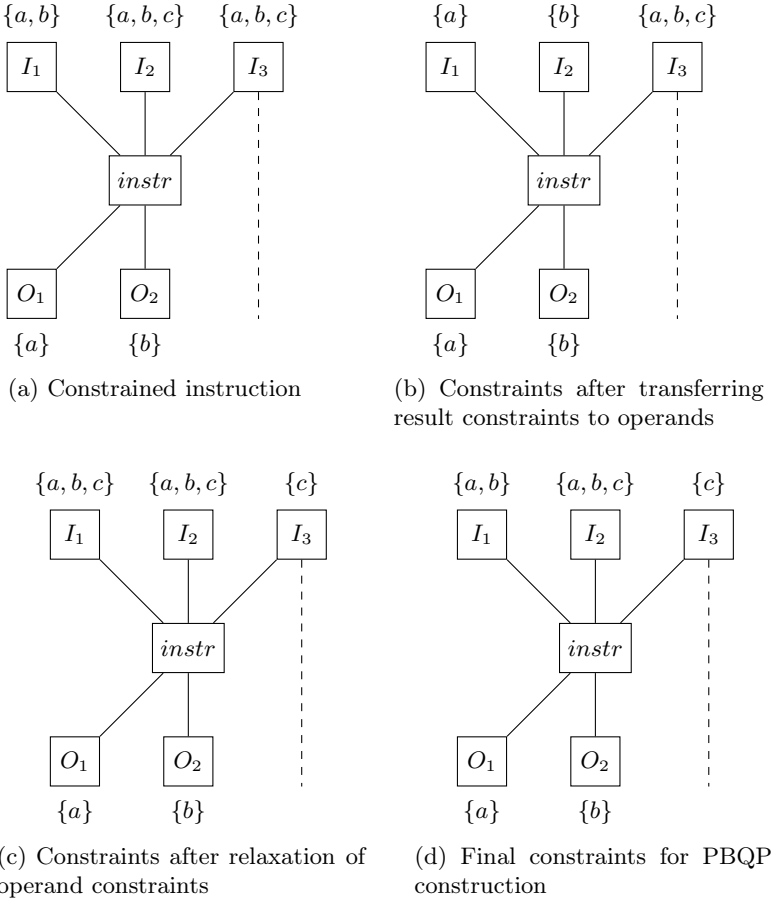


Fig. 3. Handling of constrained instructions

Definition 1 (hierarchical constraints). Let \mathcal{C} be a set of register constraints. \mathcal{C} is hierarchical if for all constraints $C_1 \in \mathcal{C}$ and $C_2 \in \mathcal{C}$ holds:

$$C_1 \cap C_2 \neq \emptyset \Rightarrow C_1 \subseteq C_2 \vee C_2 \subseteq C_1.$$

This definition excludes “partially overlapping” constraints, like $C_1 = \{a, b\}$ and $C_2 = \{b, c\}$. As a result, the constraints form a tree with respect to strict inclusion, which we call *constraint hierarchy*. For instance, the constraint hierarchy for the general purpose registers of the IA-32 architecture consists of $C_{all} = \{A, B, C, D, SI, DI\}$, a subset $C_s = \{A, B, C, D\}$ for instructions on 8- or 16-bit subregisters, and all constraints that consist of a single register.

For hierarchic constraints we obtain a valid register assignment of an interference clique by successively coloring a most constrained node. However, for a constrained instruction we also have to ensure that after coloring the values,

Algorithm 2. Restricting constraints of input operands.

```

1: procedure RESTRICTINPUTS( $Ins, Outs$ )
2:    $\mathcal{C} \leftarrow Ins \cup Outs$ 
3:   while  $\mathcal{C} \neq \emptyset$  do
4:      $C_{min} \leftarrow getMinimalElement(\mathcal{C})$ 
5:      $C_{partner} \leftarrow getMinimalPartner(\mathcal{C}, C_{min})$ 
6:      $\mathcal{C} \leftarrow \mathcal{C} \setminus \{C_{min}, C_{partner}\}$ 
7:     if  $C_{partner}$  from  $Ins$  then
8:       assign  $C_{min}$  to partner value

```

which are live before an instruction $instr$, we still can color the values live after $instr$. In [Figure 3a](#) O_1 and O_2 are constrained to a and b , respectively, and thus c must be assigned to I_3 . Unfortunately, c may also be chosen for I_2 according to its constraints, if it is colored before I_3 . To avoid such situations, we want to modify the constraints in a way that forces the first two operands to use the same registers as the results. This is done in three steps:

1. Add unconstrained pseudo operands/results until the register pressure equals the number of available registers.
2. Match results and dying operands to assign result constraints to the corresponding operand, if they are more restrictive.
3. Try to relax the introduced constraints of the previous step, to enable more affinities to be fulfilled.

The first step ensures that the number of dying operands and the number of results are equal, which is required by the second step. For our example in [Figure 3a](#) we have nothing to do, since the register pressure before and after $instr$ is already equal to the number of available registers.

4.1 Restricting Operands

We employ [Algorithm 2](#) for the second step. The algorithm has two parameters: A multiset of input constraints and a multiset of output constraints. It iteratively pairs an input constraint with an output constraint. For this pairing we select a minimal constraint (with respect to inclusion) C_{min} . Then we try to find a minimal partner $C_{partner}$, which is a constraint of the other parameter such that $C_{min} \subseteq C_{partner}$. If C_{min} is an output constraint we transfer the constraint to the partner. It is not necessary to restrict output constraints, since the inputs are colored first and restrictions propagate from there.

For our example in [Figure 3a](#) the algorithm input is $\{I_1, I_2\}$ and $\{O_1, O_2\}$. The constraints of O_1 and O_2 are both minimal. We assume that the function *getMinimalElement* chooses O_1 and thus $C_{min} = \{a\}$. Since O_1 is a result, the corresponding partner must be an operand. We select the only minimal partner I_1 which leads to $C_{partner} = \{a, b\}$. We now have our first match (I_1, O_1) and remove both values from the value sets. Since the result constraint is more restrictive, we assign this constraint to the operand I_1 . In the next iteration we

match I_2 and O_2 and restrict I_2 to $\{b\}$. The resulting constraints are shown in [Figure 3b](#). Due to the introduced restrictions, the dying operands have to use the same registers as the results.

In the following, we prove that *getMinimalPartner* always finds a minimal partner. Furthermore, we show that [Algorithm 2](#) cannot restrict the operands constraints in a way that renders a coloring of the values, which are live before the instruction, impossible.

Theorem 3. *Let $G = (V = \mathcal{I} \cup \mathcal{O}, E)$ a bipartite graph with $\mathcal{I} = \{I_1, \dots, I_n\}$ and $\mathcal{O} = \{O_1, \dots, O_n\}$. Further, let $\mathcal{R} = \{R_1, \dots, R_n\}$ be a set of colors (registers) and $\text{constr} : V \rightarrow \mathcal{P}(\mathcal{R})$ a function that assigns each node its feasible colors. Moreover, let $c : v \mapsto R_v \in \text{constr}(v)$ be a coloring of V that assigns each color to exactly one element of \mathcal{I} and one element of \mathcal{O} . Let $M \subseteq E$ be a perfect bipartite matching of G such that*

$$\{u, v\} \in M \Rightarrow c(u) = c(v).$$

Then, [Algorithm 2](#) finds a perfect bipartite matching M' such that there is a coloring $c' : v \mapsto R'_v \in \text{constr}(v)$ that assigns each color to exactly one element of \mathcal{I} and one element of \mathcal{O} with

$$\{u, v\} \in M' \Rightarrow c'(u) = c'(v).$$

Proof. We prove the theorem by induction on n . For $n = 1$ there is only one perfect bipartite matching and since $c(I_1) = c(O_1) \in (\text{constr}(I_1) \cap \text{constr}(O_1))$ we have $\text{constr}(I_1) \subseteq \text{constr}(O_1)$ or $\text{constr}(O_1) \subseteq \text{constr}(I_1)$. Thus, [Algorithm 2](#) finds the perfect bipartite matching which can be colored by $c' = c$.

For $n > 1$, without loss of generality, we can rename the nodes such that $\forall i : c(I_i) = c(O_i)$ and [Algorithm 2](#) selects O_1 as node with minimal constraints. If the algorithm selects I_1 to be the minimal partner, we can remove I_1 and O_1 from the graph, the color $c(I_1)$ from the set of colors \mathcal{R} and apply the induction assumption.

In case the algorithm does not select I_1 as minimal partner let I_p be the minimal partner. Our goal is to show that there is a coloring for

$$M'' = (M \setminus \{\{I_1, O_1\}, \{I_p, O_p\}\}) \cup \{\{I_1, O_p\}, \{I_p, O_1\}\}$$

and then apply the induction assumption. To obtain such a coloring we consider the corresponding constraints. Since O_1 has minimal constraints and $c(I_1) = c(O_1) \in (\text{constr}(I_1) \cap \text{constr}(O_1))$, we get $\text{constr}(O_1) \subseteq \text{constr}(I_1)$. Furthermore, we know that I_p is the minimal partner of O_1 which means $\text{constr}(O_1) \subseteq \text{constr}(I_p)$ by definition. Thus, we get $\emptyset \neq \text{constr}(O_1) \subseteq (\text{constr}(I_1) \cap \text{constr}(I_p))$ and since I_p is the minimal partner of O_1 , we get $\text{constr}(I_p) \subseteq \text{constr}(I_1)$. Using these relations, we obtain

$$c(O_1) \in \text{constr}(O_1) \subseteq \text{constr}(I_p)$$

$$c(O_p) = c(I_p) \in \text{constr}(I_p) \subseteq \text{constr}(I_1).$$

Thus, $c'' = c[I_1 \mapsto c(O_p), I_p \mapsto c(O_1)]$ is a coloring for M'' . We now remove $\{I_p, O_1\}$ from the graph and $c''(O_1)$ from the set of colors \mathcal{R} and apply the induction assumption, resulting in a matching M''' and a coloring c''' . Since c''' does not use the color $c''(O_1)$, we can extend the matching M''' to

$$M' = M''' \cup \{I_p, O_1\}$$

and the corresponding coloring c''' to

$$c'(v) = \begin{cases} c''(O_1) & , v \in \{I_p, O_1\} \\ c'''(v) & , \text{otherwise} \end{cases}$$

so $\{u, v\} \in M' \Rightarrow c'(u) = c'(v)$ holds. \square

4.2 Relaxing Constraints

The restriction of the operands ensures a feasible coloring. However, some of the operands may now be more restricted than necessary, so the third step relaxes their constraints again. For instance, in [Figure 3b](#) the operands I_1 and I_2 are pinned to register a and b , respectively, but assigning register b to I_1 and register a to I_2 is also feasible. To permit this additional solution, the constraints can be relaxed to $\{a, b\}$ for both operands. In the following, we provide some rules that modify the constraint hierarchy of the input operands in order to relax the previously restricted constraints. We introduce two predicates to determine whether a rule is applicable or not.

Dying. A node is dying if the live-range of the operand ends at the instruction.

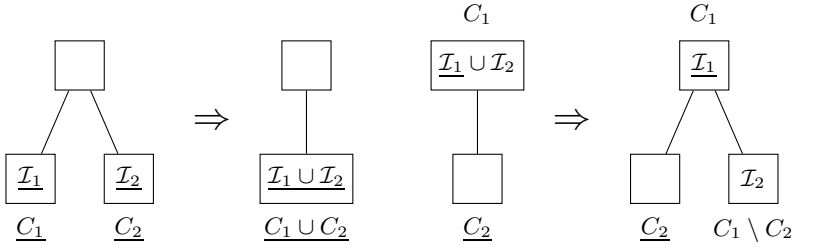
Its assigned register is available for result values.

Saturated. A constraint C is saturated, if it contains as many registers $|C|$ as there are nodes, which must get one of those registers assigned $|\{I \in \mathcal{I} \mid C_I \subseteq C\}|$. This means, every register in C will be assigned in the end.

[Figure 4](#) shows the transformation rules for constraint hierarchies. The rules are applied greedily from left to right. A constraint C_i is underlined if it is saturated. Each constraint has a set of dying nodes \mathcal{I}_i and a set of non-dying nodes \mathcal{I}_j .

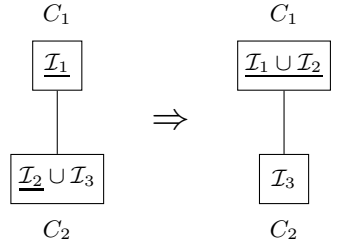
The rule shown in [Figure 4a](#) combines two saturated constraints that contain only dying nodes. Applying this rule to the constraint hierarchy of our example in [Figure 3b](#) lifts the constraints of I_1 and I_2 to $\{a, b\}$. Since both values die, it is not important which one is assigned to register a and which one to register b .

We now apply the rule of [Figure 4b](#). This rule removes registers from a node constraint if we know that these registers are occupied by other nodes, i.e. the constraints consisting of these registers are saturated. Reconsidering the example shown in [Figure 4d](#), the nodes I_1 and I_2 can only be assigned to register a and b . Thus, I_3 cannot be assigned to one of these registers and we remove them from the constraint of I_3 . The transformation removes all nodes from the upper constraint. Usually, we delete such an empty node after reconnecting all children

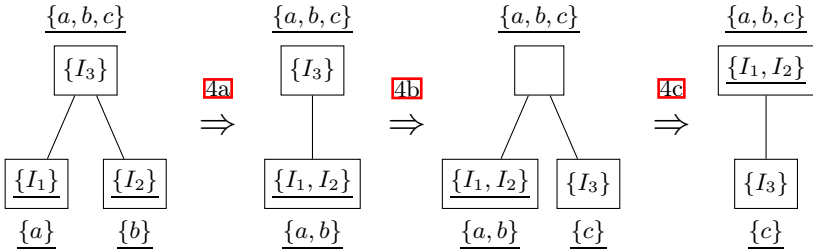


(a) Merging two saturated constraints consisting of dying nodes

(b) Restricting non-dying nodes due to a saturated constraint



(c) Moving dying nodes along the constraint hierarchy



(d) Application of the transformation rules to relax the operand constraints shown in [Figure 3b](#)

Fig. 4. Rules to relax constraints and a usage example

to its parent, because an empty node serves no purpose and the removal may enable further rule applications. However, since $\{a, b, c\}$ is the root of our tree—holding only unconstrained nodes—we keep it.

Since we want to relax the constraints of dying nodes as much as possible, the rule shown in [Figure 4c](#) moves dying nodes upwards in the constraint hierarchy. This is only allowed, if the constraint C_1 does not contain non-dying nodes. For our example of [Figure 4d](#) we relax the constraints of I_1 and I_2 further to $\{a, b, c\}$. This would be disallowed without the application of [4b](#), because a or b could then be assigned to I_3 , which would render a coloring of the results impossible.

4.3 Obtaining a Coloring Order

After exhaustive application of the transformation rules, we obtain an ordering of the constraints by a post-order traversal of the constraint hierarchy, so more constrained nodes are colored first. For example, in [Figure 4d](#) the node I_3 must be colored first due to this order. Within each constraint of the hierarchy, the associated values are further ordered with respect to a post-order traversal over the original constraint hierarchy. The second traversal ensures that “over-relaxed” values, i.e. values with a constraint that is less restrictive than their original constraint, are colored first. For our example in [Figure 4d](#) this means that we have to color I_1 before I_2 , although their relaxed constraints are equal. The final node order is I_3, I_1, I_2 . For the PBQP, we intersect the original ([Figure 3a](#)) and the relaxed constraints ([Figure 3c](#)); resulting in the constraints shown in [Figure 3d](#). We now have an order to color the values live immediately before the constrained instruction. Likewise, we obtain an order for the results by coloring the more constrained values first. Finally, we obtain a coloring order for the whole PBQP graph by employing the PEO for the remaining (unconstrained) nodes. This order ensures that our PBQP solver finds a register assignment even in presence of constrained instructions.

5 Evaluation

In this section we evaluate the impact of our adaptations. First, the late decision is compared to early decision making. Also, we investigate the effects of RM. Finally, our approach is compared to the current LIBFIRM allocator in terms of speed and result quality.

5.1 Early vs. Late Decision

As mentioned in [Section 3.3](#) we implemented early decision as well as late decision. We evaluated both approaches using the C programs of the SPEC CINT2000 benchmark suite. The programs compiled with late decision do not reach the performance of the programs compiled with early decision for any benchmark, showing a slowdown of 3.9% on average. Especially the 253.perlbnk benchmark performs nearly 20% slower.

We think that the quality gap stems from the different handling of affinities. An early decision takes account of the surrounding affinity costs and propagates them during the reduction. For a late decision a node and incident affinity edges are removed from the PBQP graph first; then the decisions at adjacent nodes are made without accounting the affinity costs. When the late decision is made, the affinities may not be fulfilled due to decisions at interference neighbors that were not aware of these affinities.

5.2 Effects of RM

We added RM to our PBQP solver and [Table 1](#) shows that 3.4% of the PBQP reductions during a SPEC compilation are RM. The number of nodes, remaining

Table 1. Percentages of reduction types

Reduction	RM disabled		RM enabled	
	Applications	Ratio	Applications	Ratio
R0	2,047,038	—	2,013,003	—
RE	126,002	—	33,759	—
R1	106,828	13.9%	94,529	11.9%
R2	363,221	47.2%	382,705	48.0%
RN	298,928	38.9%	292,872	36.7%
RM		0.0%	26,850	3.4%

after the graph is completely reduced, is given in the R0 row, but technically these nodes are not “reduced” by the solver, so they are excluded from the ratio calculation. RE is also excluded, since it reduces edges instead of nodes. The heuristic RN makes up 36.7% of the reductions, so these decisions are significant. The number of independent edge reductions decreases to nearly a fourth in total, which suggests that a significant number of RE stem from nodes, whose assignment is determined by a heuristic reduction of a neighbor. In case of RM, those edges are “redirected” to this neighbor instead. Another effect is that the number of heuristic decisions decreases by 2%. This reflects nodes that can be optimally reduced after merging the neighbors into them. Altogether, the costs of the PBQP solutions decreased by nearly 1% on average, which shows that RM successfully improved the heuristic decisions.

5.3 Speed Evaluation

To evaluate the speed of the compilation process with PBQP-based copy coalescing, we compare our approach to the recoloring approach [12]. Both approaches are implemented within the LIBFIRM compiler backend, so all other optimizations are identical. The SPEC CINT2000 programs ran on an 1.60GHz Intel Atom 330 processor on top of an Ubuntu 10.04.1 system. We timed the relevant phases within both register allocators and compare the total time taken for a compilation of all benchmark programs. The recoloring approach uses 11.6 seconds for coloring and 27.3 seconds for copy coalescing, which is 38.9 seconds in total. In contrast, the PBQP approach integrates copy coalescing into the coloring, so the coloring time equals the total time. Here, the total time amounts to 31.5 seconds, which means it takes 7.4 seconds less. Effectively, register assignment and copy coalescing are 19% faster when using the PBQP approach instead of recoloring.

5.4 Quality Evaluation

To evaluate the quality of our approach, we compare the best execution time out of five runs of the SPEC CPU2000 benchmark programs with the recoloring approach. The results in Table 2 show a slight improvement of 0.1% on average.

Table 2. Comparison of execution time in seconds with recoloring and PBQP

Benchmark	Recoloring		PBQP	Ratio
164.gzip	345	350	101.4%	
175.vpr	446	444	99.7%	
176.gcc	179	179	99.8%	
181.mcf	336	335	99.6%	
186.crafty	233	231	99.4%	
197.parser	468	467	99.7%	
253.perlbnk	355	354	99.8%	
254.gap	252	253	100.4%	
255.vortex	417	418	100.1%	
256.bzip2	374	371	99.4%	
300.twolf	684	680	99.4%	
Average				99.9%

Table 3. Dynamic copy instructions in a SPEC run (in billions)

Benchmark	PBQP			Recoloring			Ratio	
	Instr.	Swaps	Copies	Instr.	Swaps	Copies	Swaps	Copies
164.gzip	332	3.14%	0.71%	326	2.01%	0.19%	156.2%	374.7%
175.vpr	202	4.38%	0.29%	201	4.32%	0.25%	101.4%	113.8%
176.gcc	165	4.20%	0.30%	165	3.95%	0.28%	106.4%	108.9%
181.mcf	50	4.22%	0.00%	50	4.72%	0.00%	89.4%	1047207.5%
186.crafty	208	8.14%	0.56%	209	8.36%	0.64%	97.4%	87.7%
197.parser	365	4.13%	0.56%	366	4.48%	0.28%	92.2%	198.5%
253.perlbnk	396	4.64%	0.28%	408	5.97%	0.14%	77.7%	202.1%
254.gap	259	7.02%	0.08%	259	6.86%	0.40%	102.4%	20.1%
255.vortex	379	3.08%	0.53%	377	3.12%	0.16%	98.9%	339.2%
256.bzip2	295	6.30%	0.14%	298	6.15%	0.93%	102.4%	14.9%
300.twolf	306	4.80%	0.90%	306	4.33%	1.30%	110.6%	69.1%
Average	269	4.91%	0.40%	269	4.93%	0.42%	99.6%	95.2%

In addition, we assess the quality of our copy minimization approach by counting the inserted copies due to unfulfilled register affinities. We instrumented the Valgrind tool [15] to count these instructions during a SPEC run. Despite dynamic measuring, the results in Table 3 are static, because the input of the benchmark programs is static. Since the number of instructions varies between programs, we examine the percentage of copies. We observe that nearly 5% of the executed instructions are swaps and around 0.4% are copies on average. Because of the small number of copies a difference seems much higher, which results in the seemingly dramatic increase of 1047208% swaps for 181.mcf. On average the percentages decrease by 0.4% and 4.8%, respectively.

6 Future Work

Some architectures feature irregularities which are not considered in the context of SSA-based register allocation. The PBQP has been successfully used to model a wide range of these irregularities by appropriate cost matrices [16]. While the modelling can be adopted for SSA-based register assignment, guaranteeing a polynomial time solution is still an open problem.

7 Conclusion

This work combines SSA-based with PBQP-based register allocation and integrates copy coalescing into the assignment process. We introduced a novel PBQP reduction, which improves the quality of the heuristic decisions by merging nodes. Additionally, we presented a technique to handle hierarchic register constraints, which enables a wider range of options within the PBQP. Our implementation achieves an improvement over the SSA-based recoloring approach. On average, the relative number of swap and copy instructions for the SPEC CINT2000 benchmark was reduced to 99.6% and 95.2%, respectively, while taking 19% less time for assignment and coalescing.

References

1. Bouchez, F., Darte, A., Rastello, F.: On the complexity of register coalescing. In: CGO 2007: Proceedings of the International Symposium on Code Generation and Optimization, pp. 102–114 (2007)
2. Braun, M., Mallon, C., Hack, S.: Preference-guided register assignment. In: Gupta, R. (ed.) CC 2010. LNCS, vol. 6011, pp. 205–223. Springer, Heidelberg (2010)
3. Briggs, P., Cooper, K.D., Torczon, L.: Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.* 16(3), 428–455 (1994)
4. Brisk, P., Dabiri, F., Macbeth, J., Sarrafzadeh, M.: Polynomial time graph coloring register allocation. In: 14th International Workshop on Logic and Synthesis. ACM Press, New York (2005)
5. Buchwald, S., Zwinkau, A.: Instruction selection by graph transformation. In: Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2010, pp. 31–40. ACM, New York (2010)
6. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register allocation via coloring. *Computer Languages* 6(1), 47–57 (1981)
7. Dirac, G.: On rigid circuit graphs. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg* 25, 71–76 (1961)
8. Ebner, D., Brandner, F., Scholz, B., Krall, A., Wiedermann, P., Kadlec, A.: Generalized instruction selection using SSA-graphs. In: LCTES 2008: Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 31–40. ACM, New York (2008)
9. Eckstein, E., König, O., Scholz, B.: Code instruction selection based on SSA-graphs. In: Anshelevich, E. (ed.) SCOPES 2003. LNCS, vol. 2826, pp. 49–65. Springer, Heidelberg (2003)

10. Grund, D., Hack, S.: A fast cutting-plane algorithm for optimal coalescing. In: Adsul, B., Vetta, A. (eds.) CC 2007. LNCS, vol. 4420, pp. 111–125. Springer, Heidelberg (2007)
11. Hack, S.: Register allocation for programs in SSA form. Ph.D. thesis, Universität Karlsruhe (October 2007)
12. Hack, S., Goos, G.: Copy coalescing by graph recoloring. In: PLDI 2008: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (2008)
13. Hack, S., Grund, D., Goos, G.: Register allocation for programs in SSA-form. In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 247–262. Springer, Heidelberg (2006)
14. Hames, L., Scholz, B.: Nearly optimal register allocation with PBQP. In: Lightfoot, D.E., Ren, X.-M. (eds.) JMLC 2006. LNCS, vol. 4228, pp. 346–361. Springer, Heidelberg (2006)
15. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. 42(6), 89–100 (2007)
16. Scholz, B., Eckstein, E.: Register allocation for irregular architectures. In: LCTES-SCOPES, pp. 139–148 (2002)

Probabilistic Points-to Analysis for Java

Qiang Sun¹, Jianjun Zhao^{1,2}, and Yuting Chen²

¹ Department of Computer Science and Engineering

² School of Software

Shanghai Jiao Tong University

800 Dongchuan Road, Shanghai 200240, China

Abstract. Probabilistic points-to analysis is an analysis technique for defining the probabilities on the points-to relations in programs. It provides the compiler with some optimization chances such as speculative dead store elimination, speculative redundancy elimination, and speculative code scheduling. Although several static probabilistic points-to analysis techniques have been developed for C language, they cannot be applied directly to Java because they do not handle the classes, objects, inheritances and invocations of virtual methods. In this paper, we propose a context-insensitive and flow-sensitive *probabilistic points-to analysis for Java* (JPPA) for statically predicting the probability of points-to relations at all program points (i.e., points before or after statements) of a Java program. JPPA first constructs an interprocedural control flow graph (ICFG) for a Java program, whose edges are labeled with the probabilities calculated by an algorithm based on a static branch prediction approach, and then calculates the probabilistic points-to relations of the program based upon the ICFG. We have also developed a tool called *Lukewarm* to support JPPA and conducted an experiment to compare JPPA with a traditional context-insensitive and flow-sensitive points-to analysis approach. The experimental results show that JPPA is a precise and effective probabilistic points-to analysis technique for Java.

Keywords: points-to analysis, probability, Java.

1 Introduction

Points-to analysis is an analysis technique which is widely used in compiler optimization and software engineering [1,2]. The goal of points-to analysis is to compute points-to relations between variables of pointer types and their allocation sites. Context-sensitivity and flow-sensitivity are two major aspects of points-to analysis for improving the precision of the analysis [3]. While the context-sensitive points-to analysis [4,5] distinguishes the different contexts in which a method is invoked and then analyzes the method individually for each context, the flow-sensitive points-to analysis [6,7] takes into account the control flows inside a program or a method, and computes the solutions (i.e., points-to relations) for the program points on the control flow of each method. Especially, a flow-sensitive points-to analysis helps deduce that for each points-to relation whether it *definitely* exists or *maybe* exists at any program point.

Probabilistic points-to analysis [8], which defines the probability of each points-to relation, provides the compiler with some optimization chances. With the probabilistic

points-to relations, a compiler may perform speculative dead store elimination, redundancy elimination, and code scheduling [9,10,11]. For example, by speculation a compiler with a recovery mechanism may characterize a variable as redundant if its points-to relation is of high probability of the same as those of other variables. A probabilistic points-to analysis usually follows two steps to compute, during the executions of a program, the quantitative information of the likelihood a points-to relation *maybe* holds. First, all the execution paths and their frequencies are collected at runtime. Second, the points-to relations are deduced and the probabilities of points-to relations are computed according to the path frequencies. However, a challenge about decreasing the costs of computation of the probabilities in a large-scale program still exists because a program may run hundreds of times and a large amount of time and memory can be consumed.

One possible solution to above problem is to conduct static probabilistic points-to analyses of programs. Although several static probabilistic points-to analysis techniques have been developed for C language, they cannot be applied directly to the analysis of Java programs due to the differences between Java and C languages. For example, the techniques for analyzing C function pointers usually compute the points-to set for each function pointer and the probabilities of the elements inside. We could not use these techniques to analyze the invocation of Java virtual methods, while a Java virtual method is invoked by an object, and an analysis of the invocation of the method requires the determination of the receiver and the exploration of the distributions over the points-to set for the receiver.

In this paper, we propose a context-insensitive and flow-sensitive *probabilistic points-to analysis for Java* (JPPA) for statically predicting the probability of each points-to relation at each program point of a Java program. JPPA first constructs a call graph through the traditional static analysis [12,13,14], and then builds an intraprocedural control flow graph (CFG) with probabilities for each method in the call graph. The probabilities can be computed according to the result of a static branch prediction. After that JPPA combines the call graph and CFGs to construct an interprocedural CFG (ICFG), and then carries out the data-flow analysis on the ICFG for constructing the probabilistic points-to graph at each program point. In JPPA, a points-to relation is not confined to yes/no but is associated with a real number representing the probability. We have also implemented JPPA with a tool called *Lukewarm* and conducted an experiment to evaluate the effectiveness of JPPA. The experimental results show that, for our benchmark programs, JPPA provides a cost-effective manner in computing the probabilistic points-to relations in Java programs.

This paper makes the following contributions:

- **Abstraction.** We define the probabilities on the edges of ICFG, and these probabilities, which share one destination node, forms a distribution. We also use a discrete probability distribution to represent a points-to set with probabilities.
- **Analysis Approach.** We develop a probabilistic points-to analysis technique for Java named JPPA, which takes into account the object-oriented features such as inheritance and polymorphism. Especially, in order to improve the performance of analysis, JPPA computes a partial probabilistic points-to graph before parameter

```

1 public class Shape {
2     public Double area = null;
3     public Double set(Double s) {
4         this.area = s;
5         return s;
6     }
7     public static void main(String args[]) {
8         int a = randomInt();
9         int b = randomInt();
10        Shape p = null;
11        if(a>0 && b>0) {
12            p = new Circle(); //o1
13        }
14        else {
15            p = new Square(); //o2
16        }
17        Double q =
18            new Double(Math.abs(a*b)); //o3
19        Double r = p.set(q);
20        System.out.println(p.area);
21    }
22    private static double randomInt() {
23        return Math.floor(Math.
24            random()*11-5);
25    }
26 }
27
28 public class Circle extends Shape {
29     public Double area = null;
30     public Double radius = null;
31     public Double set(Double s) {
32         this.area = s;
33         this.radius =
34             new Double(Math.sqrt(s/3.14)); //o5
35         return this.radius;
36     }
37 }
38
39 public class Square extends Shape {
40     public Double sideLength = null;
41     public Double set(Double s) {
42         this.area = s;
43         this.sideLength =
44             new Double(Math.sqrt(s)); //o6
45         return this.sideLength;
46     }
47 }

```

Fig. 1. A sample Java program

passing; in order to improve the precision of analysis, JPPA computes the probabilities over the edges of ICFG dynamically. We have also developed a tool to support the JPPA approach.

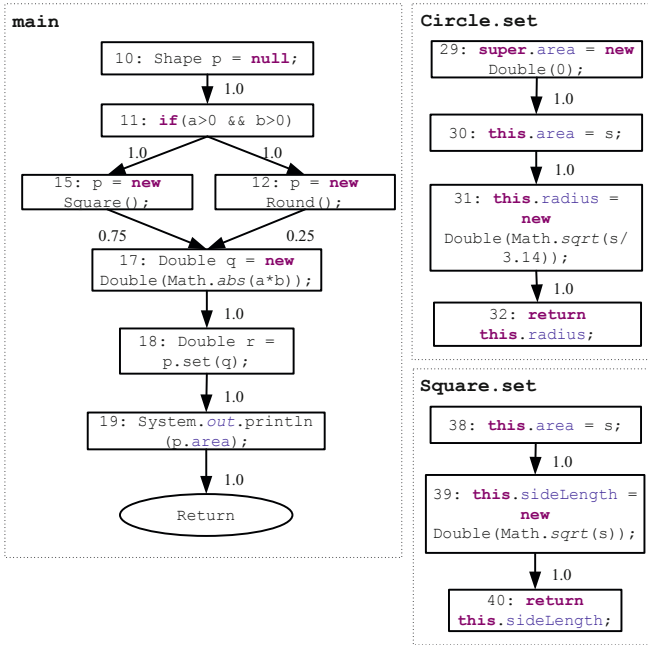
- **Analysis Results.** We have conducted an experiment to evaluate JPPA, and the experimental results show that our analysis can be used to compute precisely the probabilistic points-to information in Java programs.

2 Example

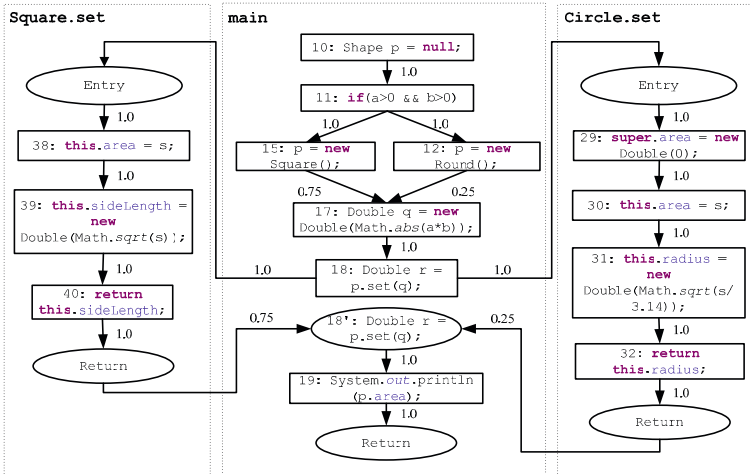
We next present an example to illustrate how our static probabilistic points-to analysis works. Fig. 1 shows a Java program with three classes: Shape, Circle, and Square. Class Shape declares shapes and their sizes, and classes Circle and Square extend Shape with two fields radius and sideLength respectively. The method randomRealNum() in class Shape randomly generates values to simulate the inputs from the real world. The main() method in class Shape receives an input and then creates a shape (either a circle or a square) and prints its field of area.

Suppose randomRealNum() generates the real numbers complying to the uniform distribution in the range $[-5, +5]$. The probabilities of the if-else branches (see lines 11-16) in the main() method can be easily referred: the if branch is of a probability of 0.25 and the else branch a probability of 0.75. Suppose the objects created in the sample program in Fig. 1 are o_i ($i=1..6$).

Our JPPA analysis consists of four steps. At the beginning, a call graph is constructed through the traditional static analysis [12,13,14] in order to remove the unreachable methods to reduce the analyzing costs. Although the call graph built by static analysis may not be precise enough as it still contains some unreachable methods, JPPA can refine it based on the points-to information. The points-to set $\{o_1, o_2\}$ of variable p at line 18 can be computed through analyzing the statements at lines 12 and 15.



(a) The CFGs for methods main, Circle.set and Square.set



(b) The ICFG for the whole sample program

Fig. 2. The CFGs and ICFG of the sample program in Fig. 1

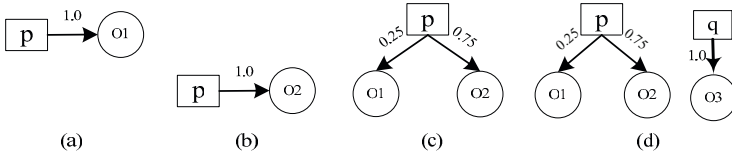


Fig. 3. The probabilistic points-to analysis for method main in (part I)

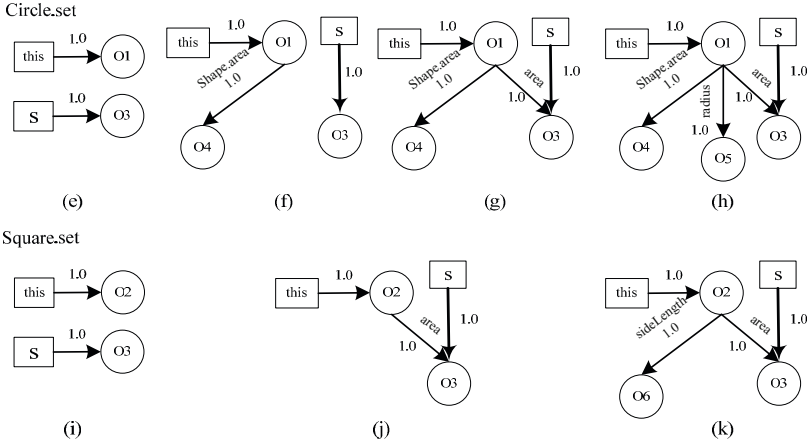


Fig. 4. The probabilistic points-to analysis for methods Circle.set and Square.set

Through identifying the classes of o_1 and o_2 , both the methods `Circle.set()` and `Square.set()` can be invoked at line 18. Therefore, the analyzed methods includes `main()`, `Circle.set()`, and `Square.set()`.

The second step is to construct a CFG for each method in the call graph. Fig. 2(a) shows the CFGs for methods `main()`, `Circle.set()` and `Square.set()`, in which each node in the CFG represents a statement and each edge is labeled with a real number for the predecessor-dependent probability (see Section 3). Since the traverse of node 15 must be preceded by the traverse of node 11, the probability of the edge from node 11 to node 15 is 1.0; a traverse of node 17 may succeed the traverse of node either 12 or 15 and the probabilities of the **if-else** branches are 0.75 and 0.25, and therefore the probabilities of the edges from nodes 15 and 12 to 17 are 0.75 and 0.25, respectively.

The third step is to combine the CFGs of the methods of interests into an interprocedural CFG (ICFG) according to the call graph. Fig. 2(b) shows the ICFG of the sample program with the probabilities on the edges. Note that we add some new nodes to the ICFG in order to simplify our discussion. Node 18' copies the node 18 in order to receive the return values of the method invocation `Circle.set()` or `Square.set()`. For each method, an entry node and a return node are added to assign the parameters and return values, respectively. The probabilities on the interprocedural return edges are initialized by 0.0 at first and then adjusted in the final step.

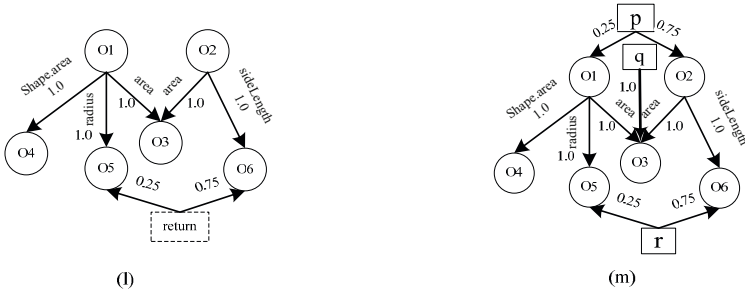


Fig. 5. The probabilistic points-to analysis for method `main` (part II)

The final step is to analyze each node in the ICFG to compute the probabilistic points-to graph. Fig. 3 illustrates the probabilistic points-to analysis of the statements between lines 11 and 17. The probabilistic points-to graphs (Fig. 3 (a) and Fig. 3 (b)) are calculated after analyzing the statements at lines 12 and 15, respectively. Before line 17, Fig. 3(c) is calculated by merging the two graphs in Fig. 3(a) and 3(b). After analyzing the statement at line 17, the graph in Fig. 3(d) is generated. Because of the analysis in the first step, either `Circle.set()` or `Square.set()` can be invoked in line 18. The probabilities on the interprocedural edges are adjusted according to the points-to probabilities of the receiver objects. The variable `p` points to the objects `o1` and `o2` with the probabilities 0.25 and 0.75 respectively. The probabilities of the edges from the return nodes of methods `Circle.set()` and `Square.set()` to the node 18' are adjusted by 0.25 and 0.75. A detailed algorithm about the calculation of the probabilistic points-to graphs is given in Section 3.

Since the field `area` is overloaded in the class `Circle`, the field declared in class `Shape` is marked as `Shape.area`. The problem does not occur in class `Square` because it inherits `area` from class `Shape`. Fig. 4 manifests the analysis of `Circle.set()` and `Square.set()`, and Fig. 5 shows the analysis from the end of method call to the end of method `main()`. In Fig. 5 the graph (l) is built by merging the two graphs (h) and (k), in which the local variables are eliminated and the return value is marked by the dish square node. The graph (m) represents the points-to graph after the method call that is generated by updating the graph (d) before the method call uses the graph (l) and replacing the return node by `r` node.

3 Probabilistic Points-to Analysis for Java

3.1 Probabilistic Points-to Graph

Traditionally, in order to conduct a points-to analysis of a Java program, three sets are defined [4]:

- *Ref*: a set containing all reference variables in the program and all static fields in its classes.

- *Obj*: a set containing the names of all objects created at the object allocation sites.
- *Field*: a set containing instance fields but not the static fields in the classes.

In addition, we define $R \subseteq Ref \cup Obj \times Field$ containing all object references if they belong to *Ref* or have a form $\langle o, f \rangle \in Obj \times Field$. Two relations are defined on the three sets: $(r, o) \in Ref \times Obj$ representing that a reference variable r points to an object o and $(\langle o, f \rangle, o') \in (Obj \times Field) \times Obj$ representing that a field f of the object o points to an object o' .

The goal of probabilistic points-to analysis is to compute the probability of each points-to relation holding at every program point. A *program point* is a code location before or after a statement executed. In order to do this, we define an expected probability as follows:

$$Prob(l, d) = \begin{cases} \frac{Expected(l, d)}{Expected(l)} & Expected(l) \neq 0, \\ 0 & Expected(l) = 0. \end{cases}$$

where d is a points-to relationship, $Expected(l)$ is the times which a program point l is expected to turn up on the program execution paths and $Expected(l, d)$ is the times which d is expected to hold dynamically at a program point l [15].

For a given program point l , the points-to set of a reference r or $\langle o, f \rangle$ with probabilities satisfies a discrete probability distribution over the object set *Obj*, and the probabilistic points-to relationships at l is a set of the distributions. A discrete probability distribution over a set *Obj* is a mapping

$$\Delta : Obj \mapsto [0, 1], \quad \sum_{o \in Obj} \Delta(o) = 1.$$

The *support* of Δ is $[\Delta] := \{o \in Obj \mid \Delta(o) > 0\}$. If a points-to relation d is in the form of (r, o) or $(\langle o_1, f \rangle, o_2)$ at the program point l , the distributions of the references satisfy $\Delta_r(o) = Prob(l, (r, o))$ or $\Delta_{o_1.f}(o_2) = Prob(l, (\langle o_1, f \rangle, o_2))$. \bar{o} is the point distribution over *Obj* satisfying

$$\bar{o}(t) = \begin{cases} 1 & t = o, t \in Obj, \\ 0 & otherwise. \end{cases}$$

If Δ_i is a distribution for each i in some finite index set I and $\sum_{i \in I} p_i = 1$, $\sum_{i \in I} p_i \cdot \Delta_i$ is also a distribution ($\sum_{i \in I} p_i \cdot \Delta_i(o) = \sum_{i \in I} p_i \cdot \Delta_i(o), o \in Obj$). Every distribution can be represented as a linear combination of the point distributions with the form of $\Delta = \sum_{o \in [\Delta]} \Delta(o) \cdot \bar{o}$.

► **Example.** The distribution of variable p before line 18 is represented by $\Delta_p = 0.25\bar{o}_1 + 0.75\bar{o}_2$. ◀

In our research, we use *probabilistic points-to graph* (PPG) for probabilistic points-to analysis of Java programs. A PPG is a directed multi-graph whose nodes are the elements belonging to *Ref* and *Obj*. Each edge has a probability and represents a

probabilistic points-to relation either from a variable to an object or from an object to another object. Specially, an edge from an object o_1 to another o_2 (e.g., the edge from node o_2 to node o_3 in Fig. 4(j)) means a field of o_1 points to o_2 , and holds the field information. Thus a PPG contains a distribution set of the reference variables.

► **Example.** For the program point after line 18 in Fig. 1 the PPG (see Fig. 5(m)) shows the distributions of p , q , r and all field references.

$$\begin{aligned} \Delta_p &= 0.25\tilde{o}_1 + 0.75\tilde{o}_2 & \Delta_r &= 0.25\tilde{o}_5 + 0.75\tilde{o}_6 & \Delta_q &= 1.0\tilde{o}_3 \\ \Delta_{o1.radius} &= 1.0\tilde{o}_5 & \Delta_{o1.Shape.area} &= 1.0\tilde{o}_4 & \Delta_{o1.area} &= 1.0\tilde{o}_3 \\ \Delta_{o2.area} &= 1.0\tilde{o}_3 & \Delta_{o2.sideLength} &= 1.0\tilde{o}_6 & & \blacktriangleleft \end{aligned}$$

In the research we provide a program with predecessor-dependent probabilities information so that PPGs can be precisely calculated. In order to perform probabilistic points-to analysis, a program is represented by a ICFG (e.g., Fig. 2(b)) whose edges are labeled with probabilities. We call this probability predecessor-dependent probability to distinguish the branch probability. A branch probability [16] ($edge_prob(s_i \rightarrow s_j)$) is an estimate of the likelihood that a branch will be taken. For any two statement s_1 and s_2 in a method, the predecessor-dependent probability is an estimate of the likelihood that s_1 directly reaches s_2 . The branch probabilities calculated by the branch prediction algorithm [16] can not be directly applied to the merge operation in our framework. The branch probabilities on the edges of the intra-method CFG can be used to compute the predecessor-dependent probabilities. Given a path (s_0, \dots, s_n) , the path probability is computed by

$$path_prob(s_0, \dots, s_n) = \prod_{i=0}^{n-1} edge_prob(s_i \rightarrow s_{i+1})$$

where s_0 represents the method entry statement. The predecessor-dependent probability for the edge (s_i, s_j) is computed by

$$EP((s_i, s_j)) = \frac{\sum path_prob(s_0, \dots, s_i)}{\sum path_prob(s_0, \dots, s_i, s_j)}$$

► **Example.** After the branch prediction analysis, we have

$edge_prob(10 \rightarrow 11) = 1.0$, $edge_prob(15 \rightarrow 17) = 1.0$, $edge_prob(12 \rightarrow 17) = 1.0$, $edge_prob(11 \rightarrow 12) = 0.25$, $edge_prob(11 \rightarrow 15) = 0.75$.

The predecessor-dependent probability of the edge from 15 to 17 is calculated as follows:

$$\begin{aligned} EP((15, 17)) &= \frac{path_prob(10, 11, 15, 17)}{path_prob(10, 11, 15, 17) + path_prob(10, 11, 12, 17)} \\ &= \frac{1.0 \times 0.75 \times 1.0}{1.0 \times 0.75 \times 1.0 + 1.0 \times 0.25 \times 1.0} = 0.75 \quad \blacktriangleleft \end{aligned}$$

Since in Java there are four ways to assign a value to a reference variable that may change a points-to relation. The statements in these four forms should be analyzed. These forms are:

- Create an object: $v = new C$;
- Assign a value: $v = r$;

- Read an instance field : $v = r.f$;
- Write an instance field : $v.f = r$.

► **Example.** Nodes 32 and 21 in Fig. 2 (b) create an object and write an instance field, respectively. ◀

3.2 Intraprocedural Analysis

A points-to analysis can be formulated as a data flow framework which includes transfer functions formulating the effect of statements on points-to relations [6]. As a result, our probabilistic points-to analysis framework can be represented by a tuple

$$(L, \sqcup, Fun, P, Q, E, \iota, M, EP)$$

where:

- L is a lattice and a PPG can be regarded as an element of L
- \sqcup is the meet operator
- $Fun \subseteq L \mapsto L$ is a set of monotonic functions
- P is the set of the statements
- $Q \subseteq P \times P$ is the set of flows between statements
- E is the initial set of statements
- ι specifies the initial analysis information
- $M : P \mapsto Fun$ is a map from statements to transfer functions
- $EP : Q \mapsto [0, 1]$ is an predecessor-dependent probability function

The partial order over L is determined by

$$\forall G_1, G_2 \in L, G_1 \sqsubseteq G_2 \text{ iff } \forall r \in R, [\Delta_r^{G_1}] \subseteq [\Delta_r^{G_2}].$$

And the meet operation of two graphs G_1 and G_2 is

$$G_1 \sqcup G_2 = \{p \cdot \Delta_r^{G_1} + (1 - p) \cdot \Delta_r^{G_2} \mid r \in R, p \in [0, 1]\}.$$

Let $f_s \in Fun$ be the transfer function of the statement s , and $G_{in}(s)$ and $G_{out}(s)$ represent the PPGs at the program points before and after the statement s , respectively, we have

$$G_{in}(s) = \begin{cases} \iota & \text{if } s \in E \\ \bigsqcup \{G_{out}(s') \mid (s', s) \in Q\} & \text{otherwise} \end{cases}$$

$$G_{out}(s) = f_s(G_{in}(s))$$

The statement s is associated with the transfer function that transforms $G_{in}(s)$ to $G_{out}(s)$, and the analysis iteratively computes the $G_{in}(s)$ and $G_{out}(s)$ for all nodes until convergence.

In a probabilistic points-to analysis, E only contains the first statement during the program execution and ι is a special probabilistic points-to graph in which all the reference variables point to undefined target (i.e., UND) with total probabilities. Next we describe the transfer functions for assignments and branches.

Table 1. Computing distributions

Statement	Updating the distributions of $G_{out}(s)$
$v = new\ C$	$\Delta_v^{G_{out}(s)} \leftarrow \bar{o}$
$v = r$	$\Delta_v^{G_{out}(s)} \leftarrow \Delta_r^{G_{in}(s)}$
$v = r.f$	$\Delta_v^{G_{out}(s)} \leftarrow \sum_{o \in [\Delta_r^{G_{in}(s)}]} \Delta_r^{G_{in}(s)}(o) \cdot \Delta_{o.f}^{G_{in}(s)}$
$v.f = r$	$\Delta_{o.f}^{G_{out}(s)} \leftarrow \Delta_v^{G_{in}(s)}(o) \cdot \Delta_r^{G_{in}(s)} + (1 - \Delta_v^{G_{in}(s)}(o)) \cdot \Delta_{o.f}^{G_{in}(s)}, o \in [\Delta_v^{G_{in}(s)}]$

Assignments. For any assignment statement s , there is a corresponding transfer function F_s . F_s takes $G_{in}(s)$ as input and computes the result $G_{out}(s)$. A transfer function first copies $G_{in}(s)$ to $G_{out}(s)$ and then updates $G_{out}(s)$. Table 1 describes transfer functions for assignment statements: the transfer function for $v = new\ C$ updates the distribution $\Delta_v^{G_{out}(s)}$ with the point distribution of o created by $new\ C$ expression; the transfer function for $v = r$ replaces the distribution $\Delta_v^{G_{out}(s)}$ with the distribution $\Delta_r^{G_{in}(s)}$; the transfer function for $v = r.f$ composes the distributions of the field f of all the objects in the *support* set of $\Delta_r^{G_{in}(s)}$; and the transfer function for $v.f = r$ updates multiple distributions in the form of $o.f$.

A probabilistic points-to analysis also needs to take arrays into account, each of which may contain multiple references. Since an array $array$, is initialized as

$$C\ [\]\ array = new\ C\ [n];$$

where n can be either a constant or a variable, it is difficult to infer the range of n . We can obtain the array elements of multiple references and estimate the total number of references they point to if we cannot determine the points-to set of each element in a static manner. When an element o is stored to $array$ (i.e., $array[i] = o$), o is added to the points-to set of $array$, say $Pt(array)$. Then each points-to probability is recalculated

$$p = \frac{1}{|Pt(array)|}$$

where $|Pt(array)|$ is the number of the elements in $Pt(array)$. When $array[i]$ is accessed (e.g., $v = array[i]$), the distribution of v is updated by the distribution of $array$.

Branch. When multiple nodes directly reach a destination node s in a CFG, the *meet* operation is adopted in the calculation of $G_{in}(s)$. Suppose s is the successor of the nodes $s_i (i \in I)$ in a CFG. The following condition is satisfied

$$\sum_{i \in I} EP((s_i, s)) = 1$$

where $EP((s_i, s))$ represents the probability of edge (s_i, s) . The *meet* operation can be described by

$$G_{in}(s) = \bigsqcup \{G_{out}(s_i) \mid i \in I\} \triangleq \sum_{i \in I} EP((s_i, s)) \cdot G_{out}(s_i).$$

► **Example.** In an *if-then-else* statement, suppose $G_{out}(s_{then})$ and $G_{out}(s_{else})$ are the PPGs at the exit points of the *then* and *else* branches respectively, p_t and p_f are the probabilities of the *then* and *else* branches respectively, and $p_t + p_f = 1$. A PPG at s_{join} , the statement succeeding the *if-then-else* statement, can be computed by using *meet* operation: $G_{in}(s_{join}) = p_t \cdot G_{out}(s_{then}) + p_f \cdot G_{out}(s_{else})$. ◀

Loop. The loop body B can be unfolded for arbitrary times. The computation can be formulated as

$$G_{in} = \sum_{\alpha \leq i \leq \beta} p_i \cdot F^i(G_0) + p_0 \cdot G_0, \quad \sum_{\alpha \leq i \leq \beta} p_i + p_0 = 1$$

where α and β are the upper-bound and lower-bound of iteration number, G_0 represents the initial PPG before entering the loop, F is the transfer function of loop body B , p_0 represents the probability of not entering the loop, p_i represents the probability of i times iteration. Two problems arise when a loop is analyzed: (1) how to estimate the upper- and lower-bounds of iteration number, and (2) how to estimate the probability of some iteration number.

In JPPA, the loop body is unfolded for the upper-bound times. The lower-bound of iteration number is 0. The upper-bound of iteration number is the minimum N , which satisfies $\forall r \in R, \lceil \Delta_r^{F^N(G_0)} \rceil = \lceil \Delta_r^{F^{N+1}(G_0)} \rceil$. The probability of each iteration number is $1/(N + 1)$.

Exception Handling. The exception handling in Java encapsulates exception in class, uses the exception handling mechanism of *try-catch-finally* and gets more robust exception handling code finally. In JPPA framework, the probabilistic points-to analysis can easily go deep into the exception blocks through building the CFG for them. In the *try-catch-finally* structure, the exceptions are thrown out from every program point in the try block, then the edges from the program points to the entry point of catch block are generated. In our study we adopt Soot to construct CFGs for *try-catch-finally* structures. However, a precise calculation of points-to relation information in exceptions requires obtain all program points that may throw these exceptions, which remains in our future work.

3.3 Interprocedural Analysis

Interprocedural probabilistic points-to analysis analyzes points-to relations crossing the boundary between methods. At each call site, points-to relations are mapped from the actual parameters to the formal parameters, and the results are mapped back to the variables in the caller.

Since a Java program may rely heavily on libraries with a number of irrelevant methods, JPPA adopts RTA algorithm [13] to produce an approximation of the corresponding call graph so that some irrelevant methods can be ignored. After that, JPPA uses the call graph to construct the ICFG. At each call site, two extra nodes are generated for each callee: an entry node recording the passing of parameters, and a return node recording the returns of the callee. Specially, all values returned by the callee are assigned to a unique variable in the return node. JPPA then refines the ICFG by adjusting the probabilities on the interprocedural edges according to the PPGs at all method call sites.

When a method call is analyzed, a partial PPG is propagated through an interprocedural edge to the callee method. JPPA takes a set of actual parameters and a PPG as its inputs, and then computes all the objects that can be accessed by these actual parameters. The partial PPG as a result includes all the points-to relations with the probabilities that may be manipulated by the callee method.

Since JPPA is a context-insensitive analysis without distinguishing the contexts under which a method is invoked, it may share one callee method with different calling contexts. Thus a meet operation can be defined as

$$G_{in}(s_m) = \sum_{cs_i \in CS} EP(e_{cs_i}) \cdot G_{out}(s_{cs_i}), \quad e_{cs_i} = (s_{cs_i}, s_m)$$

where m is the callee method, $G_{in}(s_m)$ is the PPG before the program point entering the method m , CS denotes the set of call sites at each of which m is invoked, $EP(e_{cs_i})$ denotes the probability of the invocation of m at cs_i , and $G_{out}(s_{cs_i})$ represents the PPG after the passing of parameters at cs_i .

When the return value is assigned to the variable in the caller method, all the points-to relations updated by the callee method need to be reflected upon the PPG after the method invocation.

Virtual Invocation. In Java, virtual method is a method whose behavior can be overridden within an inheriting class by a method with the same signature. The compiler and loader can guarantee the correct correspondence between objects and the methods applied to them.

A virtual method m is usually invoked explicitly or implicitly by a *this* object, and thus *this* needs to be mapped to the receiver object invoking m . Suppose m is declared in class C and at each call site $cs_i \in CS$, m may be invoked in a form $r_i.m()$ where r_i is the receiver object invoking m at cs_i . At each call site cs_i , there exists an object set

$$set_i = \{o \mid o \in [\Delta_{r_i}] \wedge o.Class \in MatchedClass(C, m)\}$$

where $MatchedClass(C, m)$ is a set of classes containing class C and all its subclasses in each of which m is inherited. The distribution of *this* can be computed

$$\Delta_{this} = \sum_{cs_i \in CS} \sum_{So \in set_i} \frac{EP(e_{cs_i}) \cdot \Delta_{r_i}(o)}{a} \bar{o},$$

$$a = \sum_{cs_i \in CS} \sum_{So \in set_i} EP(e_{cs_i}) \cdot \Delta_{r_i}(o), \quad e_{cs_i} = (s_{cs_i}, s_m)$$

where $EP(e_{cs_i})$ denotes the probability of invocation of m at cs_i .

► **Example.** In Fig. 11 the distribution of *this* in method `Circle.set` can be calculated as $1.0\bar{o}_1$, which is more precise than that calculated by using the formula in Section 3.3 (i.e., $0.25\bar{o}_1 + 0.75\bar{o}_2$). ◀

The return value of a virtual invocation also needs to be taken into account in order to achieve a conservative resolution. Suppose a virtual invocation at a call site cs is

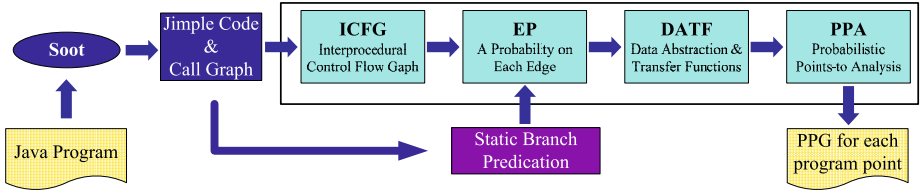


Fig. 6. A framework of *Lukewarm* tool

$v = r.m()$. With the points-to set of r , we can deduce the set of methods that can be invoked at cs and denote each method remained in the set M_v . The distribution of the variable v can be calculated by

$$\Delta_v = \sum_{m_j \in M_v} EP(e) \cdot \Delta_{return_{m_j}}, \quad e = (s_{m_j}^{ret}, s'_{cs})$$

where $EP(e)$ denotes the probability of m_j invoked at cs , and $\Delta_{return_{m_j}}$ the distribution of the return value of m_j . $EP(e)$ is then adjusted on the base of the distribution of r .

► **Example.** In Fig. 1, the distribution of variable x is $0.25\vec{o}_5 + 0.75\vec{o}_6$. ◀

4 Implementation

We have developed a tool called *Lukewarm* to support our method. Fig. 6 shows the framework of *Lukewarm*: the inputs are Jimple code (a typed 3-address *intermediate representation*) [17,18], the call graph of the program which are generated by Soot [19], and the predecessor-dependent probabilities computed by a static branch prediction analysis [16]; the outputs are the PPGs at all program points. *Lukewarm* provides engineers with support in probabilistic points-to analysis of Java programs by the following four steps:

- Use RTA algorithm in Soot to construct the call graph covering all reachable methods, and then construct the CFGs for these methods and combine them to form an ICFG;
- Annotate each edge with a probability calculated based on the static branch prediction approach;
- Extract all the references and objects from the ICFG and generate the transfer function for each node;
- Adopt a worklist algorithm on the nodes of the ICFG to compute the PPG for each program point. A worklist is initialized to contain a node which is associated with an empty PPG. Everytime one node n in the worklist is retrieved, and the PPGs of its G_{in} and G_{out} are calculated on the basis of the transfer function. If G_{in} or G_{out} of n is changed (The graphs G_a and G_b are equivalent *iff* $\|G_a - G_b\|_e < \epsilon$), all the successive nodes of n are added to the worklist. This procedure is repeated until the worklist is empty. Based on the observations, we choose ϵ to be 0.01 in this paper for the reason that it can balance the efficiency and precise.

Table 2. Java benchmarks

Program	#Statement	#Block	Description
HashMap	20929	12307	A small program using HashMap in Java library.
ArrayList	150	27	A small program using ArrayList in Java library.
antlr	52058	23167	A parser generator and translator generator.(DaCapo)
xalan	24186	13716	An XSLT processor for transforming XML documents.(DaCapo)
luindex	24947	14144	A text indexing tool.(DaCapo)
hsqldb	24466	13714	An SQL relational database engine written in Java.(DaCapo)
toba-s	34374	16661	A tool translating Java class files into C source code.(Ashes)
Jtopas	32226	21042	A Java library for the common problem of parsing text data. (SIR)
JLex	32058	16120	A lexical analyzer generator for Java.
java_cup	37634	18049	A LALR parser generator written in Java.

5 Experiments

We have conducted three experiments on JPPA by comparing it with a context-insensitive and flow-sensitive points-to analysis (TPA for short) proposed by Hind et al [20]. The principle of TPA is to use an iterative dataflow analysis framework to compute the points-to set for each reference variable on each node of the CFG of the program. The first experiment was conducted to evaluate the precision of points-to sets calculated through JPPA, the second one was to evaluate the capability of JPPA to calculate the points-to relations *maybe* holding, and the third one was to evaluate the performance of JPPA. All experiments were conducted on a machine with an AMD Sempron™ 1.80GHz CPU and 1G heap size (option -Xmx1024m).

Table 2 shows benchmark programs used in the experiments, which include the SIR suite [21], programs from the Ashes suite [22], programs from the DaCapo suite [23], and two programs for testing `java.util.HashMap` and `java.util.ArrayList`. Table 2 also shows the total number of statements (#Statement) in Jimple code and that of basic blocks (#Block) of each benchmark program.

5.1 Precision of Points-to Analysis

In the first experiment, the points-to sets computed by JPPA and TPA for each reference were the same at each program point. Table 3 shows the average and the maximum sizes of the points-to sets of each benchmark program as well the percentage of the points-to set with only one object inside. It can be seen that for each benchmark program the average size of points-to sets is less than 2, and the maximum size is less than 20. A rational claim is that the closer to 1 the average size of points-to sets is, the more traditional optimization techniques we can use because optimizations usually require the information about the points-to relations *definitely* holding. For example, if the points-to set of the receiver variable contains only one object, it can reduce the direct overhead of dispatching the message and provide the opportunities for method inlining and interprocedural analysis.

In addition, most points-to sets of the benchmark programs have only one object inside, as the row OneObject.Pt (%) indicates. It means that both JPPA and TPA can explore most points-to relations.

Table 3. JPPA measurements

Program	HashMap	ArrayList	antlr	xalan	luindex	hsqldb	toba-s	Jtopas	JLex	java_cup
#Avg.Size.Pt	1.49	1.32	1.05	1.60	1.95	1.49	1.29	1.86	1.04	1.72
#Max.Size.Pt	6	8	13	13	13	13	19	13	9	18
OneObject.Pt(%)	79.57	93.80	98.50	86.22	77.58	81.34	91.33	74.12	99.15	87.58

5.2 Precision of Probabilities

In the second experiment, we chose some program points. For each program point l , we obtained the corresponding dynamic probabilistic points-to graph G_d as a standard and then evaluated JPPA and TPA by calculating their probabilistic points-to graphs at l (say G_s^{JPPA} and G_s^{TPA} , respectively) and the corresponding average graph distances to G_d . In this experiment, G_d was computed by using a dynamic method:

- Perform a program instrumentation in order to collect at runtime the hashcodes of all objects in the program and variables at l ;
- Execute the program and use the hashcodes of the objects to explore the points-to relations at l ;
- Execute the corresponding benchmark program multiple times and record the frequency for each points-to relation.

The average of distance can be calculated by

$$AVG_{distance} = \frac{\sum \|G_s - G_d\|_e}{N}$$

where N is the number of program points of interest, and G_s is a probability points-to graph of any program point generated either by JPPA or TPA, and $\|G_s - G_d\|_e$ calculates the average of the normalized Euclidean distances between the distributions of G_s and G_d , which can be calculated by

$$\|G_s - G_d\|_e = \frac{\sum_{r \in R} \sqrt{0.5 \cdot \sum_{o \in Obj} (\Delta_r^s(o) - \Delta_r^d(o))^2}}{|R|}.$$

$AVG_{distance}$ ranges from zero to one and was used to measure the divergence between a probability points-to graph computed by JPPA or TPA and that computed by the dynamic method. $AVG_{distance} = 0$ means that the PPG computed by JPPA or TPA is regarded as correct, and $AVG_{distance} = 1$ means that it may be totally wrong.

In the experiment, when using JPPA, we assumed that all incoming edges of a node of ICFG have the same probability, and then computed the probabilistic points-to relations. When using TPA, we assumed that the probability of each points-to relation belonging to a points-to set is equal. Fig. 7 shows the average distances between the PPGs computed by JPPA and TPA and those computed by the dynamic method. For each benchmark program, the average distance corresponding to JPPA is shorter than that corresponding to TPA. However, for the programs **hsqldb** and **Jtopas**, the distances corresponding to JPPA are not of significant divergences to those corresponding

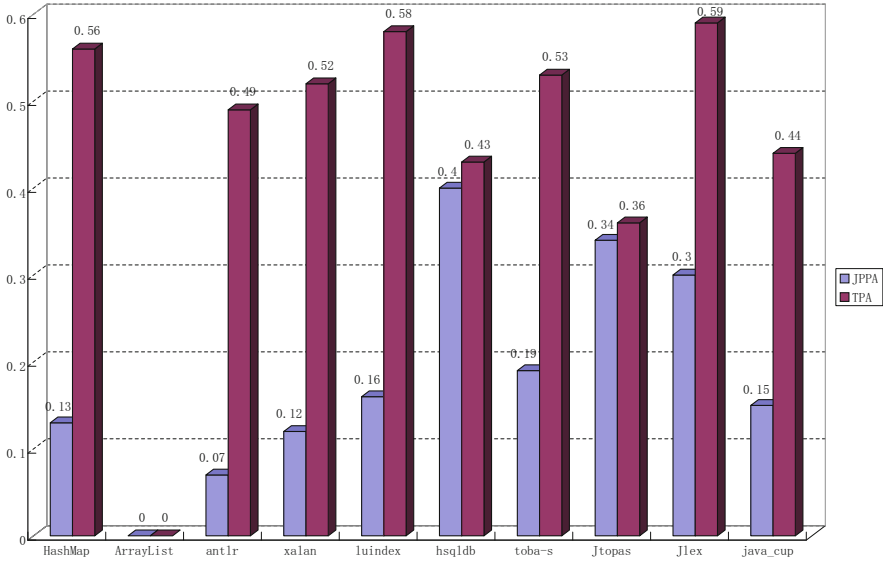


Fig. 7. The comparison of graph distances between JPPA and TPA

to TPA. After a thorough investigation, we found that the arrays of objects used in these programs can decrease the precision of JPPA to compute probabilities. It is one of our future research directions to find out how to improve the precision of JPPA when a number of arrays of objects are available in the programs.

JPPA reaches the same precision of points-to sets as that of TPA, but has a much higher precision of probabilities than that of TPA. In addition, the precision of probabilities of JPPA is very close to that of the dynamic method. It demonstrates that JPPA can be an effective approach to probabilistic points-to analysis of Java programs.

5.3 Analysis Performance

In the third experiment, an investigation of the execution time and memory usage was conducted after the benchmark programs were executed several times. Table 4 shows the running time and memory usage for each benchmark program. A vertical line | is used to divide a result of using JPPA from that of using TPA. We also counted the number of the references (#reference), static fields (#static-field), and objects (#object) for each benchmark program in order to find out the factors that will affect the performance.

After analyzing the results, we found that the distributions of the static fields significantly affects the performance of both JPPA and TPA because the points-to sets of these fields can propagate to all nodes in the ICFG. Specially, when a points-to set of a static field is of a large number of elements, the performance will decrease rapidly because of the propagation of a great amount of data. In addition, the use of arrays of objects in Java also decrease the performance of JPPA because they are handled conservatively and thus a number of redundant objects are contained in points-to sets and also will be propagated.

Table 4. Performance

Program	#reference	#static-field	#object	Time (Sec)		Memory (MB)	
HashMap	4200	104	296	32.98	33.47	176.36	195.82
ArrayList	42	0	9	1.41	1.36	0.38	0.31
antlr	8651	158	972	329.03	369.52	239.79	203.98
xalan	4906	133	398	36.54	40.98	206.56	228.24
luindex	5095	133	419	43.06	43.87	218.97	237.01
hsqldb	4883	114	417	41.26	41.03	213.33	227.99
toba-s	5876	140	750	131.63	103.96	316.97	331.69
Jtopas	5896	107	440	139.70	167.45	298.07	330.80
JLex	5491	129	481	56.90	54.23	296.81	306.58
java_cup	6431	132	825	215.14	146.64	303.02	334.37

From the experiment, we found that JPPA spends more time and memory in computing the probabilities for the points-to relations than TPA does. However, the cost of using JPPA is only about 1.2 times of that of using TPA, which is still acceptable.

5.4 Threats to Validity

A threat to validity is that when using JPPA, we assumed that all incoming edges of a node in the ICFG have the same probability, and when using TPA, we assumed that the probability of each points-to relation belonging to a points-to set is equal. In practice, these assumptions are very strong in that edges of ICFG may hold different probabilities, which may be hardly determined precisely before the analysis. A more reasonable solution is to use the edge profiling information estimated by the machine learning techniques to compute the predecessor-dependent probabilities which will be explored in our future work.

6 Related Work

In the past several years, points-to analysis has been an active research field. A survey of algorithms and metrics for points-to analysis has been given by Hind [24].

Traditional points-to analysis. Context-sensitivity and flow-sensitivity are two major dimensions of pointer analysis precision. Context-sensitive and flow-sensitive algorithms (CSFS) [25,26,27,28] are usually precise, but are difficult to scale to large programs. Recently Yu et al. [29] proposed a level-by-level algorithm that improves the scalability of the context-sensitive and flow-sensitive algorithm. Context-insensitive and flow-insensitive (CIFI) algorithms [30,31] have the best scalability on the large programs with overly conservative results. Equality-based analysis and inclusion-based analyses have become the two widely accepted analysis styles. Through carrying out the experiments, Foster et al. [32] compared several variations of flow-insensitive points-to analysis for C, including polymorphic versus monomorphic and equality-based versus inclusion-based.

How well the algorithms scale to large programs is an important issue. Trade-offs are made between efficiency and precision by various points-to analyses, including

context-sensitive and flow-insensitive analyses [4,5,33] and context-insensitive and flow-sensitive analyses [6,20,7]. However, these conventional points-to analyses do not provide with the probabilities for the possible points-to relations, which is one of main goals of JPPA proposed in this paper.

Probabilistic pointer analysis for C. With the proposition of the speculative optimizations, the probability theory has been introduced into the traditional program analysis. In earlier work, Ramalingam [15] proposed a generic data flow frequency analysis framework that uses the edge frequencies propagation to compute the probability a fact holds true at every control flow node. Chen et al. [8] developed a context-sensitive and flow-sensitive probabilistic point-to analysis algorithm. This algorithm is based on an iterative data flow analysis framework, which computes the transfer function for each control flow node and propagates probabilistic information additionally. In addition, this algorithm also handles interprocedural points-to analysis on the basis of Emami's algorithm [25]. Their experimental results demonstrates that their technique can estimate the probabilities of points-to relations in benchmark programs with reasonable accuracy although they have not disambiguated heap and array elements. Compared with Chen et al.'s algorithm, JPPA is also on the basis of an iterative data flow analysis framework but with context-insensitive analysis. In addition, the concept of the discrete probability distribution are introduced to JPPA.

Silva and Steffan [34] proposed a one-level context-sensitive and flow-sensitive probabilistic pointer analysis algorithm that statically predicts the probability of each points-to relation at every program point. Their algorithm computes points-to probabilities through the use of linear transfer functions that are efficiently encoded as sparse matrices. Their experimental results demonstrates that their analysis can provide accurate probabilities, but omits handling alias between the shadow variables. JPPA provides a safer result in its analysis because the transfer functions of accessing of instance fields do not use the shadow variables during its propagation of the distributions.

7 Conclusions

In this paper we have presented JPPA, a context-insensitive and flow-sensitive algorithm for calculating the probabilistic points-to relations in Java programs. JPPA also predicts the likelihood of points-to relations without depending on runtime profiles. In order to ensure the safety of the result, JPPA takes Java libraries into account. We have conducted experiments to validate JPPA by comparing it with the traditional context-insensitive and flow-sensitive points-to analysis. The experimental results show that JPPA not only produces precise probabilities of points-to relations for Java, but also maintains a similar performance to the traditional context-insensitive and flow-sensitive points-to analysis.

In the future, we would like to continue our efforts to improve and extend JPPA to support more powerful functions concerned with points-to analysis, such as handling of arrays of objects. Also with a more mature approach and the corresponding tool, we would like to apply them in some large-scale projects to investigate how our approach can benefit real projects. We would also like to extend JPPA to a context-sensitive analysis approach in order to improve its precision.

Acknowledgements

We would like to thank the anonymous reviewers for their useful comments and suggestions. We would also like to thank Longwen Lu, Hongshen Zhang, Yu Kuai and Cheng Zhang for their discussions on this work. This work was supported in part by National Natural Science Foundation of China (NSFC) (Grant No. 60970009 and 60673120).

References

1. Das, M., Liblit, B., Fähndrich, M., Rehof, J.: Estimating the impact of scalable pointer analysis on optimization. In: Proceedings of the 8th International Static Analysis Symposium, pp. 260–278 (2001)
2. Hind, M., Pioli, A.: Which pointer analysis should I use? In: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 113–123 (2000)
3. Chatterjee, R., Ryder, B.G., Landi, W.A.: Complexity of points-to analysis of Java in the presence of exceptions. *IEEE Transactions on Software Engineering* 27(6), 481–512 (2001)
4. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology* 14, 1–41 (2002)
5. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, pp. 131–144 (2004)
6. Choi, J.D., Burke, M., Carini, P.: Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 232–245 (1993)
7. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 226–238 (2009)
8. Chen, P.S., Hwang, Y.S., Ju, R.D.C., Lee, J.K.: Interprocedural probabilistic pointer analysis. *IEEE Transactions on Parallel and Distributed Systems* 15(10), 893–907 (2004)
9. Oancea, C.E., Mycroft, A., Harris, T.: A lightweight in-place implementation for software thread-level speculation. In: Proceedings of the 21st Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 223–232 (2009)
10. Scholz, B., Horspool, R.N., Knoop, J.: Optimizing for space and time usage with speculative partial redundancy elimination. In: Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 221–230 (2004)
11. Dai, X., Zhai, A., chung Hsu, W., chung Yew, P.: A general compiler framework for speculative optimizations using data speculative code motion. In: Proceedings of the 2005 International Symposium on Code Generation and Optimization, pp. 280–290 (2005)
12. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Proceedings of the 9th European Conference on Object-Oriented Programming, pp. 77–101 (1995)
13. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, pp. 324–341 (1996)
14. Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical virtual method call resolution for Java. In: Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 264–280 (2000)

15. Ramalingam, G.: Data flow frequency analysis. In: Proceedings of the 1996 Conference on Programming Language Design and Implementation, pp. 267–277 (1996)
16. Wu, Y., Larus, J.R.: Static branch frequency and program profile analysis. In: Proceedings of the 27th Annual International Symposium on Microarchitecture, pp. 1–11 (1994)
17. Vallée-Rai, R., Hendren, L.J.: Jimple: simplifying Java bytecode for analyses and transformations. Sable technical report, McGill (1998)
18. Vallée-Rai, R.: The Jimple framework. Sable technical report, McGill (1998)
19. Soot: <http://www.sable.mcgill.ca/soot>
20. Hind, M., Burke, M., Carini, P., deok Choi, J.: Interprocedural pointer alias analysis. ACM Transactions on Programming Languages and Systems 21(4), 848–894 (1999)
21. Do, H., Elbaum, S., Rothermel, G.: Infrastructure support for controlled experimentation with software testing and regression testing techniques. Empirical Software Engineering: An International Journal 10, 405–435 (2004)
22. Ashes Suite Collection, <http://www.sable.mcgill.ca/software>
23. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiederemann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 169–190 (2006)
24. Hind, M.: Pointer analysis: haven't we solved this problem yet? In: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 54–61 (2001)
25. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, pp. 242–256 (1994)
26. Landi, W., Ryder, B.G., Zhang, S.: Interprocedural modification side effect analysis with pointer aliasing. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, pp. 56–67 (1993)
27. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for Java programs. In: Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, pp. 187–206 (1999)
28. Wilson, R., Lam, M.S.: Efficient context-sensitive pointer analysis for C programs. In: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, pp. 1–12 (1995)
29. Yu, H., Xue, J., Huo, W., Feng, X., Zhang, Z.: Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In: Proceedings of the 8th International Symposium on Code Generation and Optimization, pp. 218–229 (2010)
30. Steensgaard, B.: Points-to analysis in almost linear time. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 32–41 (1996)
31. Andersen, L.: Program analysis and specialization for the C programming language. DIKU report 94-19, University of Copenhagen (1994)
32. Foster, J.S., Fähndrich, M., Aiken, A.: Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In: SAS 2000. LNCS, vol. 1824, pp. 175–199. Springer, Heidelberg (2000)
33. Lhoták, O., Hendren, L.: Context-sensitive points-to analysis: Is it worth it? In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 47–64. Springer, Heidelberg (2006)
34. Silva, J.D., Steffan, J.G.: A probabilistic pointer analysis for speculative optimizations. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 416–425 (2006)

Faster Alias Set Analysis Using Summaries

Nomair A. Naeem and Ondřej Lhoták

University of Waterloo, Canada
{nanaeem, olhotak}@uwaterloo.ca

Abstract. Alias sets are an increasingly used abstraction in situations which require flow-sensitive tracking of objects through different points in time and the ability to perform strong updates on individual objects. The interprocedural and flow-sensitive nature of these analyses often make them difficult to scale. In this paper, we use two types of method summaries (callee and caller) to improve the performance of an interprocedural flow- and context-sensitive alias set analysis. We present callee method summaries and algorithms to compute them. The computed summaries contain sufficient escape and return value information to selectively replace flow-sensitive analysis of methods without affecting analysis precision. When efficiency is a bigger concern, we also use caller method summaries which provide conservative initial assumptions for pointer and aliasing relations at the start of a method. Using caller summaries in conjunction with callee summaries enables the alias set analysis to flow-sensitively analyze only methods containing points of interest thereby reducing running time. We present results from empirically evaluating the use of these summaries for the alias set analysis. Additionally, we also discuss precision results from a realistic client analysis for verifying temporal safety properties. The results show that although caller summaries theoretically reduce precision, empirically they do not. Furthermore, on average, using callee and caller summaries reduces the running time of the alias set analysis by 27% and 96%, respectively.

1 Introduction

Inferring properties of pointers created and manipulated by programs has been the subject of intense research [12, 24]. A large spectrum of pointer analyses, from efficient points-to analyses to highly precise shape analyses, have been developed. A useful tradeoff between the two extremes, and an increasingly used abstraction, is the alias set analysis. This static abstraction represents each runtime object with the set of all local pointers that point to it, and no others. The abstraction is neither a may-point-to nor a must-point-to approximation of runtime objects. Instead, each alias set represents exactly those pointers that reference the particular runtime object. As a result, like in a shape abstraction [26], every alias set (except the empty one) corresponds to at most one concrete object at any given point in time during program execution. This ability to statically pinpoint a runtime object enables strong updates which makes the abstraction suitable for analyses that track individual objects [6, 21, 10, 18]. We discuss the alias set analysis in more detail in Section 2.

Unlike a shape analysis which emphasizes the precise relationships between objects, and is expensive to model, an alias set analysis, like a pointer abstraction, focuses on local pointers to objects. This makes computing the alias set abstraction faster than

shape analyses. However, since the analysis is flow-sensitive and inter-procedural it is still considerably slower than most points-to analyses. In this paper we propose two ways to further speed-up the alias set analysis; callee summaries providing effect and return value information and caller summaries that make conservative assumptions at method entry.

Flow sensitive analyses take into account the order of instructions in the program and compute a result for each program point. Although typically more precise than those that are insensitive to program flow, flow-sensitive analyses often have longer execution times than their insensitive counterparts. Computing such precise information for each program point is often overkill; clients of the analysis need precise results only at specific places. Long segments of code might exist where a client neither queries the analysis nor cares about its precision. As an example, consider a static verification tool that determines whether some property of lists and iterators is violated by the code in Figure 1. The verification tool is a client of the alias set analysis as it requires flow-sensitive tracking of individual objects to statically determine runtime objects involved in operations on lists and iterators. Notice that precise alias sets are required only when operations of interest occur. For the example, these are the two calls to `next` at lines 7 and 10 and the call to `add` at line 11. On the other hand, a typical alias set analysis computes flow-sensitive results for all program points irrespective of the fact that it is likely to be queried only at a few places.

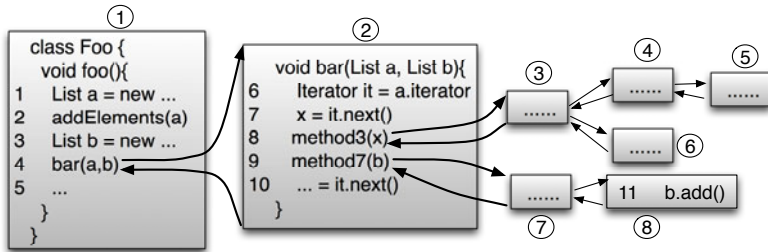


Fig. 1. Sample code illustrating the use of callee and caller summaries

In such situations, we propose the use of a selectively flow-sensitive alias set analysis that uses callee method summaries as a cheaper option. Only methods that contain a point of interest (which we call *shadows*), or transitively call methods containing shadows, are analyzed flow-sensitively. For all other methods, callee summaries providing effect information for the parameters of a method invocation and the possible return value are used. If callee summaries were available, only methods 1, 2, 7 and 8 from Figure 1 would have to be analyzed flow-sensitively since they contain shadows or call methods containing shadows. For the entire segment of code represented by methods 3-6, flow-sensitive information is not required and callee summaries can be used instead. In particular, while analyzing method 2 the alias set analysis need not propagate the analysis into method 3 at line 8 and instead its callee summary can be used. From the client's perspective this is acceptable since it does not query any program point within methods 3-6. In fact, as long as callee summaries contain sufficient information so that

foregoing flow-sensitive analysis of methods without shadows does not affect alias set precision in methods with shadows, the client’s precision will be unaffected. Details of the construction of callee summaries and their use in the alias set analysis are given in Section 3.

The advantage any static analysis derives from interprocedurally analyzing a program is that the analysis need not make conservative worst case assumptions at method entry. This certainly holds true for the alias set analysis. At a callsite, the analysis ensures an appropriate mapping from the caller scope arguments to the callee scope parameters so that alias sets in the callee precisely represent aliasing at the start of the method. However, when efficiency is a bigger concern, we propose the use of caller summaries which are conservative and sound approximations of incoming alias sets. A direct benefit of using such summaries at method entries is that methods that were previously analyzed flow-sensitively only to obtain precise entry mappings for methods containing shadows no longer require flow-sensitive analysis. For example, since methods 1 and 7 in Figure 1 were analyzed flow-sensitively only because they contain calls to methods 2 and 8, with the added use of caller summaries this is no longer required. Only methods 2 and 8 will be analyzed flow-sensitively with caller summaries used to seed their initial alias sets and callee summaries used at all callsites.

Unlike callee summaries, caller summaries can affect the precision of the alias set abstraction since important aliasing information available at a particular callsite might not be propagated into the callee and instead some conservative assumption is made. The degree to which the use of caller summaries affects precision is dependent on the choice of caller summary as well as the client analysis.

This paper makes the following three contributions:

- We describe callee method summaries for the alias set analysis which provide sufficient information at a method callsite to forego flow-sensitive analysis of the callee without a loss of precision in the caller. We present algorithms to compute such summaries and a transfer function that employs the computed summary. (Section 3)
- We present the simplest caller summary as a proof of concept to using such summaries to flow-sensitively analyze even fewer methods. A transfer function for the alias set abstraction that uses both callee and caller summaries is also presented. (Section 4)
- We empirically evaluate the effect of caller summaries on the precision of a realistic client analysis and present precision metrics for the alias set abstraction. The effect on the running time of different incarnations of the alias set analysis is discussed. (Section 5)

2 Alias Set Analysis

The alias set abstraction employs abstract interpretation to summarize all possible runtime environments. The abstraction contains an alias set for every concrete object that could exist at run time at a given program point. The merge operation is a union of the sets of alias sets coming from different control flow paths. A given alias set o^{\sharp} is exactly the set of local variables that point to the corresponding concrete object at run time. Individual alias sets do not represent may- or must- points-to approximations of

runtime objects, although the abstraction subsumes these relationships. If two pointers must point to the same object at a program point, then all alias sets in the abstraction for that point will either contain both pointers or neither. Similarly, if two pointers point to distinct objects at a program point then the abstraction at that point will not contain any alias sets containing both pointers.

2.1 Intermediate Representation and Control Flow Graph

We assume that the program has been converted into an SSA-based intermediate representation containing the following kinds of instructions:

$$s ::= \text{Copy}(x \leftarrow y) \mid \text{Store}(y.f \leftarrow x) \mid \text{Load}(x \leftarrow y.f) \mid \\ \text{Null}(x \leftarrow \mathbf{null}) \mid \text{New}(x \leftarrow \mathbf{new}) \mid \text{Call}(m(p_0 \cdots p_k))$$

The instructions copy pointers between variables, store and load objects to and from fields, assign null to variables, create new objects and call a method m . For method calls, the receiver is specified as the first argument p_0 followed by the arguments p_1 to p_k . ϕ instructions, introduced during SSA conversion, act as copy instructions with a different multi-variable copy for each incoming control flow edge.

The interprocedural control flow graph is created in the standard way; nodes represent instructions and edges specify predecessor and successor relationships. Each procedure begins with a unique *Start* node and ends at a unique *Exit* node. By construction, a call instruction is divided into two nodes; call and return. A call edge connects the call node in the caller with the start node in the callee. A return edge connects the exit node in the callee with the return node in the caller. A *CallFlow* edge connects a call node to its return node completely bypassing the callee (Figure 2). This edge is parameterized with the method it bypasses and the variable the return from the call is assigned to.

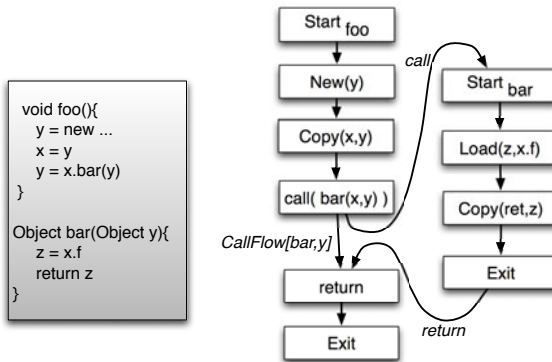


Fig. 2. Interprocedural control flow graph with *call*, *return* and *CallFlow* edges

2.2 Intra-procedural Alias Set Analysis

Flow-sensitivity enables the alias set analysis to precisely track abstract objects through different points in time. The analysis mimics the effect of program instructions in changing the targets of pointers and accordingly updates the alias sets representing each

runtime object. For example, consider the instruction $x \leftarrow \mathbf{new}$. At runtime an object o is allocated in the heap and x points to that object. Correspondingly, the static abstraction creates the alias set $\{x\}$ representing the object's abstraction o^\sharp . Since a pointer can only point to one concrete object at a time, x points to the newly created object and none other. If a copy instruction $y \leftarrow x$ creates a new reference to the runtime object o the analysis mimics this effect by updating the alias set to $\{x, y\}$. Hence, at all times each concrete object is represented by some alias set, though due to the conservative nature of the analysis there may be alias sets which represent no runtime object. For most instructions in the program, given an alias set representing some runtime object o , it is possible to compute the exact set of pointers which will point to o after the execution of the instruction.

An exception to this is the load from the heap ($v \leftarrow \mathbf{e}$). Since the abstraction only tracks local variables, the analysis is uncertain whether the object being loaded is represented by a given alias set o^\sharp before the instruction, and whether the destination variable v should therefore be added to o^\sharp . To be conservative, the analysis accounts for both possibilities and creates two alias sets, one containing v ($o^\sharp \cup \{v\}$) and one not containing v ($o^\sharp \setminus \{v\}$). At this point a straightforward optimization can be applied; only objects that had previously escaped to the heap via a Store can be loaded. We have implemented this optimization in the alias set abstraction. At each program point, the abstraction computes two sets of alias-sets ρ^\sharp and h^\sharp with the condition that $h^\sharp \subseteq \rho^\sharp$ and that h^\sharp contains only those alias sets which are abstractions of run time objects that have escaped into the heap.

In previous work [18] we presented the intra-procedural transfer functions for the alias set abstraction which we reproduce in Figure 3. The core of the transfer function is the helper function $\llbracket s \rrbracket_{o^\sharp}^1$ which, depending on the instruction, updates an existing

$$\begin{aligned}
\llbracket s \rrbracket_{o^\sharp}^1(o^\sharp) &\triangleq \begin{cases} o^\sharp \cup \{v_1\} & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \in o^\sharp \\ o^\sharp \setminus \{v_1\} & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \notin o^\sharp \\ o^\sharp \setminus \{v\} & \text{if } s \in \{v \leftarrow \mathbf{null}, v \leftarrow \mathbf{new}\} \\ o^\sharp & \text{if } s = \mathbf{e} \leftarrow v \\ \text{undefined} & \text{if } s = v \leftarrow \mathbf{e} \end{cases} \\
\mathit{focus}[h^\sharp](v, o^\sharp) &\triangleq \begin{cases} \{o^\sharp \setminus \{v\}\} & \text{if } o^\sharp \notin h^\sharp \\ \{o^\sharp \setminus \{v\}, o^\sharp \cup \{v\}\} & \text{if } o^\sharp \in h^\sharp \end{cases} \\
\llbracket s \rrbracket_{O^\sharp}^1[h^\sharp](O^\sharp) &\triangleq \begin{cases} \bigcup_{o^\sharp \in O^\sharp} \llbracket s \rrbracket_{o^\sharp}^1(o^\sharp) & \text{if } s \neq v \leftarrow \mathbf{e} \\ \bigcup_{o^\sharp \in O^\sharp} \mathit{focus}[h^\sharp](v, o^\sharp) & \text{if } s = v \leftarrow \mathbf{e} \end{cases} \\
\llbracket s \rrbracket_{\rho^\sharp}^1(\rho^\sharp, h^\sharp) &\triangleq \llbracket s \rrbracket_{\text{gen}}^1 \cup \llbracket s \rrbracket_{O^\sharp}^1[h^\sharp](\rho^\sharp) \\
\llbracket s \rrbracket_{\text{gen}}^1 &\triangleq \begin{cases} \{\{v\}\} & \text{if } s = v \leftarrow \mathbf{new} \\ \emptyset & \text{otherwise} \end{cases} \\
\llbracket s \rrbracket_{h^\sharp}^1(\rho^\sharp, h^\sharp) &\triangleq \llbracket s \rrbracket_{O^\sharp}^1[h^\sharp] \left(\begin{cases} h^\sharp \cup \{o^\sharp \in \rho^\sharp : v \in o^\sharp\} & \text{if } s = \mathbf{e} \leftarrow v \\ h^\sharp & \text{otherwise} \end{cases} \right) \\
\llbracket s \rrbracket_{\rho, h^\sharp}^1(\rho^\sharp, h^\sharp) &\triangleq \langle \llbracket s \rrbracket_{\rho^\sharp}^1(\rho^\sharp, h^\sharp), \llbracket s \rrbracket_{h^\sharp}^1(\rho^\sharp, h^\sharp) \rangle
\end{aligned}$$

Fig. 3. Transfer functions on individual alias sets. The superscript¹ identifies the version of the transfer function; we will present modified versions of the transfer functions later in the paper.

alias set. For a copy instruction ($v_1 \leftarrow v_2$) any alias set that contains the source variable v_2 is modified by adding the target variable v_1 , since after the instruction the source and target both point to the same location. Since a pointer can only point to one location at a time, instructions that overwrite a variable v modify an existing alias set by removing v as after the instruction v no longer points to the runtime object abstracted by this set. The store instruction ($\mathbf{e} \leftarrow v$) has no effect on an alias set since alias sets by definition only track local variables.

The *focus* operator in Figure 3 handles the uncertainty due to heap loads. As discussed earlier, only objects that were previously stored in the heap can be loaded. Therefore, for alias-sets not in h^\sharp , $\text{focus}(v, o^\sharp)$ removes v from o^\sharp since the loaded object cannot possibly be represented by o^\sharp and after the assignment v no longer points to o^\sharp . On the flip side, if o^\sharp represents an escaped object, then it is split into two, one representing the single concrete object that may have been loaded, and the other representing all other objects previously represented by o^\sharp .

Two additional special cases are handled. First, for a store ($\mathbf{e} \leftarrow v$), all abstract objects that contain the variable v are added to h^\sharp . Second, for an allocation instruction, a new alias set containing only the destination variable v is created and added to ρ^\sharp .

Figure 4 graphically shows the effect of a sequence of three instructions on the alias set abstraction. For illustration we assume that before the first instruction, ρ^\sharp and h^\sharp already contain an alias set $\{x, z\}$ i.e. an abstraction of an object that is pointed to by local variables x and z and might also have external references from the heap. Note also the presence of a single empty alias set which represents all runtime objects that are not referenced through any local variables. This keeps the abstraction finite. With the allocation instruction, a new alias-set $\{x\}$ is added to ρ^\sharp . At the same time, $\llbracket s \rrbracket_o^\sharp$ removes x from the alias set $\{x, z\}$ since x no longer points to this runtime object. After the copy instruction both y and z point to the same runtime object. The heap load highlights a number of analysis features. First, note that the analysis determines that the loaded object cannot be the newly created object from instruction 1. Second, since $\{x, z\}$ was in h^\sharp , so is $\{y, z\}$. The analysis applies the *focus* operator. Third, notice the creation of the alias set $\{z\}$ which represents a loaded object that previously had no local variable references. Figure 4 illustrates the two key properties of alias set analysis (i) the abstraction can distinguish individual objects i.e. each alias set represents at most one runtime object and (ii) the transfer functions flow-sensitively track the effect of instructions on pointers. Each column represents what happens to a particular concrete

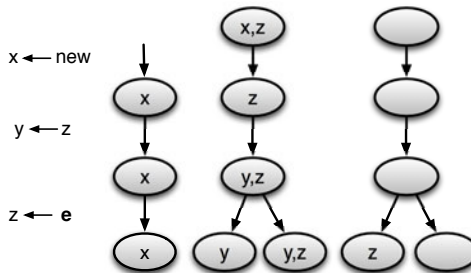


Fig. 4. An illustration of the transfer functions for computing the alias set abstraction

object as different instructions execute; all that changes is the set of pointers pointing to the object at different program points.

2.3 Inter-procedural Alias Set Analysis

The intra-procedural transfer function can be extended to be inter-procedural by defining the transfer functions for *call* and *return*. The overall effect of calling a function m is $\llbracket \text{return} \rrbracket \circ \llbracket m \rrbracket \circ \llbracket \text{call} \rrbracket$ for each possible callee. To determine the callees possible at each call site, we used a call graph computed using the default subset-based points-to analysis implemented in Spark [16]. The function $\llbracket \text{call} \rrbracket_{o^\sharp}$ is straightforward to define; actual arguments in each alias set are replaced by the corresponding parameters and all other variables are removed. Given a substitution r that maps each argument to its corresponding parameter, the function is defined in Figure 5.

$$\begin{aligned} \llbracket \text{call} \rrbracket_{o^\sharp}^1(o^\sharp) &\triangleq \left\{ r(v) : v \in o^\sharp \cap \text{dom}(r) \right\} \\ rv(o_c^\sharp, o_r^\sharp) &\triangleq \begin{cases} o_c^\sharp & \text{if } p \text{ does not return a value} \\ o_c^\sharp \cup \{v_t\} & v_s \in o_r^\sharp \\ o_c^\sharp \setminus \{v_t\} & v_s \notin o_r^\sharp \end{cases} \\ \llbracket \text{return} \rrbracket_{o^\sharp}^1(o_c^\sharp) &\triangleq \{rv(o_c^\sharp, o_r^\sharp) : o_r^\sharp \in \llbracket m \rrbracket \circ \llbracket \text{call} \rrbracket\} \end{aligned}$$

Fig. 5. Transfer functions for $\llbracket \text{call} \rrbracket_{o^\sharp}$ and $\llbracket \text{return} \rrbracket_{o^\sharp}$

Defining the return from a function m is more challenging since any object that might be returned by m is abstracted by some alias set containing variables local to m . On its own this is insufficient to map variables from a callee alias set to one in the caller since it is unknown which caller variables, if any, pointed to the object before the call. Instead, the analysis uses a function that, given a call site and the computed flow function $\llbracket m \rrbracket \circ \llbracket \text{call} \rrbracket$, computes the appropriate caller-side alias sets after the function returns. For the $\llbracket \text{return} \rrbracket_{o^\sharp}$ function in Figure 5, o_c^\sharp is the caller-side abstraction of an object existing before the call and the set $\llbracket m \rrbracket \circ \llbracket \text{call} \rrbracket$ contains all possible callee-side alias sets (o_r^\sharp) that could be returned. The function rv takes each such pair (o_c^\sharp, o_r^\sharp) , where v_s is the callee variable being returned and v_t is the caller variable to which the return value is assigned. Intuitively, if the object that was represented by o_c^\sharp in the caller before the call is returned from the callee (i.e. $v_s \in o_r^\sharp$), then v_t is added to o_c^\sharp . If some other object is returned, then v_t is removed from o_c^\sharp , since v_t gets overwritten by the return value. In the case of an object newly created within the callee, the empty set is substituted for o_c^\sharp , since no variables of the caller pointed to the object before the call. Overall, $\llbracket \text{return} \rrbracket_{o^\sharp}$ yields the set of possible caller-side alias sets of the object after the call. We refer the interested reader to our previous work [20] for more details.

3 Callee Summaries

Although precise, the alias set analysis in its original form is expensive to compute. Using efficient data structures [19] and algorithms [22, 20] only improves the efficiency

to some extent. In situations where a faster running time is desired we propose the use of method summaries. In this section, we discuss the use of callee summaries that decrease the computation load, without any effect on a client analysis.

The key insight is that clients of a flow-sensitive whole program analysis often need precise information at a small subset of program points. On the other hand, a flow-sensitive program analysis computes precise information at all program points and therefore computes a lot more information than required. Computing this unnecessary information is wasteful and should be avoided. We use callee summaries to achieve this.

Before we explain the contents of a callee summary let us see how the alias set analysis can use such summaries. Consider a callsite, with a target method m . If an oracle predicts that a client of the alias set analysis never queries any program point within m or any methods transitively called by m , then computing flow-sensitive alias results for all methods in the transitive closure of m is unnecessary. Instead a callee summary, which provides information regarding the parameters and return value, could be used. For many client analyses such an oracle exists. In Section 5 we discuss one such client analysis that leverages alias sets in proving temporal properties of objects. The points of interest for this analysis i.e. the shadows, are operations that change the state an object is in and are statically known ahead of time. Additionally, callee summaries can be used for methods in the standard library; the alias set analysis can be seeded to use callee summaries for all chains of calls into the library. Analyses such as those detecting memory leaks and automatically deallocating objects [6, 21], that already use alias sets, could benefit from such summaries to only analyze application code.

The key requirement we put on a callee summary is that it should enable the analysis to bypass flow-sensitively analyzing a method without impacting precision in the caller. Table 1 provides a summary of the contents of such a summary. The summary is divided into escape (α_{esc}) and return value (α_{ret}) information.

Table 1. Callee Summary for a callsite with target method m

Escape Information (α_{esc})	
params	set of parameters (including receiver) that may be stored into the heap by m or procedures transitively called by m
Return Value Information (α_{ret})	
params	set of parameters (including receiver) that might be returned by m .
heap	might an object loaded from the heap be returned?
fresh	might a newly created object be returned?
escaped	might a newly created object be stored in the heap before being returned?
null	might a null reference be returned?

To determine the contents of a callee summary one must understand the effect of a method call on the alias set abstraction. First, the callee might escape the receiver or arguments of the call. This might occur *directly*, when a callee’s parameter is stored in a field, or *indirectly*, when a parameter is copied to a local reference which is then stored. In Figure 6 the function $\text{f}\circ\circ$ escapes both its parameters, p directly via a store to

field f of class `Foo` and q indirectly by first copying the reference to y and then storing in `Foo.f`. Therefore, a callee summary analysis must track such copies and ultimately provide a list of all parameters that might have escaped.

Second, the return value from the callee might be assigned to a reference in the caller. To see how this might affect aliasing in the caller consider once again the example in Figure 6. The function `foo` returns the pointer y which is a copy of q , one of `foo`'s parameters. Therefore, the returned reference is the argument which is mapped to q , in this case variable b . At run time, the effect of calling `foo` is that after the call, a and b must point to the same object. Let us examine the effect on the abstraction at the callsite if the interprocedural transfer functions from Figure 5 were used. $\llbracket call \rrbracket$ determines that b and q point to the same location and $\llbracket foo \rrbracket$ determines that q and y point to the same location. This leads $\llbracket return \rrbracket$ to infer that since b and y point to the same location and y is assigned to a , b and a must point to the same location after the call; an alias set containing both a and b is created in the caller. In order to forego flow-sensitive analysis of `foo` in favour of a callee summary, the summary must specify which of the callee's parameters might be returned so that similar updates can be made at the callsite. Other possible returned references include references to newly created objects or those loaded from the heap.

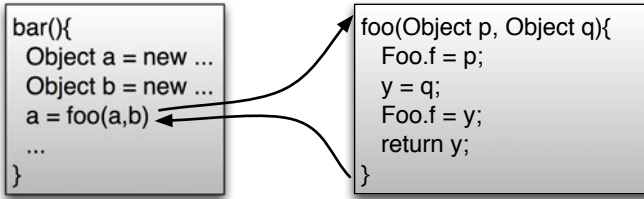


Fig. 6. An example illustrating the effect of a method call on alias sets in the caller

3.1 Computing Callee Summaries

The algorithm to compute the set of parameters that escape (α_{esc}) from a method m is presented in Figure 7. The algorithm takes as input a SSA-based control flow graph of the method and returns a set of indices which refer to the positions of parameters in the method's signature which might have escaped [1]. Lines 1-10 populate a worklist with variables that either escaped through a store or through a function call from within m . The algorithm then proceeds through each variable v in the worklist. Using the SSA property that each variable has a single reaching definition the algorithm retrieves the unique definition def of v (line 15). If def represents the Start node then v is a receiver or a parameter and the appropriate index is added to the mayEscape set. For a copy instruction $v \leftarrow s$, s is added to the worklist, since v and s both point to the same escaped object. Notice that the order between the instruction that escapes v and the copy from s to v does not matter, since in SSA-form once a variable is defined its value

¹ Recall from Section 2 that we write a function call as $m(p_0, \dots, p_k)$ where p_0 denotes the receiver of the call and p_1 to p_k are the arguments.

```

input: SSA-based CFG of method  $m$ 
output: mayEscape
declare mayEscape : Set[Int], WorkList: FIFOWorklist[Var], seen : Set[Var]
1 foreach instruction  $inst \in \text{cfg}$  do
2   switch  $inst$ 
3     case  $inst = \text{Store}(v)$  : add  $v$  to WorkList end-case
4     case  $inst = \text{CallSite}(\text{args}, \text{retval})$  :
5       foreach  $tgt \in \text{callees}(inst)$  do
6         WorkList += {  $\text{args}(i) : i \in \text{EscapeSummaries}(tgt)$  }
7       od
8     end-case
9   end-switch
10 od
11 while WorkList not empty
12   Select and Remove variable  $v$  from WorkList
13   if seen contains  $v$  then continue fi
14   add  $v$  to seen
15    $def = \text{uniqueDef}(\text{cfg}, v)$ 
16   switch  $def$ 
17     case  $def = \text{Start}(p_0 \dots p_k)$ : mayEscape += {  $i : p_i = v$  } end-case
18     case  $def = \text{Copy}(v, s)$ : add  $s$  to WorkList end-case
19     case  $def = \text{CallSite}(\text{args}, \text{retval})$ :
20       foreach  $tgt \in \text{callees}(def)$  do
21         WorkList += {  $\text{args}(i) : i \in \text{RetValSummaries}(tgt).\text{params}$  }
22       od
23     end-case
24     case  $def = \text{Phi}$ : foreach  $\text{Copy}(v, s) \in \text{phi.defs}(v)$  do add  $s$  to WorkList od end-case
25   end-switch
26 od

```

Fig. 7. Algorithm to compute callee escape summary (α_{esc}) for a method m

remains unchanged. If variable v is assigned the return value from a function call then all arguments corresponding to the parameters that might be returned are added to the worklist since these might have escaped (lines 19-23). A SSA ϕ instruction acts as a multi-variable copy statement.

Figure 8 presents the algorithm to compute the return value summary for a function m . The algorithm maintains a worklist of variables that might be returned. The worklist is seeded with the unique return variable of m . For each variable v in the worklist, depending on its unique definition, the return value summary and the worklist are updated. In lines 12-14, if v is defined at the `Start` node then, since a `Start` node defines the receiver or parameters of method m , the corresponding index of the parameter is stored in `params`. This represents the situation when the receiver or a parameter to m might be returned. Lines 15-23 update the return value summary if v is assigned the return value at a callsite. The return value summaries of all possible target methods at the callsite are consulted and the `fresh`, `heap` and `null` fields of the summary of m appropriately updated. If any of the return value summaries indicate that a receiver or parameter might be returned the corresponding argument is added to the worklist. *Copy*

```

input: SSA-based CFG of method  $m$ 
output: retValSum
declare WorkList: FIFOWorklist[Var], seen : Set[Var]
1  retValSum = RetValSum { params: Set[Int], heap = fresh = escaped = null = false }
4  if  $m.isVoid$  then return retValSum fi
5  Insert unique return variable into WorkList
6  while WorkList not empty
7    Select and remove variable  $v$  from WorkList
8    if seen contains  $v$  then continue fi
9    add  $v$  to seen
10  $def = uniqueDef(cfg, v)$ 
11 switch  $def$ 
12   case  $def = Start(p_0 \dots p_k)$  :
13     retValSum.params += {  $i : p_i = v$  }
14   end-case
15   case  $def = CallSite( args, retval )$  :
16     foreach  $tgt \in callees(def)$  do
17       calleeRetValSum = RetValSummaries( $tgt$ )
18       if calleeRetValSum.fresh then retValSum.fresh = true fi
19       if calleeRetValSum.heap then retValSum.heap = true fi
20       if calleeRetValSum.null then retValSum.null = true fi
21       WorkList += {  $args(i) : i \in calleeRetValSum.params$  }
22     od
23   end-case
24   case  $def = Copy(v, s)$  : add  $s$  to WorkList end-case
25   case  $def = Phi$  :
26     foreach  $Copy(v, s) \in phi.defs(v)$  do add  $s$  to WorkList od
27   end-case
28   case  $def = Load$  : retValSum.heap = true end-case
29   case  $def = New$  : retValSum.fresh = true end-case
30   case  $def = Null$  : retValSum.null = true end-case
31 end-switch
32 foreach  $inst \in cfg$  do
33   switch  $inst$ 
34     case  $inst = Store(v)$  :
35       if seen contains  $v$  then retValSum.escaped = true fi
36     end-case
37     case  $inst = CallSite( args, retval )$  :
38       foreach  $tgt \in callees(inst)$  do
39         if seen contains  $args(i) : i \in EscapeSummaries(tgt)$  then
40           retValSum.escaped = true
41         fi
42       od
43     end-case
44   end-switch
45 od

```

Fig. 8. Algorithm to compute the return value summary (α_{ret}) for a method m

and *Phi* instructions add sources of assignments to the worklist. *Load*, *New* and *Null* instructions require an update to the corresponding heap, *fresh* and *null* fields of the return value summary. Two special cases must also be handled; if any possibly returned variable was stored in a field or escaped by a function called by *m*, *escaped* is set to true.

Since the callee summary of a function *m* depends on summaries of functions called by *m*, the algorithms presented must be wrapped in an interprocedural fixed-point computation. A worklist keeps track of all functions whose summaries may need to be recomputed. Whenever the summary of a function changes, all of its callers are added to the worklist. The computation iterates until the worklist becomes empty.

3.2 Using Callee Summaries

To leverage callee summaries in the alias set analysis the transfer functions from Figures 3 and 5 are modified. These modifications are presented in Figure 9. We denote the set of methods that contain shadows or transitively call methods containing shadows by M^* . The function $\llbracket call \rrbracket_{o^\sharp}^1$ is modified so that arguments in the caller are mapped to parameters in the callee only for methods in M^* , the methods that are still analyzed flow-sensitively. Since return instructions are only encountered in methods not using callee summaries no change is required to the return function $\llbracket return \rrbracket_{o^\sharp}^1$.

For methods not in M^* , we define the transfer function $\llbracket CallFlow \rrbracket$ for the similarly named edge connecting a call node to a return node in the caller. The $\llbracket CallFlow \rrbracket$ function uses two helper functions `mustReturn` and `mightReturn` which employ the return value summary α_{ret} to update alias sets by simulating the effect of analyzing the callee. The function `mustReturn` is true only when the object represented by o^\sharp before the call is returned by the callee. Therefore the *null*, *fresh* and *heap* flags of the return value summary should be false since a non-null object, that was not allocated in the callee nor loaded from the heap should be returned. Additionally, o^\sharp must contain the corresponding arguments of all parameters that might be returned by the callee. Parameters that might be returned are given by $\alpha_{ret}.params$ and the corresponding arguments are retrieved through the inverse function \bar{r} , where *r* is the function mapping arguments to parameters.

The helper function `mightReturn` determines whether o^\sharp might be returned. This is true if o^\sharp represents an escaped object ($o^\sharp \in h^\sharp$) and an object loaded from the heap might be returned. An object representing o^\sharp might also be returned if at least one parameter, whose corresponding argument is in o^\sharp , might be returned. To handle the uncertainty when `mightReturn` is true, $\llbracket CallFlow \rrbracket$ accounts for both possibilities similarly to the *focus* operation. If the returned object must not be o^\sharp then the variable assigned from the return of the callee cannot possibly point to o^\sharp after the call.

The callee escape summary α_{esc} is utilized to update alias sets representing objects that might escape due to the function call (the function *escape* in Figure 9). If any of the corresponding arguments to parameters that escape ($\alpha_{esc}.params$) are in an alias set in ρ^\sharp , the alias set is added to h^\sharp since the function call escapes the parameter. The transfer function must also handle situations when the callee allocates and returns a new object. If $\alpha_{ret}.fresh$ is true and the return from the callee is assigned to variable *v*, an alias set

$$\begin{aligned}
\llbracket call \rrbracket_{o^\sharp}^2(o^\sharp) &\triangleq \begin{cases} \llbracket call \rrbracket_{o^\sharp}^1(o^\sharp) & \text{if } s = call \wedge target(call) \in M^* \\ \emptyset & \text{otherwise} \end{cases} \\
mustReturn(o^\sharp, \alpha_{ret}) &\triangleq \begin{cases} true & \text{if } !\alpha_{ret}.fresh \wedge !\alpha_{ret}.heap \wedge !\alpha_{ret}.null \wedge \\ & \forall p, p \in o^\sharp : p \in \overline{\tau}(\alpha_{ret}.params) \\ false & \text{otherwise} \end{cases} \\
mightReturn[h^\sharp](o^\sharp, \alpha_{ret}) &\triangleq \begin{cases} true & \text{if } (o^\sharp \in h^\sharp \wedge \alpha_{ret}.heap) \vee \\ & \exists p, p \in o^\sharp : p \in \overline{\tau}(\alpha_{ret}.params) \\ false & \text{otherwise} \end{cases} \\
\llbracket CallFlow \rrbracket_{o^\sharp}^2[h^\sharp, m, v](o^\sharp) &\triangleq \begin{cases} \emptyset & \text{if } m \in M^* \\ \{o^\sharp \cup v\} & \text{if } m \notin M^* \wedge mustReturn(o^\sharp, m.\alpha_{ret}) \\ \{o^\sharp \setminus v, o^\sharp \cup v\} & \text{if } m \notin M^* \wedge mightReturn[h^\sharp](o^\sharp, m.\alpha_{ret}) \\ \{o^\sharp \setminus v\} & \text{otherwise} \end{cases} \\
\llbracket s \rrbracket_{ret}^2 &\triangleq \begin{cases} \{\{v\}\} & \text{if } s = CallFlow[m, v] \wedge m \notin M^* \wedge m.\alpha_{ret}.fresh \\ \emptyset & \text{otherwise} \end{cases} \\
\llbracket s \rrbracket_{O^\sharp}^2[h^\sharp](O^\sharp) &\triangleq \begin{cases} \bigcup_{o^\sharp \in O^\sharp} \llbracket s \rrbracket_{o^\sharp}^1(o^\sharp) & \text{if } s \notin \{v \leftarrow e, CallFlow[m, v]\} \\ \bigcup_{o^\sharp \in O^\sharp} \llbracket CallFlow \rrbracket_{o^\sharp}^2[h^\sharp, m, v](o^\sharp) & \text{if } s = CallFlow[m, v] \\ \bigcup_{o^\sharp \in O^\sharp} focus[h^\sharp](v, o^\sharp) & \text{if } s = v \leftarrow e \end{cases} \\
\llbracket s \rrbracket_{\rho^\sharp}^2(\rho^\sharp, h^\sharp) &\triangleq \llbracket s \rrbracket_{gen}^1 \cup \llbracket s \rrbracket_{ret}^2 \cup \llbracket s \rrbracket_{O^\sharp}^1[h^\sharp](\rho^\sharp) \\
escape(\rho^\sharp, h^\sharp, m) &\triangleq h^\sharp \cup \{o^\sharp \in \rho^\sharp : \exists p, p \in o^\sharp : p \in \overline{\tau}(m.\alpha_{esc}.params)\} \\
\llbracket s \rrbracket_{esc}^2(m) &\triangleq \begin{cases} \{\{v\}\} & \text{if } m.\alpha_{ret}.fresh \wedge m.\alpha_{ret}.escaped \\ \emptyset & \text{otherwise} \end{cases} \\
\llbracket s \rrbracket_{h^\sharp}^2(\rho^\sharp, h^\sharp) &\triangleq \begin{cases} \llbracket s \rrbracket_{O^\sharp}^2[h^\sharp](h^\sharp \cup \{o^\sharp \in \rho^\sharp : v \in o^\sharp\}) & \text{if } s = e \leftarrow v \\ \llbracket s \rrbracket_{esc}^2(m) \cup \llbracket s \rrbracket_{O^\sharp}^2[h^\sharp]escape(\rho^\sharp, h^\sharp, m) & \text{if } s = CallFlow[m, v] \wedge m \notin M^* \\ \llbracket s \rrbracket_{O^\sharp}^2h^\sharp & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 9. Modified transfer functions for the alias set analysis using callee summaries

containing only v is added to ρ^\sharp . If the freshly created object might have been stored in the heap before being returned ($\alpha_{ret}.escaped$ is true) a similar alias set is added to h^\sharp .

4 Caller Summaries

In this section we present caller summaries as a mechanism to speed up the interprocedural context-sensitive alias set analysis. Although static analyses that infer properties of pointers can be useful even when the analysis is carried out locally on individual methods, such analyses shine most when computed interprocedurally. The added ability to carry forward computed pointer and aliasing information from a caller into a callee by mapping arguments to parameters can significantly improve precision. However, when efficiency is a bigger concern a natural trade-off is to forego some precision by conservatively assuming initial pointer and aliasing relationships for the parameters of a method.

The reason using caller summaries improves efficiency is that it decreases the number of the methods the must be analyzed flow-sensitively. Let us revisit the example in

Figure 11. Using callee summaries enables the alias set analysis to discard flow-sensitive analysis of methods 3-6 since they do not contain any shadows. However, even though methods 1 and 7 do not contain shadows, they are analyzed flow-sensitively to ensure that at a callsite to a method containing a shadow, precise information can be mapped into the callee. In an analysis that uses caller summaries to make conservative assumptions at every method entry, flow-sensitively analyzing such methods is un-needed since the precise information computed at the callsite will never be propagated into the callee.

We have implemented a conservative mechanism for computing caller summaries. In our summaries, initial alias sets are created for the parameters of a method such that the abstraction at the start of the method specifies that any two parameters might be aliased. In our intermediate representation the method `bar` in the example from Figure 11 has three parameters; the `this` receiver and the two `List` references `a` and `b`. The caller summary for this method contains the following sets: $\{\}$, $\{this\}$, $\{a\}$, $\{b\}$, $\{this,a\}$, $\{this,b\}$, $\{a,b\}$ and $\{this,a,b\}$. Notice that given these alias sets the only conclusion that can be drawn is that the three parameters might be aliased i.e. no must or must-not relationships exist between the parameters. This is overly conservative. Firstly, the caller summary does not take into account any type information. Although `a` and `b` are both references to a `List` data structure, the receiver `this` is of type `Foo` and, unless `Foo` is declared a supertype of `List`, a reference of type `Foo` can never point to a `List` object. Secondly, the caller summaries do not leverage any pointer information. For example, subset-based points-to analyses that use allocation sites as their object abstraction are often performed at onset for constructing a callgraph. Using this type of pointer analysis could potentially improve the precision of the caller summary in situations where the pointer analysis can specify that parameters `a` and `b` were created at different allocation sites.

Our reason for using a naive caller summary was to investigate the maximum precision degradation due to such summaries. Whereas the callee summaries presented in the preceding section do not affect precision, caller summaries do. As an example let us look more closely at the example in Figure 11. The method `bar` receives two `List` references, `a` and `b`. An alias set analysis which does not utilize caller summaries is able to differentiate between the two references. In particular, at the start of method `bar` the analysis infers that `a` and `b` must-not alias (two separate lists were created at lines 1 and 3 and assigned to `a` and `b` respectively, and a reference of one is never copied to the other). However, the naive caller summary assumes that `a` and `b` could be aliased. Hence the precision of the alias set analysis degrades i.e. fewer must-not facts are computed.

This decrease in precision can cascade into client analyses. For example suppose a client of the alias set analysis is a verification tool for the property that an iterator's underlying list structure has not been modified when its `next` method is invoked (executing such code results in a runtime exception). If caller summaries are not used, the analysis infers that the iterator's underlying list i.e. the list referenced by `a`, is never modified since `a` and `b` must-not point to the same object and the code only modifies the list referenced by `b`. Hence, the client analysis can prove that line 10 is not a violation of the property. However, when caller summaries are used, the client analysis infers that the list pointed to by reference `a` might be modified (the caller summary suggests that `a` and `b` might be aliased and an element is added at line 11 to the list pointed to by

reference b). Hence the client analysis loses precision since it can no longer prove that the `next` operation at line 10 is safe w.r.t. the property being verified. We empirically evaluate the loss in precision of using caller summaries on the alias set analysis and a client analysis in Section 5.

$$\begin{aligned}
\llbracket call \rrbracket_{o^\sharp}^3(o^\sharp) &\triangleq \text{callerSummaries}(\text{target}(call)) \\
\llbracket CallFlow \rrbracket_{o^\sharp}^3[h^\sharp, m, v](o^\sharp) &\triangleq \begin{cases} \{o^\sharp \cup v\} & \text{if } \text{mustReturn}(o^\sharp, m, \alpha_{ret}) \\ \{o^\sharp \setminus v, o^\sharp \cup v\} & \text{if } \text{mightReturn}[h^\sharp](o^\sharp, m, \alpha_{ret}) \\ \{o^\sharp \setminus v\} & \text{otherwise} \end{cases} \\
\llbracket s \rrbracket_{h^\sharp}^3(\rho^\sharp, h^\sharp) &\triangleq \begin{cases} \llbracket s \rrbracket_{o^\sharp}^2[h^\sharp] (h^\sharp \cup \{o^\sharp \in \rho^\sharp : v \in o^\sharp\}) & \text{if } s = e \leftarrow v \\ \llbracket s \rrbracket_{\text{esc}}^2(m) \cup \llbracket s \rrbracket_{o^\sharp}^2[h^\sharp] \text{escape}(\rho^\sharp, h^\sharp, m) & \text{if } s = \text{CallFlow}[m, v] \\ \llbracket s \rrbracket_{o^\sharp}^2[h^\sharp] (h^\sharp) & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 10. Updated transfer functions for the alias set analysis using callee and caller summaries

Modifying the alias set analysis to use caller summaries is straightforward. Figure 10 shows those transfer functions which have been modified from their earlier version (Figure 9). First, the `call` function is modified. Instead of mapping arguments to parameters, the caller summary provides the set of alias sets to seed the callee’s analysis. Second, callee summaries are used for all methods instead of only those not in M^* .

5 Experiments

Any standard dataflow analysis framework can be used to compute the alias set abstraction using the original transfer functions from Section 2 or the subsequently modified versions from Sections 3 and 4. We have chosen to implement these incarnations as instances of the efficient interprocedural finite distributive subset (IFDS) algorithm of Reps et al. [22]. The algorithm requires an analysis domain $\mathcal{P}(D)$ for some finite set D , and transfer functions that are distributive. The alias set abstraction satisfies these conditions where D is the set of all possible alias sets. The key to IFDS’s efficiency is the distributivity of transfer functions which enables it to evaluate the functions on individual alias sets, rather than on the entire set of alias sets at a program point. Space restrictions limit us in providing interesting details of the algorithm. Instead we refer the interested reader to the ever growing body of work discussed in Section 6.

For the experiments we used the DaCapo Benchmark suite, version 2006-10-MR2 with the standard library from JDK 1.3.1_12 for antlr, pmd and bloat, and JDK 1.4.2_11 for the rest, since they use features not present in JDK 1.3. The intermediate representation is constructed using the Soot Framework [29] with reflective class loading modelled through reflection summaries obtained using ProBe [15] and *J [9]. To give an indication of the size of these benchmarks we computed the number of methods statically reachable in the control flow graph created by Soot and present these in Table 2. Time taken to pre-compute the callee summaries using the algorithms discussed in Section 3 are also shown.

In our experiments we have used a static analysis that verifies conformance to temporal properties specified using a statemachine-based specification [2] as a client of the

Table 2. Number of statically reachable methods and the time to precompute callee summaries

Benchmark	antlr	bloat	chart	fop	hsqldb	jspx	luindex	lusearch	pmd	xalan
Reachable Methods	4452	5955	14912	27408	11418	14437	7358	7821	9365	14961
Callee SummaryTime(s)	8	12	29	72	28	50	10	10	15	29

alias set abstraction. In previous work [18], we presented a two stage approach to verifying such properties. In the first stage an alias set abstraction of objects in the program is computed. The second stage uses this abstraction to compute an abstraction for the state an object, or group of objects, is in. This enables statically verifying whether the state machine might end up in an error state indicating a violation of the property.

Our choice of client analysis was dictated by two reasons. First, each temporal property specifies its own points of interest; only events that transition the state machine of that property are considered shadows. By choosing different properties we ensure a varying set M^* , the set of methods for which callee summaries are used. The properties we experimented with are presented in Table 3. The code fragment from Figure 1 uses the FSI property. Shadows for FSI are the next operation on an iterator and updates on the `Collection` type e.g. `add`, `clear`, `remove`. A second reason for choosing this client analysis is that it cleanly teases apart the computation of the alias set abstraction and its use in computing the state abstraction. This enables us to measure the precision and efficiency of the alias set abstraction in a real-world scenario.

Table 3. Temporal properties investigated to obtain a varying set M^*

FailSafeEnum (FSE): A vector should not be updated while enumerating it
FailSafeEnumHash (FSEH): A hashtable should not be updated while enumerating its keys/values
FailSafeIter (FSI): A collection should not be updated while iterating over it
HasNext (HN): The <code>hasNext</code> method should be called prior to every call to <code>next</code> on an iterator
HasNextElem (HNE): The <code>hasNextElem</code> method should be called prior to every call to <code>nextElement</code> on an enumeration
Reader (R): A Reader should not be used after its <code>InputStream</code> has been closed
Writer (W): A Writer should not be used after its <code>OutputStream</code> has been closed

We call a pair containing a benchmark and temporal property a *test case*. Since not all benchmarks exercise all temporary properties we have chosen to present results only for test cases when a temporal property is applicable for a benchmark e.g. the `antlr` benchmark never uses a `Writer` and hence the corresponding temporal property is inapplicable.

5.1 Shadow Statistics

Section 3 proposed the use of callee summaries for methods not in M^* and Section 4 proposed the use of caller summaries for all methods thereby requiring flow-sensitive analysis of only methods containing shadows (S). We measured the percentage of reachable methods that are in M^* and S and present these in Table 4. The maximum percentage of methods in M^* was for the test case `jspx-FSI` where 59.9% of the methods are in M^* . Notice that the methods containing shadows for `jspx-FSI` are only 0.6%

Table 4. Percentage of reachable methods that contain shadows or transitively call methods with shadows (M^*) and methods that contain shadows (S)

	antlr		bloat		chart		fop		hsqldb		jython		luindex		lusearch		pmd		xalan	
	M^*	S	M^*	S	M^*	S	M^*	S	M^*	S	M^*	S	M^*	S	M^*	S	M^*	S	M^*	S
FSE	56.3	0.7					47.6	0.1	1.6	0.1	59.8	0.4	1.6	0.4	1.1	0.2	9.5	0.1	50.0	0.6
FSEH	56.3	1.1							2.8	0.1	59.8	0.4	1.1	0.3	0.8	0.2			49.9	0.3
FSI			56.3	4.2	50.6	0.6	47.7	0.5	54.0	0.1	59.9	0.6	53.1	0.4	52.4	0.6	52.4	1.0	50.0	0.5
HN			56.0	2.6	50.5	0.4	47.6	0.1	54.0	0.1	59.8	0.2	53.1	0.2	52.3	0.3	52.2	0.5	0.1	0.1
HNE	56.3	0.7	0.1	0.1					1.6	0.1	59.8	0.2	0.5	0.2	0.3	0.1	6.6	0.1	49.9	0.2
R	7.5	0.2							4.2	0.3	59.8	0.2	0.9	0.1	2.3	0.3	7.7	0.1	49.9	0.1
W			55.8	0.5			47.6	0.3	5.7	0.6			0.8	0.1	1.7	0.3	0.2	0.1	49.9	0.4

indicating that most methods are in M^* since they call methods containing shadows. On average (geometric mean) M^* contains 11.9% of the reachable methods implying that callee summaries are used for the remaining 88.1%. When using both callee and caller summaries a mere 0.3% of reachable methods (average of set S) require flow-sensitive analysis.

5.2 Efficiency

To measure the effect of summaries on the time required to compute the alias set abstraction we computed the abstractions using the three versions of the transfer functions. In Table 5 we show the running time of the original alias set abstraction (ORIG), the alias set abstraction which uses only callee summaries (CS) and the abstraction using both callee and caller summaries (CCS). The times for CS and CCS include the time for computing the callee summary and CCS also includes the time to compute the caller summary.

For all test cases, the time required to compute the abstraction is reduced when callee summaries are used for methods not in M^* . The greatest reduction is for pmd-writer which takes 99.6% less time to compute (6670 vs 29 seconds). The reason for this is quite obvious; for pmd-writer, M^* contains only 12 methods out of the 9365 reachable methods. On average the use of callee summaries reduces the time to compute the alias set abstraction by 27%. Introducing caller summaries has a more significant impact; an average reduction of 96% is witnessed over the entire test set.

5.3 Client Analysis Precision

As discussed earlier we chose a static analysis that verifies conformance to temporal properties as a client of the alias set analysis. The analysis represents temporal properties as state machines where operations on object(s) transition the state machine associated with the object(s). To distinguish the objects on which operations are performed, the analysis uses an alias set abstraction. The result of the analysis is a list of shadows that cannot be verified by the analysis; these include actual violations and false positives.

To evaluate the effect of using caller summaries on the precision of the analysis (callee summaries have no effect on precision) we executed the client analysis using

Table 5. Time taken to compute the alias set abstraction for the original transfer functions (ORIG), the transfer functions leveraging Callee Summaries (CS) and the transfer functions employing both Callee and Caller Summaries (CCS)

		antlr	bloat	chart	fop	hsqldb	jython	luindex	lusearch	pmd	xalan
FSE	ORIG	484			5934	1351	2390	1017	580	2638	5052
	CS	349			5422	220	1524	118	151	37	4484
	CCS	15			108	40	71	19	18	26	112
FSEH	ORIG	500				1426	2020	1054	576		6395
	CS	386				226	1397	120	133		5834
	CCS	18				39	77	17	17		51
FSI	ORIG		1810	1653	4057	1316	2335	1057	553	3685	5100
	CS		1683	1022	4051	651	1671	391	407	2069	5099
	CCS		563	72	109	37	69	17	17	29	119
HN	ORIG		1601	1665	3735	1406	2225	1019	485	5273	5262
	CS		1556	1035	3289	714	1390	393	388	5031	164
	CCS		455	44	115	41	70	18	17	29	45
HNE	ORIG	457	1607			1450	2233	1098	562	5182	4361
	CS	358	119			220	1481	137	121	32	3588
	CCS	16	18			40	77	18	17	26	44
R	ORIG	511				1416	2205	1097	563	4348	3280
	CS	55				238	1487	123	123	35	3172
	CCS	13				39	73	16	16	27	48
W	ORIG		1551		3840	1450		1067	607	6670	3468
	CS		1411		3553	318		120	146	29	3323
	CCS		19		447	37		17	18	28	66

the ORIG and CCS abstractions. As per our discussion in Section 4, we expected a decrease in precision since caller summaries cause the alias set abstraction to compute fewer aliasing facts. However, the results surprised us; none of the 54 test cases showed any degradation in the client analysis. The CCS abstraction contained sufficient must and must-not aliasing at each shadow of a test case to produce the same transitions on the abstract state machine.

Our conclusion from this experiment is that even though caller summaries cause a theoretical decrease in precision, this does not automatically translate into precision loss for the client analysis. Situations exist where the benefits of using caller summaries heavily outweigh the slight chance of losing precision.

5.4 Fine-Grained Precision Metrics

When the client analysis did not show a loss of precision, we set out to develop a fine-grained metric for evaluating precision of alias sets. Using the alias set abstraction we compute must and must-not alias pairs for variables live at the shadows of each test case. Then we sum the alias pairs for all shadows in a test case to give us two precision metrics: MA the aggregated must-alias pairs and MNA the aggregated must-not alias pairs. As expected, the metric values for ORIG and CS are identical indicating that no precision is lost by using callee summaries. Table 6 presents the results of ORIG

(alternately CS) vs CCS². For each test case the MA and MNA values for ORIG are presented. Below this is a number indicating the number of alias pairs that are lost with CCS. For example, the MA value for jython-FSE is 51 indicating that 51 different alias-pairs were identified at the shadows of this test case. The absence of a number below indicates no decrease in precision when using caller summaries. The MNA value for jython-FSE is 189. The -7 below indicates that 7 must-not alias pairs were lost when caller summaries were used.

Table 6. Alias set abstraction precision in terms of aggregated must aliasing (MA) and must not aliasing (MNA) metrics computed at the shadows for each test case

	bloat		chart		jython		luindex		lusearch		pmd		xalan	
	MA	MNA	MA	MNA	MA	MNA	MA	MNA	MA	MNA	MA	MNA	MA	MNA
FSE					51	189	6	98	18	91	0	35	371	1180
FSEH					930	1546	4	63	1	17			179	384
FSI	1152	18858	3344	6244	404	583	76	226	77	233	459	1505	350	1042
		-611	-212	-254		-32		-1		-1	-3	-79		-22
HN	606	7584	704	1529	322	386	95	202	58	112	127	839	0	13
		-328	-14	-214						-1		-53		
HNE	0	21			11	133	8	91	0	13	0	41	45	194
		0				-10		-2						-1
R					253	427	59	80	203	222	56	130	671	1395
					-9	-14	-44		-44			-7		-20
W	7	306					56	83	200	219	0	22	524	799
		-55					-41		-41					-21

Of the 54 test cases, only 9 showed a degradation in the MA precision metric. The four highest degradations were for luindex-R (75%), luindex-W (73%), lusearch-R (22%) and lusearch-W (21%). The average (geometric mean) degradation for the 9 test cases was 8%. 31 of the 54 test cases also noted a decrease in the MNA metric. The maximum decrease was 17% for bloat-W with an average decrease of 4%.

6 Related Work

Kildall’s framework [14] for intraprocedural dataflow analysis was extended by Sharir and Pnueli [27] to perform context-sensitive interprocedural dataflow analysis using either the *call-strings* or *functional* approach. The functional approach computes the effect of each procedure by composing functions for individual instructions in the procedure thereby obtaining a summary function $f_p : \mathcal{D} \rightarrow \mathcal{D}$, where \mathcal{D} is the dataflow analysis domain. Once the summary function for a procedure has been computed it is used at each call site of the procedure to model the effect of the call. Sagiv, Reps and Horwitz [22] extended the original formalism to $\mathcal{P}(\mathcal{D})$ for a finite set \mathcal{D} with the

² Due to space limitations we do not present results for antlr, fop and hsqldb.

condition that the functions on individual dataflow facts should be distributive. Distributivity of transfer functions enables the graphical representation of these functions as bipartite graphs with $O(D)$ nodes. The IFDS algorithm has been used to solve both locally separable problems such as reaching definitions, available expressions and live variables, and non-locally-separable problems such as uninitialized variables and copy-constant propagation. Its efficiency makes it suitable for computing a variety of useful static abstractions [10, 23, 28, 31, 13, 25, 19].

Other frameworks for computing procedure summaries have also been proposed. Gulwani and Tiwari [11] developed procedure summaries in the form of constraints that must be satisfied for some generic assertion to hold at the end of the procedure. Their key insight was to use weakest preconditions of such generic assertions. Furthermore, for efficiency they use strengthening and simplification of these preconditions to ensure early termination. The approach has been used to compute two useful abstractions; unary uninterpreted functions and linear arithmetic. Recently, Yorsh et al. [32] introduced an algorithm which also computes weakest preconditions and relies on simplification for termination. They describe a class of complex abstract domains (including the class of problems solvable using IFDS) for which they can generate concise and precise procedure summaries. Their approach uses symbolic composition of the transfer functions for the instructions in the program to obtain a compact representation for the possibly infinite calling contexts.

In contrast to the related work discussed above, we propose a technique to reduce the number of methods that must be analyzed using any of the approaches discussed above (our implementation uses the IFDS algorithm [22] to compute the alias set abstraction). Under certain conditions, instead of computing expensive procedure summaries through IFDS, our analysis uses cheaper callee summaries without a loss of precision.

Cherem and Rugina [7] present a flow-insensitive, unification-based context-sensitive analysis to construct method summaries that describe heap effects. The analysis is parameterized for specifying the depth of the heap to analyze (k) and the number of fields to track per object (b). Varying the values for k and b results in different method summaries; smaller values produce lightweight summaries whereas larger values result in increased precision. Method summaries were shown to significantly improve a client analysis that infers uniqueness of variables i.e. when a variable holds the only reference to an object.

Also related are analyses which traverse the program callgraph (mostly bottom-up but some top-down analyses have also been proposed) and compute a summary function for each procedure [30, 4, 5]. This summary function is then used when analyzing the callers.

Escape analysis has been widely studied [8, 3, 1, 30] and used in a variety of applications ranging from allocating objects on the stack to eliminating unnecessary synchronization in Java programs. To determine whether an object can be allocated on the stack and whether it is accessed by a single thread, Choi et al. [8] compute object escape information using *connected graphs*. A connected graph summarizes a method and helps identify non-escaping objects in different calling contexts. In their work on inferring aliasing and encapsulation properties for Java [17], Ma and Foster present a static analysis for demand-driven predicate inference. Their analysis computes predicates such as checking for uniqueness of pointers (only reference to an object), parameters that are lent (callee does not change uniqueness) and those that do not escape a callee.

7 Summary

This paper presented callee and caller summaries as a means to improve the efficiency of an alias set analysis. We described the information required from a callee summary to ensure that their use does not decrease precision at a callsite. Algorithms to compute the callee summary and the alias set transfer function leveraging the summaries were also presented. Through experimental evidence we showed that a client analysis and alias set precision metrics are unaffected by the use of callee summaries. On average a 27% reduction in the running time to compute the abstraction was witnessed.

In situations where some loss of precision is acceptable in favour of larger gains in efficiency, we showed how caller summaries that make assumptions about pointer and aliasing relationships at method entry can be employed. In order to gauge the maximum decrease in precision, we chose to use a conservative caller summary which assumes that any two parameters of a method might be aliased. Empirical evaluation of the effect of using caller summaries on the precision of the client analysis revealed no decrease in the abilities of the client analysis. For a fine-grained evaluation of precision, two metrics deriving aggregated must and must-not aliasing between variables were calculated. The average decrease was 8% for the must- and 4% for the must-not alias metric. The running time for computing the alias set abstraction decreases by 96% on average if both callee and caller summaries are used.

Future directions include experimenting with other client analyses of the alias set abstraction, using callee and caller summaries for the standard library, and developing less conservative caller summaries such as those briefly mentioned in Section 4.

Acknowledgements. This work was supported, in part, by the Natural Sciences and Engineering Research Council of Canada and Ontario Ministry of Research and Innovation.

References

1. Aldrich, J., Chambers, C., Sireer, E.G., Eggers, S.J.: Static analyses for eliminating unnecessary synchronization from Java programs. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 19–38. Springer, Heidelberg (1999)
2. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: OOPSLA 2005, pp. 345–364 (2005)
3. Blanchet, B.: Escape analysis for object-oriented languages: application to Java. In: OOPSLA 1999, pp. 20–34 (1999)
4. Chatterjee, R., Ryder, B.G., Landi, W.A.: Relevant context inference. In: POPL 1999, pp. 133–146 (1999)
5. Cheng, B.-C., Hwu, W.-M.W.: Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In: PLDI 2000, pp. 57–69 (2000)
6. Cherem, S., Rugina, R.: Compile-time deallocation of individual objects. In: ISMM 2006, pp. 138–149 (2006)
7. Cherem, S., Rugina, R.: A practical escape and effect analysis for building lightweight method summaries. In: Adsul, B., Vetta, A. (eds.) CC 2007. LNCS, vol. 4420, pp. 172–186. Springer, Heidelberg (2007)
8. Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V.C., Midkiff, S.: Escape analysis for Java. In: OOPSLA 1999, pp. 1–19 (1999)
9. Dufour, B.: Objective quantification of program behaviour using dynamic metrics. Master’s thesis, McGill University (June 2004)

10. Fink, S.J., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective tpestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.* 17(2), 1–34 (2008)
11. Gulwani, S., Tiwari, A.: Computing procedure summaries for interprocedural analysis. In: De Nicola, R. (ed.) *ESOP 2007*. LNCS, vol. 4421, pp. 253–267. Springer, Heidelberg (2007)
12. Hind, M.: Pointer analysis: haven't we solved this problem yet? In: *PASTE 2001*, pp. 54–61 (2001)
13. Horwitz, S., Reps, T., Sagiv, M.: Demand interprocedural dataflow analysis. In: *SIGSOFT FSE 1995*, pp. 104–115 (1995)
14. Kildall, G.A.: A unified approach to global program optimization. In: *POPL 1973*, pp. 194–206 (1973)
15. Lhoták, O.: Comparing call graphs. In: *PASTE 2007*, pp. 37–42 (2007)
16. Lhoták, O., Hendren, L.: Scaling Java points-to analysis using Spark. In: Hedin, G. (ed.) *CC 2003*. LNCS, vol. 2622, pp. 153–169. Springer, Heidelberg (2003)
17. Ma, K.-K., Foster, J.S.: Inferring aliasing and encapsulation properties for Java. In: *OOPSLA 2007*, pp. 423–440 (2007)
18. Naeem, N.A., Lhoták, O.: Typestate-like analysis of multiple interacting objects. In: *OOPSLA 2008*, pp. 347–366 (2008)
19. Naeem, N.A., Lhoták, O.: Efficient alias set analysis using SSA form. In: *ISMM 2009*, pp. 79–88 (2009)
20. Naeem, N.A., Lhoták, O., Rodriguez, J.: Practical extensions to the IFDS algorithm. In: Gupta, R. (ed.) *CC 2010*. LNCS, vol. 6011, pp. 124–144. Springer, Heidelberg (2010)
21. Orlovich, M., Rugina, R.: Memory leak analysis by contradiction. In: Yi, K. (ed.) *SAS 2006*. LNCS, vol. 4134, pp. 405–424. Springer, Heidelberg (2006)
22. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: *POPL 1995*, pp. 49–61 (1995)
23. Rinetzky, N., Sagiv, M., Yahav, E.: Interprocedural shape analysis for cutpoint-free programs. In: Hankin, C., Siveroni, I. (eds.) *SAS 2005*. LNCS, vol. 3672, pp. 284–302. Springer, Heidelberg (2005)
24. Ryder, B.G.: Dimensions of precision in reference analysis of object-oriented programming languages. In: Hedin, G. (ed.) *CC 2003*. LNCS, vol. 2622, pp. 126–137. Springer, Heidelberg (2003)
25. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167(1-2), 131–170 (1996)
26. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.* 20(1), 1–50 (1998)
27. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Muchnick, S.S., Jones, N.D. (eds.) *Program Flow Analysis: Theory and Applications*, ch. 7, pp. 189–233. Prentice-Hall, Englewood Cliffs (1981)
28. Shoham, S., Yahav, E., Fink, S., Pistoia, M.: Static specification mining using automata-based abstractions. In: *ISSTA 2007*, pp. 174–184 (2007)
29. Vallée-Rai, R., Gagnon, E., Hendren, L.J., Lam, P., Pominville, P., Sundaresan, V.: Optimizing Java bytecode using the soot framework: Is it feasible? In: Watt, D.A. (ed.) *CC 2000*. LNCS, vol. 1781, pp. 18–34. Springer, Heidelberg (2000)
30. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for Java programs. In: *OOPSLA 1999*, pp. 187–206 (1999)
31. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) *CAV 2008*. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)
32. Yorsh, G., Yahav, E., Chandra, S.: Generating precise and concise procedure summaries. In: *POPL 2008*, pp. 221–234 (2008)

JPure: A Modular Purity System for Java

David J. Pearce

School of Engineering and Computer Science,
Victoria University of Wellington, New Zealand
djp@ecs.vuw.ac.nz

Abstract. Purity Analysis is the problem of determining whether or not a method may have side-effects. This has applications in automatic parallelisation, extended static checking, and more. We present a novel purity system for Java that employs purity annotations which can be checked modularly. This is done using a flow-sensitive, intraprocedural analysis. The system exploits two properties, called *freshness* and *locality*, to increase the range of methods that can be considered pure. JPure also includes an inference engine for annotating legacy code. We evaluate our system against several packages from the Java Standard Library. Our results indicate it is possible to uncover significant amounts of purity efficiently.

1 Introduction

Methods which don't update program state may be considered *pure* or *side-effect free*. Knowing which methods are pure in a program has a variety of applications, such as: specification languages [19,13,10], model checking [31], compiler optimisations [9,18,36], atomicity [13], query systems [21,34] and memoisation of function calls [15].

Several existing techniques are known for determining purity in OO languages (e.g. [26,30,22,23]). The majority employ interprocedural pointer analysis as the underlying algorithm. While this yields precise results, there are a number of drawbacks: firstly, it requires the whole program be known in advance [29]; secondly, it takes a significant amount of time to run, which is prohibitive in normal day-to-day development.

A useful alternative is to use annotations. Here, pure methods are first annotated with @Pure; then, a *purity checker* is provided to help enforce the purity protocol in programs. For this approach to be practical, the purity checker must be efficient to fit within normal day-to-day development. A sensible way of ensuring this is to require that the annotations be *modularly checkable*. That is, the purity annotations on one method can be checked in isolation from others. Unfortunately, the majority of previous works on purity analysis, particularly those which depend upon interprocedural pointer analysis, do not generate modularly checkable annotations.

An obvious difficulty with any annotation-based purity system is the vast amount of legacy code that would first need to be annotated. In particular, the Java standard library has not been annotated to identify pure methods. To address this, we present a purity system that is split into two components: a *purity inference* and a *purity checker*. The purity inference operates as a source-to-source translation, taking in existing Java code and adding modularly checkable @Pure annotations (and any auxiliary annotations required). The purity checker can then check these annotations are correct efficiently

at compile-time. The idea behind this is simple: users take their existing applications, infer the `@Pure` annotations once using the (potentially expensive) purity inference; then, they *maintain* them using the (efficient) purity checker.

An important requirement for an annotation-based purity system is that the annotations themselves must be simple to use. This is because, once the annotations are inferred, we expect programmers to use and understand them. Our system uses only three annotations, `@Pure`, `@Local` and `@Fresh`, but remains sufficiently flexible for many real-world examples.

The contributions of this paper are as follows:

1. We present a novel purity system which employs modularly checkable annotations. This exploits *freshness* and *locality* to increase the number of methods which can be considered pure. The system comprises a *purity checker* and a *purity inference*. The latter operates as a source-source translation for annotating legacy code.
2. We formalise the intraprocedural analysis that underpins both the purity checking and purity inference algorithms.
3. We report on experiments using our system on several packages from the Java Standard Library. Our results indicate that at least 40% of methods in these packages are pure.

2 A Simple Purity System

We start by considering a simple purity system which is surprisingly effective in practice and, crucially, employs modularly checkable annotations. We then highlight several problems which stem from code found in the Java Standard Library. Later, in §3 we refine this simple system to resolve these problems.

2.1 Overview

In the simple purity system, pure methods are indicated by a `@Pure` annotation. The following characterises the meaning of purity within the system:

Definition 1 (Pure Method). *A method is considered pure if it does not assign (directly or indirectly) to any field or array cell that existed before it was called.*

This implies that, for a method to be pure, any method it calls must also be pure. Therefore, for any call-site, we must conservatively approximate the set of methods that may be invoked. To check annotations modularly, we can only rely on the static type information available at a given call-site. For example:

```
1 public void f(List<String> x) { x.add("Hello"); }
```

As we do not know what implementations of `List` may be supplied for `x`, we must assume any is possible and, hence, that the invocation may dispatch to any implementation of `List.add()`. Thus, for method `f()` above to be considered pure, every implementation of `List.add()` must itself be pure.

The simple purity system must also follow a covariant typing protocol. This requires that pure methods can only be overridden by methods which are also pure. The following illustrates:

```

1 class Parent {
2   @Pure void f() {}
3 }
4
5 class Child extends Parent {
6   int x;
7   void f() { x=1; }
8 }

```

If we considered `Parent.f()` in isolation, one would conclude it is pure. However, `Child.f()` is clearly not pure, since it assigns field `x`. Thus, following the covariant typing protocol, the purity checker must reject this code.

2.2 Modular Checking

We now give an informal argument as to why the annotations produced by the simple purity system are modularly checkable. Essentially, there are three cases to consider:

- (1) **Direct Field Assignment.** A method annotated `@Pure` cannot assign to fields. This can be easily checked by inspecting its implementation.
- (2) **Indirect Field Assignment.** A method annotated `@Pure` may only call methods which are themselves pure. This is checked by ensuring, for each call-site, the method invoked is annotated `@Pure`. Since this is determined using static type information, it may not be the actual method invoked (due to dynamic dispatch). However, it is safe as the covariant typing protocol ensures all overriding methods must be pure.
- (3) **Method Overriding.** A method annotated `@Pure` can only be overridden by methods which are also annotated `@Pure`. This ensures the covariant typing protocol is followed and is easily checked by comparing the annotations on a method with those it overrides.

The argument here is that: one can check a method annotated `@Pure` is indeed pure simply by inspecting its implementation, and those signatures of methods it calls or overrides. This follows the general approach to type checking, as adopted by e.g. Java's type checker.

2.3 Problem 1 — Iterator

We now consider several problems that arose when using the simple purity system on real code. The first is that of `java.util.Iterator`. The following illustrates:

```

1 public Test {
2   private List<String> items;
3   boolean has(String x) {
4     for(String i : items) {
5       if(x == i) { return true; }
6     }
7   return false;
8 }}

```

At a glance, method `Test.has()` appears pure. But, this is not the case because the **for**-loop uses an iterator. Roughly speaking, the loop is equivalent to the following:

```

1 boolean has(String x) {
2   Iterator iter = items.iterator();
3   while(iter.hasNext()) {
4     String i = iter.next();
5     if(x == i) { return true; }
6   }
7   return false;
8 }

```

Here, `iter.next()` is called to get the next item on the list. However, this method updates its `Iterator` object and, hence, cannot be considered pure. In section §3 we resolve this by extending the system to make explicit the fact that `items.iterator()` only ever returns *fresh* — i.e. newly allocated — objects.

2.4 Problem 2 — Append

The second kind of problem one encounters with the simple system is illustrated by the following, adapted from `java.lang.AbstractStringBuilder`:

```

1 class AbstractStringBuilder {
2   char[] data;
3   int count; // number of items used
4   AbstractStringBuilder append(String s){
5     ...
6     ensureCapacity(count + s.length());
7     s.getChars(0, s.length(), data, count);
8     ...
9     return this;
10  }}

```

Here, `getChars()` works by copying the contents of `s` into `data` at the correct position. Because of this, `getChars()` and, hence, `append()` cannot be considered pure under Definition 1. So, why is this a problem? Well, consider the following:

```

1 String f(String x) { return x + "Hello"; }

```

Again, at a glance, this method appears pure. However, the bytecode generated for this method indirectly calls `AbstractStringBuilder.append()` and, hence, it cannot be considered pure.

Clearly, we want methods such as `f()` above to be considered pure. In section §3 we achieve this by extending the system to determine that: firstly, the `StringBuilder` object created for the string append is fresh; secondly, that the array referred to by `data` is in the *locality* of that `StringBuilder` object. When an object (the child) is in the locality of another (the parent), we have a guarantee that the child is fresh if the parent is fresh. Thus, assignments to the array referenced by `data` are permitted as it is known to be fresh (since the `StringBuilder` object is fresh).

3 Our Improved Purity System

In this section, we detail our purity system which improves upon the simple approach outlined in §2. Our system is implemented in a tool called JPure, and we report on experiments using it in §5.

3.1 Freshness and Locality

Let us recall the first problem encountered with the simple purity system, as discussed in §2.3:

```

1 @Pure boolean has(String x) {
2   Iterator iter = items.iterator();
3   while(iter.hasNext()) {
4     String i = iter.next();
5     if(x == i) { return true; }
6   }
7   return false;
8 }

```

To show that this method is pure under Definition 1 requires two guarantees:

1. That `items.iterator()` always returns a *fresh* (i.e. newly allocated) object.
2. That `iter.next()` can only modify the *locality* (i.e. local state) of the `Iterator` object.

Intuitively, the idea is that, when an object is fresh, so is its locality. Then, by Definition 1, state in its locality can be modified as if it did not exist prior to the method (i.e. `has()`) being called. In §3.2 we will give a more precise definition of locality.

To indicate a method returns a fresh object, or that it only ever modifies an object's locality, we employ two additional annotations: `@Fresh` and `@Local`. Thus, for the `Collection` and `Iterator` interfaces, we would add the following annotations:

```

1 interface Collection {
2   @Fresh Object iterator();
3   ...
4 }
5 interface Iterator {
6   @Pure boolean hasNext();
7   @Local Object next();
8   ...
9 }

```

Here, `@Fresh` implies all implementations of `iterator()` must return a fresh object and, furthermore, must be pure (as, for brevity, we say `@Fresh` implies `@Pure`). Likewise, since `hasNext()` is annotated `@Pure`, its implementations must be pure. Finally, the `@Local` annotation on `next()` implies its implementations may only modify the `Iterator`'s locality.

3.2 Understanding Locality

To make our notions of freshness and locality more precise, we must consider which parts of an object they apply to. For example, an `Iterator` instance returned from `iterator()` may be freshly allocated; but, it is almost certainly not the case that all objects reachable from it are (e.g. the items being iterated over). Thus, we need some way to determine how much of an object's reachable state is included in its locality.

Definition 2 (Locality). *The locality of an object includes every field defined by its class and, for those annotated `@Local`, the locality of the referenced object.*

Fields of primitive type are always in the locality of their containing object. For fields of reference type, the field itself is always in the locality, but the referenced object may or may not be (depending on whether the field is annotated `@Local` or not). Our definition of locality is, in some ways, similar to the notion of *ownership* (see e.g. [4,8]); however, we are able to exploit some counter-intuitive properties of pure methods to get a simpler, more flexible system.

Figure 1 illustrates the main ideas. For an instance of `TypedList`, its locality includes the fields `length`, `data` and `elementType`. The locality of the object referenced by `data` is also included, whilst that referred to by `elementType` is not. The presence of an `@Local` annotation on `copy()` indicates it is a *local method*:

Definition 3 (Local Method). *A local method may modify the locality of any parameter annotated `@Local` but, in all other respects, must remain pure. The method receiver (i.e. `this`) is treated as a special parameter, with `@Local` placed on the method itself.*

By Definition 3, `copy()` may modify the locality of `this` and, by Definition 1, any state created during its execution.

The following rules clarify in more detail what a local method conforming to Definition 3 is permitted to do:

```

1 class TypedList {
2   private int length;
3   private @Local Object[] data;
4   private Type elementType;
5
6   @Local public TypedList(Type type, int maxSize) {
7     length = 0;
8     data = new Object[maxSize];
9     elementType = type;
10  }
11
12  @Local public void copy(TypedList dst) {
13    length = dst.length;
14    type = dst.type;
15    data = new Object[dst.length];
16    for(int i=0; i!=length; ++i) { data[i] = dst.data[i]; }
17  }

```

Fig. 1. An example illustrating the main aspects of locality

Rule 1. *A local method may assign fresh objects to any field in the locality of a parameter annotated @Local.*

Rule 2. *A local method may assign any reference to a field in the locality of a parameter annotated @Local, provided that field is not itself annotated @Local.*

To better understand these rules, consider them in the context of Figure 11. The assignment to `data` on Line 15 is permitted under Rule (1) above. Similarly, the assignments to `length` and `type` on Lines 13 and 14 are permitted under Rule (2) since they are both in the locality of `this`, but are not annotated @Local. Finally, the assignment to elements of `data` on Line 16 is permitted under Rule (2). This is because elements of an array object are treated as though they were fields. Since these “fields” cannot be annotated with @Local, Rule (2) must apply.

3.3 Locality Invariants

The rules for checking local methods given in the previous section may seem strange, but they are needed to preserve the *locality invariants*:

Locality Invariant 1 (Construction). *When a new object is constructed, its locality is always fresh.*

The purpose of Locality Invariant 1 is to ensure we can safely modify the locality of fresh objects within pure methods.

Locality Invariant 2 (Preservation). *When the locality of a fresh object is modified, its locality must remain fresh.*

The purpose of Locality Invariant 2 is to ensure that the locality of a fresh object remains fresh, even after modifications to it. Without this guarantee, we become limited in how we can subsequently modify this locality. For example:

```

1 class Link {
2   private @Local Link next;
3
4   public @Local void set(Link link) {
5     this.next = link; // violates invariant 2
6   }
7   public @Fresh Link create() {
8     Link c = new Link();
9     c.set(this.next);
10    c.next.set(null); // problem
11    return c;
12  }

```

Here, method `set()` violates Locality Invariant 2 by assigning an arbitrary reference to field `next`. This is a problem because the locality of a fresh `Link` may no longer be fresh after this assignment (i.e. if the `link` assigned is not fresh). This problem manifests itself in `create()` as, after `set()` is called on Line 9, the locality of the object referenced by `c` is no longer entirely fresh (i.e. `c.next` is not fresh). As a result, the call to `set()` on Line 10 causes a side-effect — meaning `create()` should not be considered pure (recall @Fresh implies @Pure).

3.4 The Law of Locality

Locality can be regarded as a simplified form of ownership suited to purity analysis. Unlike ownership we can be more flexible regarding object aliasing. In particular, the seemingly counter-intuitive *law of locality* is useful:

Definition 4 (Law of Locality). *When checking @Local annotations, one can safely assume parameters are not aliased.*

This law seems strange, but it relates to the overall goal of purity analysis. To understand it better, consider the following:

```
1 void f(T a, @Local T b) { b.field = 0; }
```

Here, it is clear that `b` must be annotated `@Local`, since its locality is modified. However, the question is: should `a` be annotated `@Local` as well? Given that `a` and `b` could be aliased on entry, it seems as though we should assume they are. However, under the law of locality, we can assume they are not.

So, *why does the law of locality work?* Well, the key lies in the way `@Local` annotations are used to show methods are pure. Consider the following:

```
1 @Pure void g() { T x = new T(); f(x, x); }
```

This method is considered pure precisely because the object passed in for parameter `b` is fresh. Thus, if `a` and `b` are aliased on entry to `f()` we have one of two things: either, the caller is impure (in which case it doesn't matter); or, the object passed in for `b` is fresh. In the latter, it immediately follows that `a` is fresh — hence, neither the Locality Invariants nor Definition 1 are violated by the assignment in `f()`.

4 Implementation

We have implemented the purity system outlined in §3 as part of a tool called JPure, which supports *purity checking* and *purity inference*. The former can be done efficiently in a modular fashion. The latter performs a source-to-source translation of Java source code, whilst annotating it with `@Pure`, `@Local` and `@Fresh` annotations where appropriate. Both tools employ an intraprocedural analysis to determine the freshness and locality of variables within a method. The purity inference propagates that information interprocedurally using *Static Class Hierarchy Analysis* [11] to ensure annotations remain modularly checkable. In this section, we formalise the dataflow analysis which underpins both modes of operation.

4.1 Intermediate Language

Before presenting the details of our analysis, we first introduce an Intermediate Language (IL) to base this on. The IL is small and compact, and we deliberately omit many features of the Java language. Despite this, it provides a useful vehicle for presenting the key aspects of our analysis.

The syntax of our intermediate language is given in Figure 2. The IL uses unstructured control-flow, and employs only very simple statements. We also assume our variables are class references, and ignore other types altogether (since they are of no concern here). Likewise, we provide only very limited forms of expression in **if** conditions. A simple IL program is given below:

```

1 void meth(Object x) {
2   Object y;
3   y = new MyClass();
4   if(x == null) goto label1;
5   y = x;
6 label1:
7   y.f();
8 }

```

Our intraprocedural analysis will determine that method `f()` may be called on the object referred to by `x`. If this method is impure, the entire method must be impure; or, if this method has a `@Local` receiver, then `x` will be annotated `@Local`; otherwise, if `f()` is `@Pure` then the whole method may be annotated `@Pure` (i.e. provided the `MyClass` constructor is).

4.2 Overview

The intraprocedural analysis employs an *abstract environment*, Γ , which conservatively models the freshness and locality of visible objects. This maps variables to a set of *abstract references* which range over $\{?, \epsilon, \ell_x, \ell_y, \dots\}$. Here, $?$ indicates an unknown object, ϵ indicates a fresh object or primitive value and, finally, ℓ_x, ℓ_y, \dots are *named objects*. One named object is provided for each parameter, and represents the object it referenced on entry. The analysis tracks how these flow through the method, in a manner similar to an intraprocedural pointer analysis.

Figure 3 illustrates the analysis operating on a simple local method, `copy()`. Recall from Definition 3 that, since the method is annotated `@Local`, it is entitled to modify the locality of the receiver (i.e. **this**), but in all other respects must remain pure.

The intraprocedural analysis assumes the following abstract environment holds on entry to `copy()` (i.e. immediately after Line 4):

$$\Gamma \downarrow (4) = \{\mathbf{this} \mapsto \{\ell_{\mathbf{this}}\}, \mathbf{src} \mapsto \{\ell_{\mathbf{src}}\}\}$$

At this stage, `l` and `t` are undefined and, hence, not present in the abstract environment. Thus, we see that **this** references named object $\ell_{\mathbf{this}}$, and `src` references $\ell_{\mathbf{src}}$. The abstract environment immediately following the assignment to `l` on Line 6 is:

$$\Gamma \downarrow (6) = \{\mathbf{this} \mapsto \{\ell_{\mathbf{this}}\}, \mathbf{src} \mapsto \{\ell_{\mathbf{src}}\}, \mathbf{l} \mapsto \{\epsilon\}\}$$

Here, `l` maps to the special value ϵ which represents both primitive values (as in this case) and fresh objects. The abstract environment immediately following the assignment to `t` on Line 8 is:

$$\Gamma \downarrow (8) = \{\mathbf{this} \mapsto \{\ell_{\mathbf{this}}\}, \mathbf{src} \mapsto \{\ell_{\mathbf{src}}\}, \mathbf{l} \mapsto \{\epsilon\}, \mathbf{t} \mapsto \{?\}\}$$

Intermediate Language Syntax:

$$M ::= T \text{ m}(\overline{T \text{ x}}) \{ \overline{\text{Object } v} \ [L:] \overline{S} \}$$

$$S ::= v = w \mid v = c \mid v.[T]f = w \mid v = w.[T]f \mid v = \text{u.m}[T_f](\overline{w}) \mid v = \text{new } [T_f](\overline{w})$$

$$\quad \mid \text{return } v \mid \text{if}(v == w) \text{ goto } L \mid \text{goto } L$$

$$c ::= \text{null}, \dots, -1, 0, 1, \dots$$

$$T ::= C \mid \text{int}$$

$$T_f ::= \overline{T} \rightarrow T$$

Fig. 2. Syntax for a simple intermediate language. Here, C represents a valid class name, whilst c represents a constant. We only consider class reference and `int` types, since these are sufficient to illustrate the main ideas. Finally, field accesses and method calls are annotated with the static type of the field/method in question.

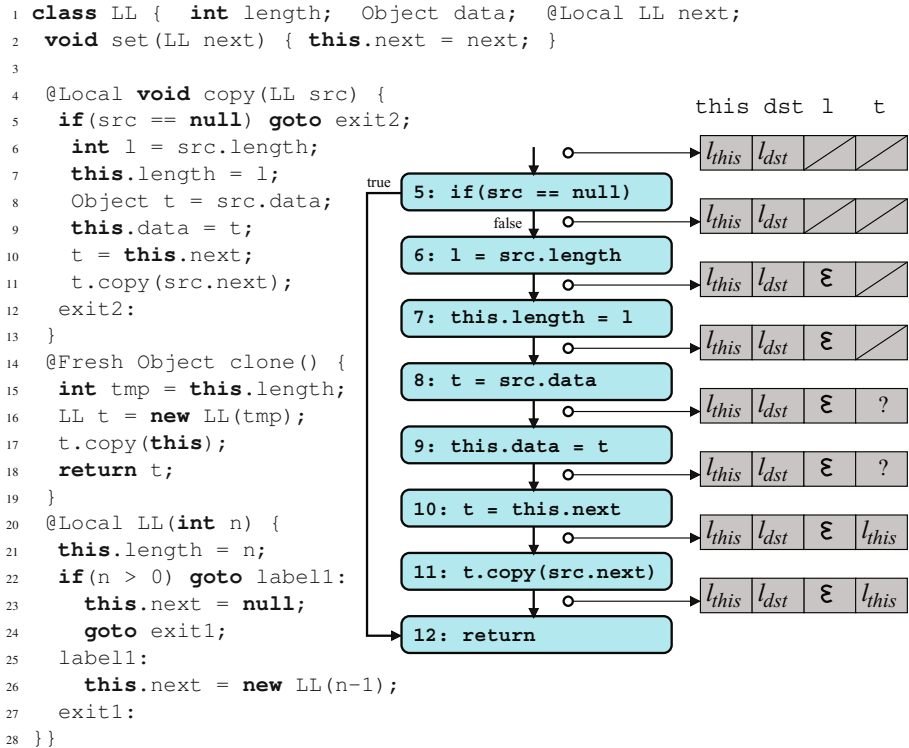


Fig. 3. A simple linked list example which contains (among other things) a local method `copy`, which updates the locality of the receiver `this`. Alongside, the result of our intraprocedural analysis are shown for this method.

In the above, ? represents an *unknown object reference*, and indicates that there is no information available at this point about the object τ refers to. Nevertheless, this unknown reference can be safely assigned to `this.data` on Line 9 under Rule (2) from §3.2

The second assignment to τ on Line 10 is treated differently from the first, because field `next` is annotated `@Local`. The abstract environment immediately following this assignment is:

$$\Gamma \downarrow (10) = \{\text{this} \mapsto \{\ell_{\text{this}}\}, \text{src} \mapsto \{\ell_{\text{src}}\}, \text{l} \mapsto \{\epsilon\}, \tau \mapsto \{\ell_{\text{this}}\}\}$$

This captures the fact that τ still refers to an object in the locality of ℓ_{this} . This is needed to determine that the subsequent invocation, `τ.copy(src.next)`, is safe. That is, since `copy()` is permitted to modify the locality of its receiver and, at this point, τ refers to an object within this locality, the invocation `τ.copy(src.next)` is permitted.

4.3 Abstract Semantics

The effect of a statement on an abstract environment is determined by its *abstract semantics*, which we describe using transition rules. These summarise the abstract environment immediately after the instruction in terms of that immediately before it. The abstract semantics for the intraprocedural analysis are given in Figure 4. Here, $\Gamma[v \mapsto \phi]$ returns an abstract environment identical to Γ , except that v now maps to ϕ . Similarly, $\Gamma[v]$ returns the abstract reference for v in Γ . Several helper functions and constants are used in the semantics:

- `isFresh(m, Tf)` — true iff the given method (determined by its name m and static type T_f) is annotated `@Fresh`.
- `isImpure(m, Tf)` — true iff the given method (determined by its name m and static type T_f) is impure. That is, it is neither annotated with `@Pure`, nor any of its parameters are marked with `@Local`. Note, `$` indicates a constructor.
- `isLocal(f, T)` — true iff the given field (determined by its name f and static type T) is annotated `@Local`.
- `isLocal(i, m, Tf)` — true iff the parameter at position i in the given method (determined by its name m and static type T_f) is annotated `@Local`.
- `isLocalOrFresh(ls)` — true iff the parameters identified in $ls \subseteq \{\ell_1, \dots, \ell_n, \epsilon\}$ are annotated `@Local` in the method being analysed. Note, `? ∈ ls` is not permitted.
- `thismeth` — expands to (m, T_f) where m is the name of the method being analysed, and T_f is its type.

As another example, let us consider how the intraprocedural analysis applies to the method `clone()` from Figure 3. This is annotated `@Fresh` which implies: firstly, it must return an object that did not exist prior to its invocation; secondly, it may not modify any state that existed prior to its invocation (since `@Fresh` implies `@Pure`).

The intraprocedural analysis assumes the following abstract environment holds on entry to `clone()`:

$$\Gamma \downarrow (14) = \{\text{this} \mapsto \{\ell_{\text{this}}\}\}$$

$$\begin{array}{c}
 \frac{c \in \{\text{null}, \dots, -1, 0, 1, \dots\}}{\text{tf}(v = c, \Gamma) \longrightarrow \Gamma[v \mapsto \{\epsilon\}]} \quad \text{[S-C]} \qquad \frac{}{\text{tf}(v = w, \Gamma) \longrightarrow \Gamma[v \mapsto \Gamma[w]]} \quad \text{[S-V]} \\
 \\
 \frac{\begin{array}{c}
 \neg \text{isImpure}(m, T_f) \\
 \text{isFresh}(m, T_f) \Longrightarrow \phi = \{\epsilon\} \quad \neg \text{isFresh}(m, T_f) \Longrightarrow \phi = \{?\} \\
 \text{isLocal}(u, m, T_f) \Longrightarrow \text{isLocalOrFresh}(\Gamma[u]) \\
 \text{isLocal}(w_1, m, T_f) \Longrightarrow \text{isLocalOrFresh}(\Gamma[w_1]) \\
 \dots \\
 \text{isLocal}(w_n, m, T_f) \Longrightarrow \text{isLocalOrFresh}(\Gamma[w_n])
 \end{array}}{\text{tf}(v = u.m[T_f](\bar{w}), \Gamma) \longrightarrow \Gamma[v \mapsto \phi]} \quad \text{[S-M]} \\
 \\
 \frac{}{\text{tf}(v = w.[T]f, \Gamma) \longrightarrow \Gamma[v \mapsto \{\epsilon\}]} \quad \text{[S-F1]} \qquad \frac{T \neq \text{int} \quad \neg \text{isLocal}(f, T)}{\text{tf}(v = w.[T]f, \Gamma) \longrightarrow \Gamma[v \mapsto \{?\}]} \quad \text{[S-F2]} \\
 \\
 \frac{\text{isLocalOrFresh}(\Gamma[w])}{T \neq \text{int} \quad \text{isLocal}(f, T)} \quad \text{[S-F3]} \qquad \frac{\text{isLocalOrFresh}(\Gamma[v])}{\text{isLocal}(f, T) \Longrightarrow \Gamma[w] = \{\epsilon\}} \quad \text{[S-W]} \\
 \text{tf}(v = w.[T]f, \Gamma) \longrightarrow \Gamma[v \mapsto \Gamma[w]] \qquad \text{tf}(v.[T]f = w, \Gamma) \longrightarrow \Gamma \\
 \\
 \frac{\begin{array}{c}
 \neg \text{isImpure}(\$, T_f) \\
 \text{isLocal}(w_1, \$, T_f) \Longrightarrow \text{isLocalOrFresh}(\Gamma[w_1]) \\
 \dots \\
 \text{isLocal}(w_n, \$, T_f) \Longrightarrow \text{isLocalOrFresh}(\Gamma[w_n])
 \end{array}}{\text{tf}(v = \text{new}[T_f](\bar{w}), \Gamma) \longrightarrow \Gamma[v \mapsto \{\epsilon\}]} \quad \text{[S-N]} \\
 \\
 \frac{}{\text{tf}(\text{if}(v == w) \text{goto } L, \Gamma) \longrightarrow \Gamma} \quad \text{[S-I]} \qquad \frac{}{\text{tf}(\text{goto } L, \Gamma) \longrightarrow \Gamma} \quad \text{[S-G]} \\
 \\
 \frac{\text{isFresh}(\text{this}_{\text{meth}}) \Longrightarrow \Gamma[v] = \epsilon}{\text{tf}(\text{return } v, \Gamma) \longrightarrow \Gamma} \quad \text{[S-R]}
 \end{array}$$

Fig. 4. Abstract semantics for checking the correctness of methods annotated @Pure, @Fresh or @Local. The rules assume the method being analysed has at least one of these annotations.

Then, by application of rule S-F1, it computes the following to hold immediately after Line 15:

$$\Gamma \downarrow (15) = \{\text{this} \mapsto \{\ell_{\text{this}}\}, \text{tmp} \mapsto \{\epsilon\}\}$$

At this point, a call to the LL (**int**) constructor is encountered. Constructors are treated like other methods, and may be annotated with @Pure, @Local or not at all (i.e. if they are impure). The LL (**int**) constructor is annotated with @Local, and is treated in the same way as a local method. Hence, it is permitted to modify the locality of **this**

(but must remain pure in all other respects). Therefore, rule S-N applies here, and so the following is determined to hold immediately after Line 16:

$$\Gamma \downarrow (16) = \{\mathbf{this} \mapsto \{\ell_{\mathbf{this}}\}, \mathbf{tmp} \mapsto \{\epsilon\}, \mathbf{t} \mapsto \{\epsilon\}\}$$

Recall that `clone()` must be pure (since `@Fresh` implies `@Pure`), and that the `LL(int)` constructor is `@Local` (hence, it may modify state). Rule S-N safely embodies these requirements as, in a constructor, `this` is (by definition) fresh.

Finally, the analysis applies rule S-M to determine the abstract environment immediately after Line 17. This is identical to $\Gamma \downarrow (16)$ as `copy(LL)` has no return value. The analysis then applies rule S-R to confirm the return value is indeed fresh.

4.4 Dataflow Equations

We formalise the intraprocedural analysis in the usual way by providing *dataflow equations* over the control-flow graph:

$$\begin{aligned} \Gamma \uparrow (n) &= \bigsqcup_{m \rightarrow n} \Gamma \downarrow (m) \\ \Gamma \downarrow (n) &= \mathbf{tf}(\mathbf{S}(n), \Gamma \uparrow (n)) \end{aligned}$$

Here, `m` and `n` represent nodes in the control-flow graph, and `m → n` the directed edge between them. Similarly, `tf` denotes the *transfer function*, whose operation is determined by the semantics of Figure 4, whilst `S(n)` gives the statement at node `n`. The abstract environment immediately before node `n` is given by $\Gamma \uparrow (n)$, whilst that immediately after is given by $\Gamma \downarrow (n)$. Finally, we define the meet of two abstract environments as follows:

$$\Gamma_1 \sqcup \Gamma_2 = \{\mathbf{x} \mapsto \phi_1 \cup \phi_2 \mid \mathbf{x} \in \mathbf{dom}(\Gamma_1) \cup \mathbf{dom}(\Gamma_2) \wedge \phi_1 = \Gamma_1[\mathbf{x}] \wedge \phi_2 = \Gamma_2[\mathbf{x}]\}$$

To be complete, we must detail the initial store used in the dataflow analysis. This is defined as follows:

$$\Gamma \uparrow (0) = \{\mathbf{x} \mapsto \{\ell_{\mathbf{x}}\} \mid \mathbf{x} \in \mathbf{Params}\} \cup \{\mathbf{this} \mapsto \epsilon\}$$

Here, `Params` is the set of all parameters accepted by the method being analysed. Furthermore, we assume that node 0 is the entry point of the control-flow graph. Observe that the analysis assumes parameters are unaliased on entry. Whilst this may seem unsound, it is safe under the law of locality (see §3.4).

4.5 Purity Checking

As discussed previously, our purity system breaks into two components: a *purity checker* and a *purity inference*. The former checks the annotations in a given program are used correctly; the latter infers `@Pure`, `@Local`, and `@Fresh` annotations on legacy code.

In this section, we consider the *purity checker* in more detail. This checks each method in isolation from others using the rules from §3.2 and the intraprocedural analysis discussed earlier. For any method `m`, the purity checker ensures a covariant typing

protocol is followed for `@Fresh` and `@Pure` annotations, and a contra-variant protocol is followed for `@Local` annotations. This is done by examining the annotations on those methods overridden by `m` (which is the same approach used in the Java compiler for checking generic types).

For methods annotated with `@Pure`, `@Fresh` or `@Local`, the intraprocedural analysis is used to check the annotations are properly adhered to. In this case, the rules of Figure 4 are treated in a similar manner to normal typing rules. If the analysis can construct a valid abstract environment before and after each statement, then the method is considered safe (with respect to its purity annotations). Or, if it is unable to do this, an error is reported.

4.6 Purity Inference

The purity inference is also based on the intraprocedural analysis; however, the rules from Figure 4 are treated differently and information may be propagated interprocedurally. For example, some rules (e.g. S-F3) require that fields be annotated `@Local`, whilst others (e.g. S-W) can require that parameters be annotated `@Local`. Other rules (e.g. S-R) are indifferent on whether an annotation actually has to be present or not.

In order to uncover as much purity as possible, the inference adopts a greedy approach. Initially, it assumes all methods are annotated `@Fresh` or `@Pure` (depending on their return type) and all fields are annotated `@Local`. Then, it processes each method in turn and, upon encountering something which invalidates these assumptions, downgrades them as necessary. For example, consider the following method:

```
1 void f(Counter t) { t.count = 1 }
```

The inference initially assumes `f()` is `@Pure`. Upon encountering the assignment on Line 2 this assumption becomes untenable under rule S-W (which requires `t` be fresh or annotated `@Local`). Since `t` is not fresh, it removes the `@Pure` annotation on `f()`, and replaces it with a `@Local` annotation on `t`.

When the annotations on a method are downgraded, this can have a knock-on effect for other methods. In particular, if one method `g()` calls `f()` and, subsequently, it transpires that `f()` is not `@Pure`, then this implies `g()` is no longer `@Pure`. To address this, the purity inference propagates information interprocedurally using static class hierarchy analysis. The following example illustrates:

```
1 class Parent {
2   void f(Test x, Test y) { g(x) }
3   void g(Test z) { z.field = 1; }
4 }
5 class Child extends Parent {
6   int field;
7   void f(Test u, Test v) { }
8 }
```

Let us assume the inference initially visits `Parent.f()`, then `Parent.g()`. The inference will conclude that `Parent.f()` is `@Pure`, since `Parent.g()` is assumed `@Pure`. However, when subsequently examining `Parent.g()` it will realise that

`z` must be annotated `@Local`. At this point, it identifies all potential call sites of `Parent.g()` using *Static Class Hierarchy Analysis* [11]. The method `Parent.f()` contains one such call-site and, hence, is re-examined. As a result, `Parent.f()` is downgraded from being `@Pure` and, instead, `x` is annotated `@Local`. The inference must ensure all `@Local` annotations adhere to a contravariant typing protocol. Therefore, it propagates the new `@Local` annotation up the class hierarchy, resulting in `u` being annotated `@Local` in `Child.f()`.

A similar strategy is employed for propagating information about other annotations, such as `@Fresh` and `@Local` on fields, interprocedurally. The inference will continue doing this until no further changes are necessary (i.e. it has reached a fixed-point). Finally, since the inference only relies on static class hierarchy analysis, the resulting annotations are guaranteed to be modularly checkable.

5 Experimental Results

We have implemented our analysis as part of a tool called JPure. This is open source and freely available from <http://www.ecs.vuw.ac.nz/~djp/jpure>. Our main objective with the tool is to develop a set of modularly checkable purity annotations for the Java Standard Library. We are interested in this because it represents the first, and most difficult, obstacle facing any purity system based on annotations.

Our experimental data is presented in Figure 5. Here, column “#Method” counts the total number of (non-synthetic) methods; “#Pure” counts the total number of pure methods (i.e. those annotated `@Pure`, or those annotated with `@Fresh` but with no `@Local` parameters); “#Local” counts the total number of methods with one or more parameters annotated `@Local`, compared with the total number accepting one or more parameters of reference type; “#Fresh” counts the total number of methods guaranteed to return fresh objects, compared with the total number which return a reference type.

When generating this data, our system assumed all classes being inferred (i.e. all those in the packages shown in Figure 5) were *internal*, and all others were *external*. Then, since annotations were not generated for external classes, their methods were conservatively regarded as impure. Thus, we would expect to see greater amounts of purity if more of the standard library were considered in one go (i.e. because some internal methods call out to external methods).

One issue is the treatment of native methods, which our inference assumed were pure. Whilst this is not ideal, it remains for us to manually identify native methods with side-effects. We would not expect this to affect the data since it mostly relates to I/O, and methods such as `Writer.write()` were inferred as impure anyway.

5.1 Discussion

The results presented in Figure 5 show surprising amounts of purity can be uncovered using our purity inference and (modularly) checked with our purity checker. Recall the inference assumed methods in external packages (i.e. packages other than those listed) were impure. By analysing more of the standard library in one go, we may uncover more purity in those packages listed (since they call methods in external packages).

pkg	#Methods	#Pure	#Local	#Fresh
java.lang	1624	995 (61.2%)	103 / 599 (17.1%)	113 / 520 (21.7%)
java.util.prefs	202	75 (37.1%)	5 / 125 (4.0%)	25 / 80 (31.2%)
java.lang.management	130	105 (80.7%)	0 / 16 (0.0%)	34 / 60 (56.6%)
java.lang.instrument	15	15 (100.0%)	0 / 9 (0.0%)	3 / 5 (60.0%)
java.util.concurrent	525	142 (27.0%)	16 / 242 (6.6%)	15 / 164 (9.1%)
java.util.regex	371	181 (48.7%)	32 / 181 (17.6%)	24 / 70 (34.2%)
java.util	2151	647 (30.0%)	171 / 1043 (16.3%)	108 / 745 (14.4%)
java.util.concurrent.atomic	170	41 (24.1%)	8 / 80 (10.0%)	3 / 21 (14.2%)
java.util.concurrent.locks	98	39 (39.7%)	1 / 35 (2.8%)	8 / 15 (53.3%)
java.io	1017	374 (36.7%)	111 / 491 (22.6%)	22 / 153 (14.3%)
java.util.zip	255	131 (51.3%)	36 / 90 (40.0%)	6 / 23 (26.0%)
java.lang.annotation	17	10 (58.8%)	1 / 8 (12.5%)	2 / 10 (20.0%)
java.util.jar	134	39 (29.1%)	3 / 75 (4.0%)	8 / 44 (18.1%)
java.util.logging	238	49 (20.5%)	8 / 140 (5.7%)	2 / 69 (2.8%)
Total	6947	2843 (40.9%)	495 / 3134 (15.7%)	373 / 1979 (18.8%)

Fig. 5. Experimental data on packages from the Java Standard Library

An interesting question is whether or not we could more purity in these packages by further extending our system. By manually inspecting the inferred annotations, we found a few surprises. In particular, neither `java.lang.Object.equals()` nor `java.lang.Object.hashCode()` were inferred as `@Pure`. This was surprising as: firstly, we did not expect implementations of these methods to have side-effects; secondly, these methods are so widely used that their impurity must be having a large knock-on effect. Through a detailed examination of methods which override `java.lang.Object.equals()`, we identified various reasons why it could not be annotated `@Pure`. For example, `java.util.GregorianCalendar.equals()` has side-effects. A common pattern in such methods is to use one or more fields as a cache. The first time the method is called, objects are created and assigned to these fields, whilst subsequent invocations reuse them. This causes a problem for our system, since the field assignment forces the method to be local or — worse still — impure (e.g. if the fields are `static`).

6 Related Work

Interprocedural side-effect analysis has a long history, with much of the early work focused on compiler optimisation for languages like C and FORTRAN [7,17]. The use of pointer analysis as a building block quickly became established, and remains critical for many modern side-effect and purity systems (e.g. [26,30,22]). In such cases, the precision and efficiency of the side-effect analysis is largely determined by that of the underlying pointer analysis. Numerous pointer analyses have been developed which offer different precision-time trade-offs (see e.g. [27,33,25]). Almost all of these perform *whole-program analysis* and, as such, are inherently unmodular.

There are several good examples of side-effect systems built on top of pointer analysis. Salcianu and Rinard employ a combined pointer and escape analysis, and generate

regular expressions to characterise externally mutated heap locations [30]. Rountev’s system is designed to work with incomplete programs [26]. It assumes a library is being analysed, and identifies methods which are observationally pure to its clients. Side-effects are permitted on objects created within library methods, provided they do not escape. The system uses fragment analysis [28] to approximate the possible information flow and is parameterised on the pointer analysis algorithm. Thus, it could be considered a modularly checkable system, provided the underlying pointer analysis was. A critical difference from our work, is the lack of a concept comparable to locality for succinctly capturing side-effects. Instead, raw points-to information feeds the analysis, meaning that any modularly checkable annotations used would necessarily reflect this — making them cumbersome for a human to maintain. In experiments conducted with this system, around 22% of methods were found to be side-effect free. Nguyen and Xue adopt a similar approach for dealing with dynamic class loading [23]. Benton and Fischer present a lightweight type and effect system for Java, which characterises initialisation effects (i.e. writes to object state during construction) and quiescing fields (i.e. fields which are never written after construction) [2]. Their approach is parameterised on the pointer analysis algorithm. As above, this means that, while it could be considered modularly checkable, it would require cumbersome annotations that were hard to maintain. Finally, they demonstrate that realistic Java programs exhibit a high-degree of mostly functional behaviour.

Systems have also been developed which do not rely on interprocedural analysis. Instead, they typically rely on *Static Class Hierarchy Analysis (SCHA)* [11] to approximate the call-graph, as we do. The advantage of this, as discussed in §11 is that it lends itself more easily to modular checking. Clausen developed a Java bytecode optimiser using SCHA which exploits knowledge of side-effects [9]. In particular, it determines whether a method’s receiver and parameters are *pure*, *read-only*, *write-only* or *read-write*. Here, *pure* is taken to mean: *is not accessed at all*. Cherem and Rugina describes a mechanism for annotating methods with summaries of heap effects in Java programs [6]. In principle, these could be checked modularly, although they did not directly address this. An interprocedural, context-sensitive analysis is also provided for inferring summaries. This differs from our work, in that it is more precise, but generates larger, and significantly harder to understand, annotations.

Aside from compiler optimisations, another important use of purity information lies with specification and assertion languages. The issue here is that, in the specification of a method, one cannot invoke other methods unless they are pure. The Java Modelling Language (JML) provides a good example [20]. Here, only methods marked pure may be used in pre- and post-conditions. In [19] a simple approach to checking the purity of such methods is given — they may not assign fields, perform I/O or call impure methods. However, as discussed in §2 this is insufficient for real-world code, such as found in Java’s standard libraries. Barnett *et al.* also considered this insufficient in practice and, instead, proposed a notion of *observational purity* [1]. Thus, a pure method may have side-effects, provided they remain invisible to callers. They permit field writes in pure methods, provided those fields are annotated as *secret*. JML supports an annotation — *modifies* — which identifies the locations a method may modify. The ES-C/Java tool attempts to statically check JML annotations [14]. Cataño identified that it

does not check `modifies` clauses, and presented an improved system [5]. However, their system ignores the effect of pointer aliasing altogether. `Spec#` is another specification language which requires methods called from specifications be pure [10]. Finally, Nordio *et al.* employ pure methods to model pre- and post-conditions for function objects [24].

There are numerous other works of relevance. In [18], an interprocedural pointer analysis is used to infer side-effect information for use in the Jikes RVM. This enabled upto a 20% improvement in performance for a range of benchmarks. Zhao *et al.* took a simpler approach to inferring purity within Jikes [36]. Whilst few details were given regarding their method, it appears similar to that outlined in §2. However, they achieved a 30% speedup on a range of benchmarks. In [13], pure methods are used to verify atomocity of irreducible procedures. However, no mechanism for checking their purity was given, and instead the authors assume an existing analysis that annotates methods appropriately. Finifter *et al.* adopt a stricter notion of purity, called *functional purity*, within the context of Joe-E — a subset of Java [12]. A method is considered functionally pure if it is both side-effect free, and deterministic. Here, methods are allowed to return different objects for the same inputs, provided that they are equivalent, and their reachable object graphs are isomorphic. The authors report on their experiences identifying (manually) pure methods in several sizeable applications. DPJizer infers method effect summaries and annotates the program accordingly [32]. This is used to help porting of Java programs to DPJ — a language for writing safe parallel programs [16]. DPJ provides a type system that guarantees noninterference of parallel tasks.

Finally, Xu *et al.* consider a dynamic notion of purity, rather than the more common static approach [35]. They examined the number of methods which exhibit pure behaviour on a given program run. They considered different strengths of purity, and found that, while weak definitions exposed significant purity, this information was not always that useful in practice.

7 Conclusion

We have presented a novel purity system that is specifically designed to generate and maintain modularly checkable purity annotations. The system employs only three annotations, `@Pure`, `@Local` and `@Fresh`, but remains sufficiently flexible for many real-world examples. The key innovation lies in the concepts of locality and, particularly, in the locality invariants and the law of locality. We have evaluated our system against several packages from the Java Standard Library, and found that over 40% of methods were inferred as pure.

Acknowledgments. Thanks to Art Protin for useful feedback on an earlier draft.

References

1. Barnett, M., Naumann, D.A., Schulte, W., Sun, Q.: 99.44% pure: Useful abstractions in specification. In: Proc. FTFJP, pp. 11–19 (2004)
2. Benton, W.C., Fischer, C.N.: Mostly-functional behavior in Java programs. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 29–43. Springer, Heidelberg (2009)

3. Bierhoff, K., Aldrich, J.: Lightweight object specification with tpestates. In: ESEC/SIGSOFT FSE, pp. 217–226. ACM Press, New York (2005)
4. Boyapati, C., Liskov, B., Shriram, L.: Ownership types for object encapsulation. In: Proc. POPL, pp. 213–223. ACM Press, New York (2003)
5. Cataño, N., Huisman, M.: CHASE: A static checker for JML's *Assignable* clause. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 26–40. Springer, Heidelberg (2002)
6. Cherem, S., Rugina, R.: A practical escape and effect analysis for building lightweight method summaries. In: Adul, B., Vetta, A. (eds.) CC 2007. LNCS, vol. 4420, pp. 172–186. Springer, Heidelberg (2007)
7. Choi, J.-D., Burke, M., Carini, P.: Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In: Proc. POPL, pp. 232–245. ACM Press, New York (1993)
8. Clarke, D., Potter, J., Noble, J.: Ownership Types for Flexible Alias Protection. In: Proc. OOPSLA, pp. 48–64. ACM Press, New York (1998)
9. Clausen, L.R.: A Java bytecode optimizer using side-effect analysis. *Concurrency - Practice and Experience* 9(11), 1031–1045 (1997)
10. Darvas, Á., Leino, K.R.M.: Practical reasoning about invocations and implementations of pure methods. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 336–351. Springer, Heidelberg (2007)
11. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 77–101. Springer, Heidelberg (1995)
12. Finifter, M., Mettler, A., Sastry, N., Wagner, D.: Verifiable functional purity in Java. In: Proc. CCS, pp. 161–174. ACM Press, New York (2008)
13. Flanagan, C., Freund, S.N., Qadeer, S.: Exploiting purity for atomicity. In: Proc. ISSTA, pp. 221–231. ACM Press, New York (2004)
14. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proc. PLDI, pp. 234–245. ACM Press, New York (2002)
15. Heydon, A., Levin, R., Yu, Y.: Caching function calls using precise dependencies. In: Proc. PLDI, pp. 311–320 (2000)
16. Jr., R.L.B., 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: Proc. OOPSLA, pp. 97–116. ACM Press, New York (2009)
17. Landi, W., Ryder, B.G., Zhang, S.: Interprocedural side effect analysis with pointer aliasing. In: PLDI, pp. 56–67 (1993)
18. Le, A., Lhoték, O., Hendren, L.: Using inter-procedural side-effect information in JIT optimizations. In: Bodik, R. (ed.) CC 2005. LNCS, vol. 3443, pp. 287–304. Springer, Heidelberg (2005)
19. Leavens, G.T.: Advances and issues in JML. In: Presentation at Java Verification Workshop (2002)
20. Leavens, G.T., Leino, K.R.M., Poll, E., Ruby, C., Jacobs, B.: JML: notations and tools supporting detailed design in Java. In: OOPSLA Companion, pp. 105–106 (2000)
21. Lencevicius, R., Holzle, U., Singh, A.K.: Query-based debugging of object-oriented programs. In: Proc. OOPSLA, pp. 304–317. ACM Press, New York (1997)
22. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to and side-effect analyses for Java. *SIGSOFT Softw. Eng. Notes* 27(4), 1–11 (2002)
23. Nguyen, P.H., Xue, J.: Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. In: Proc. ACSC, pp. 9–18 (2005)

24. Nordio, M., Calcagno, C., Meyer, B., Müller, P., Tschannen, J.: Reasoning about function objects. In: Vitek, J. (ed.) TOOLS 2010. LNCS, vol. 6141, pp. 79–96. Springer, Heidelberg (2010)
25. Pearce, D.J., Kelly, P.H.J., Hankin, C.: Efficient field-sensitive pointer analysis for C. *ACM TOPLAS* 30 (2007)
26. Rountev, A.: Precise identification of side-effect-free methods in Java. In: Proc. ICSM, pp. 82–91. IEEE Computer Society, Los Alamitos (2004)
27. Rountev, A., Milanova, A., Ryder, B.G.: Points-to analysis for Java using annotated constraints. In: Proc. OOPSLA, pp. 43–55 (2001)
28. Rountev, A., Milanova, A., Ryder, B.G.: Fragment class analysis for testing of polymorphism in Java software. In: Proc. ICSE, pp. 210–220 (2003)
29. Rountev, A., Ryder, B.G.: Points-to and side-effect analyses for programs built with pre-compiled libraries. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 20–36. Springer, Heidelberg (2001)
30. Salcianu, A., Rinard, M.: Purity and side effect analysis for Java programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 199–215. Springer, Heidelberg (2005)
31. Tkachuk, O., Dwyer, M.B.: Adapting side effects analysis for modular program model checking. *SIGSOFT Softw. Eng. Notes* 28(5), 188–197 (2003)
32. Vakilian, M., Dig, D., Bocchino, R.L., Overbey, J., Adve, V.S., Johnson, R.: Inferring method effect summaries for nested heap regions. In: Proc. ASE, pp. 421–432 (2009)
33. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using Binary Decision Diagrams. In: Proc. PLDI, pp. 131–144. ACM Press, New York (2004)
34. Willis, D., Pearce, D.J., Noble, J.: Caching and incrementalisation in the java query language. In: Proc. OOPSLA, pp. 1–18. ACM Press, New York (2008)
35. Xu, H., Pickett, C.J.F., Verbrugge, C.: Dynamic purity analysis for Java programs. In: Proc. PASTE, pp. 75–82. ACM Press, New York (2007)
36. Zhao, J., Rogers, I., Kirkham, C., Watson, I.: Pure method analysis within jikes rvm. In: Proc. ICPOOLPS (2008)

Tainted Flow Analysis on e-SSA-Form Programs

Andrei Rimsa¹, Marcelo d’Amorim², and Fernando Magno Quintão Pereira¹

¹ UFMG – 6627 Antônio Carlos Av, 31.270-010, Belo Horizonte, Brazil

² UFPE – Av. Prof. Luis Freire, 50.740-540, Recife, Brazil

rimsa@live.com, damorim@cin.ufpe.br, fpereira@dcc.ufmg.br

Abstract. Tainted flow attacks originate from program inputs maliciously crafted to exploit software vulnerabilities. These attacks are common in server-side scripting languages, such as PHP. In 1997, Ørbæk and Palsberg formalized the problem of detecting these exploits as an instance of type-checking, and gave an $O(V^3)$ algorithm to solve it, where V is the number of program variables. A similar algorithm was, ten years later, implemented on the Pixy tool. In this paper we give an $O(V^2)$ solution to the same problem. Our solution uses Bodik *et al.*’s extended Static Single Assignment (e-SSA) program representation. The e-SSA form can be efficiently computed and it enables us to solve the problem via a sparse data-flow analysis. Using the same infrastructure, we compared a state-of-the-art data-flow solution with our technique. Both approaches have detected 36 vulnerabilities in well known PHP programs. Our results show that our approach tends to outperform the data-flow algorithm for bigger inputs. We have reported the bugs that we found, and an implementation of our algorithm is now publicly available.

1 Introduction

Web applications are pervasive in the Internet. They are broadly used and often manipulate sensitive information. It comes to no surprise that web applications are common targets of *cyber attacks* [24]. A cyber attack typically initiates with a remote attacker carefully forging inputs to corrupt a running system. A study performed by CVE¹ with statistics for the year 2006 shows that *cross-site scripting* accounts for 18.5% of the web vulnerabilities, while *PHP includes* and *SQL injection* account, respectively, for 13.1% and 13.6%. All three vulnerabilities are commonly found in web applications. To put the significance of these threats in perspective, the annual SANS’s report² estimates that a particular type of attack – malicious SQL injection – has happened approximately 19 million times in July of 2009. Therefore, the static detection of potential vulnerabilities in web applications is an important problem.

Many web vulnerabilities are described as *Tainted Flow Attacks*. Examples include: SQL injection, cross-site scripting, malicious file inclusion, unwanted command executions, eval injections, and file system attacks [24,29,31]. This

¹ <http://cve.mitre.org/docs/vuln-trends/index.html>

² <http://www.sans.org/top-cyber-security-risks/origin.php>

pattern consists of a remote individual exploring potential leaks in the system via its public interface. In this context, the interface is the web and the vulnerability is the lack of “sanity” checks on user-provided data before using it on sensitive operations. To detect this kind of attack one needs to answer the following question: does the target program contain a path on which data flows from some input to a sensitive place without going through a sanitizer function? A sanitizer is a function that either “cleans” malicious data or warns about the potential threat. We call the previous question the *Tainted Flow Problem*.

The tainted flow problem was formalized by Ørbæk and Palsberg in 1997 as an instance of type-checking [16]. They wrote a type system to the λ -calculus, and proved that if a program type-checks, then it is free of tainted flow vulnerabilities. Ten years later, Jovanovic *et al.* provided an implementation of an algorithm that solves the tainted flow problem for PHP 4.0 on the Pixy tool. This algorithm was a data-flow version of Ørbæk and Palsberg’s type system. It has an average $O(V^2)$ running-time complexity, yet the Pixy’s implementation suffers from worst case $O(V^4)$ complexity. Ørbæk and Palsberg’s solution, when seen as a data-flow problem, admits a worst case $O(V^3)$ solution [16, p.30].

This paper improves on the complexity of these previous results. The algorithm that we propose is, in the worst case, quadratic on the number of variables in the source program, both in terms of time and space. The low asymptotic complexity is justified by the use of a program representation called *Extended Static Single Assignment* (e-SSA) form, introduced by Bodik *et al.* [5], which can be computed in linear time on the program size. This intermediate representation makes it possible to solve the tainted flow problem as a *sparse* analysis, which associates constraints directly to program variables, instead of associating them to variables at every program point. This paper brings forward the following contributions:

- An efficient algorithm to solve the tainted flow problem. A distinguishing feature of this algorithm is the use of the e-SSA representation to generate constraints. See Section 4.4.
- An implementation of the algorithm on top of `phc` [3,4], an open source PHP compiler³. Our implementation of e-SSA is now part of the compiler’s official distribution.
- An evaluation of the proposed approach on public PHP applications, including benchmarks used in previous works [13,14,31], and the consequent exposure of previously unknown vulnerabilities. See Section 5.

Our analysis can be generalized to other procedural languages. We chose PHP for two reasons. First, it is popular for developing server-side web applications. For example, PHP programs can be found in over 21 million Internet domains⁴. Second, PHP has been the focus of previous research on static detection of tainted flow vulnerabilities, and benchmarks are easily available.

³ <http://www.phpcompiler.org/>

⁴ <http://php.net/usage.php>

2 Examples of Tainted Flow Attacks

A tainted flow attack is characterized by a subpath from a source to a sink function that does not include calls to sanitizing functions. A source function reads information from an input channel (e.g., from an HTML form) and passes it to the program. Sinks are functions that perform sensitive operations, such as writing information into the program's output channel (e.g., to a dynamically generated webpage). Sanitizers are functions that protect the program. For instance, proving that untrusted information is safe, removing malicious contents from tainted data, or firing exceptions when necessary. The literature describes many kinds of tainted flow attacks. Some noticeable examples are cross-site scripting (XSS) [9,24], SQL injection [29,31], malicious evaluations⁵, local/remote file inclusions⁶, and unwanted command execution⁷. In this section, we explain two of these vulnerabilities in more detail; however, in the rest of the paper we chose to focus on cross site scripting attacks only. Important to note that our framework is capable of handling other types of attacks. In particular, we have showed elsewhere [21] that it can be used to search for SQL injections.

2.1 Cross-Site Scripting

A cross-site scripting attack can occur when a user is able to dump HTML text into a dynamically-generated page. An attacker uses this vulnerability to inject JavaScript code into the page, usually trying to steal cookie information to acquire session privileges. The program below illustrates this situation. In this case, the user provides the input “<script>does.something.evil;</script>” to the variable `name` from the code fragment below.

```
<?php $name = $_GET['name']; echo $name; ?>
```

Note that a potentially malicious JavaScript could be used instead of `does.something.evil`. A workaround for this threat is to strip HTML-related data from the user input. In this case, from the JavaScript passed as input. The function `htmlentities`, shown below, does the trick by encoding special characters into their respective HTML entities. For example, this function translates the symbol “<” to “<”.

```
<?php $name = htmlentities($_GET['name']); echo $name; ?>
```

Cross-site scripting attacks fit into the tainted flow problem framework. A possible input configuration, in this case, would be:

Sources : `$_GET`, `$_POST`, ...

Sinks : `echo`, `print`, `printf`

Sanitizers : `htmlentities`, `htmlspecialchars`, `strip_tags`

⁵ <http://cwe.mitre.org/data/definitions/95.html>

⁶ <http://projects.webappsec.org/Remote-File-Inclusion>

⁷ <http://secunia.com/advisories/26201/>

2.2 SQL Injection Attacks

The SQL injection attack is another common type of security flaw. In this attack an adversary uses the parameters of SQL queries to manipulate a database. The effect can go from reporting incorrect results to the user to modifying database contents. The program below contains a vulnerability of this kind.

```
<?php
  $userid = $_GET['userid'];
  $passwd = $_GET['passwd'];
  ...
  $result = mysql_query("SELECT userid FROM users WHERE
                        userid=$userid AND passwd='$passwd'");
?>
```

Note that this program does not sanitize its inputs. A malicious user could obtain access to the application by providing the text “1 OR 1 = 1 --” in the `userid` field. The double hyphen starts a comment in MySQL. The following query is obtained with the input variables replaced: `SELECT userid FROM users WHERE userid=1 OR 1 = 1 -- AND passwd='ANY PASSWORD'`. The execution of this query outputs one row and therefore bypass the authentication procedure.

A workaround for this threat is to sanitize the variable `userid` to ensure that it only contains numerical characters; a task that we perform either casting it to integer or checking its value with functions like `is_numeric`. One can sanitize variable `$passwd` using the `addslashes` function, which inserts slashes (escape characters) before a predefined set of characters, including single quotes. A typical configuration of SQL injection is given below:

```
Sources : $_GET, $_POST, ...
Sinks   : mysql_query, pg_query, *_query
Sanitizers : addslashes, mysql_real_escape_string, *_escape_string
```

3 Formal Definition and Previous Solution

Nano-PHP. We use the assembly-like Nano-PHP language to define the tainted flow problem. A label $l \in L$ refers to a program location and is associated to one instruction. A Nano-PHP program is a sequence of labels, $l_1 l_2 \dots l_{exit}$. Figure [1](#) shows the six instructions of the language. We use the symbol \otimes to denote any operation that uses a sequence of variables to define another variable.

Semantics. We define the semantics of Nano-PHP programs with an abstract machine. The state M of this machine is characterized with a tuple (Σ, F, I) , informally defined as follows:

Store Σ : $Var \rightarrow Abs$	e.g., $\{x_1 \mapsto clean, \dots, x_n \mapsto tainted\}$
Code Heap F : $L \rightarrow [Ins]$	e.g., $\{l_1 \mapsto i_1 \dots i_a, \dots, l_n \mapsto i_b\}$
Instruction Sequence I : $[Ins]$	e.g., $i_5 i_6 \dots i_n$

Name	Instruction	Example
Assignment from source	$x = o$	<code>\$a = \$_POST['content']</code>
Assignment to sink	$\bullet = v$	<code>echo(\$v)</code>
Simple assignment	$x = \otimes(x_1, \dots, x_n)$	<code>\$a = \$t1 * \$t2</code>
Branch	$\mathbf{bra} \ l_1, \dots, l_n$	general control flow
Filter	$x_1 = \mathbf{filter}$	<code>\$a = htmlentities(\$t1)</code>
Validator	$\mathbf{validate} \ x, l_c, l_t$	<code>if (!is_numeric(\$1)) abort();</code>

Fig. 1. The Nano-PHP syntax

$$\begin{array}{l}
\text{[S-SOURCE]} \quad (\Sigma, F, x = o; S) \rightarrow (\Sigma \setminus [x \mapsto \text{tainted}], F, S) \\
\text{[S-SINK]} \quad \frac{\Sigma \vdash v = \text{clean}}{(\Sigma, F, \bullet = v; S) \rightarrow (\Sigma, F, S)} \\
\text{[S-SIMPLE]} \quad \frac{\Sigma \vdash \sqcup(x_1, \dots, x_n) = v}{(\Sigma, F, x = \otimes(x_1, \dots, x_n); S) \rightarrow (\Sigma \setminus [x \mapsto v], F, S)} \\
\text{[S-BRANCH]} \quad \frac{\{l_i\} \subseteq \text{dom}(F) \quad F(l_i) = S' \quad 1 \leq i \leq n}{(\Sigma, F, \mathbf{bra} \ l_1, \dots, l_n; S) \rightarrow (\Sigma, F, S')} \\
\text{[S-FILTER]} \quad (\Sigma, F, x = \mathbf{filter}; S) \rightarrow (\Sigma \setminus [x \mapsto \text{clean}], F, S) \\
\text{[S-VALIDC]} \quad \frac{\Sigma \vdash x = \text{clean} \quad \{l_c\} \subseteq \text{dom}(F) \quad F(l_c) = S'}{(\Sigma, F, \mathbf{validate}(x, l_c, l_t); S) \rightarrow (\Sigma, F, S')} \\
\text{[S-VALIDT]} \quad \frac{\Sigma \vdash x = \text{tainted} \quad \{l_t\} \subseteq \text{dom}(F) \quad F(l_t) = S'}{(\Sigma, F, \mathbf{validate}(x, l_c, l_t); S) \rightarrow (\Sigma, F, S')}
\end{array}$$

Fig. 2. Operational semantics of Nano-PHP

The symbol Var denotes the domain of program variables. The symbol Abs denotes the domain of abstract states $\{\perp, \text{clean}, \text{tainted}\}$. The store Σ binds each variable name, say $x \in Var$, to an abstract value $v \in Abs$. The code heap F is a map from a program label to a sequence of instructions. Each sequence corresponds to one *basic block* from the Nano-PHP program. Only labels associated to entry basic block instructions appear in F . The list I denotes the next instructions for execution. We say that the abstract machine can *take a step* if from a state M it can make a transition to state M' . More formally, we write $M \rightarrow M'$. We say that the machine is *stuck* at M if it cannot make any transition from M .

Figure 2 illustrates the transition rules describing the semantics of Nano-PHP programs. Rule S-SOURCE states that an assignment from source binds the left-hand side variable to the tainted abstract state. Rule S-SINK is the only one that can cause the machine to get stuck: the variable on the right hand side *must* be

bound to clean in order to execute a safe assignment to sink. Rule S-SIMPLE says that, given an assignment $x = \otimes(x_1, x_2, \dots, x_n)$, the abstract state of x is defined by folding the join operation (as described on Table II) onto the list of variables in the right hand side, e.g.: $x_1 \sqcup x_2 \dots \sqcup x_n$. Rule S-BRANCH defines a non-deterministic branch choice: the machine chooses one target in a range of possible labels and branches execution to the instruction at this label.

Nano-PHP organizes the sanitizer function in two groups: filters and validators. Filters correspond to functions that take a value, typically of string type, and return another value without malicious fragments from the input. For simplicity we do not show the input parameter in the syntax of Nano-PHP. Rule S-FILTER shows that an assignment from a filter binds the variable on the left side to the clean state. We can use this syntax to define assignments from constants (e.g., $v = 1$). Validators are instructions that combine branching with a boolean function that checks the state for tainting. The instruction `validate(x, l_c, l_t)` has two possible outcomes. If x is bound to the clean state, the machine branches execution to $F(l_c)$. If x is bound to the tainted state, execution branches to $F(l_t)$. Again, we omit the boolean function itself from the syntax for simplicity. Rules S-VALIDC and S-VALIDT define these cases. We assume that in any Nano-PHP program every variable must be defined before being used; therefore, we rule out the possibility of passing x to a validator when $\Sigma \vdash x = \perp$.

Important consideration. Before we move on to describe the traditional data flow solution to the tainted flow problem, a note about functions is in order. In this paper we describe an intraprocedural analysis. Thus, we conservatively consider that input parameters and the return values of called functions are all definitions from source. A context insensitive, interprocedural version of the algorithms in this paper can be produced by creating assignments from actual to formal parameters. We opted for not doing it due to an engineering shortcoming: our limited knowledge of `phc` has hindered us thus far from crossing the boundaries of functions.

The problem. We define the tainted flow problem as follows.

Definition 1. THE TAINTED FLOW PROBLEM

Instance: a Nano-PHP program P .

Problem: determine if the machine can get stuck executing P .

Data Flow Analysis. Given a Nano-PHP program, we can solve the tainted flow problem using a *forward-must* data flow analysis. Our analysis binds information

Table 1. Definition of least upper bound over pairs of abstract values

\sqcup	\perp	clean	tainted
\perp	\perp	clean	tainted
clean	clean	clean	tainted
tainted	tainted	tainted	tainted

Table 2. Data-Flow equations to solve the Tainted Flow Problem

l	$\llbracket - \rrbracket$
$x = \circ$	$\llbracket l, l_+ \rrbracket = JOIN(l) \setminus [x \mapsto \text{tainted}]$
$\bullet = x$	$\llbracket l, l_+ \rrbracket = JOIN(l)$
$x = \otimes(x_1, \dots, x_n)$	$\llbracket l, l_+ \rrbracket = JOIN(l) \setminus [x \mapsto JOIN(l)(x_1) \sqcup \dots \sqcup JOIN(l)(x_n)]$
bra l_1, \dots, l_n	$\llbracket l, l_i \rrbracket = JOIN(l), 1 \leq i \leq n$
$x = \text{filter}$	$\llbracket l, l_+ \rrbracket = JOIN(l) \setminus [x \mapsto \text{clean}]$
validate x, l_c, l_t	$\llbracket l, l_c \rrbracket = JOIN(l) \setminus [x \mapsto \text{clean}]$ $\llbracket l, l_t \rrbracket = JOIN(l)$

to *program points*, which are the regions between pairs of consecutive Nano-PHP labels. We define a lattice $(Abs, <)$ by augmenting the set Abs with the following ordering $\perp < \text{clean} < \text{tainted}$. Table 1 shows the least upper bound for subsets of Abs including pairs of elements from Abs . The map lattice $(Var \rightarrow Abs, <')$ is obtained with the typical lifting of the lattice associated to Abs . Recall that the set Var is finite. We represent data-flow information with the function $\llbracket - \rrbracket : L \rightarrow L \rightarrow Var \rightarrow Abs$. This function associates to each program point (l, l') a map storing the abstract values of each program variable. We use the notation $\llbracket l_1, l_2 \rrbracket$ to denote information at (l_1, l_2) . It abbreviates the function application $(\llbracket - \rrbracket)_{l_1} l_2$. Note that $\llbracket - \rrbracket$ is also a lattice.

Table 2 defines the transfer functions $(Var \rightarrow Abs) \rightarrow (Var \rightarrow Abs)$ associated to each instruction. The initial state of the analysis associates undefined to all program variables at every point, i.e., $\llbracket - \rrbracket = \lambda l_1 . \lambda l_2 . \lambda v . \perp$. We let $PRED(l)$ be the set of program points immediately before label l , and define the auxiliary function $JOIN$ as follows:

$$JOIN(l) = \bigsqcup \llbracket l_i, l \rrbracket, \quad l_i \in PRED(l)$$

Given two functions $\llbracket k', k \rrbracket$ and $\llbracket l', l \rrbracket$, we define $\bigsqcup \{\llbracket k', k \rrbracket, \llbracket l', l \rrbracket\}$ as $\lambda v . (\llbracket k', k \rrbracket v) \sqcup (\llbracket l', l \rrbracket v)$, with \sqcup given by Table 1. The combined transfer function $tr : \llbracket - \rrbracket \rightarrow \llbracket - \rrbracket$ is defined as usual with the composition of all individual transfer functions. Function tr admits fix-points as the lattice is finite and all individual transfer functions are monotone.

The join operation denotes accumulation of information across control flow edges. In this case, information flows from the predecessor edges of a node. Note that we define operation $JOIN$ over a map lattice. Informally, the semantics of this operation is to apply \sqcup over elements on the image of the functions according to the definition on Table 1. For example $\{x \mapsto \text{clean}, y \mapsto \text{clean}\} \sqcup \{x \mapsto \text{tainted}, y \mapsto \perp\} = \{x \mapsto \text{clean} \sqcup \text{tainted}, y \mapsto \text{clean} \sqcup \perp\}$.

Illustrative Example. Figure 3 illustrates the result of a data-flow analysis. We let DB to denote a global database, and we assume that $DB.get$ might produce tainted data. The function $DB.isMember$ works as a validator. We have replaced

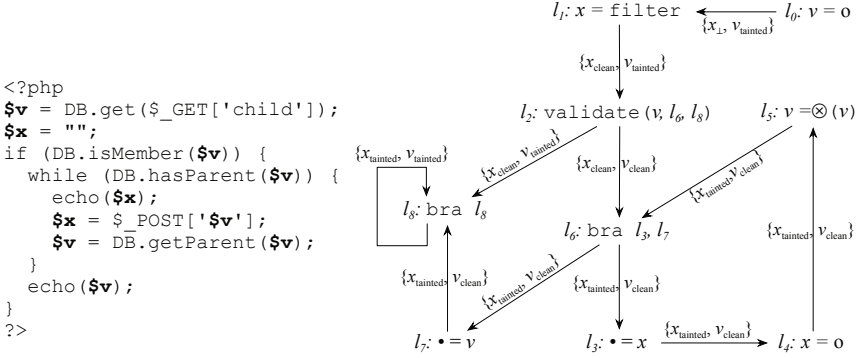


Fig. 3. A simple PHP program (left), and its equivalent Nano-PHP version (right), augmented with the result of data-flow analysis

a call to `DB.hasParent` by the simple branch at l_6 , as this operation does not create new data. Similarly, we have replaced the call to `DB.getParent` by $v = \otimes(v)$. We use l_8 , a label jumping to itself, to mark the end of the program. We show the maps produced by the data-flow analysis on the edges of the Nano-PHP program. In this program the data-flow analysis obtains a fix-point in two iterations. The example contains a tainted flow vulnerability, given by the path $l_4 \rightarrow l_5 \rightarrow l_6 \rightarrow l_3$. At l_4 we read variable x , e.g., `$x = $_POST['$v']`, and at l_3 we feed it to a sink function, e.g., `echo($x)`. Note that variable v cannot be used in a tainted flow attack, because it is sanitized by the function `DB.isMember`.

Complexity. We can solve this data-flow analysis using the chaotic iteration model. If the CFG of the input program has I instructions and V variables then we can perform $O(I \times V)$ iterations. Each union is $O(V)$, and we may have $O(I)$ unions per iteration. Thus, our data-flow analysis has complexity $O(V^2 \times I^2)$. However, it is possible to speedup the algorithm executing the transfer functions in a topological order of the program’s dominator tree [2]. In particular, Palsberg [17] gives an $O(V^3)$ type-inference algorithm that solves the tainted flow problem. In practice, this data-flow analysis is $O(V \times I)$ [2, p.209].

4 The Proposed Solution

In this section we describe our solution to the tainted flow problem. Our approach is divided into the three parts below. We give time complexity in terms of the number of variables (V) in the source program.

1. Convert the input program to the *Extended Static Single Assignment* (e-SSA) form. The construction of the dominator tree is $O(V\alpha(V))$, where α is the inverse Ackerman function, normally regarded as constant, and the insertion of ϕ -functions is $O(V^2)$, yet linear in practice [2, p.408].

2. Traverse the e-SSA-form program collecting use-chains: $O(V)$.
3. Use the algorithm in Figure 8 to find tainted flow vulnerabilities: $O(V^2)$, but $O(V)$ in practice.

4.1 E-SSA Form Is the Linchpin of Fast Tainted Flow Analysis

We use the *Extended Static Single Assignment* (e-SSA) representation to simplify our tainted flow analysis. The e-SSA program representation is a superset of the well known *Static Single Assignment* (SSA) form [10]. This representation has been used by Bodik *et al.* [5] to eliminate array bound checks. Its main advantage, in our case, is the possibility of acquiring useful information from the outcome of conditional tests, and then binding this information directly to variables, instead of pairs of variables and program points. We convert a Nano-PHP program to e-SSA form using the algorithm below:

1. For each instruction $i = \text{validate } x, l_c, l_t$:
 - (a) replace i by a new instruction $\text{validate } x, x_c, l_c, x_t, l_t$, where x_c and x_t are fresh variables;
 - (b) rename every use of x dominated by l_c to x_c . A label l dominates a use of variable x at label l_u if, and only if, every path from the program’s entry point to l_u goes across l .
 - (c) rename every use of x dominated by l_t to x_t ;
2. Convert the resulting program into SSA form. For a fast algorithm, see Appel and Palsberg [2, p.410].

In order to represent Nano-PHP program in e-SSA form, we modify the syntax of this language in two ways. First, we add ϕ -functions to the language. These special instructions are an abstraction first introduced by Cytron *et al.* [10] to represent SSA-form programs. ϕ -functions are used at control-flow join points, and they receive as parameter one variable name associated to each control-flow predecessor. A ϕ -function such as $x_n = (x_1, \dots, x_m)$, placed at label l has the effect of assigning $x_i, 1 \leq i \leq m$ to x_n , depending on which predecessor of l was last visited before execution reaches l . The use of a variable in SSA-form programs is associated to only one definition. Thus, to convert a program into the SSA form, we rename each definition of a variable v to a different name, and join definitions of v that reach a common program point by ϕ -functions. These new ϕ -functions produce fresh definitions of v ; thus, the process continues until the program stabilizes. There exist almost linear time algorithms to convert programs to SSA-form [15]. E-SSA-form programs are also SSA-form programs; thus, they have the property that each variable has only one definition.

Second, we modify the syntax of the validator instruction, which become $\text{validator } (x, x_c, l_c, x_t, l_t)$ [8]. Conceptually, the validator splits the live range of variable x in two parts, depending on whether or not its abstract value is tainted. Note that when converting a program into e-SSA form, we rename every use of

⁸ Bodik *et al.* use special instructions called π -functions to create x_c and x_t [5].

x in labels dominated by l_c to x_c , and rename every use of x in labels dominated by l_t to x_t . The new instruction has the following semantics:

$$[\text{S-ESSAC}] \frac{\Sigma \vdash x = \text{clean} \quad \{l_c\} \subseteq \text{dom}(F) \quad F(l_c) = S'}{(\Sigma, F, \text{validate}(x, x_c, l_c, x_t, l_t); S) \rightarrow (\Sigma \setminus [x_c \mapsto \text{clean}], F, S')}$$

$$[\text{S-ESSAT}] \frac{\Sigma \vdash x = \text{tainted} \quad \{l_t\} \subseteq \text{dom}(F) \quad F(l_t) = S'}{(\Sigma, F, \text{validate}(x, x_c, l_c, x_t, l_t); S) \rightarrow (\Sigma \setminus [x_t \mapsto \text{tainted}], F, S')}$$

Rule S-ESSAC says that a validator, upon receiving a clean variable x , guarantees that the variable will be clean henceforth. Given that every use of x dominated by l_c has been renamed to x_c beforehand, we simply continue the program execution in an environment where x_c is bound to clean. Rule S-ESSAT does the opposite: if a validator fails on a variable x , we know that x is tainted; hence, we continue the program execution in an environment where x_t is bound to tainted.

The e-SSA representation allows us to acquire static information from the outcome of conditionals. Hence, we can associate unique constraints to variables, as Figure 4 illustrates. The original program in Figure 3 contains two variables, x and v . We know that these variables are clean in some program points, but not in all. The e-SSA representation allows us to identify these program points precisely. The modified program has five variables created after v : $\{v_0, v_5, v_9, v_{2c}, v_{2t}\}$, plus three variables created after x : $\{x_1, x_4, x_9\}$. Let's consider the first group of variables. Given that v_0 is produced by source assignment, we know that it is tainted. Variable v_{2c} must be necessarily clean, as it is produced by the validation of v_9 . On the other hand, v_{2t} must be necessarily tainted, for the opposite reason. Variable v_5 , which results from the application of an operation – assignment – on a clean variable, is also clean. Finally, v_9 , which may be assigned either a clean or a tainted value, is tainted, as this is the most conservative choice to detect security vulnerabilities.

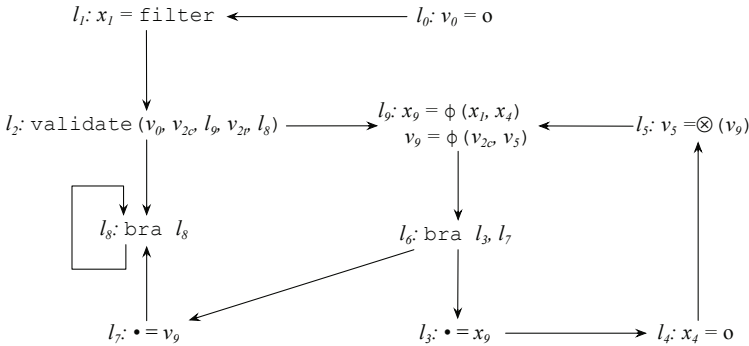


Fig. 4. The example of Figure 3 converted into e-SSA form

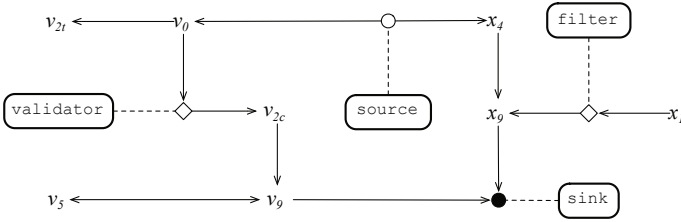


Fig. 5. The reachability graph built after the program in Figure 4

4.2 Tainted Analysis as Graph Reachability

Given a Nano-PHP program P , we represent it as a graph G , in which each node $n_v \in G$ denotes a variable $v \in P$. We build the reachability graph directly from the e-SSA-form Nano-PHP program. Each particular type of instruction produces a specific configuration of nodes in the reachability graph, as Table 3 shows. Roughly, there is an edge linking n_u to n_v if information flows from variable u to v . Notice that, were it not for filters and validators, our reachability graph would represent the def-use chains of the Nano-PHP program. The program from Figure 4 gives origin to the reachability graph in Figure 5.

Table 3. Mapping program instructions to nodes in the reachability graph

Instruction	Example	Nodes
$v = \circ$	$\$v = \$_POST['id']$	$\$_POST['id'] \cdots \circ \rightarrow \v
$\bullet = v$	$\text{echo}(\$v)$	$\$v \rightarrow \bullet \cdots \text{echo}$
$v = \otimes(v_1, \dots, v_2)$	$\$a = \$t1 * \$t2$	$\$t1 \rightarrow \a $\$t2 \rightarrow \a
$v = \text{filter}$	$\$a = \text{stripslashes}(\$t1)$	stripslashes $\diamond \rightarrow \$a$
$v = \phi(v_1, \dots, v_2)$	$\$v = \text{phi}(\$v1, \$v2)$	$\$v1 \rightarrow \v $\$v2 \rightarrow \v
$\text{validate}(v, v_c, l_c, v_t, l_t)$	$\text{if}(\text{is_num}(\$i))$	$\$i \rightarrow \diamond \rightarrow \$i2$ $\diamond \downarrow$ $\$i1 \rightarrow \text{is_num}$

Definition 2 rephrases the tainted flow problem as an instance of graph reachability. The traversal of the reachability graph is related to the notion of program slicing [30]. Any node u that reaches a node v is part of the program slice that defines the behavior of v .

Definition 2. THE TAINTED FLOW PROBLEM AS GRAPH REACHABILITY

Instance: a graph G that describes a Nano-PHP program P .

Problem: determine if G contains a path from a source to a sink that does not cross any sanitizer.

4.3 Addressing Aliasing with HSSA

Aliasing is a phenomenon typical of imperative languages, in which two names reference the same memory location. Aliasing complicates static analyses because it requires the analyzer to understand that updates in the state of a variable may also apply to other variables. To see the implications of aliasing on tainted flow analysis, let's consider the PHP program in Figure 6 (Left). Assuming that `$_GET` is a source and `echo` is a sink, then the program is logically bug free. That is, the name `$i`, which is used in a sink, has been sanitized as name `$j`, because both names, `$i` and `$j` represent the same variable. The ordinary e-SSA representation will not catch this subtlety, as Figure 6 shows. There is a clear path from `$i0` to the sink that does not go across any sanitizer.

In order to deal with aliasing we use an augmented flavor of the e-SSA representation, that we derive from a representation called *Hashed Static Single Assignment (HSSA)* form [7]. This last program representation is used internally by `phc` [3, Sec 6.5], our baseline compiler. For each assignment $v = E$ in a SSA-form program, the equivalent HSSA-form program contains an assignment $(v, a_1, \dots, a_n) = E$, where a_1, \dots, a_n are the aliases of v at the assignment location. Following this strategy, our augmented representation generates new names for each variable created by a sanitizer. The literature contains a plethora of methods to conservatively estimate the set of aliases of a variable. We use the flow sensitive, interprocedural analysis [18] that we obtain from `phc`. Moving on with our example, Figure 7 shows the program and the reachability graph after

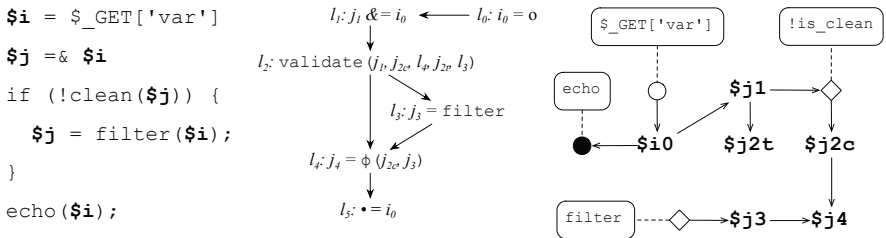


Fig. 6. An example of how aliasing complicates the tainted flow analysis. In the right side we show the reachability graph built for the e-SSA form program.

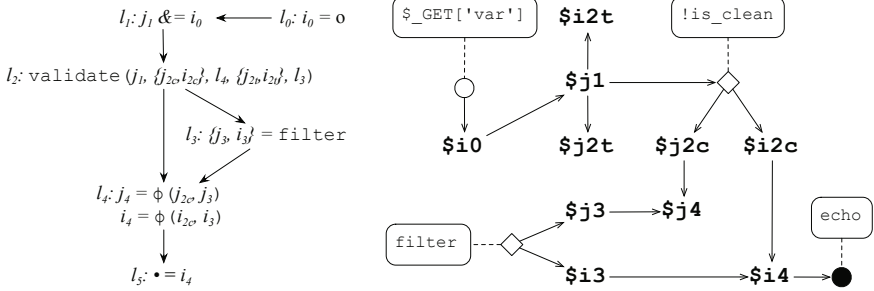


Fig. 7. (Left) input program in e-SSA form augmented with the results of alias analyses. (Right) final reachability graph.

augmenting the e-SSA form program in Figure 6 with the results of alias analysis. In the new reachability graph there is no path from a source to a sink that does not go across a sanitizer. Thus, we report that the program is bug-free.

4.4 A Solution Quadratic in Time and Space

The function `markTaintedVars`, given in Figure 8 finds bugs in e-SSA-form NanoPHP programs. We use SML/NJ’s syntax plus Erlang-style guards in pattern matching, as in the auxiliary function `hasTaintedChild`. This function simulates a traversal of the reachability graph that we described in Section 4.2, but it does not really build the graph. Instead, it relies on the *use-chains* of the variables to guide the traversal. The use-chain of a variable x is a function USE that maps x to every instruction where this variable is used.

Function `markTaintedVars` receives three parameters: a set $\{i, i_1, \dots, i_n\}$ of instructions to process, an environment Σ that maps variables to either clean or tainted, and a set of visited instructions, which we keep to avoid visiting the same instruction twice. `MarkTaintedVars` processes each instruction forwardly, i.e, an instruction that defines a variable x is buggy if any of the instructions that use x is buggy. We assume that every variable used in a sink function is buggy. We use the auxiliary function `hasTaintedChild` to check if any of the instructions in the use chain of a variable x defines a variable that has been set as tainted in the environment. Notice that neither `markTaintedVars` or `hasTaintedChild` deals with switches or filter instructions. These instructions will never define or use tainted variables, and will never be found by any of these functions.

Complexity. The function `markTaintedVars` is quadratic in time and space. Because `markTaintedVars` keeps the use-chains of every variable, this function uses $O(V \times I)$ space, where V is the number of variables in the input program, and I is the number of instructions in this program. The function is recursively called at most once per each program instruction. When the function is called, it might do a linear search on the use-chain of a variable, inside the function `doUseChainSearch`. Therefore, this function has time complexity $O(I^2)$.

```

fun hasTaintedChild  $\_ \{ \dots, (\bullet = x), \dots \} \Rightarrow \text{true}$ 
| hasTaintedChild  $\Sigma \{ \dots, (x = \otimes(\dots)), \dots \} \wedge \Sigma \vdash x = \text{clean} \Rightarrow \text{true}$ 
| hasTaintedChild  $\Sigma \{ \dots, (x = \phi(\dots)), \dots \} \wedge \Sigma \vdash x = \text{clean} \Rightarrow \text{true}$ 
| hasTaintedChild  $\Sigma \{ \dots, (\text{validate}(\_, \_, x, \_)), \dots \} \wedge \Sigma \vdash x = \text{clean} \Rightarrow \text{true}$ 
| hasTaintedChild  $\_ \_ \Rightarrow \text{false}$ 
fun markTaintedVars  $\emptyset \Sigma \_ \Rightarrow \Sigma$ 
| markTaintedVars  $\{i, i_1, \dots, i_n\} \Sigma V \Rightarrow$ 
  let
    val  $V' = \{i\} \cup V$ 
    fun doUseChainSearch  $v =$ 
      let
        val  $N = \text{USE}(v) \setminus V'$ 
        val  $\Sigma' = \text{markTaintedVars} (\{i_1, \dots, i_n\} \cup N) \Sigma V'$ 
      in
        if hasTaintedChild  $\Sigma' \text{USE}(v)$ 
          then  $\Sigma'[v \mapsto \text{tainted}]$ 
          else  $\Sigma'$ 
        end
      in
        case  $i$  of
           $\bullet = x \rightarrow \text{markTaintedVars} \{i_1, \dots, i_n\} \Sigma[x \mapsto \text{tainted}] V'$ 
           $x = \circ \rightarrow \text{doUseChainSearch } x$ 
           $x = \otimes(\dots) \rightarrow \text{doUseChainSearch } x$ 
           $x = \phi(\dots) \rightarrow \text{doUseChainSearch } x$ 
          validate  $x, x_c, l_c, x_t, l_t \rightarrow \text{doUseChainSearch } x_t$ 
        end
    end
  end

```

Fig. 8. The algorithm that finds bugs in Nano-PHP programs

5 Experiments

We have implemented the data-flow analysis discussed in Section 3 and our e-SSA based analysis from Section 4 on top of the `phc` open source compiler [34]. This compiler, started in 2005 by Edsko de Vries and John Gilbert, is implemented in C++, and currently uses our implementation of e-SSA as an internal representation. Our implementation of data-flow analysis uses a standard working list algorithm, and runs on a quasi-topological ordering of the CFG of the input program [2, pag.360].

Benchmarks: We have run our analysis on 20,900 files publicly available in 30 PHP content management systems (CMS). Most of these applications appear in previous works [13,14,31]. The names of these applications are given in Figure 9. In this section we show results for 13,297 files out of the 20,900 inputs (63.6%). The omissions are due to the fact that `phc`, being a static compiler, is not able to analyze some features of PHP, such as dynamic file inclusion or dynamic code evaluation. None of these failures are due to our implementations, i.e, they happen before we have the chance to run the tainted flow analyses. A detailed account of each `phc` failure is provided by Rimsa [21].

Set up: Currently our tool reads a configuration file that determines which functions (user defined or from libraries) are sinks, sources and sanitizers. For these experiments we use a configuration file that identifies cross-site scripting attacks, which we describe in Section 2.1. Notice that by properly pointing sources, sinks and sanitizers our analysis can be easily modified to handle other vulnerabilities, such as SQL injections (Section 2.2).

Efficiency: We compare the time to run the data-flow analysis (Section 3) and the time to run our sparse analysis (Section 4). We run the data-flow analysis on the original program, before the conversion to SSA (and e-SSA) form. In order to produce e-SSA form programs, we start from a non-SSA form program, and augment it with special instructions, i.e, π and ϕ -functions [5,10]. Figure 9 shows that the e-SSA based approach is faster than the data-flow approach as the size of the input functions grow. Each bar is the average sum of the times to process each function of the benchmark, over 10 runs. On the average, our sparse analysis is 28% faster than the traditional data-flow approach. We measure the time to analyze each function individually, and we do not consider functions containing less than 100 assembly instructions, for in this case time measurements are too imprecise. Our benchmarks have provided us with 1,122 function above this threshold. The largest function that we have analyzed contains 1,141 instructions. We speculate that once we cross the boundaries of functions, and analyze whole PHP applications, which might contain thousands of functions, and millions of lines of code, our analysis will be much more efficient than the data-flow approach.

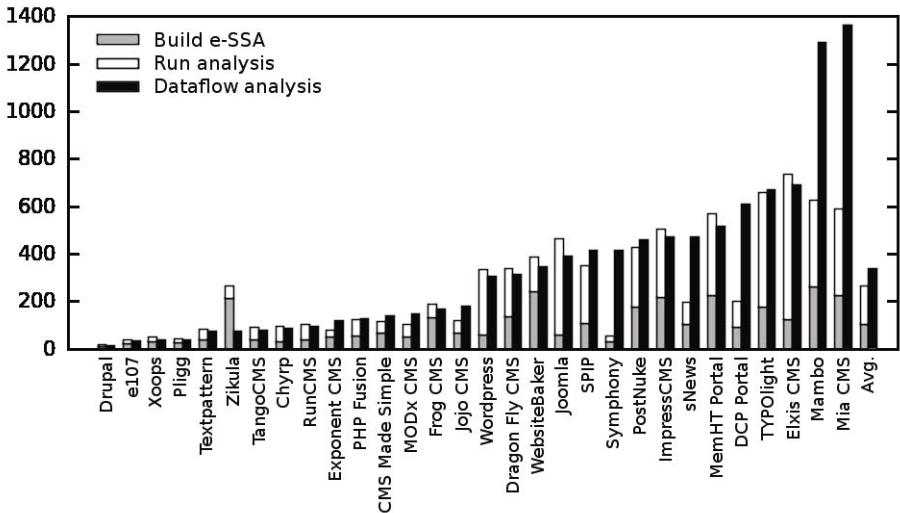


Fig. 9. Average execution time (ms) per benchmark for data-flow and e-SSA-based analyses. Bars are sorted by the time to run the data-flow based analysis.

Table 4. Precision results. F is the number of files, and LOC/F is the number of lines of PHP code per file. Affected is the number of files containing tainted flow vulnerabilities. TP are true positives, and FP are false positives.

benchmark	version	files					warnings	
		total		processed		affected	TP	FP
		F	LOC / F	F	LOC / F			
MODx	1.0.3	472	231	308	228	3	1	1
Exponent CMS	0.97	3456	42	2833	32	3	28	11
DCP Portal	7.0 beta	535	97	392	61	7	5	11
Pligg	1.0.4	380	146	179	154	3	1	0
RunCMS	2.1	737	134	361	86	2	1	6
avg.	-	-	-	-	-	3.60	7.20	5.8

Precision: Both our e-SSA based analysis and the data-flow analysis have succeeded on the same inputs, reporting 63 warning messages across 25 distinct PHP files. Table 4 details these numbers for the subjects that contain confirmed vulnerabilities. Manual inspection of each of these warnings revealed actual vulnerabilities in 36 of these reports, i.e., a 45% false positive ratio. The false positives are due to the lack of whole program analysis, which force us to assume that every function parameter is tainted. We used this list of bugs to perform cross-site scripting attacks in 9 distinct PHP files. To the best of our knowledge, none of these vulnerabilities have been previously reported. We have submitted all these vulnerabilities to the bugtraq at <http://www.securityfocus.com/>. For a detailed account of each bug, see Rimsa [21,22].

5.1 An Example of a Real-World Bug

In order to illustrate our analysis, we will show an actual bug that our implementation found in the content management system MODx CMS version 1.0.3. We have reported this bug to the developers⁹, who acknowledge the presence of the bug. In this example we use the PHP program in Figure 10, which was publicly available on 2010-5-4.

One of the steps of the installation process lets the user choose a database collation from a small suite of options. Users specify this database via three parameters: `host`, `uid` and `pwd`. Users also specify their choice for a collation system via a string, which the PHP program stores in the variable `database_collation`. The PHP file queries a database, using this variable as a key. However, in case the parameters `host`, `uid` or `pwd` do not determine a valid database, the module receives a collation option from a variable originated from a post request, i.e., a form. This string, stored in `database_collation`, is printed in the output without sanitization, as we see in Line 17 of Figure 10. Therefore, in order to print a malicious script in the user's webpage, we can choose an invalid host for the database, and

⁹ <http://www.securityfocus.com/bid/41454>

```

<?php
1 $host = $_POST['host'];
2 $uid = $_POST['uid'];
3 $pwd = $_POST['pwd'];
4 $database_collation = $_POST['database_collation'];
5 $output = '<select id="database_collation" name="database_collation">
<option value="'. $database_collation.'" selected >
.' . $database_collation.'" </option></select>';
6 if ($conn = @mysql_connect($host, $uid, $pwd)) {
    // get collation
7     $getCol = mysql_query("SHOW COLLATION");
8     if (@mysql_num_rows($getCol) > 0) {
9         $output = '<select id="database_collationse_collation"
name="database_collation">';
10        while ($row = mysql_fetch_row($getCol)) {
11            $selected = ( $row[0]==$database_collation ? ' selected' : '' );
12            $output .= '<option value="'. $row[0].'"' . $selected.'" . $row[0].
' </option>';
13        }
14        $output .= '</select>';
15    }
16 }
17 echo $output;
?>

```

Fig. 10. An installation file used in MODx CMS version 1.0.3. This file contains a XSS vulnerability, which we have highlighted in boldface.

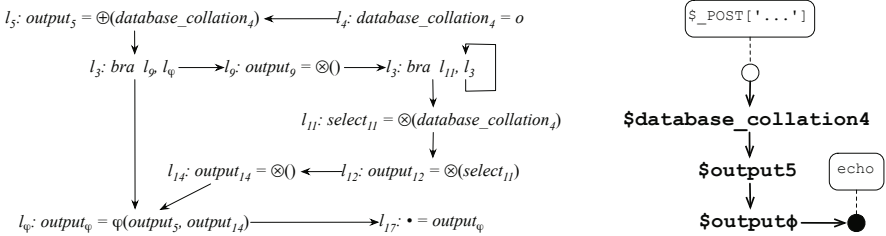


Fig. 11. (Left) the Nano-PHP representation of the program in Figure 10 – we show only the highlighted lines. (Right) The reachability graph.

write the script code directly in the form that feeds `database_collation`. For instance, we can steal cookies from the user’s browsing environment with the string `</option></select><script>window.alert(document.cookie);</script>`. Our analysis finds this vulnerability, as we illustrate in Figure 11. The reachability graph that we build for the example program contains a path from the variable `database_collation`, which is initialized from a source, to the function `echo`, which we qualify as a sink.

6 Related Work

The tainted flow problem is well known in the literature [13,19,28,29,31]. Wasserman and Su [29] have used context-free grammars and string analysis [8] to prove

that functions manipulate strings safely. Another strategy, which uses symbolic execution to solve the tainted flow problem, was proposed by Xie and Aiken [31]. While our analysis has conditional validators powered by the e-SSA representation, these other approaches try to infer new functions as validators. However, a direct comparison between these previous two works and ours is not possible, because the tools are not publicly available. We can only speculate that, by using symbolic execution or string analysis, they are more expensive than ours, although likely more precise. There exist, however, publicly available tools that perform tainted flow analysis. One of them is MARCO [19], a Java bytecode analyzer. Another is Pixy [13], a PHP analyzer. MARCO relies on program slicing [30] to find the set of tainted variables, whereas Pixy uses a variation of the data-flow analysis from Section 3. Neither tool takes the results of conditional tests into consideration; hence, both are path insensitive – a problem that our intermediate representation permits us to circumvent.

Many compiler analyses are based on the notion of graph reachability. In this case, the subject graph normally represents part of a *program slice* [30]. This strategy was made popular by the pioneering works of Choi *et al.* [6] and Reps *et al.* [20]. For a clear explanation of the use of graphs to model data-flow problems, we recommend the work of Scholz *et al.* [23]. The tainted flow problem has been modeled as instances of graph reachability before [11,12,28]. In particular, relying on a modified notion of *thin slicing* [26], Tripp *et al.* [28] have been able to analyze remarkably large benchmarks. However, to the best of our knowledge, we present the first algorithm that uses the e-SSA representation to handle conditional validators inside the graph reachability framework. Conditional validators increase the precision of our analysis, as a given variable might be treated as clean in some program path, and tainted in others, and the e-SSA representation makes it possible to model this flow sensitivity sparsely.

The e-SSA intermediate program representation [5] allows us to model a data-flow problem *sparsely*. There exist many program representations that have been designed with this purpose. The most well known member of this family is the Static Single Assignment (SSA) form [10]. Another program representation that has been conceived with similar objectives is the Static Single Information (SSI) form [1,25], which deals with backward data-flow analyses. We opted to use the e-SSA form because, contrary to SSA form, it allows us to capture information from conditional tests. The SSI representation also gives us this type of information; however, it inserts almost seven times more copies into the source program when compared to the e-SSA form and takes almost 15 times longer to build [27].

7 Conclusion

This paper presented a novel and efficient approach to statically identify security vulnerabilities in code that can result in tainted flow attacks. Key to our speedup was the e-SSA program representation. This enabled us to encode our analysis as a graph reachability problem using a non-iterative data flow algorithm. We have implemented our analysis on top of `phc`, an open source PHP compiler, and have

used it to find real bugs in well known web applications. We reported all the new bugs that we found to the maintainers of the target applications. Some of these developers acknowledged and fixed the vulnerabilities. Our implementation of the e-SSA representation is currently available in the `phc` compiler, our analysis code is available at <http://www.dcc.ufmg.br/11p/projects/phc-tainted/>.

Acknowledgment. Andrei Rimsa is supported by CAPES. We thank Paul Biggar for invaluable help with the `phc` compiler, and Roberto Bigonha plus the anonymous reviewers for helping to improve the text.

References

1. Ananian, S.: The Static Single Information Form. Master's thesis, MIT (September 1999)
2. Appel, A.W., Palsberg, J.: *Modern Compiler Implementation in Java*, 2nd edn. Cambridge University Press, Cambridge (2002)
3. Biggar, P.: *Design and Implementation of an Ahead-of-Time Compiler for PHP*. Ph.D. thesis. Trinity College, Dublin (2009)
4. Biggar, P., de Vries, E., Gregg, D.: A practical solution for scripting language compilers. In: SAC, pp. 1916–1923. ACM, New York (2009)
5. Bodik, R., Gupta, R., Sarkar, V.: ABCD: eliminating array bounds checks on demand. In: PLDI, pp. 321–333. ACM, New York (2000)
6. Choi, J.D., Cytron, R., Ferrante, J.: Automatic construction of sparse data flow evaluation graphs. In: POPL, pp. 55–66 (1991)
7. Chow, F.C., Chan, S., Liu, S.M., Lo, R., Streich, M.: Effective representation of aliases and indirect memory operations in SSA form. In: Gyimóthy, T. (ed.) CC 1996. LNCS, vol. 1060, pp. 253–267. Springer, Heidelberg (1996)
8. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003)
9. Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged information flow for javascript. In: PLDI, pp. 50–62. ACM, New York (2009)
10. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. *TOPLAS* 13(4), 451–490 (1991)
11. Foster, J.S., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: PLDI, pp. 1–12. ACM, New York (2002)
12. Hammer, C., Krinke, J., Snelting, G.: Information flow control for java based on path conditions in dependence graphs. In: ISSSE, pp. 1–10. IEEE, Los Alamitos (2006)
13. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In: S&P, pp. 258–263. IEEE, Los Alamitos (2006)
14. Jovanovic, N., Kruegel, C., Kirda, E.: Precise alias analysis for static detection of web application vulnerabilities. In: PLAS, pp. 27–36. ACM, New York (2006)
15. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. *TOPLAS* 1(1), 121–141 (1979)
16. Ørbæk, P., Palsberg, J.: Trust in the λ -calculus. *Journal of Functional Programming* 7(6), 557–591 (1997)

17. Palsberg, J.: Efficient inference of object types. *Inf. Comput.* 123(2), 198–209 (1995)
18. Pioli, A., Burke, M., Hind, M.: Conditional pointer aliasing and constant propagation. *Tech. Rep. 99-102*, SUNY at New Paltz (1999)
19. Pistoia, M., Flynn, R.J., Koved, L., Sreedhar, V.C.: Interprocedural analysis for privileged code placement and tainted variable detection. In: Gao, X.-X. (ed.) *ECOOP 2005*. LNCS, vol. 3586, pp. 362–386. Springer, Heidelberg (2005)
20. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: *POPL*, pp. 49–61. ACM, New York (1995)
21. Rimsa, A.: Efficient detection of tainted flow vulnerabilities. Master's thesis, Federal University of Minas Gerais (UFMG) (December 2010)
22. Rimsa, A.A., d'Amorim, M., Pereira, F.M.Q.: Efficient static checker for tainted variable attacks. In: *SBLP*. SBC (2010)
23. Scholz, B., Zhang, C., Cifuentes, C.: User-input dependence analysis via graph reachability. *Tech. rep.*, Sun Microsystems, Inc. (2008)
24. Scott, D., Sharp, R.: Specifying and enforcing application-level web security policies. *Trans. on Knowl. and Data Eng.* 15, 771–783 (2003)
25. Singer, J.: *Static Program Analysis Based on Virtual Register Renaming*. Ph.D. thesis, University of Cambridge (2006)
26. Sridharan, M., Fink, S.J., Bodik, R.: Thin slicing. In: *PLDI*, pp. 112–122. ACM, New York (2007)
27. Tavares, A.L.C., Pereira, F.M.Q., Bigonha, M.A.S., Bigonha, R.: Efficient SSI conversion. In: *Brazilian Symposium on Programming Languages (SBLP)*, pp. 1–14 (2010)
28. Tripp, O., Pistoia, M., Fink, S., Sridharan, M., Weisman, O.: TAJ: Effective taint analysis of web applications. In: *PLDI*, pp. 87–97. ACM, New York (2009)
29. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: *PLDI*, pp. 32–41. ACM, New York (2007)
30. Weiser, M.: Program slicing. In: *ICSE*, pp. 439–449. IEEE, Los Alamitos (1981)
31. Xie, Y., Aiken, A.: Static detection of security vulnerabilities in scripting languages. In: *USENIX-SS*. USENIX Association (2006)

Clean Translation of an Imperative Reversible Programming Language

Holger Bock Axelsen

DIKU, Dept. of Computer Science, University of Copenhagen
funkstar@diku.dk

Abstract. We describe the translation techniques used for the code generation in a compiler from the high-level reversible imperative programming language Janus to the low-level reversible assembly language PISA. Our translation is both semantics preserving (correct), in that target programs compute exactly the same functions as their source programs (*cleanly*, with no extraneous garbage output), and efficient, in that target programs conserve the complexities of source programs. In particular, target programs only require a constant amount of temporary garbage space.

The given translation methods are generic, and should be applicable to any (imperative) reversible source language described with reversible flowcharts and reversible updates. To our knowledge, this is the first compiler between reversible languages where the source and target languages were independently developed; the first exhibiting both correctness and efficiency; and just the second compiler for reversible languages overall.

1 Introduction

Reversible computing is the study of computation models that exhibit both forward and backward determinism [2]. Historically, reversible computing originates in the physics of computation: Irreversible computations (for example, seen in our inability to recover the input to a NAND-gate from its output, or the previous value of a variable or register after most assignments) can be shown to have a physical effect on the machines that execute them, in the form of heat dissipation and power consumption [7]. For reversible computations these physical consequences are no longer implied, and it should therefore be possible to lower the power consumption of computing machinery by using reversible components [5]. Lowering power demands is increasingly important as microprocessor technology bottoms out at the atomic level.

To obtain maximal benefit from reversibility a reversible computer should be reversible at every abstraction layer, so reversible hardware [16,13] demands reversible software. However, reversible programming languages are rare and underdeveloped. This is unfortunate, given that reversible programming finds use in many diverse areas of computer science. As an example, in quantum computing [14] programs are necessarily reversible (modulo measurements, which are destructive.) Other application include bidirectional model transformation [10],

static analysis of program properties such as average case time complexity [11], and complex program transformations such as inversion [15].

In the context of compiler theory, we find that there are *no* well-established principles for translation between reversible languages. Two baseline criteria for such a compiler `comp` are its *correctness* and *efficiency*. Correctness means that the translation should be *semantics preserving*, and efficiency here means that the translation should be *complexity preserving*: Given a source program `p`, the target program `q = [[comp]] p` should compute the same function as the source program modulo data representation, *i.e.*, `[[p]] = [[q]]`, and the source and target programs should have the same asymptotic complexities *wrt* resource usage. That such translations are possible is critical for the usefulness of reversible languages.

Both correctness and efficiency present novel challenges for the translation of reversible programming languages: High-level reversible languages are usually reversible at a coarser level than low-level reversible languages. In our source language, Janus, reversibility bottoms out with fairly complex (reversible) statements like `x += a *(5+b) - 17/y`, whereas in the target language, PISA, reversibility bottoms out with individual instructions like `ADDI r1 10`, which is much more fine-grained. The problem is that the components of the source language that exist *below* the granularity of reversibility (for instance, the irreversible expressions in Janus) must *still* be implemented reversibly in the target language, without generating any extraneous *garbage data* for the output. Targeting a reversible assembly language also means that fundamental conventional techniques, such as using scratch registers for temporary values, must be revised, as we can *not* allow overwrite registers. Finally, the source languages contain novel program features that no classical languages exhibit, such as the procedure *uncall* statement in Janus, which executes a procedure *backwards*.

In this paper, we present the techniques used for code generation in a correct and efficient translation from the high-level imperative reversible language Janus [9,19,17] to the low-level reversible assembly language PISA [16,6,3,13]. We consider these to be the most developed and well-understood extant reversible languages in their respective classes. A compiler based on the techniques was implemented and tested. To our knowledge, this paper presents the first translation of reversible languages where the source and target languages have been independently developed, and only the second compiler overall (the other being Frank's R-compiler [6, Apps. C & D].) Our main contributions are as follows.

- We provide a full description of code generation for a *correct* and *efficient* translation of Janus to PISA.
- We give general clean translation methods for generic reversible control flow operators, which avoids code duplication of the computation and uncomputation of their conditional expressions.
- We give methods for explicit register allocation in reversible evaluation of expression trees.

Program

$$p ::= d^* (\text{procedure } id \ s)^+$$

$$d ::= x \mid x[c]$$
Statements

$$s ::= x \oplus = e \mid x[e] \oplus = e$$

$$\mid \text{call } id \mid \text{uncall } id$$

$$\mid \text{if } e \text{ then } s \text{ else } s \text{ fi } e$$

$$\mid \text{from } e \text{ do } s \text{ loop } s \text{ until } e$$

$$\mid \text{skip} \mid s \ s$$
Expressions, operators, constants

$$e ::= c \mid x \mid x[e] \mid e \otimes e$$

$$\otimes ::= \oplus \mid * \mid \&\& \mid <= \mid \dots$$

$$\oplus ::= + \mid - \mid \wedge$$

$$c ::= \dots \mid -1 \mid 0 \mid 1 \mid \dots$$
Syntax domains

$p \in \text{Program}$	$e \in \text{Expression}$	$\otimes \in \text{Operator}$
$s \in \text{Statement}$	$x \in \text{Variable}$	$id \in \text{ProcedureID}$

Fig. 1. Syntax of Janus

Overview. Sect. 2 presents the target and source languages, Sect. 3 motivates the guiding principles underlying the translation, and Sect. 4 provides schemes for code generation. We discuss the implemented compiler in Sect. 5, related work in Sect. 6, and give conclusions and directions for future work in Sect. 7.

2 Languages

Here, we provide a brief overview of the source and target languages.

2.1 Source Language: Janus

The source language for the translation is *Janus*, an imperative structured high-level reversible language designed in the early 80's [9]. Its recent formalization [19] and extension [17] makes it one of the most well-understood and well-developed reversible language in existence. We specifically use the version of Janus defined in [19].

A Janus program consists of a list of global variable declarations, and a list of procedure (subroutine) declarations, see Fig. 1 (some operators are omitted for space reasons). A global variable is a (32-bit signed) integer x , or a (zero-indexed) static size array of integers $x[c]$. All variables and arrays are initialized to zero.

¹ In this version of Janus there are no local variables.

A procedure is a non-parameterized list of statements. A statement s is a reversible assignment² to an integer variable $x \oplus = e$ or array entry $x[e_1] \oplus = e_2$; a (recursive) procedure call `call id` ; a procedure uncall `uncall id` ; a reversible conditional selection `if e_1 then s_1 else s_2 fi e_2` , or a reversible loop `from e_1 do s_1 loop s_2 until e_2` .

Briefly, the semantics of Janus is as follows. A reversible assignment updates the left value by adding (`+=`), subtracting (`-=`) or (bitwise) xoring (`^=`) the value of the expression on the right hand side to the original value. The variable being updated must neither occur in the right hand side nor in the array index expression, to conserve reversibility. A reversible conditional selection is similar to a classical `if-then-else`, but the value of the `fi`-expression must be true after execution if the `then`-branch was taken, and false otherwise. A reversible loop is similar to a classical `do-while`, but the `from`-assertion must be true when entering the loop, and false in all subsequent iterations (*cf.* Fig. 7). A `call` recursively executes a named procedure, and an `uncall` executes the procedure with its inverse functionality. Such direct programming access to the inverse semantics of a languages is a unique feature of reversible languages.

To provide Janus with a stand-alone execution behaviour, the last procedure in the procedure list acts as the `main` procedure for a program, and is executed at run-time.³

2.2 Target Language: PISA

We target the *Pendulum Instruction Set Architecture* assembly language (PISA), as described in [63]. The Pendulum [16] is a RISC architecture, with 32 general purpose 32-bit signed integer registers, designated r_0 to r_{31} .⁴ A formal semantics for PISA and an abstract von Neumann machine on which to run it, is described in [3]. A minimal assembly language BobISA, inspired by PISA, is being developed [13] for future implementation in reversible logic [12]. PISA is thus representative of a large class of reversible machine languages.

A PISA program is a list of (possibly labeled) RISC-style machine instructions, see Fig. 2 for a representative excerpt. An instruction is either a data instruction, or a branching instruction, or a special instruction used in program control.

A data instruction has no direct influence on control flow. The data instructions are reversible versions of classical RISC instructions. As an example, the `ADD r_1 r_2` instruction performs the reversible update $r_1 \leftarrow r_1 + r_2$ (in Janus syntax $r_1 += r_2$). To conserve reversibility the source and target registers must

² A reversible assignment follows the pattern of a *reversible update*: It is a function of the form $g(x, y) = (x \oplus f(y), y)$, where \oplus is a binary operator that is injective in its first argument, *i.e.*, that $b \oplus a = c \oplus a \Rightarrow b = c$. This makes g injective. Note that f is unrestricted, allowing us to use irreversible operators, such as logical conjunction, in the right hand side of a reversible assignment. See also [3][17].

³ An online interpreter can be found at <http://topps.diku.dk/pirc/janus>

⁴ By convention, r_0 is usually preserved as 0, r_1 is the call stack pointer and r_2 is used to store return offsets in procedure calls.

Program

$$p ::= ([l :] i)^+ \\ i ::= a \mid b \mid s$$

Data instructions

$$a ::= \text{ADD } r \ r \mid \text{SUB } r \ r \mid \text{NEG } r \mid \text{XOR } r \ r \mid \dots \\ \mid \text{ADDI } r \ c \mid \text{SUBI } r \ c \mid \text{XORI } r \ c \mid \dots \\ \mid \text{ORX } r \ r \ r \mid \text{ANDX } r \ r \ r \mid \text{SLTX } r \ r \ r \mid \dots \\ \mid \text{EXCH } r \ r$$

Branching instructions

$$b ::= \text{BRA } l \mid \text{RBRA } l \\ \mid \text{BEQ } r \ r \ l \mid \text{BNE } r \ r \ l \mid \text{BGEZ } r \ l \mid \dots \\ \mid \text{SWAPBR } r$$

Special instructions

$$s ::= \text{DATA } c \mid \text{START} \mid \text{FINISH}$$

Immediates

$$c ::= \dots \mid -1 \mid 0 \mid 1 \mid \dots$$

Syntax domains

$$p \in \text{Program} \quad a \in \text{DataInst} \quad s \in \text{Special} \\ l \in \text{Label} \quad b \in \text{BranchInst} \quad r \in \text{Register}$$

Fig. 2. Syntax of PISA (Excerpt)

be different. One may also use immediates in place of a source register. For instructions which do not mimic an operation which is injective in an argument, expanding instructions such as `ANDX $r_1 \ r_2 \ r_3$` (which performs $r_1 \leftarrow r_1 \text{ XOR } (r_2 \text{ AND } r_3)$) are available. This is also used in comparison operations, such as *set-if-less-than* `SLTX`. Reversible memory access is provided by `EXCH $r_1 \ r_2$` which exchanges the contents of r_1 with the value in the memory cell pointed to by r_2 .

A branching instruction is an unconditional (*e.g.*, `BRA l`) or conditional branch (*e.g.*, `BGEZ $r \ l$`) to a label. Branching is made reversible through the architectural design of PISA: At load time, labels are replaced with the *relative offset* of a branch instruction and its target. A branch instruction does not overwrite the program counter pc directly, but instead adds the offset to a control register, the branch register br . In between the execution of any two instructions, the pc is reversibly updated by adding the br to it if $br \neq 0$, and proceeding to the next instruction, if not.

To program jumps in PISA one can use *paired branches*: A branch target should contain a branching instruction pointing back to the jump point, in order to clear the br and resume normal step-wise execution. A final control register, the direction bit *dir*, controls the interpretation and execution direction, allowing

for programmer control of the execution direction through the ingenious reverse branch instruction RBRA. To allow for a labeled instruction to be jumped to from many source points, the SWAPBR r instruction exchanges the value of br with that in register r . Further details in [6,3].

The special instructions are not executed at run-time: The DATA c instruction is used to initialize the memory cell at that point with c at load-time. Execution begins at the START instruction, and halts at the FINISH instruction.

3 Motivation

Before providing technical details, we motivate the guiding principles for the translation.

Historically, the study of reversible computing (starting with Landauer [7] and Bennett [4]) has been focused on transformations from irreversible to reversible programs (for Turing machines), so-called *reversibilizations*. Critically, such transformations are neither semantics nor complexity preserving: The target program of such a transformation computes a different function from the source program, with additional *garbage data* in the output (*e.g.*, a trace of every computation step), and can be asymptotically very inefficient (*e.g.*, requiring as much space as time, regardless of the space usage of the source program.)

Bennett suggests a method for programs computing *injective* functions, see [4, p. 530], which is semantics preserving. While this is *extensionally* clean (*i.e.*, correct), it is still not satisfactory: The method requires the use of a complete execution trace, so target programs would be extremely inefficient, using as much space as time. This is clearly not acceptable for any realistic computing device.

Thus, using existing reversibilizations is unsuitable when we want both correct and efficient translation between reversible language. In fact, no general reversibilization (from irreversible source to reversible target) will be able to guarantee both properties without breaking widely believed conjectures in complexity theory (such as the existence of one-way functions.) However, we also believe that translations between strictly reversible languages *can* be both correct and efficient. Thus we should *not* rely on general reversibilizations.

The central idea of our translation is to exploit that Janus is reversible at the level of *individual statements* in addition to the overall reversibility between input and output. We can aim for an *intensionally* clean translation where each individual statement is translated cleanly, without having to accumulate garbage. We still need to evaluate expressions as a subpart of statements, and this *will* require some use of reversibilization (leading to temporary garbage) as expression evaluation is inherently irreversible. However, we should be able to remove the garbage data immediately following any use of the expression value, exactly because the individual statements are reversible. This will allow us to *reuse the space that would otherwise be filled* by accumulating garbage data!

This has a dramatic effect on efficiency: Expressions are non-recursive, so their size is effectively a (rough) bound on how much garbage we can accumulate by

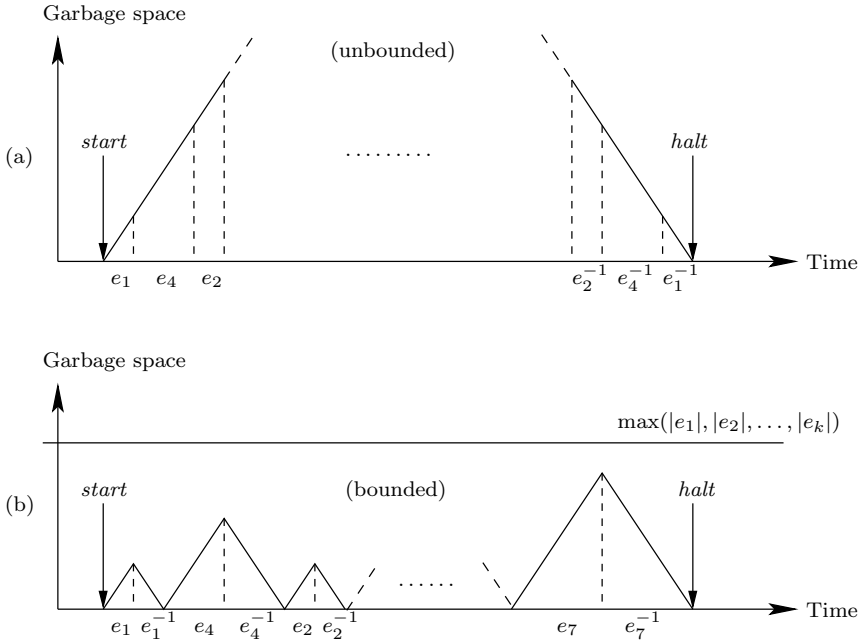


Fig. 3. Garbage space usage during target execution for two different approaches to reversible translation. (a) corresponds to Bennett’s method for injective functions. (b) is our translation. $\{e_1, e_2, \dots, e_k\}$ is the (finite) set of expressions in a particular program.

simulating any single one. This leads to a *constant* upper bound for garbage use for any given program (which only has finitely many expressions, each of fixed size), regardless of how many times we need to evaluate them in a program execution, in stark contrast to what happens with general reversibilization. Fig. 3 shows a conceptual representation of this.

The removal of any garbage data serves to make the translation *correct*, and doing so immediately makes the translation *efficient* (assuming we can translate the other component of Janus correctly and efficiently as well.) Thus, by actively exploiting the reversibility of the source language, we expect the irreversible subparts to be simulable without any impact on neither semantics nor complexity. Of course, this means that we have to come up with good, garbage-free translations for the remaining parts of the source language.

We now turn to the details of the translation based on these ideas.

4 Translation

In this section we present the translation techniques used in our compiler. In particular, we show the development of the *code generation* schemes used in

```

      < variable defs >
      < procedure code >
start : START           ; Start program here
      ADDI rsp size(q) ; Init stack pointer
      BRA  main         ; Call main procedure
      SUBI rsp size(q) ; Clear stack pointer
finish : FINISH        ; Exit program here

```

Fig. 4. Overall layout of a translated program q

the translation. Although we show the translation for two particular languages, the presentation is intended to be sufficiently abstract that one may adapt the schemes for use with other reversible source and target languages (or parts thereof) as well. Many parts of the compiler (parsing, syntax analysis etc.) were straightforwardly implemented using classical methods, and will not be discussed.

Janus is a *structured* language, we can inductively define the translation over the Janus syntax, and the presentation follows this structure:

- Overall target program structure is shown in Sect. [4.1](#)
- Procedure encapsulation and procedure calls are translated in Sect. [4.2](#)
- Reversible assignments (atomic statements) are translated in Sect. [4.3](#)
- Control flow operators are translated in Sect. [4.4](#)
- Expression evaluation is implemented in Sect. [4.5](#)

4.1 Overall Program Structure

A source program consists of declaration lists of global variables and procedures. The variables are global and of fixed size, so we can use the DATA 0 instruction to allocate space for them in the translated program. We use the order of declarations from the source, and use the names as labels, *e.g.*

$$\left. \begin{array}{l} l_x : \text{DATA } 0 \\ \quad \vdots \\ \quad \text{DATA } 0 \end{array} \right\} n \text{ cells for array } x[n], \\
 l_y : \text{DATA } 0 ; 1 \text{ cell for integer } y.$$

This will allow us to read the result of a program execution directly from these memory locations when the target program halts. Variables are also kept in memory across statements that do not refer to them, to simplify the translation.

Following the space for variables comes a list of translated procedures, see below for details. Finally, a small section of code defines the execution behaviour. We assume that the program is loaded at address 0. Program execution begins with the *pc* at the **START** instruction labeled *start*. We shall need a call stack for recursive program calls, so a stack pointer r_{sp} (any of the general purpose registers, but usually r_1) is initialized to point at the first free memory cell

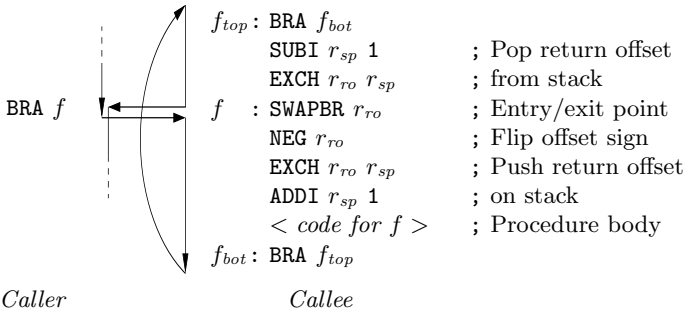


Fig. 5. Translation and calling convention for (recursive) procedures. The arrows show the control flow of a procedure call to f .

above the program by offsetting it with the size of the program $size(q)$ (which is a constant). Then, we call $main$, (where $main$ is the name of the last defined procedure in the program) using the calling convention below. Returning from this procedure call leads to the termination of the program with the `FINISH` instruction labeled $finish$. Following this, the call stack will be empty, and we clear the stack pointer, for cleanliness.

4.2 Procedure Definitions and Procedure Calls

Procedure declarations are translated using a generalized version of the calling convention for PISA defined in [6], with added support for recursion.⁵ The translation is most easily explained by considering how a caller and callee interact, see Fig. 5 for code.

The caller uses a simple `BRA f` instruction to call a procedure. This leads to a jump to the instruction labeled f . This entry/exit point is common to all callers, so the jump offset from caller to callee (the value currently in the branch register br), is moved to a general purpose *return offset* register r_{ro} , using the `SWAPBR` instruction. If r_{ro} is 0 when the call is made, the combined effect of the `BRA` and `SWAPBR` is to jump from caller to callee and proceed with normal step-wise execution from the callee entry point.

When returning from the procedure, the desired offset from callee to caller is the same distance as the original jump offset but with reverse sign, so we negate r_{ro} with `NEG` to get the return offset. The `ADDI` and `EXCH` instruction pushes the return address onto the call stack. The procedure body (which can include recursive calls to f) is then executed. After this, the paired `BRA` instructions send

⁵ In the extended version of Janus [17] procedures are equipped with call-by-reference parameters. The above calling convention can support this straightforwardly in the call sequence by placing the references in an activation record on the call stack. These can be popped into procedure-dependent dedicated registers for formal parameters in the callee prologue (or when needed) and dereferenced when formal parameters are used.

- (1) $\langle \text{code for } r_a \leftarrow \llbracket e_1 \rrbracket \rangle$; Generates garbage G_1
- (2) **ADDI** $r_a \ l_x$; Add base address to index
- (3) $\langle \text{code for } r_e \leftarrow \llbracket e_2 \rrbracket \rangle$; Generates garbage G_2
- (4) **EXCH** $r_d \ r_a$; Swap array entry into r_d
- (5) **ADD** $r_d \ r_e$; Update array entry
- (6) **EXCH** $r_d \ r_a$; Swap back array entry
- (7) $\langle \text{inverse code of 3} \rangle$; Removes garbage G_2
- (8) **SUBI** $r_a \ l_x$; Subtract base address
- (9) $\langle \text{inverse code of 1} \rangle$; Removes garbage G_1

Fig. 6. Translation of reversible (array) assignment $x[e_1] += e_2$. For assignments which use $-=$ or $\hat{+}=$ substitute the instruction in line 5 with **SUB** or **XOR**.

control to the top of the procedure encapsulation. Here, we pop the return offset from the stack and put it into r_{ro} , with the **SUBI** and **EXCH** instructions. Then, the **SWAPBR** returns control to the caller, where the **BRA** instruction restores the br to 0 again, and the caller continues its execution.

Note that the use of a call stack does *not* lead to garbage data: Any data that is added to the stack by a call is also *cleared* when returning from the call, so the size of the stack is conserved over, though not during, calls. Since the stack is initially *empty* when the program calls the *main* procedure, it will also be so when the program terminates.

Finally, procedure uncalls are supported by the use of the **RBRA** instruction in place of **BRA** in the caller. Nothing needs to change in the callee, which means that the *same* code can be shared for both calls and uncalls.

4.3 Reversible Assignments

Janus assignment statements are reversible updates. These can be implemented using a generalized version of Bennett’s method, where the intermediate copying phase is replaced by in-place updating the left hand side variable, see [3]. The main difficulty we face is that expression evaluation is, in general, irreversible, and embedding this evaluation in PISA will necessarily generate garbage data. For a *clean* translation, this garbage must be disposed of, *i.e.*, it must be reversibly cleared.

We shall detail the case of assignment to an array variable, $x[e_1] += e_2$. The translation of this is as follows (see Fig. 6 for a corresponding code template):

- (1) Reversibly evaluate the index expression e_1 , placing the result in (zero-cleared) register r_a . This will generate some garbage data G_1 .
- (2) Add the base address l_x of the array x to r_a , yielding the exact address of the entry in memory.
- (3) Reversibly evaluate the update expression e_2 , placing the result in zero-cleared register r_e . This will generate garbage data G_2 .
- (4) Swap the array entry from its location in memory (given by r_a) with some register r_d , which need *not* be zero-cleared, but which must be different from r_a and r_e .
- (5) Update the array entry value in r_d by adding (subtracting, xoring) r_e .
- (6) Swap the updated array entry back to its memory location, restoring r_d to its original

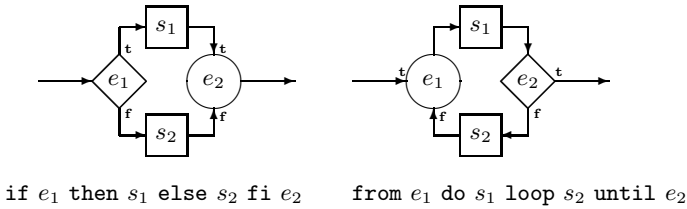


Fig. 7. Reversible flowcharts for the Janus CFOs

value. (7) *Uncompute* (unevaluate) expression e_2 , removing all the garbage data G_2 generated in the forwards evaluation of this expression and clearing r_e . This can be done with the *inverse* of the code from step 3, see below. (8) *Subtract* the base address of x from r_a , leaving only the index. (9) *Uncompute* the index expression e_1 , clearing garbage data G_1 and register r_a . For this, use the inverse of the code generated for step 1.

Steps 1–6 are almost completely conventional, except for the fact that the computations in steps 1 and 3 have the side effect of generating garbage data. If we ignore garbage we would not need to perform steps 7–9, but seeing as we want a clean semantics-preserving translation at the statement level these uncomputations are necessary.

All PISA instructions have inverses that are single PISA instructions as well. The inverse of `ADD` is `SUB`, the inverse of `ANDX` is itself, *etc.* This can be exploited to generate the inverse code for the uncomputations in steps 7 and 9 in a very straightforward manner: Reverse the list of instructions, and invert each instruction in the code from steps 1 and 3. This also means that steps 6–9 are actually the inverses of steps 1–4, so the entire effect of the translated code will be to update the array entry (step 5), with no garbage data left afterwards.

The value in register r_a , once we have computed the specific address in step 2, *must* be conserved over the evaluation of e_2 in step 3. It is only because we *know* that the register is later cleared that we may use it as a free register in future computations. In other words, no instruction in PISA can by itself be used to declare a register *dead*, making general register allocation in PISA non-trivial.

4.4 Control Flow Operators

Frank provides some (informal) guidelines for programming control structures in PISA [6, Ch. 9]. However, it is unclear how these can be used for a clean translation of Janus: There is little to no discussion of evaluation of conditionals, garbage handling, or any such concepts. Furthermore, the discussion of loops is largely limited to fixed iteration for-loops, which are much less general than Janus loops. However, Janus CFOs are fully implementable in PISA without generating any garbage data, as we shall demonstrate.

We shall use a bottom-up approach to explain our translation. It is easy to see that the Janus CFOs (Fig. 7) can be decomposed using the simpler reversible

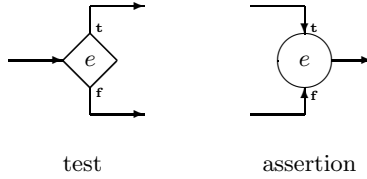


Fig. 8. General reversible flowchart nodes

constructs in Fig. 8. The *test* is simply a conventional conditional branch, while the *assertion* is a join of control flow, where the conditional expression must be *true* when entering the node via the edge labeled **t**, and *false* if entering via the edge labeled **f**, see [18]. We shall examine how each of these constructs can be translated cleanly to PISA and combined for the translation of Janus CFOs.

Translation of Tests. An initial attempt at a PISA translation of the test would be to follow the standard method: Evaluate the expression, placing the result in (zero-cleared) register r_e . Use a conditional jump on r_e for the control flow, jumping to another place in the code if the expression evaluated to zero (coding for *false*) and falling through if not (coding for *true*). In PISA this looks as follows.

```

    <code for  $r_e \leftarrow \llbracket e \rrbracket$ > ; Evaluate  $e$ 
test : BEQ  $r_e$   $r_0$   $test_{false}$  ; Jump if  $\llbracket e \rrbracket = 0$ 
    <code for true branch>
        :
test_false : BRA test ; Receive jump
    <code for false branch>
        :

```

While functional, this approach poses several problems, the major of which is the massive generation of garbage: The value of the evaluation is left in register r_e after being used by the conditional branch, and the evaluation of e will likely leave garbage data in other registers and possibly in memory as well. The simplest way of removing the garbage data would be to push it onto a “garbage stack”, but this would break the cleanness of the translation. Furthermore, since the test is a reversible flowchart, we expect to be able to simulate it in PISA without having to resort to the Landauer embedding.

We observe that we can use the inverse of the expression evaluation code for a Lecerf reversal [8]:

```

    <code for  $r_e \leftarrow \llbracket e \rrbracket$ > ; Evaluate  $e$ 
    <inverse code for  $r_e \leftarrow \llbracket e \rrbracket$ > ; Unevaluate  $e$ 

```

The total effect of this is to compute the identity. (We shall write $r_e \rightarrow \llbracket e \rrbracket$ as a shorthand for the *uncomputation*.) Because branch instructions can not alter

the contents of r_e , we can place the uncomputation code at the start of both the *true* and *false* branch, and the total effect will be to compute the test without generating garbage.

```

                                <code for  $r_e \leftarrow \llbracket e \rrbracket$ > ; Evaluate  $e$ 
test      : BEQ  $r_e$   $r_0$   $test_{false}$  ; Jump if  $\llbracket e \rrbracket = 0$ 
                                <code for  $r_e \rightarrow \llbracket e \rrbracket$ > ; Unevaluate  $e$ 
                                <code for true branch>
                                ⋮
test_{false} : BRA test ; Receive jump
                                <code for  $r_e \rightarrow \llbracket e \rrbracket$ > ; Unevaluate  $e$ 
                                <code for false branch>
                                ⋮

```

Note that the uncomputation code occurs twice in this translation. Because the expressions may be very large, this can have a significant impact on program size, with all its drawbacks, especially *wrt* debugging. This *code duplication* can be avoided per the following observation: Only the value of r_e has an effect on the control flow. Every other piece of garbage data from the expression evaluation may be safely uncomputed *before* the conditional jump. To clear r_e itself we require some knowledge of its value at the branches. If we jumped to the *false* branch then r_e is necessarily zero, and so already cleared. If we fell through to the *true* branch, then r_e was non-zero. Expression evaluation may in general return any integer result for r_e , so it seems that we have no way of deterministically clearing r_e in the *true* branch.

However, for *conditional* evaluations we may reduce the expression result further to being either 0 (for *false*) or 1 (for *true*), since only two distinct values are actually necessary to make the jump work (we write $\llbracket e \rrbracket_c$ for such conditional evaluation). This is easily implemented in PISA: If r_x contains the integer value of $\llbracket e \rrbracket$, we may compute $\llbracket e \rrbracket_c$ in r_e as follows.

```

cond_{top} : BEQ  $r_x$   $r_0$   $cond_{bot}$  ; If  $\llbracket e \rrbracket \neq 0$ ,
              XORI  $r_e$  1 ; set  $r_e = \llbracket e \rrbracket_c = 1$ ,
cond_{bot} : BEQ  $r_x$   $r_0$   $cond_{top}$  ; else leave  $r_e = 0$ .

```

With this strategy, r_e contains either 0 or 1 after evaluation of the conditional expression. Copy this result into an (initially zero-cleared) register r_t . We can now clear r_e along with the other garbage in the uncomputation before the jump.⁶ After the jump r_t can now be deterministically cleared in the *true* branch by a single XORI instruction, since we know it must contain a 1. The full translation of a test is shown in Fig. 9; it avoids code duplication, and generates no garbage data.⁷ (We discuss the error check below.)

⁶ Technically, we do not need the extra register r_t . We can organize the uncomputation to clear everything *except* r_e . This is more efficient, but the details depend on the evaluation strategy for conditional expressions; in particular, on the top-level operator in e .

⁷ An alternative strategy is to implement the expression evaluation as a subroutine. This removes the need for the inverse code as well, as we can use an RBRA instruction for the uncomputation.


```

      BNE  $r_t r_0 error$  ; Error check
      <code for  $r_e \leftarrow \llbracket e \rrbracket_c$ > ; Evaluate  $e$ 
      XOR  $r_t r_e$  ; Set  $r_t = r_e$ 
      <code for  $r_e \rightarrow \llbracket e \rrbracket_c$ > ; Unevaluate  $e$ 
test   : BEQ  $r_t r_0 test_{false}$  ; Jump if  $\llbracket e \rrbracket = 0$ 
      XORI  $r_t 1$  ; Clear  $r_t$ 
      <code for true branch>
      :
test_{false} : BRA test ; Receive jump
      <code for false branch>
      :

```

Fig. 9. Code template for translation of the conditional test in Fig. 8

```

      :
      <code for true branch>
assert_{true} : BRA assert ; Jump
      :
      <code for false branch>
assert   : BNE  $r_t r_0 assert_{true}$  ; Receive jump
      <code for  $r_e \leftarrow \llbracket e \rrbracket_c$ > ; Evaluate  $e$ 
      XOR  $r_t r_e$  ; Clear  $r_t$ 
      <code for  $r_e \rightarrow \llbracket e \rrbracket_c$ > ; Unevaluate  $e$ 
      BNE  $r_t r_0 error$  ; Error check

```

Fig. 10. Code template for translation of the assertion in Fig. 8

Translation of Assertions. We observe that an assertion is actually the *inverse* of a test. Since we already have a garbage-free translation of tests, we can use this symmetry property on the translation template in Fig. 9 to get a similarly garbage-free translation of *assertions*, practically for free. A translation of assertions is shown in Fig. 10.

Due to programmer errors, an assertion might *fail*: e might evaluate to *true* when coming from the *false* branch, and vice versa. In the given translations, such an error will manifest itself in the value of r_t : A failed assertion means that r_t will *not* be zero-cleared after the assertion code is executed. Rather than let the machine continue with erroneous (though reversible) behaviour, we can catch failed assertions by *dynamic error checks*. This is accomplished by checking the final value of r_t . If it is non-zero then the assertion failed, and we jump to an error handling routine *error*. Translated code may be executed in reverse (*e.g.*, in procedure uncalls). In that case a test acts as an assertion, so we need dynamic error checks in the translation of tests as well. Hence the seemingly superfluous check at the beginning of the translated test.

```

      BNE  $r_t r_0 error$  ; Error check
      <code for  $r_e \leftarrow \llbracket e_1 \rrbracket_c$ > ; Evaluate  $e_1$ 
      XOR  $r_t r_e$  ; Set  $r_t = r_e$ 
      <code for  $r_e \rightarrow \llbracket e_1 \rrbracket_c$ > ; Unevaluate  $e_1$ 
test   : BEQ  $r_t r_0 test\_false$  ; Jump if  $\llbracket e \rrbracket_c = 0$ 
      XORI  $r_t 1$  ; Clear  $r_t$ 
      <code for true branch>
      XORI  $r_t 1$  ; Set  $r_t = 1$ 
asserttrue : BRA assert ; Jump
testfalse : BRA test ; Receive jump
      <code for false branch>
assert : BNE  $r_t r_0 assert\_true$  ; Receive jump
      <code for  $r_e \leftarrow \llbracket e_2 \rrbracket_c$ > ; Evaluate  $e_2$ 
      XOR  $r_t r_e$  ; Clear  $r_t$ 
      <code for  $r_e \rightarrow \llbracket e_2 \rrbracket_c$ > ; Unevaluate  $e_2$ 
      BNE  $r_t r_0 error$  ; Error check

```

Fig. 11. Translation of a Janus `if e_1 then s_1 else s_2 fi e_2`

Complete Translation of CFOs. The translations for the reversible control flow nodes in Figs. 9 and 10 can now be combined to yield the complete translation of a Janus conditional selection, see Fig. 11. The principles carry over directly to the translation of a reversible loop, where the placement of code is a little more intricate, see Fig. 12.

The only Janus CFO left is the *sequence* operator. The other translated CFOs are structured with only one entry and exit point, so this amounts to simple code concatenation. Thus, all of Janus' control structures (and statements in general) are implementable in PISA with garbage-free translations.

4.5 Expression Evaluation

Janus expression evaluation is irreversible: different stores may evaluate a given expression to the same value. This means that we cannot, in any way, implement expression evaluation *cleanly* in a reversible language: Evaluating an expression necessarily leaves garbage. This was recognized in the translation of reversible assignments and CFOs, where we chose to *unevaluate* the expression immediately after the use of the expression value, to clear any such garbage generated. This makes the translation of statements clean, which is the best we can achieve.

The problem of reversible expression evaluation then reduces to finding a way of forwards evaluating expressions in general using reversible instructions, generating garbage as necessary. Because expressions are uncomputed after use, any reversibilization will work, but we should still aim for efficiency.

Expressions in Janus are trees, so we can use simple post-order traversal to generate code (*Maximal Munch*, cf. [1, Ch. 9]). A leaf node on the tree (representing a constant or variable) is simply copied into a free register. An internal

```

XORI r_t 1                ; Set r_t = 1
entry : BEQ r_t r_0 assert ; Receive jump
       <code for r_e ← [e_1]_c > ; Evaluate e_1
XOR r_t r_e                ; Clear r_t = [e_1]_c
       <code for r_e → [e_1]_c > ; Unevaluate e_1
BNE r_t r_0 error         ; Error check
       <code for s_1 >
BNE r_t r_0 error         ; Error check
       <code for r_e ← [e_2]_c > ; Evaluate e_2
XOR r_t r_e                ; Set r_t = [e_2]_c
       <code for r_e → [e_2]_c > ; Unevaluate e_2
test  : BNE r_t r_0 exit   ; Exit if [e_2]_c = 1
       <code for s_2 >
assert : BRA entry         ; Jump to top
exit   : BRA test          ; Receive exit jump
XORI r_t 1                ; Clear r_t

```

Fig. 12. Translation of a Janus `from e1 do s1 loop s2 until e2 loop`

node (an expression $e_1 \otimes e_2$, ignoring unary operators) is (recursively) translated, yielding the following code structure.

$$e_1 \otimes e_2 \implies \begin{array}{l} 1. \text{ <code for } r_{e_1} \leftarrow [e_1] \text{ >} \\ 2. \text{ <code for } r_{e_2} \leftarrow [e_2] \text{ >} \\ 3. \text{ <code for } r_e \leftarrow r_{e_1} [\otimes] r_{e_2} \text{ >} \end{array}$$

Here, r_{e_1} , r_{e_2} and r_e are zero-cleared registers. This mimics the conventional translation of expressions, but in the reversible setting we are faced with two interesting problems. How do we allocate registers? How can we translate irreversible operators?

Register Allocation for Expression Trees. In irreversible languages this can be done optimally using the Sethi-Ullman algorithm. This is not available to us in reversible languages, as scratch registers cannot be indiscriminately overwritten. The Sethi-Ullman algorithm assumes them to be *dead*, but in this translation they are still live, as they will be reversibly cleared in uncomputation. In any case, overwriting a register simply is not possible in PISA. We thus also expect register pressure to be somewhat higher in reversible machine code. It is therefore important to know how we can free registers.

Instead of the usual categories of *live* and *dead* registers, we partition the register file into the following sets.

- *Free registers.* We know these to be zero-cleared, and may use them freely.
- *Commit registers.* These contain values that we shall need at a future point in the expression evaluation.
- *Garbage registers.* These contain values that are no longer needed for the computation.

In the translation of $e_1 \otimes e_2$ above, r_{e_1} is a *free* register before step 1. During step 2 it is a *commit* register as we need it for step 3, after which it becomes

a *garbage* register. By maintaining the partitioning explicitly during the code generation for the expression, we can use several strategies to free registers.

- *Pebbling*. Garbage registers can be locally uncomputed. This is space-wise efficient, but can be very costly *wrt* time, if done recursively.
- *Garbage spills*. Garbage registers can be pushed unto the stack. This requires space, but reduces the number of executed instructions needed for an evaluation.
- *Commit spills*. We can push a *context* of commit registers onto the stack, and restore them when they are needed.

In the implemented compiler a mixture of all three strategies is used: At leaf nodes in the expression tree *pebbling* is used as the uncomputations at leafs are extremely short. For inner nodes *garbage spills* are used, with *commit spills* only as a last resort. With global variables allocated to memory, and no local variables, the compiler uses explicit register allocation throughout, without the use of virtual registers.

Operator Translation. With very few exceptions (unary minus, bitwise negation) the operators in Janus expressions are irreversible. However, most operators are still supported in PISA in various guises. Assume that we want to evaluate $x \otimes y$, with the values of x and y stored in r_x and r_y , respectively.

Addition, subtraction and exclusive-or are directly supported. For example, $x + y$ can be evaluated by `ADD r_x r_y` . This has the added advantage of reusing r_x as the commit register for the total expression, leaving only r_y as garbage. Other operators, such as bitwise disjunction, $x | y$, have expanding support which consumes a free register (r_e): `ORX r_e r_x r_y` .

The most interesting, however, are those with no clear support in PISA, expanding or otherwise. As an example, we shall look at the equality test $x = y$. We shall need the `SLTX` (*set-less-than-xor*) instruction,

$$\llbracket \text{SLTX } r_d \ r_s \ r_t \rrbracket = r_d \leftarrow r_d \oplus ((r_s < r_t) ? 1 : 0),$$

where \oplus is (bitwise) exclusive-or. We can use simple logical identities to reduce equality to the *less-than* comparisons. An obvious choice would seem to be

$$x = y \Leftrightarrow \neg(x < y \vee y < x)$$

However, both the logical NOR operation and *less-than* are only available as expanding instructions: Naïvely using this identity requires *three* free registers (here r_s , r_t and r_e) to compute $x = y$ as follows.

```
SLTX  $r_s$   $r_x$   $r_y$ 
SLTX  $r_t$   $r_y$   $r_x$ 
NORX  $r_e$   $r_s$   $r_t$ 
```

Since registers are a scarce commodity, we want to do better, and indeed we can. Note that at most one of $x < y$ or $y < x$ can be true ($<$ is antisymmetric).

$$x = y \Leftrightarrow \neg(x < y \oplus y < x).$$

Exclusive-or is directly supported, and logical negation is reversible, so we can evaluate $x = y$ using only one free register:

```
SLTX  $r_e r_x r_y$ 
SLTX  $r_e r_y r_x$ 
XORI  $r_e 1$ 
```

Some operators (such as logical conjunction) still require three registers. Finally, there are also operators in Janus that have no short implementation in PISA, *e.g.*, multiplication or modulus, which can require executing hundreds of PISA instructions. For these one can inline and specialize a reversible simulation of the operator, or use the reversible simulations as subroutines, with proper parameter handling. In the latter case, it is important that the callee subroutine does not perturb any registers other than those of the arguments.

5 Implementation

A compiler based on the above translation methods was implemented in ML (Moscow ML, version 2.01), in approximately 1500 lines of code. Target programs were tested on the PendVM Pendulum simulator⁸ and compared with source runs in a Janus interpreter. All tests corroborated the correctness and efficiency of the translation, so target programs do not leave garbage data in neither registers nor memory.

The largest program translated was a PISA interpreter written in Janus, specialized to a PISA program running a simple physical simulation of a falling object. Weighing in at slightly more than 500 lines of Janus code, this is possibly the largest reversible program ever written, and was certainly the most complex available to the author. The compiled code is *ca.* 12K PISA instructions long. This program had still negligible compile and execution times, so we omit timing statistics. In general, target programs were about 10–20 times larger (in lines of code) than their source programs.

6 Related Work

The only other work on compilers for reversible languages known to the author is Frank's R-to-PISA compiler [6]. The R-compiler is described mainly by commented code in [6, App. D], and is not being maintained, which makes it difficult to discern how the compiler is intended to work abstractly, and verify its correctness and/or efficiency.

R is a prototype reversible procedural language developed specifically for compilation to PISA. R shares some features with Janus, but also has a number of significant differences. For example, R's control flow operators (CFOs) are fairly weak. R-loops are made for definite iteration, and there is no if-then-else CFO. The R if-then CFO uses just a single conditional expression as both if- and fi-conditional. Also, all subexpressions of the conditional must be conserved across

⁸ C. R. Clark, *The Pendulum Virtual Machine*. Available at

<http://www.cise.ufl.edu/research/revcomp/users/cclark/pendvm-fall2001/>

the branch body. In particular, this means that no variables occurring in the conditional expression can be updated in the body, severely limiting the expressiveness of R. On the other hand, R does have some advanced features that Janus does not, like direct access to memory, and input/output facilities. However, the emerging picture is still that R is somewhat limited in its expressiveness as a programming language, compared to Janus.

We believe such restrictions were imposed on R to simplify compilation: The R-compiler can leave any garbage values used for conditional expressions *in place* across the branch bodies, computing and uncomputing the expression (which removes the garbage) only once, and only *after* the conditional is exited. While the analogous translation in Janus requires *two* computations and uncomputations, this also implies that the translation of R is not intentionally clean. This strategy is furthermore *not* applicable to Janus translation, where if- and fi-conditional expressions are allowed to be different, and variables occurring therein are allowed to be updated freely in the branch bodies. (However, our translation could easily be applied to R.) Finally, by not clearing the garbage value before entering the branch body, target programs can generate unbounded garbage data at run-time in recursive procedure calls, which breaks efficiency.

7 Conclusion and Future Work

We presented a correct and efficient translation for compiling the reversible high-level programming language Janus to the reversible low-level machine language PISA. Target programs produced using this translation conserve both the semantics (correctness) and space/time complexities (efficiency) of the source programs. We achieved this by making the translation *intensionally* (as well as extensionally) clean: Reversibility in Janus bottoms out at the statement level, and the compilation reflects this by translating each *individual* statement in the source program cleanly. This has the effect that at no point in the execution of any translated program do we accumulate more than a constant amount of temporary garbage data.

By breaking down the control flow operators of Janus into simpler reversible flowchart nodes, we found that we could exploit the symmetry properties of reversible flowcharts to simplify the translation. We also eliminated the need for code duplication in the translation. The developed translation methods are generic, and will work for all languages with control flow describable by reversible flowcharts and statements describable by reversible updates.

The aim here was to produce a working compiler that demonstrates the fundamental structure of a correct and efficient translation between reversible languages, leaving plenty of opportunities for development and future research. General register allocation methods for reversible assembly languages must be developed, and might benefit from the novel partitioning of registers we use for register allocation for expression trees. The heavy use of uncomputation of expression evaluations in both reversible assignments and conditionals suggest that novel as well as conventional optimizations (such as common subexpression elimination) could be very useful in compilers for reversible languages.

Acknowledgments. A preliminary version of this work was presented at the 2nd Workshop on Reversible Computing in Bremen, July 2010.

References

1. Appel, A.W.: Modern Compiler Implementation in ML. Camb. Uni. Press, New York (1998)
2. Axelsen, H.B., Glück, R.: What do reversible programs compute? In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 42–56. Springer, Heidelberg (2011)
3. Axelsen, H.B., Glück, R., Yokoyama, T.: Reversible machine code and its abstract processor architecture. In: Diekert, V., Volkov, M.V., Voronkov, A. (eds.) CSR 2007. LNCS, vol. 4649, pp. 56–69. Springer, Heidelberg (2007)
4. Bennett, C.H.: Logical reversibility of computation. IBM Journal of Research and Development 17, 525–532 (1973)
5. De Vos, A.: Reversible Computing: Fundamentals, Quantum Computing and Applications. WILEY-VCH, Weinheim (2010)
6. Frank, M.P.: Reversibility for Efficient Computing. PhD thesis, MIT (1999)
7. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development 5(3), 183–191 (1961)
8. Lecerf, Y.: Machines de Turing réversibles. Récursive insolubilité en $n \in \mathbf{N}$ de l'équation $u = \theta^n u$, où θ est un “isomorphisme de codes”. Comptes Rendus Hebdomadaires 257, 2597–2600 (1963)
9. Lutz, C.: Janus: a time-reversible language. Letter written to R. Landauer (1986), <http://tetsuo.jp/ref/janus.html>
10. Mu, S.-C., Hu, Z., Takeichi, M.: An algebraic approach to bi-directional updating. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 2–20. Springer, Heidelberg (2004)
11. Schellekens, M.: MOQA; unlocking the potential of compositional static average-case analysis. Journal of Logic and Algebraic Programming 79(1), 61–83 (2010)
12. Thomsen, M.K., Axelsen, H.B.: Parallelization of reversible ripple-carry adders. Parallel Processing Letters 19(2), 205–222 (2009)
13. Thomsen, M.K., Glück, R., Axelsen, H.B.: Towards designing a reversible processor architecture (work-in-progress). In: Reversible Computation. Preliminary Proceedings, pp. 46–50 (2009)
14. Thomsen, M.K., Glück, R., Axelsen, H.B.: Reversible arithmetic logic unit for quantum arithmetic. J. of Phys. A: Math. and Theor. 42(38), 2002 (2010)
15. van de Snepscheut, J.L.A.: What computing is all about. Springer, Heidelberg (1993)
16. Vieri, C.J.: Reversible Computer Engineering and Architecture. PhD thesis, MIT (1999)
17. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: Proceedings of Computing Frontiers, pp. 43–54. ACM Press, New York (2008)
18. Yokoyama, T., Axelsen, H.B., Glück, R.: Reversible flowchart languages and the structured reversible program theorem. In: Aceto, L., Danggård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfssdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 258–270. Springer, Heidelberg (2008)
19. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: Proceedings of Partial Evaluation and Program Manipulation, pp. 144–153. ACM Press, New York (2007)

Interpreter Instruction Scheduling

Stefan Brunthaler

Institut für Computersprachen
Technische Universität Wien
Argentinerstraße 8,
A-1040 Wien
`brunthaler@complang.tuwien.ac.at`

Abstract. Whenever we extend the instruction set of an interpreter, we risk increased instruction cache miss penalties. We can alleviate this problem by selecting instructions from the instruction set and re-arranging them such that frequent instruction sequences are co-located in memory. We take these frequent instruction sequences from hot program traces of external programs and we report a maximum speedup by a factor of 1.142. Thus, interpreter instruction scheduling complements the improved efficiency of an extended instruction set by optimizing its instruction arrangement.

1 Motivation

For compilers instruction scheduling is an important optimization that re-arranges assembler instructions of a program to optimize its execution on a native machine without changing the semantics of the original program, i.e., the native machine is constant, but we can change the order of assembler instructions of the program. Interestingly, the situation is actually the other way around in interpreters. Usually, the bytecode instructions of an interpreter cannot be re-arranged without changing the semantics of the corresponding programs. However, we can re-arrange the instructions of an interpreter, such that frequently executed sequences of instructions become co-located in memory, which allows for better instruction cache utilization. So, for interpreters, the program is constant, but we can change the virtual machine to optimize the execution of a program.

Interpreter instruction scheduling becomes increasingly important when an interpreter has a large instruction set, because in such an interpreter not all instructions can be held in caches at all times. Consequently, there is a trade-off between the optimizations and their benefit being influenced by possible cache miss penalties. Our own previous work [\[43\]](#) on improving the efficiency of interpreters using purely interpretative optimization techniques, relies heavily on instruction set extension. Fortunately, these optimization techniques are efficient enough to offset increased cache-miss penalties, however, we feel that by using interpreter instruction selection, the gains of these optimization techniques can be noticeably improved.

Other optimization techniques like superinstructions and replication [5] focus on improving branch prediction and instruction cache utilization by copying instruction implementations together into one instruction or distributing copies of the same instruction over the interpreter dispatch routine to improve locality. Contrary to these approaches, interpreter instruction scheduling does not increase the size of the interpreter’s dispatch loop, but focuses on improving instruction cache utilization instead of improving branch prediction.

Our contributions are:

- We formalize the general concept of interpreter instruction scheduling.
- We present detailed examples of how interpreter instruction scheduling works, along with an implementation of our algorithm; the implementation is complemented by a detailed description and results of running the algorithm on an actual benchmark program.
- We report a maximum speedup of 1.142 and an average speedup of 1.061 when using interpreter instruction scheduling on the Intel Nehalem architecture, and provide results of our detailed evaluation.

2 Background

We present a formal description of the problem of interpreter instruction scheduling.

$$\begin{aligned}
 I &:= i_0, i_1, \dots, i_n \\
 A &:= a_0, a_1, \dots, a_n \\
 P &:= p_0, p_1, \dots, p_m \\
 \forall p \in P : \exists j : i_j \in I \wedge a_j \in A &\Leftrightarrow p = (i_j, a_j)
 \end{aligned} \tag{1}$$

$$\begin{aligned}
 T &:= \{(p, f) \mid p \in P \wedge f \in \mathbb{N}\} \\
 K &:= \{p \mid (p, f) \in T \wedge f \geq L\} \\
 K &\subset P
 \end{aligned}$$

We define an interpreter I as a set of n instructions i . To each instruction i corresponds a native machine address a of the set of n addresses A , i.e., the address for some interpreter instruction i_j is a_j . Next, we define a program P consisting of m instruction occurrences, which are tuples of an instruction i and the corresponding address a . This concludes the definition of the static view on an interpreter. However, our optimization requires profiling information obtained at run-time. Thus, we define the trace T of a program P as the set of all tuples of an instruction occurrence p and its execution frequency f . Since a trace contains much more information than we need, we define a kernel K , that contains all instruction occurrences p of our program P that have execution frequencies above some definable threshold L .

Given these definitions, the following functions allow us to precisely capture the concept of the distance between instructions.

$$\begin{aligned}
 s(p_i) &:= |a_{i+1} - a_i| \\
 d(p_i, p_j) &:= \begin{cases} |a_j - a_i| - s(p_i) & \text{if } i \leq j, \\ |a_i - a_j| - s(p_j) & \text{if } i > j. \end{cases} \\
 d_{\text{overall}}(P) &:= \sum_{j=1}^m d(p_{j-1}, p_j)
 \end{aligned} \tag{2}$$

First, we define a function s that computes the size of an instruction i . Next, we define a function d that computes the distance of two arbitrary instructions. Here, the important thing is to note, that if two instruction occurrences p_i and p_j refer to two adjacent instructions, i.e., $p_i = (i_i, a_i)$ and $p_j = (i_{i+1}, a_{i+1})$, then the distance between them is zero. ($d(p_i, p_j) = |a_{i+1} - a_i| - |a_{i+1} - a_i|$) Finally, the overall distance of a program is the sum of all of its sequential distances. Using static program data, this makes no sense, because we do not a priori know which parts of program P are hot. Here, we use our kernel K , which contains only relevant parts of the program, with respect to the overall computational effort. Thus, we define interpreter instruction scheduling as finding a configuration of interpreter instructions that results in a minimal overall distance over some kernel K .

For further illustration, we introduce a working example here. We are going to take a close look on how interpreter instruction scheduling works, using the `fasta` benchmark of the computer language benchmarks game [6]. Running the `fasta` program on the Python interpreter, for example with an argument of 50,000, results in the execution of 10,573,205 interpreter instructions. If we instrument a Python interpreter to trace every instruction executed, with additional location information, such as the instruction offset and the function it belongs to, we can easily extract the computationally relevant kernels from a running program. If we restrict ourselves to only consider kernels for interpreter instruction scheduling, we can significantly reduce the amount of information to consider. For example, an aggregated trace of the `fasta` program shows that the interpreter executes 5,976,237 instructions while interpreting the `genRandom` function, i.e., more than half of the totally executed instructions can be attributed to just one function (cf. Table 1.) Another function—an anonymous list comprehension—requires 4,379,824 interpreted instructions (cf. Table 2.) Together, the `genRandom` function and the list comprehension represent 97.95% of all executed instructions.

Though Tables 1, and 2 indicate that our trace gathering tool is imprecise, since it seems to lose some instruction traces, it is precise enough to indicate which parts of the instructions are kernels. For example, the kernel of function `genRandom` includes all 15 instructions between the offsets 64 and 184, whereas the kernel of the anonymous list comprehension includes all 12 instructions between the offsets 24 and 104. In consequence, our interpreter instruction

Table 1. Dynamic bytecode frequency for `genRandom` function of benchmark program `fasta`

Frequency	Offset	Instruction Identifier
1	16	STORE_FAST_A
1	24	LOAD_GLOBAL_NORC
1	32	LOAD_FAST_B_NORC
1	40	CALL_FUNCTION_NORC
1	48	STORE_FAST_C
1	56	SETUP_LOOP
396,036	64	LOAD_FAST_A_NORC
400,000	72	LOAD_FAST_NORC
400,000	80	INCA_LONG_MULTIPLY_NORC
396,037	88	LOAD_FAST_NORC
400,000	96	INCA_LONG_ADD_NORC_TOS
396,041	104	LOAD_FAST_B_NORC
400,000	112	INCA_LONG_REMAINDER_NORC_TOS
396,040	120	STORE_FAST_A
400,000	128	LOAD_FAST_D_NORC
400,000	136	LOAD_FAST_A_NORC
400,000	144	INCA_FLOAT_MULTIPLY_NORC
396,039	152	LOAD_FAST_C_NORC
400,000	160	INCA_FLOAT_TRUE_DIVIDE_NORC_TOS
396,039	168	YIELD_VALUE
399,999	184	JUMP_ABSOLUTE

Table 2. Dynamic bytecode frequency for an anonymous list comprehension of benchmark program `fasta`

Frequency	Offset	Instruction Identifier
6,600	16	LOAD_FAST_A
402,667	24	FOR_ITER_RANGEITER
396,002	32	STORE_FAST_B
399,960	40	LOAD_DEREF
396,001	48	LOAD_DEREF_NORC
396,000	56	LOAD_DEREF_NORC
396,000	64	LOAD_DEREF_NORC
396,000	72	FAST_PYFUN_DOCALL_ZERO_NORC
395,999	80	FAST_C_VARARGS_TWO_RC_TOS_ONLY
395,999	88	INCA_LIST_SUBSCRIPT
395,998	96	LIST_APPEND
395,998	104	JUMP_ABSOLUTE
6,600	112	RETURN_VALUE

scheduling algorithm only has to consider the arrangement of 27 instructions which constitute almost the complete computation of the `fasta` benchmark. If all 27 instructions are distinct, the optimal interpreter instruction scheduling consists of these 27 instructions being arranged sequentially and compiled adjacently, according to the order given by the corresponding kernel. However, because of the repetitive nature of load and store instructions for a stack-based architecture, having a large sequence of non-repetitive instructions is highly unlikely. Therefore, our interpreter instruction scheduling algorithm should be able to deal with repeating sub-sequences occurring in a kernel. In fact, our `fasta` example contains repetitions, too. The `genRandom` function:

- `LOAD_FAST_A_NORC`, at offsets: 64, 136.
- `LOAD_FAST_NORC`, at offsets: 72, 88.

The anonymous list comprehension contains the following repetition:

- `LOAD_DEREF_NORC`, at offsets: 48, 56, 64.

Fortunately, however, only single instructions instead of longer sub-sequences repeat. Therefore, for the `fasta` case, an optimal interpreter instruction scheduling can easily be computed. We generate a new optimized instruction set from the existing instruction set and move instructions to the head of the dispatch loop according to the instruction order in the kernels. We maintain a list of all instructions that have already been moved, and whenever we take a new instruction from the kernel sequence, we check whether it is already a member of that remembered list. Thus, we ensure that we do not re-reorder already moved instructions. For our `fasta` example, this means that for all of the repeated instructions, we only generate them when we process them for the first time, i.e., only at the first offset position for all occurrences. Consequently, interpreter instruction scheduling generates long chains of subsequently processed instruction sequences that correspond extremely well to the major instruction sequences occurring in the `fasta` benchmark. In fact, we report our highest speedup by a factor of 1.142 for this benchmark.

Thus, if we have an interpreter with many instructions—such as interpreters doing extensive quickening based purely interpretative optimizations, such as inline caching via quickening [4], or eliminating reference counts with quickening [3]—we can reduce potential instruction cache misses using interpreter instruction scheduling.

3 Implementation

The implementation includes all details necessary to implement interpreter instruction scheduling. First, we present an in-depth discussion of how to deal with sub-sequences and why we are interested in them (Section 3.1). Next, we are going to explain how to compile an optimized instruction arrangement with `gcc` (Section 3.2).

3.1 Scheduling in the Presence of Repeating Sub-sequences

As we have seen in the previous section (cf. Section 2), not all kernels contain repeating sub-sequences. However, all larger program traces are likely to contain sub-sequences, or at least similar sequences of instructions. Thus the overall distance of a kernel K depends substantially on the distance of its sub-sequences. We encode the whole trace into a graph data structure and will create an interpreter instruction schedule that contains the most frequent sequences.

In order to demonstrate this approach, we introduce another example from the computer language benchmarks game [6], viz. the `nbody` benchmark. Running the `nbody` benchmark with an argument of 50,000 on top of our instrumented Python interpreter for dynamic bytecode frequency analysis results in the execution of 68,695,970 instructions, of which 99.9% or 68,619,819 instructions are executed in the `advance` function. Its kernel K consists of a trace of 167 instructions, distributed among just 29 instructions.

Creating an instruction schedule using the simple algorithm of the previous section (cf. Section 2) is going to be sub-optimal, since it does not account for representation of repeating sub-sequences. To properly account for these sub-sequences, we create a weighted directed graph data-structure of all 29 instructions as nodes. Since the kernel is a sequence of instructions, we create an edge in this digraph for each pair of adjacent instructions. Whenever we add an edge between two nodes that already exists, we increment the weight of the already existing edge, instead of adding another one. (cf. Figure 1)

Once we have such a digraph, we obtain an instruction schedule with a minimum distance the following way. Given we have some node, our algorithm always chooses the next node by following along the edge with the highest weight. First, we create a list named `open` that contains tuples of nodes and the collective weight of edges leading to that node. We sort the `open` list in descending order of the collective weight component. Because we actually only need the collective weight for choosing the first node and ensuring that we process all nodes, we can now safely zero out all weights of the tuples in the `open` list. Then, we start the actual algorithm by fetching and removing the first tuple element from the `open` list; we assign the node part to n and ignore the weight. Next, we check whether n has already been scheduled by checking whether the `schedule` list contains n . If it has not been scheduled yet, we append it to the `schedule` list. Then, we start looking for a successor node m . We process the successor nodes by having them sorted in descending order of the edge-weight associated between nodes n and m . We repeatedly fetch nodes m from the list of successors until we find a node that has not already been scheduled or the list is finally empty. If we do not find a node m , then we have to restart by fetching the next node from the `open` list. If we find a node m , then we add the reachable nodes from n to m to the `open` list and sort it, such that the successors with the highest weight will be chosen as early as possible. Next we assign m to n and restart looking for m 's successors.

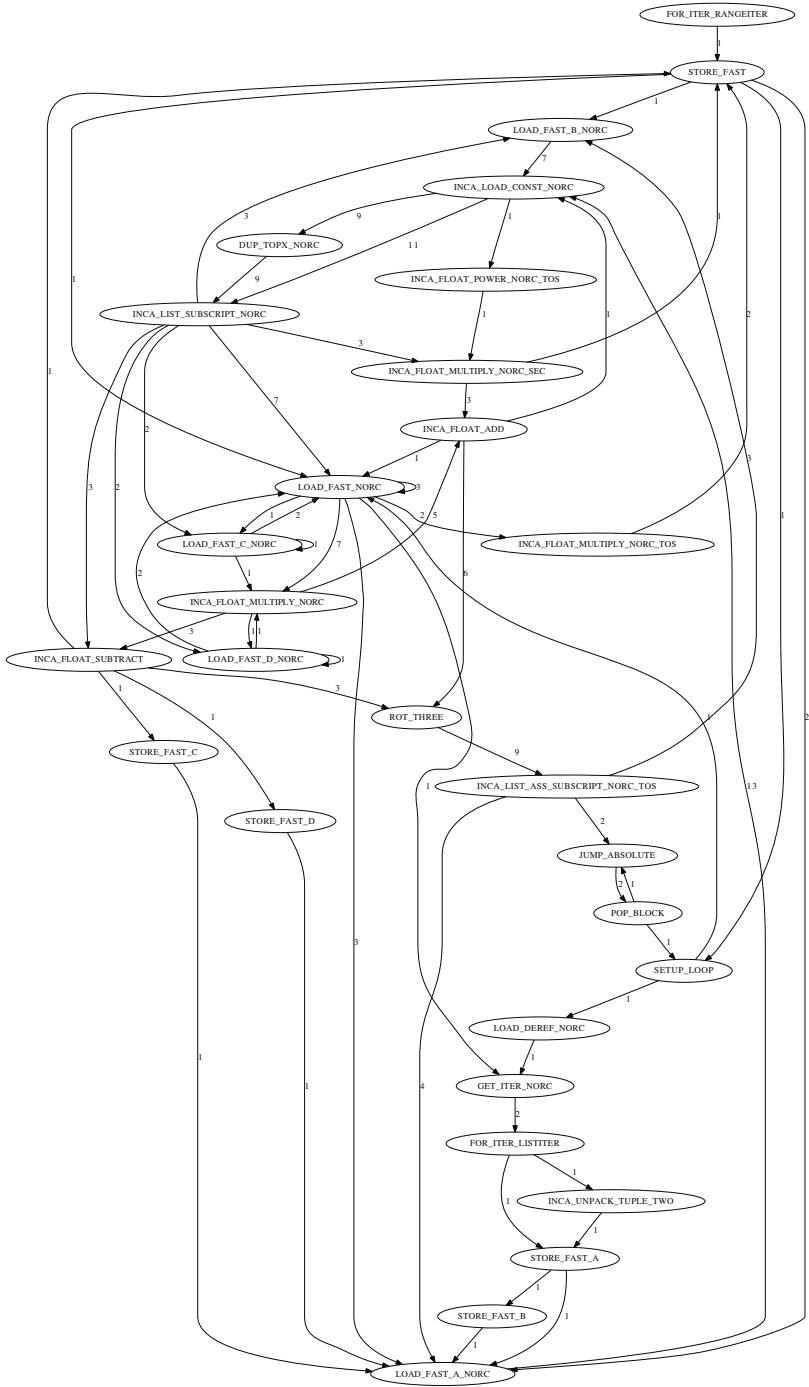


Fig. 1. Instructions of kernel for nbody benchmark

The following listing shows our implementation in Python, followed by a detailed description of how it works:

```
def rsorted(dict):
    """Sorts dictionary entries by their numeric values in
    descending order.
    """
    return sorted(dict.items(), key=lambda (key, value): -value)

def schedule_instr(graph):
    schedule= []

    open= rsorted(graph.most_frequent_vertices())
    ## open is a list of tuples (node, number of edges)
    open= [ (node, 0) for (node, edge_count) in open ]
    ## now, we erased the number of edges, such that when we add
    ## the reachable destination nodes for the current source
    ## node, and we sort the <open> list, the node that can be
    ## reached with the edge having the highest weight will be
    ## the first element on the <open> list

    while open:
        ## fetch the tuple, ignore the number of edges
        (n, _)= open.pop(0)
        while n:
            if n not in schedule:
                schedule.append( n )

            reachable= rsorted(n.get_destinations())
            if not reachable:
                break

            ## find reachable nodes that have not been scheduled yet
            (m, _)= reachable.pop(0)
            while m in schedule:
                if len(reachable) > 0:
                    (m, _)= reachable.pop(0)
                else:
                    m= None

            if m:
                n= m
                open= rsorted( reachable + open ) ## assign successor node
                ## keep reachable nodes sorted
            else:
                break

    return schedule
```

Running this algorithm on our kernel for the `nbody` benchmark computes the schedule presented in Table 3.

3.2 Compilation of the Interpreter

Once we have computed a schedule of interpreter instructions, we need to compile the interpreter with that schedule. We have extended our interpreter generator from our previous work ([3]) to generate all instructions, not just the optimized derivatives. Since we already have a schedule, it is straightforward to generate an optimized instruction set from the standard instruction set. We just process the schedule in order, move instructions from the old instruction set, and add these instructions to the optimized instruction set. Once, we have processed the plan, we just add the remaining instructions to the new optimized instruction set.

Table 3. Interpreter Instruction Schedule for the `nbody` benchmark

No.	Instruction	No.	Instruction
1	INCA_LOAD_CONST_NORC	16	LOAD_DEREF_NORC
2	INCA_LIST_SUBSCRIPT_NORC	17	GET_ITER_NORC
3	LOAD_FAST_NORC	18	FOR_ITER_LISTITER
4	INCA_FLOAT_MULTIPLY_NORC	19	STORE_FAST_A
5	INCA_FLOAT_ADD	20	STORE_FAST_B
6	ROT_THREE	21	STORE_FAST_D
7	INCA_LIST_ASS_SUBSCRIPT_NORC_TOS	22	INCA_FLOAT_MULTIPLY_NORC_SEC
8	LOAD_FAST_A_NORC	23	JUMP_ABSOLUTE
9	DUP_TOPX_NORC	24	POP_BLOCK
10	LOAD_FAST_B_NORC	25	LOAD_FAST_C_NORC
11	INCA_FLOAT_SUBTRACT	26	LOAD_FAST_D_NORC
12	STORE_FAST_C	27	INCA_UNPACK_TUPLE_TWO
13	INCA_FLOAT_MULTIPLY_NORC_TOS	28	INCA_FLOAT_POWER_NORC_TOS
14	STORE_FAST	29	FOR_ITER_RANGEITER
15	SETUP_LOOP		

There are compiler optimizations that can change the instruction order as computed by our interpreter instruction scheduling. First of all, basic block re-ordering as done by `gcc 4.4.3` will eliminate our efforts by reordering basic blocks after a strategy called “software trace-cache” [10]. Fortunately, we can switch this optimization off, by compiling the source file that contains the interpreter dispatch routine with the additional flag `-fno-reorder-blocks`. However, the instructions are still entangled in a switch-case statement. Since it is possible for a compiler to re-arrange case statements, we decided to remove the switch-case statement from the interpreter dispatch routine as well. Because our interpreter is already using the optimized threaded code dispatch technique [2], removing the switch-case statement is simple. However, we stumbled upon a minor mishap: `gcc 4.4.3` now decides to generate two jumps for every instruction dispatch. Because the actual instruction-dispatch indirect-branch instruction is shared by all interpreter instruction implementations, available expression analysis indicates that it is probably best to generate a direct jump instruction back to the top of the dispatch loop, directly followed by an indirect branch to the next instruction. On an Intel Nehalem (i7-920), `gcc 4.4.3` generates the following code at the top of the dispatch loop:

```
.L1026:
    xorl %eax, %eax
.L1023:
    jmp  *%rdx
```

And a branch back to the label `.L1026` at the end of every instruction:

```
movq opcode_targets.14198(,%rax,8), %rdx
jmp  .L1026
```

Of course, this has detrimental effects on the performance of our interpreter. Therefore, we use `gcc`’s `-save-temps` switch while compiling the interpreter

routine with `-fno-reorder-blocks` to retrieve the final assembler code emitted by the compiler. We implemented a small fix-up program that rebuilds the basic blocks and indexes their labels from the interpreter’s dispatch routine (`PyEval_EvalFrameEx`), determines if jumps are transitive, i.e., to some basic-block that itself contains only a jump instruction, and copies the intermediate block over the initial jump instruction. Thus, by using this fix-up program, we obtain the original threaded-code jump sequence:

```
movq opcode_targets.14198(,%rax,8), %rdx
xorl %eax, %eax
jmp  *%rdx
```

Finally, we need to assemble the fixed-up file into the corresponding object file and link it into the interpreter executable.

4 Evaluation

We use several benchmarks from the computer language benchmarks game [6]. We ran the benchmarks on the following system configurations:

- Intel i7 920 with 2.6 GHz, running Linux 2.6.32-25 and gcc version 4.4.3. (Please note that we have turned off Intel’s Turbo Boost Technology to have a common hardware baseline performance without the additional variances immanently introduced by it [7].)
- Intel Atom N270 with 1.6 GHz, running Linux 2.6.28-18 and gcc version 4.3.3.

We used a modified version of the `nanobench` program of the computer language benchmark game [6] to measure the running times of each benchmark program. The `nanobench` program uses the UNIX `getrusage` system call to collect usage data, for example the elapsed user and system times as well as memory usage of a process. We use the sum of both timing results, i.e., elapsed user and system time as the basis for our benchmarks. Because of cache effects, and unstable timing results for benchmarks with only little running time, we ran each program 50 successive times and use arithmetic averages over these repetitions for our evaluation. Furthermore, we compare our improvements to the performance of our most recent interpreter without interpreter instruction selection [3].

Figure 2 contains our results of running comparative benchmarks on the Intel Nehalem architecture. For each of our benchmarks, we generate a dedicated interpreter that has an optimized interpreter instruction schedule based on the profiling information obtained by running this benchmark program. We normalized our results by those of our previous work [3], such that we can tell whether interpreter instruction scheduling improves performance of an interpreter with an extended instruction set (our interpreter has 394 instructions). Similarly, Figure 3 contains our results of running this comparative setup on our Intel Atom CPU based system. First, let’s discuss the results we obtained on the Intel Atom system (cf. Figure 3). We obtained the maximum speedup by a factor of 1.1344 when running the `spectralnorm` benchmark, the minimum speedup by a factor

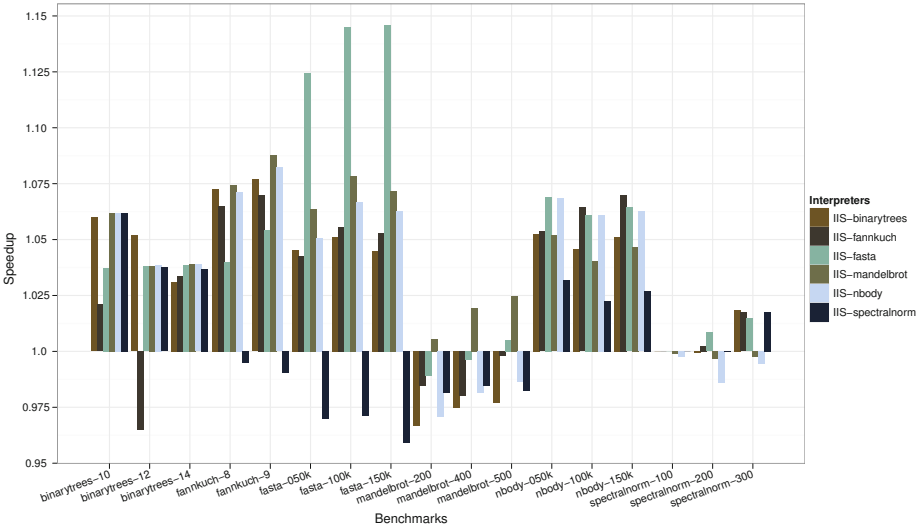


Fig. 2. Comparative benchmark results on the Intel Nehalem CPU

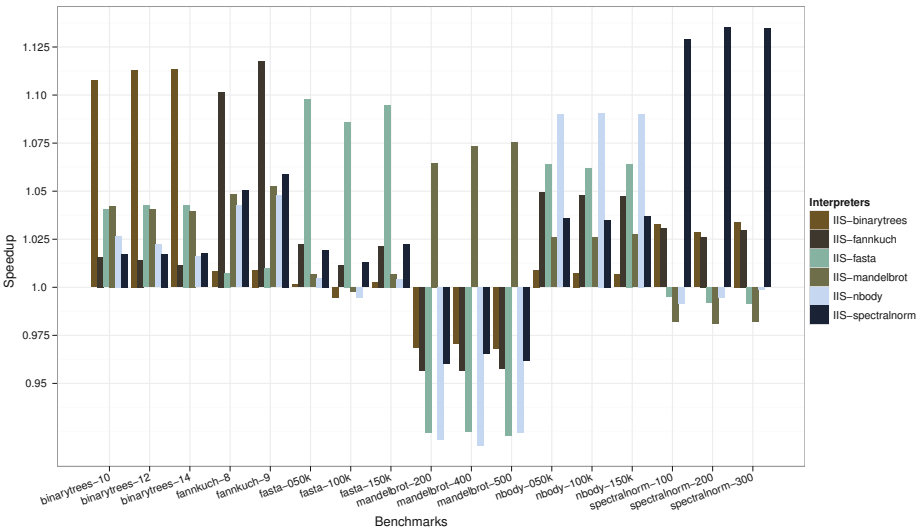


Fig. 3. Comparative benchmark results on the Intel Atom CPU

of 1.0736 when running the `mandelbrot` benchmark, and an average speedup by a factor of 1.1032. The figure clearly indicates that for every benchmark, the interpreter with the instruction scheduling corresponding to that benchmark achieves the highest speedup. Interestingly, most instruction schedules perform better on most benchmarks, with the notable exception being the `mandelbrot` benchmark—a finding that holds true for our results on the Intel Nehalem, too.

Our results on the Intel Nehalem architecture paint a different picture. While the maximum speedup by a factor of 1.142 is higher than the one we report for the Atom CPU, its average speedup by a factor of 1.061 is lower. It is reasonable to assume that this is due to the Nehalem architecture having bigger caches, which affects the performance potential of interpreter instruction scheduling. In addition, there are only two benchmarks, viz. `fasta` and `mandelbrot`, where the interpreter having an optimized instruction schedule for the corresponding benchmark actually perform better than the others. For all other benchmarks, the computed instruction schedule given the profiling information is not optimal, i.e., the schedules computed for some other benchmark allow some of the other interpreters to perform noticeably better. Further investigation is necessary, to identify the cause of this rather surprising finding—particularly in presence of the actually expected findings confirmed on the Atom CPU.

5 Related Work

We group the discussion of related work into two groups, viz., related work on compilers and related work on interpreters. First, we will discuss the related work on compilers.

Pettis and Hansen [9] present their work on optimizing compilers for the Hewlett Packard’s PA-RISC architecture. They optimize the arrangement of procedures and basic blocks based on previously obtained profiling information. Interestingly, our reordering algorithm is almost identical to their “`algo1`” algorithm; they may even be identical, but because no implementation is given, this remains unclear. Another interesting fact is that both our maximum achievable speedups are identical, i.e., both our work achieves a maximum speedup by a factor of 1.14.

More recently, Zhao and Amaral [11] demonstrate algorithms to optimize switch-case computation as well as case-statement ordering in the Open Research Compiler [1]. While both our approaches employ information gathered at run-time, the application scenario is quite different. For instance, their approach focuses on optimizing switch-case statements, and they calculate the order in which they should be generated by their rank according to frequency. In contrast, our work focuses on optimization of interpreters, particularly those without using the switch-case dispatch technique. Because of better instruction cache utilization, we choose to use another algorithm that recognizes the importance of properly covering instruction sequences. So in a sense, the major difference is that their optimization approach focuses on larger compiled programs that use switch-case statements, whereas we recognize the nature of an interpreter, where execution remains within its instruction set *at all times*. Another direct consequence of this fundamental difference is that in an interpreter we are usually not interested in the default case, since this indicates an error, i.e., an unknown opcode, which in practice happens never—the exception being malicious intent of a third party.

As for related work on interpreters, the most important work is by Lin and Chen [8]. Their work is similar to ours, since they show how to partition interpreter instructions to optimally fit into NAND flash pages. Furthermore, they describe that they too use profiling information to decide which combination of interpreter instructions to co-locate on one specific flash page. Their partitioning algorithm pays attention to the additional constraint of NAND flash page size, i.e., their algorithm computes a configuration of interpreter instructions that fits optimally within the flash pages and keeps dependencies between the pages at a minimum. For the context of our work it is unnecessary to superimpose such a constraint to our algorithm. Though, if one were to set the parameter N determining the NAND flash page size of their algorithm to the maximum representable value, all instructions would be packed into just one partition. Then, our algorithms should produce similar interpreter instruction arrangements. Another difference between our respective approaches is that ours operates on a higher level. While they post-process the assembly output generated by `gcc` to enable their optimizations, our approach is based on re-arranging the instruction at the source code level. Though we admittedly have to fix-up the generated assembly file as well, due to the detrimental effects of a misguided optimization. Because of their ties to embedded applications of the technique and its presentation in that context, we think that our presentation is more general in nature. In addition, we complement our work with extensive performance measurements on contemporary non-embedded architectures.

Ertl and Gregg [5] present an in-depth discussion of two interpreter optimization techniques—superinstructions and replication—to improve the branch prediction accuracy and instruction cache utilization of virtual machines. While the optimization technique of replication is not directly related to interpreter instruction scheduling, it improves the instruction cache behavior of an interpreter at the expense of additional memory. The idea of superinstructions is to combine several interpreter instructions into one superinstruction, thus eliminating the instruction dispatch overhead between the single constituents. While this improves branch prediction accuracy, it improves the instruction cache utilization, too: Since all instruction implementations must be copied into one superinstruction, their implementations must be adjacent, i.e., co-located in memory, which is optimal with respect to instruction cache utilization and therefore results in extremely good speedups of up to 2.45 over a threaded-code interpreter without superinstructions. However, superinstructions can only be used at the expense of additional memory, too. Since interpreter instruction scheduling happens at pre-compile, and compile time respectively, of the interpreter, there are no additional memory requirements—with the notable exception of minor changes because of alignment issues. Because the techniques are not mutually exclusive, using interpreter instruction scheduling in combination with static superinstructions will further improve the performance of the resulting interpreter.

Summing up, the major difference between the related work on compilers and our work is that the former focuses on optimizing elements visible to the compiler, such as procedures, basic blocks, and switch-case statements, whereas our

work focuses on re-arranging interpreter instructions—which are transparent to compilers. Related work on interpreters achieves a significantly higher speedup, however, at the expense of additional memory. Our work demonstrates that is possible to improve interpretation speed without sacrificing memory.

6 Conclusion

We present a technique to schedule interpreter instructions at pre-compile time in order to improve instruction cache utilization at run-time. To compute the schedule, we rely on profiling information obtained by running the program to be optimized on an interpreter. From this information, we extract a kernel, i.e., an instruction trace that consumes most of the computational resources, and construct a directed graph of that kernel. We use a simple algorithm that recognizes the importance of repeating sub-sequences occurring in that kernel when scheduling the interpreter instructions, and report a maximum speedup by a factor of 1.142 using this technique.

Future work includes investigation on the effectiveness of other scheduling algorithms—such as implementation of a dynamic programming variant, or comparing the effectiveness of algorithms mentioned in the related work section—, as well as addressing the pending question regarding the optimality of computed schedules. In addition, we are interested in devising a dynamic variant complementing our static interpreter instruction scheduling technique.

Acknowledgments

We express our thanks to Jens Knoop for reviewing early drafts of this paper, as well as to Anton Ertl for providing extensive information concerning related work. Furthermore, we are grateful to the anonymous reviewers for providing helpful remarks to improve the presentation of this paper, and making valuable suggestions for addressing future work. Finally, we thank the Austrian Science Fund (FWF) for partially funding this research under contract number P23303-N23.

References

1. Open Research Compiler (October 2010), <http://ipf-orc.sourceforge.net/>
2. Bell, J.R.: Threaded code. *Communications of the ACM* 16(6), 370–372 (1973)
3. Brunthaler, S.: Efficient interpretation using quickening. In: *Proceedings of the 6th Symposium on Dynamic Languages (DLS 2010)*, Reno, Nevada, US, October 18, ACM Press, New York (2010)
4. Brunthaler, S.: Inline caching meets quickening. In: D’Hondt, T. (ed.) *ECOOP 2010*. LNCS, vol. 6183, pp. 429–451. Springer, Heidelberg (2010)
5. Ertl, M.A., Gregg, D.: Optimizing indirect branch prediction accuracy in virtual machine interpreters. In: *Proceedings of the SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI 2003)*, pp. 278–288. ACM, New York (2003)

6. Fulgham, B.: The computer language benchmarks game, <http://shootout.alioth.debian.org/>
7. Intel: Intel Turbo Boost Technology in Intel Core microarchitecture (Nehalem) based processors. Online (November 2008), http://download.intel.com/design/processor/applnotes/320354.pdf?iid=tech%_tb+paper
8. Lin, C.-C., Chen, C.-L.: Code arrangement of embedded java virtual machine for NAND flash memory. In: Stenström, P., Dubois, M., Katevenis, M., Gupta, R., Ungerer, T. (eds.) HiPEAC 2007. LNCS, vol. 4917, pp. 369–383. Springer, Heidelberg (2008)
9. Pettis, K., Hansen, R.C.: Profile guided code positioning. SIGPLAN Notices 25(6), 16–27 (1990)
10. Ramírez, A., Larriba-Pey, J.L., Navarro, C., Torrellas, J., Valero, M.: Software trace cache. In: Proceedings of the 13th International Conference on Supercomputing (ICS 1999), Rhodes, Greece, June 20–25, pp. 119–126. ACM, New York (1999)
11. Zhao, P., Amaral, J.N.: Feedback-directed switch-case statement optimization. In: Proceedings of the International Conference on Parallel Programming Workshops (ICPP 2005 Workshops), Oslo, Norway, June 14–17, pp. 295–302. IEEE, Los Alamitos (2005)

Actor-Based Parallel Dataflow Analysis

Jonathan Rodriguez and Ondřej Lhoták

University of Waterloo,
Waterloo, Ontario, Canada
{j2rodrig, olhotak}@uwaterloo.ca

Abstract. Defining algorithms in a way which allows parallel execution is becoming increasingly important as multicore computers become ubiquitous. We present IFDS-A, a parallel algorithm for solving context-sensitive interprocedural finite distributive subset (IFDS) dataflow problems. IFDS-A defines these problems in terms of Actors, and dataflow dependencies as messages passed between these Actors. We implement the algorithm in Scala, and evaluate its performance against a comparable sequential algorithm. With eight cores, IFDS-A is 6.12 times as fast as with one core, and 3.35 times as fast as a baseline sequential algorithm. We also found that Scala's default Actors implementation is not optimal for this algorithm, and that a custom-built implementation outperforms it by a significant margin. We conclude that Actors are an effective way to parallelize this type of algorithm.

Keywords: Actors, compilers, concurrency, dataflow analysis, IFDS, Scala.

1 Introduction

Multi-core CPU architectures are becoming increasingly common in all types of computer hardware, even now in low-end consumer devices. Learning to use these additional cores is a necessary step in developing more capable software. If compilers and program analysis tools could benefit from the additional computation power available in multi-core computers, then an increase in the precision of these tools could be accomplished without compromising speed.

In this paper, we present an algorithm for solving context-sensitive IFDS (Interprocedural Finite Distributive Subset) dataflow analysis problems [14] in a way which takes advantage of any additional CPU cores which may be present. Constructing this type of algorithm using traditional thread-and-lock expressions can be a difficult exercise because it requires reasoning about shared data consistency in the presence of non-deterministic thread interleavings, reasoning which is extraordinarily difficult for human minds [9, 16]. We approach the task by expressing the algorithm using the Actor model [1, 6]. The Actor model has no notion of shared variables. Instead, each actor maintains a local state only, and communicates by passing messages to other actors. As far as we are aware, this is the first implementation of IFDS which uses a message-passing model to communicate changes in state.

Like IFDS, many other dataflow analysis algorithms use a worklist to iterate to a fixed-point solution, and therefore have the same general structures as IFDS. Although

we do not explore other dataflow analysis algorithms here, we expect the actor-based approach to parallelization to work well for many of them.

This paper is based on previous thesis work with IFDS and Scala’s Actor library [15], and we extend it here to include an alternative implementation of the runtime Actor scheduler which supports priority ordering of messages passed.

Section 3 summarizes the nature of IFDS problems and their solution. The single-threaded E-IFDS algorithm, which contains several practical extensions to the original IFDS algorithm, is presented here. Section 4 summarizes the Actor model and its semantics. Section 5 introduces the IFDS-Actors, or IFDS-A, algorithm. Section 6 discusses implementation details, including an Actor scheduler implementation which supports priority ordering of messages. Section 7 contains an empirical evaluation of the performance of IFDS-A and compares it to E-IFDS.

2 Related Work

The IFDS algorithm was originally presented by Reps, Horwitz, and Sagiv as a precise and efficient solution to a wide class of context-sensitive, interprocedural dataflow analysis problems [14]. The Extended IFDS algorithm formalized a set of extensions to the IFDS algorithm which increased its utility for a wider set of analysis problems [11]. The E-IFDS algorithm we present in this paper is essentially Extended IFDS with some minor differences.

Adapting analysis algorithms to operate using multiple CPU cores may lead to significantly improved performance of these algorithms. The Galois system approaches this problem by providing new syntactic constructs which enable thread-safe parallel iteration over unordered sets [7, 8]. Méndez-Lojo, Matthew, and Pingali [10] used the Galois system to implement a multi-core version of a points-to analysis algorithm by Hardekopf and Lin [5]. Using an eight-core computer, they were able to show performance improvements over Hardekopf and Lin for analyses which took longer than 0.5 seconds to run.

A second approach to creating multi-core analysis algorithms is to express them in terms of the Actor Model, which describes computations as sets of logical processes which communicate via message-passing [1, 6]. The Erlang language [19] has built-in support for the Actor model, and Scala [12] includes an implementation of the Actor model in its standard library [4]. Panwar, Kim, and Agha studied the application of the Actor model to graph-based algorithms, and in particular tested varying strategies of work distribution among CPU cores [13].

3 Baseline Sequential Algorithm: E-IFDS

The E-IFDS algorithm is a sequential dataflow analysis algorithm which extends the IFDS algorithm of Reps, Horwitz, and Sagiv [14]. We briefly explain the IFDS dataflow analysis problem, followed by a presentation of E-IFDS.

Dataflow analysis seeks to determine, for each instruction in an input program, facts which must hold during execution of that instruction. The types of facts which the

analysis discovers depend on the type of analysis used. For example, an uninitialized-variables analysis discovers facts of the form “ x is uninitialized at this instruction,” and a variable-type analysis discovers facts of the form “the type of the object x points to is a subtype of T .”

A *control-flow graph (CFG)* describes the structure of the input program. Each node in the CFG represents an instruction. A directed edge from instruction a to b indicates that b may execute immediately after a .

A *flow function* models the effect of each type of instruction in the input program. A flow function takes a set of facts as input, and it computes a new set of facts as output. Whenever the dataflow analysis discovers new facts holding after an instruction, it propagates the new facts along the edges in the CFG to the successors of the instruction.

```

algorithm Solve( $N^*$ ,  $s_{main}$ , successors, flow)
begin
[1]   ResultSet := {  $\langle s_{main}, \mathbf{0} \rangle$  }
[2]   WorkList := {  $\langle s_{main}, \mathbf{0} \rangle$  }
[3]   while WorkList  $\neq \emptyset$  do
[4]     Remove any element  $\langle n, d \rangle$  from WorkList
[5]     for each  $d' \in \text{flow}(n, d)$  and  $n' \in \text{successors}(n)$  do
[6]       Propagate( $\langle n', d' \rangle$ )
[7]     od
[8]   od
[9]   return ResultSet
end

procedure Propagate( $item$ )
begin
[10] if  $item \notin \text{ResultSet}$  then Insert  $item$  into ResultSet; Insert  $item$  into WorkList fi
end

```

Algorithm 1. A Naive Algorithm for Solving IFDS Problems

IFDS, or Interprocedural Finite Distributive Subset, problems are dataflow analysis problems with the following properties.

- The analysis is *interprocedural* in that it takes the effects of called procedures into account.
- Each instruction is associated with a *finite* set of facts, and each such set is a *subset* of a larger finite fact set D .
- At control flow merge points, the sets of facts coming from different control flow predecessors are combined using set union.
- The flow functions are *distributive*, i.e. for any two fact sets D_1 and D_2 , and any flow function f , $f(D_1 \cup D_2) = f(D_1) \cup f(D_2)$. The distributive property enables flow functions to be compactly represented and efficiently composed.

The distributivity of the flow function makes it possible for an analysis to evaluate each transfer function f one fact at a time. For example, consider the input set of facts $D_I = \{a, b, c\}$. This set can be written as the union $\{\} \cup \{a\} \cup \{b\} \cup \{c\}$. Therefore, the result of the transfer function $f(D_I)$ can be computed as $f(\{\}) \cup f(\{a\}) \cup f(\{b\}) \cup f(\{c\})$. In general, the transfer function can be computed for any input sets by taking the union of the results of applying the transfer function to the empty set and to singleton sets.

Algorithm 1 is a simple algorithm that finds the merge over all paths solution of an IFDS problem. Its inputs are N^* , the set of nodes in the control-flow graph; s_{main} , the entry point of the main procedure; successors, which maps a node in N^* to its control-flow successors in N^* ; and flow, the flow function. The flow function takes two parameters n , a node in N^* , and d , a fact in the set $D \cup \{\mathbf{0}\}$. A value of $d \in D$ represents the singleton set $\{d\}$, and $d = \mathbf{0}$ represents the empty set. The flow function evaluates the transfer function for the given node n on the singleton or empty set, and returns a set of facts to be propagated to successor nodes in the CFG.

The ResultSet collects all facts along with the nodes at which they were discovered. The first element put into the ResultSet is $\langle s_{main}, \mathbf{0} \rangle$, indicating that the empty set of facts reaches the beginning of the program. Every time the algorithm discovers a new fact reaching a node, it accumulates it in ResultSet and adds it into the WorkList. Elements from the WorkList may be removed and processed in any order. Whenever an element is removed, the flow function is evaluated on it and any newly generated facts are added to the ResultSet and the WorkList. When the WorkList is empty, no additional facts can be derived, so the algorithm terminates.

The actual IFDS algorithm [14] is more precise in that it computes the merge over all valid paths solution rather than the merge over all paths. A valid path is a path through the interprocedural control flow graph in which calls and returns are matched: the control flow edge taken to return from a procedure must lead to the point of the most recent call of that procedure. Here, we present and parallelize E-IFDS, a variation of the Extended IFDS algorithm [11], which in turn is an extension of the original IFDS algorithm [14]. The full E-IFDS algorithm is given in Algorithm 2.

The differences between E-IFDS and Extended IFDS are:

- The Extended IFDS algorithm maintains a SummaryEdge set, whereas E-IFDS does not.
- The Extended IFDS algorithm explicitly supports the Static Single Assignment form, or SSA, without loss of precision. E-IFDS does not make any explicit provisions for SSA [1].
- E-IFDS explicitly allows multiple called procedures at a single call-site, whereas the Extended IFDS algorithm does not.

The key idea that enables the IFDS class of algorithms to compute a solution over only valid paths is that they accumulate *path-edges* of the form $d_1 \rightarrow \langle n, d_2 \rangle$ rather than just facts of the form $\langle n, d \rangle$. Instead of representing a fact, a path-edge represents a function: the path-edge $d_1 \rightarrow \langle n, d_2 \rangle$ means that if the fact d_1 is true at the beginning of the procedure containing n , then the fact d_2 is true at n . Given a node n , the set of path-edges terminating at n defines the function:

$$f(D_{in}) = \{d_2 : d_1 \in D_{in} \cup \{\mathbf{0}\} \text{ and } d_1 \rightarrow \langle n, d_2 \rangle \in \text{path-edges}\}$$

Thus the path-edges accumulated at the return of a procedure define a function that computes the facts holding after the procedure from a given set of facts holding before that specific call of the procedure.

¹ Adding SSA support is largely just a matter of propagating predecessor nodes along with the path-edges so that the Phi nodes know which branch a given fact came from.

```

algorithm Solve( $N^*$ ,  $s_{main}$ , successors, flowi, flowcall, flowret, flowthru)
begin
[1] PathEdge := {  $\mathbf{0} \rightarrow \langle s_{main}, \mathbf{0} \rangle$  }
[2] WorkList := {  $\mathbf{0} \rightarrow \langle s_{main}, \mathbf{0} \rangle$  }
[3] CallEdgeInverse :=  $\emptyset$ 
[4] ForwardTabulate()
[5] return all distinct  $\langle n, d_2 \rangle$  where some  $d_1 \rightarrow \langle n, d_2 \rangle \in \text{PathEdge}$ 
end

procedure Propagate(item)
begin
[6] if item  $\notin$  PathEdge then Insert item into PathEdge; Insert item into WorkList fi
end

procedure ForwardTabulate()
begin
[7] while WorkList  $\neq \emptyset$  do
[8]   Remove any element  $d_1 \rightarrow \langle n, d_2 \rangle$  from WorkList
[9]   switch n
[10]    case n is a Call Site :
[11]     for each  $d_3 \in \text{flow}_{call}(n, d_2, p)$  where  $p \in \text{calledProcs}(n)$  do
[12]       Propagate( $d_3 \rightarrow \langle s_p, d_3 \rangle$ )
[13]       Insert  $\langle p, d_3 \rightarrow \langle n, d_2 \rangle$  into CallEdgeInverse
[14]       for each  $d_4$  such that  $d_3 \rightarrow \langle e_p, d_4 \rangle \in \text{PathEdge}$  do
[15]         for each  $d_5 \in \text{flow}_{ret}(e_p, d_4, n, d_2)$  do
[16]           Propagate( $d_1 \rightarrow \langle \text{returnSite}(n), d_5 \rangle$ )
[17]         od
[18]       od
[19]     od
[20]     for each  $d_3 \in \text{flow}_{thru}(n, d_2)$  do
[21]       Propagate( $d_1 \rightarrow \langle \text{returnSite}(n), d_3 \rangle$ )
[22]     od
[23]   end case
[24]   case n is the Exit node  $e_p$  :
[25]     for each  $\langle c, d_4 \rangle$  such that  $\langle p, d_1 \rightarrow \langle c, d_4 \rangle \rangle \in \text{CallEdgeInverse}$  do
[26]       for each  $d_5 \in \text{flow}_{ret}(e_p, d_2, c, d_4)$  do
[27]         for each  $d_3$  such that  $d_3 \rightarrow \langle c, d_4 \rangle \in \text{PathEdge}$  do
[28]           Propagate( $d_3 \rightarrow \langle \text{returnSite}(c), d_5 \rangle$ )
[29]         od
[30]       od
[31]     od
[32]   end case
[33]   case n is not a Call Site or Exit node :
[34]     for each  $d_3 \in \text{flow}_i(n, d_2)$  and  $n' \in \text{successors}(n)$  do
[35]       Propagate( $d_1 \rightarrow \langle n', d_3 \rangle$ )
[36]     od
[37]   end case
[38] end switch
[39] od
end

```

Algorithm 2. The E-IFDS Algorithm

Intra-procedurally, the algorithm generates new path-edges by composing existing path-edges with the flow function. If a path-edge $d_1 \rightarrow \langle n, d_2 \rangle$ exists and n' is a control flow successor of n , then for each $d_3 \in \text{flow}(n, d_2)$, a new path-edge $d_1 \rightarrow \langle n', d_3 \rangle$ is created (lines 33–37). Thus the computed path-edges represent the transitive closure of the flow function within each procedure.

The flow function for E-IFDS is separated into four functions for convenience:

- $\text{flow}_i(n, d)$: Returns the facts derivable from d at instruction node n .
- $\text{flow}_{call}(n, d, p)$: Computes call-flow edges from the call-site n to the start of a called procedure p .
- $\text{flow}_{ret}(n, d, c, d_c)$: Computes return-flow edges from the exit node n to the return-site. The fact d_c at the call-site c is the *caller context* required by some analyses; $\langle n, d \rangle$ is reachable from $\langle c, d_c \rangle$.
- $\text{flow}_{thru}(n, d)$: A convenience function which allows transmission of facts from call-site to return-site without having to propagate through called procedures.

When a call is encountered, the algorithm creates *summary edges* which summarize the effect of a procedure from the caller’s point of view. Each summary edge is the composition of a call-flow edge, a path-edge of the called procedure, and a return-flow edge. The summary edges are then composed with existing path-edges that lead to the call site to create new path-edges that lead to the return site. Lines 14–18 and lines 25–31 compute summary edges and propagate the corresponding new path-edges. Figure 1 illustrates the relationships between the CFG (left), dataflow edges determined by flow functions (right, solid lines), and dataflow edges computed by the algorithm (right, dashed lines). Whenever a summary edge is generated, E-IFDS uses it to generate new path-edges, as if the summary edge were an ordinary dataflow edge. Unlike the original IFDS algorithm, E-IFDS does not need to keep a set of all summary edges; it discards each summary edge immediately after using it to extend a path-edge.

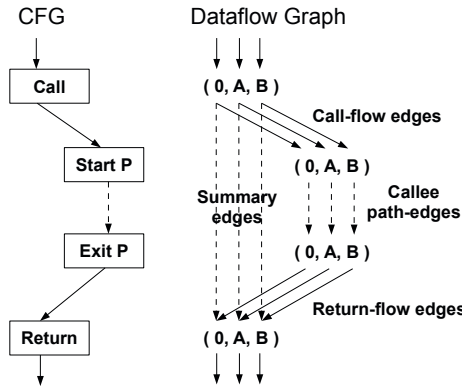


Fig. 1. Generating Call-Site Summary Edges for E-IFDS

4 The Actor Model

The basic notion behind the Actor model is that any computation can be defined as a set of entities, or *actors*, which communicate by passing messages. Each actor processes the messages it receives in some sequential order. The actor buffers received messages until it can process them, as shown in Figure 2. The theory behind this model was originally developed by Hewitt [6] and the semantics of actors were refined by a number of others, notably Agha [1].

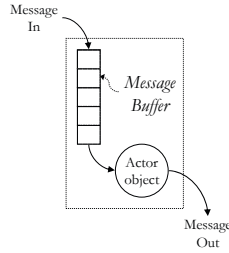


Fig. 2. The Actor Abstraction

The main difference between actor-based programming and object-oriented programming is that actor-based message-passing is *asynchronous* and *unordered*, whereas in many object-oriented systems method calls are by default synchronous and ordered. The sending actor does not wait for the receiving actor to process the message, and the time between the send and the receive may be arbitrarily long.

The notation used to define actors is shown in Figure 3. An actor definition is called an *actor class* to distinguish it from an ordinary class. This notation contains the following components: a name, a list of arguments, a statement block $stmts_{init}$ which executes upon actor construction, and a set of cases which are matched against incoming messages. A match succeeds if the message type and the number of message parameters is the same as the pattern. For example, the message $AddEdge\langle\text{“A”}, \text{“B”}\rangle$ matches the pattern $AddEdge\langle d_1, d_2 \rangle$. When the match succeeds, the values of d_1 and d_2 are “A” and “B”, respectively. Scala’s pattern matching semantics select the first pattern that matches; subsequent patterns are not tested. Actor classes may also contain a **finally** clause, which is executed immediately after the execution of any case statement.

```

def ActorName(arguments)
  stmtsinit
begin (message) switch
  case message matches pattern1 : stmts1
  ...
  case message matches patternn : stmtsn
  finally : stmtsfinal
end

```

Fig. 3. Actor Class Definition

All argument variables and all local variables created by $stmts_{init}$ persist for the lifetime of the actor and are visible to all statements $stmts_1$ through $stmts_n$ and $stmts_{final}$. These variables are analogous to member variables in object-oriented languages; they persist until the actor is garbage-collected. Any variables created by $stmts_1$ through $stmts_n$ or $stmts_{final}$ are only live and visible inside their respective statement blocks. These variables are created in response to a received message and so do not persist after the message is processed.

² As we will show later, however, applying a message prioritization policy can result in performance improvements.

When the actor receives a message, it selects at most one case statement for processing. Local variables created by the statements following **case** are stored in a temporary frame that is discarded when those statements finish executing.

5 Actor-Based Parallel Algorithm: IFDS-A

The IFDS-Actors algorithm, or IFDS-A, takes the same parameters and produces the same results as E-IFDS, but takes advantage of additional CPU cores. Algorithm 3 shows the main IFDS-A algorithm. Algorithm 4 defines the actors that respond to path-edge propagation.

```

algorithm Solve( $N^*$ ,  $s_{main}$ , successors, flowi, flowcall, flowret, flowthru)
begin
[1]   for each  $n \in N^*$  do switch
[2]     case  $n$  is a Call Site :  $N^A[n] := \text{new CallSiteActor}(n)$  end case
[3]     case  $n$  is the Exit node  $e_p$  :  $N^A[n] := \text{new ExitActor}(p)$  end case
[4]     case  $n$  is not a Call Site or Exit node :  $N^A[n] := \text{new IntraActor}(n)$  end case
[5]   end switch od
[6]   Tracker := new TrackerActor(currentThread)
[7]   Propagate( $s_{main}$ , AddPathEdge( $\mathbf{0}$ ,  $\mathbf{0}$ ))
[8]   Wait for Done( $\langle \rangle$ )
[9]   return all distinct  $\langle n, d_2 \rangle$  where some  $d_1 \rightarrow d_2 \in N^A[n].\text{PathEdge}$ 
end

pure function Propagate( $n$ , message)
begin
[10]  Send synchronous Inc( $\langle \rangle$ ) to Tracker
[11]  Send message to  $N^A[n]$ 
end

def TrackerActor(receiver)
[12]  local count := 0
begin (message) switch
[13]  case message matches Inc( $\langle \rangle$ ) :
[14]    count := count + 1
[15]  case message matches Dec( $\langle \rangle$ ) :
[16]    count := count - 1
[17]    if count = 0 then Send Done( $\langle \rangle$ ) to receiver fi
end

```

Algorithm 3. The Top-Level IFDS-A Algorithm

IFDS-A is based on a simple conceptual mapping. IFDS-A constructs one actor for each node in the CFG. For each propagated path-edge, IFDS-A sends a message. The algorithm does not need a centralized WorkList because the actor library implicitly buffers all messages until they are processed. Instead of a centralized PathEdge set, each actor records a local set of all the path-edges leading into it. Where E-IFDS stores a path-edge $d_1 \rightarrow \langle n, d_2 \rangle$, the IFDS-A node-actor corresponding to CFG node n , or $N^A[n]$, simply stores $d_1 \rightarrow d_2$.

IFDS-A defines three types of node-actors, corresponding to call-site nodes, exit nodes, and other instruction nodes. Lines 1–5 in Algorithm 3 create the node-actors, and Algorithm 4 defines the node-actor classes.

```

def CallSiteActor( $n$ )
[1]   local PathEdge :=  $\emptyset$ 
[2]   local CallEdge :=  $\emptyset$ 
[3]   local CalleePathEdge :=  $\emptyset$ 
begin ( $message$ ) switch
[4]   case  $message$  matches AddPathEdge( $d_1, d_2$ ) :
[5]     if  $d_1 \rightarrow d_2 \notin$  PathEdge then
[6]       Insert  $d_1 \rightarrow d_2$  into PathEdge
[7]       for each  $p \in$  calledProcs( $n$ ) and  $d_3 \in$  flowcall( $n, d_2, p$ ) do
[8]         Propagate( $s_p$ , AddPathEdge( $d_3, d_3$ ))
[9]         Insert  $\langle p, d_2 \rightarrow d_3 \rangle$  into CallEdge
[10]        for each  $d_4$  such that  $\langle p, d_3 \rightarrow d_4 \rangle \in$  CalleePathEdge do
[11]          for each  $d_5 \in$  flowret( $e_p, d_4, n, d_2$ ) do
[12]            Propagate(returnSite( $n$ ), AddPathEdge( $d_1, d_5$ ))
[13]          od
[14]        od
[15]      od
[16]      for each  $d_3 \in$  flowthru( $n, d_2$ ) do
[17]        Propagate(returnSite( $n$ ), AddPathEdge( $d_1, d_3$ ))
[18]      od
[19]    fi
[20]   case  $message$  matches AddCalleePathEdge( $p, d_1, d_2$ ) :
[21]     Insert  $\langle p, d_1 \rightarrow d_2 \rangle$  into CalleePathEdge
[22]     for each  $d_4$  such that  $\langle p, d_4 \rightarrow d_1 \rangle \in$  CallEdge do
[23]       for each  $d_5 \in$  flowret( $e_p, d_2, n, d_4$ ) do
[24]         for each  $d_3$  such that  $d_3 \rightarrow d_4 \in$  PathEdge do
[25]           Propagate(returnSite( $n$ ), AddPathEdge( $d_3, d_5$ ))
[26]         od
[27]       od
[28]     od
[29]   finally : Send Dec( $\rangle$ ) to Tracker
end

def ExitActor( $p$ )
[30]   local PathEdge :=  $\emptyset$ 
begin ( $message$ ) switch
[31]   case  $message$  matches AddPathEdge( $d_1, d_2$ ) :
[32]     if  $d_1 \rightarrow d_2 \notin$  PathEdge then
[33]       Insert  $d_1 \rightarrow d_2$  into PathEdge
[34]       for each  $c \in$  callers( $p$ ) do
[35]         Propagate( $c$ , AddCalleePathEdge( $p, d_1, d_2$ ))
[36]       od
[37]     fi
[38]   finally : Send Dec( $\rangle$ ) to Tracker
end

def IntraActor( $n$ )
[39]   local PathEdge :=  $\emptyset$ 
begin ( $message$ ) switch
[40]   case  $message$  matches AddPathEdge( $d_1, d_2$ ) :
[41]     if  $d_1 \rightarrow d_2 \notin$  PathEdge then
[42]       Insert  $d_1 \rightarrow d_2$  into PathEdge
[43]       for each  $d_3 \in$  flowi( $n, d_2$ ) and  $n' \in$  successors( $n$ ) do
[44]         Propagate( $n'$ , AddPathEdge( $d_1, d_3$ ))
[45]       od
[46]     fi
[47]   finally : Send Dec( $\rangle$ ) to Tracker
end

```

Algorithm 4. IFDS-A Node-Actor Classes

In addition to a PathEdge set, the CallSiteActor contains a CallEdge set. The CallEdge set retains the known call-flow edges (Algorithm 4 line 9) because summary-edge generation requires the inverse of flow_{call} (i.e. given some fact d_2 , finding all d_1 such that $d_2 \in \text{flow}_{call}(n, d_1, p)$) (line 22). CallEdge stores elements of the form $\langle p, d_1 \rightarrow d_2 \rangle$ because it must remember both the call-flow edge $d_1 \rightarrow d_2$ and the procedure p the edge goes to. The function of CallEdge in IFDS is the same as CallEdgeInverse in E-IFDS.

Node-actor operation is identical to the corresponding operations in E-IFDS, with one exception: IFDS-A’s exit node-actor forwards its edges to the call-site node-actor instead of generating summary edges itself. The reason for this is that the summary edges generated in response to facts at the exit node require access to the PathEdge and CallEdge sets present in the CallSiteActor. Summary edge generation must read both of these sets in one atomic operation to avoid missing updates to one of them. Therefore, the ExitActor sends every path-edge it receives to all of its call-site actors via the AddCalleePathEdge message.

The CallSiteActor retains the path-edges it receives in the CalleePathEdge set, for use by the normal AddPathEdge code (Algorithm 4 lines 10–14). E-IFDS call-site code accesses callee path-edges directly (Algorithm 2 lines 14–18), but the concurrent environment of IFDS-A requires each CallSiteActor to retain its own copy.

IFDS-A does not have a centralized WorkList, so it cannot use the “empty-worklist” condition to determine analysis completion. Instead, it uses a separate Tracker object to detect completion. The Tracker, created on line 6 and defined on lines 12–17 of Algorithm 3, keeps a count of unprocessed node-actor messages. Every time an actor calls Propagate to issue a new unit of work, it issues an Inc($\langle \rangle$) message to increment the Tracker’s count. Whenever an actor finishes processing a message, it sends a Dec($\langle \rangle$) message to the Tracker to decrement the count. When the count reaches zero, the Tracker sends Done($\langle \rangle$) (line 17, Algorithm 3) to wake up the main thread (line 8). Propagate sends Inc($\langle \rangle$) synchronously because the Tracker must process the Inc($\langle \rangle$) before its corresponding Dec($\langle \rangle$); otherwise the count could reach zero before all units of work complete. Communicating with the Tracker may appear to incur excessive message-passing overhead, but in practice we implement the Tracker with hardware-supported atomic-integer operations.

6 Implementation

We implement E-IFDS and IFDS-A in the Scala language, version 2.8.0 Beta 1. We use Soot [18], a Java bytecode optimization framework, to convert the input programs into CFGs suitable for our analysis.

6.1 The Variable Type Analysis

The analysis used for evaluation is the flow-sensitive variant of Variable Type Analysis (VTA) [17] defined by Naeem, et al. [11]. This analysis defines the fact-set D to be the set of all pairs $\langle v, T \rangle$ where v is a variable and T is a class type in the source program.

The presence of a fact $\langle v, T \rangle$ in the result set means that the variable v may point to an object of type T . Stated differently, the presence of $\langle v, T \rangle$ means that the analysis is unable to prove that v will *not* point to an object of type T .

Redundant Fact Removal. The original IFDS algorithm does not assume any relationships among facts, yet VTA and some other analyses do provide structured relationships. For VTA, if $\langle v, T \rangle$ and $\langle v, \text{superclass}(T) \rangle$ are in the same fact-set, then $\langle v, T \rangle$ is redundant. In our implementation of VTA, we check for redundant facts whenever we consider inserting a new element in the PathEdge set. A path-edge $d_1 \rightarrow \langle n, d_2 \rangle$ where d_2 is redundant is not inserted. If d_2 is not redundant, then any other path-edges which become redundant are removed from the PathEdge set. This prevents redundant facts from being considered in any future processing.

Priority Ordering. When using redundant fact removal, it may be advantageous to execute worklist items in a prioritized order. We implement an ordering for VTA facts which gives a higher priority to path-edges where the type T in d_2 has a smaller distance from the root type (i.e. Object). Without this ordering, the algorithms could do the work of constructing large fact sets only to discover later that much of the work was unnecessary.

6.2 Scheduling Actor Executions

Executing an actor-based algorithm on current mainstream hardware requires a *scheduler* to distribute work items among available processors. Normally, the scheduler creates some number of *worker threads* which the operating system dynamically assigns to available processors. Each unit of work is passed to the scheduler as soon as it is generated, and the scheduler eventually³ assigns it to a worker thread for execution.

Scala’s Actor Library. In our first implementation, we used Scala’s standard Actor library. Scala’s Actor implementation creates a Mailbox, or queue of unprocessed messages, for each actor created. The sending actor calls the `Actor.send` function on the receiving actor to insert a message into the receiving actor’s Mailbox. Whenever the receiving actor completes processing a message, it checks its Mailbox. If the Mailbox is non-empty, it removes a message which it passes to the underlying scheduler (by default, the Java `ForkJoinPool`) for execution. If the Mailbox is empty when the receiving actor finishes processing a message (or if no messages have yet been sent to the it), it becomes idle. If the receiving actor is idle, then calls to `Actor.send` bypass the Mailbox and submit the message directly to the scheduler.

There are two weaknesses of this implementation. The first is a performance weakness. In our preliminary experiments, we found that the overhead of handling a message was approximately 2 to 5 μs , an overhead which in some cases approached 50% of total execution time. Overhead from the `synchronized` keyword is incurred on every call to `Actor.send` and every call to the scheduler, which may be one source of inefficiency. In addition, many messages must be queued and dequeued twice: once in

³ By “eventually” we mean that every unit of work must be executed, not that the delay between time of submission to the scheduler and execution time is necessarily long.

a Mailbox, and again in the scheduler. Furthermore, the Scala Actor library seems to suffer from some scalability problems. For some inputs, the implementation runs more slowly with 8 threads than 4 threads. The second weakness of this implementation is that it does not support priority ordering of messages.

The Task-Throwing Scheduler. We created the Task-Throwing Scheduler to remedy these weaknesses. Instead of implementing an actor abstraction on top of a generic scheduler, we created a scheduler which is built specifically for actor-based programs. Algorithm 5 shows the Task-Throwing Scheduler.

There are two basic concerns when implementing an actor scheduler. The first concern is efficient scheduling of executable tasks, a concern which is common to all schedulers. In two respects we follow the lead of Arora et al. [2]. First, that work-stealing schedulers tend to make efficient use of processor resources, and second, that calling a `Yield()` statement after failing to acquire a lock is necessary for optimal performance on systems where we have no direct control over how the operating system schedules threads. Each of T threads in the scheduler runs Algorithm 5 until stopped. Each thread executes tasks from its own queue (specified by *qid*) until a failure occurs, at which time it yields control to any other threads ready to run (line 3) and then executes a task from a random queue (line 4) before checking its own queue again. Each queue is protected by a lock (from array L).

The second basic concern of an actor scheduler is maintaining mutually exclusive execution of tasks destined for the same actor. We accomplish this by including a lock for each actor which is acquired prior to running the user-defined `Actor.execute` function (lines 11–14). Each actor also includes *qid*, the queue number the currently executing task came from (line 12). If another thread attempts to execute a second task on the actor while it is busy, then lock acquisition fails and the task is re-inserted into the queue that the first task came from (line 17). This “throwing” action not only maintains mutual exclusion between tasks executing on the same actor, but also tends to group these tasks into the same work queue.

The Task-Throwing Scheduler uses boolean variables as locks, and requires the underlying hardware to support an atomic-exchange operation. Lock variables are `true` if held by some thread, `false` if not. The `AtomicExchange` function atomically writes a given value to a lock variable and returns the value previously held. The `TryAcquire` function (line 37) makes a single attempt to acquire a lock, returning `true` if successful. `ExecuteNext` uses `TryAcquire` so that it doesn’t block execution while there is potentially other useful work the thread could be doing. This non-blocking behaviour is particularly important when acquiring a lock on an actor, which could otherwise block for an arbitrarily long period of time. The `Send` function, which may be called from any thread or actor, uses `Acquire` (line 38), which blocks until the queue lock becomes free. Unlike actor locks, queue locks are only held for very short periods of time. Releasing a lock is as simple as setting the lock variable to `false` (line 39).

The Task-Throwing Scheduler is deadlock-free. An informal proof of this is as follows: After a queue lock is acquired by a thread, it is always released before the thread acquires any other locks (lines 7–10, 20 and 24–26). While holding an actor lock (lines 11–14), the user-defined `Actor.execute` function may subsequently call `Send`,

```

declare Q: static array of queues [0 .. T - 1] of pairs (Actor &, Message)
declare L: static array of volatile booleans [0 .. T - 1]

abstract class Actor:
  lock: volatile boolean
  qid: Integer
  virtual function execute (m: Message)
end

algorithm WorkerThread (qid: Integer)
begin
[1]   while true do
[2]     if ExecuteNext (qid) == false then
[3]       Yield ()
[4]     while ExecuteNext (rand () % T) == false do Yield () od
[5]     fi
[6]   od
end

static function ExecuteNext (qid: Integer)
begin
[7]   if TryAcquire (L[qid]) then
[8]     if Q[qid] is not empty then
[9]       Remove next (a, m) from Q[qid]
[10]      Release (L[qid])
[11]      if TryAcquire (a.lock) then
[12]        a.qid := qid
[13]        a.execute (m)
[14]        Release (a.lock)
[15]        return true
[16]      else
[17]        Send (a, m)
[18]      fi
[19]    else
[20]      Release (L[qid])
[21]    fi
[22]    fi
[23]    return false
end

global function Send (a: Actor &, m: Message)
begin
[24]  Acquire (L[a.qid])
[25]  Insert (a, m) into Q[a.qid]
[26]  Release (L[a.qid])
end

function TryAcquire (lock: volatile boolean &)
begin
[27]  return AtomicExchange (&lock, true) == false
end

function Acquire (lock: volatile boolean &)
begin
[28]  while AtomicExchange (&lock, true) == true do Yield () od
end

function Release (lock: volatile boolean &)
begin
[29]  lock := false
end

```

Algorithm 5. The Task-Throwing Scheduler

which acquires a queue lock. A thread holding an actor lock always releases it before attempting to acquire any other actor lock. Since any thread can hold at most one actor lock and one queue lock, and the actor lock is always acquired first, it follows that the execution of the scheduler code cannot introduce any lock acquisition cycles, and therefore cannot cause a deadlock.

7 Evaluation

We ran our performance tests on an eight-core AMD Opteron machine running the Oracle JRockit JVM version 3.1.2. We used the Oracle JVM because we were having some problems with the Sun JVM crashing.

Our test inputs are the programs `luindex`, `antlr`, and `ython` from the DaCapo Benchmark Suite version 2006-10-MR2 [3]. We analyse the source of the benchmarks only, but not the standard libraries. We make conservative worst-case assumptions about the types of objects returned from standard library methods.

7.1 Available Parallelism

Before evaluating actual performance, we wanted to estimate the maximum amount of parallelism available in the IFDS-A algorithm. To perform this estimation, we execute IFDS-A with a single-threaded worklist. Execution is performed using a doubly-nested loop, where the outer loop finds a maximal set of work units from the work-list that could be executed in parallel, and the inner loop executes this set of work units. Two work units are deemed to be executable in parallel if and only if they operate on different actors. The number of iterations the inner loop performs for a single iteration of the outer loop is the *available parallelism* (or *parallel width*), and the number of outer loop iterations is the length of a chain of sequentially dependent operations (or *parallel depth*). If all work units required the same amount of time to execute, the parallel depth is the optimal amount of time in which the algorithm could execute to completion, and the maximum parallel width is the number of processors that would be necessary to achieve this optimal time.

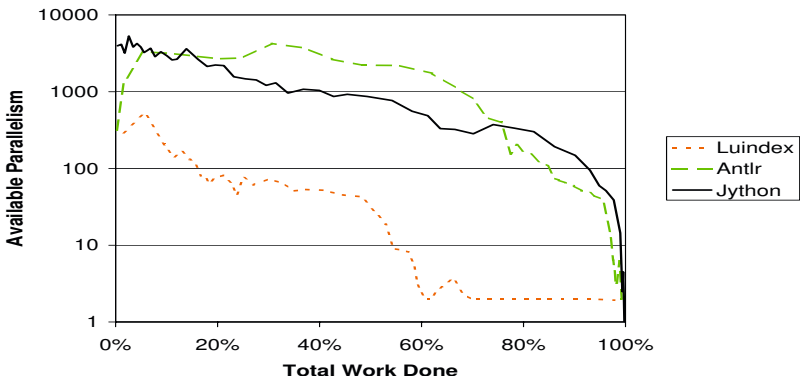


Fig. 4. Available Parallelism

Figure 4 shows the available parallelism for each input as a function of the percentage of work units completed. These charts provide an indication of how many processors the algorithm can keep busy for the given input, and for how long. For example, antlr provides sufficient parallelism to keep 100 processors busy for the first 85% of work units processed. The last 15% of work units will take longer than 15% of the total execution time because of insufficient parallelism to keep 100 processors busy. Jython has a similarly large available parallelism. In contrast, luindex has considerably smaller available parallelism, where a substantial fraction of the total work done exhibits a parallel width of only 2 units. This lack of available parallelism may limit the performance scalability of luindex.

7.2 Performance

We measured the performance of IFDS-A with the Scala Actor library and our Task Throwing Scheduler, and compared the results with the E-IFDS reference implementation. We tested the effects of priority queuing and different thread counts on the implementation's performance.

Table 1. Comparing Scheduling Methods

Policy	Scheduler	Th	LUIINDEX				ANTLR				JYTHON			
			Time	Acc	Rej	Fwd	Time	Acc	Rej	Fwd	Time	Acc	Rej	Fwd
FIFO	None (E-IFDS)	1	38	0.9	-	-	220	5.9	-	-	363	10	-	-
	ScalaActor	1	92	1.1	1.3	0.06	432	6.2	3.1	0.4	881	14	39	19
	ScalaActor	8	18	1.2	1.4	0.06	89	6.2	3.1	0.4	242	14	40	19
	TaskThrow	1	81	1.2	1.2	0.05	371	6.2	3.2	0.4	533	13	37	18
	TaskThrow	8	12	0.9	0.9	0.06	60	6.0	3.1	0.4	176	16	48	24
Priority	None (E-IFDS)	1	14	0.25	-	-	235	5.2	-	-	198	5.6	-	-
	TaskThrow	1	31	0.37	0.30	0.02	366	5.2	2.4	0.4	349	5.6	8.2	7.2
	TaskThrow	8	5	0.36	0.29	0.02	62	5.3	2.5	0.4	63	5.3	7.7	6.9

Table 1 summarizes the effects of the scheduler policy on performance. The columns of this table are:

- Policy: Indicates FIFO or priority processing of worklist elements.
- Scheduler: “None” is the single-threaded E-IFDS algorithm, “ScalaActor” is the default scheduler in the Scala Actors library, and “TaskThrow” is the new task-throwing scheduler.
- Th: Number of worker threads. Rows with a thread count of 8 are shown in bold. Each worker thread has its own worklist.
- Time: Average (geometric mean) time in seconds taken to perform one solve.
- Acc: Average number of worklist items accepted for processing, in millions.
- Rej: Average number of worklist items rejected after removal from a worklist because they had already been processed earlier, in millions.
- Fwd: Average number of worklist items forwarded from end-nodes to call-site nodes, in millions.

The scheduling policy is only enforced within a single worker thread, not across worker threads. Different worker threads can move through their worklists at different speeds depending on how long each item takes to process. Task-throwing, work-stealing, lock acquisition, and kernel scheduling activities also affect which worker processes which message. It is interesting to note that using multiple worker threads can sometimes result in execution orders that are more efficient than a strict adherence to the scheduling policy.

The task-throwing scheduler is significantly faster than Scala’s default scheduler on all the benchmarks tested. Furthermore, the task-throwing scheduler supports priority ordering of messages, whereas the default Actors implementation does not.

There are two major sources of overhead in the parallel IFDS-A algorithm compared to the sequential E-IFDS algorithm. The first source of overhead is passing messages for redundant path-edges. Unlike E-IFDS, which detects redundant path-edges before insertion into the worklist, IFDS-A checks for redundancy only after a message is received. The second source is forwarding end-node path-edges to call-site nodes. While E-IFDS can handle these edges immediately, IFDS-A must re-send each path-edge at an end-node to all associated call-site nodes.

For luindex, rejected and forwarded messages account for slightly more than half of the total messages sent by IFDS-A, more than doubling the total number of worklist items processed by E-IFDS. For antlr, rejected and forwarded messages account for a somewhat smaller proportion of the total messages, and for jython, they account for a significantly larger proportion.

Table 2. Performance with Different Thread Counts (Priority Scheduling Policy)

Scheduler	Th	LUIINDEX		ANTLR		JYTHON	
		Time	Sp/Sc	Time	Sp/Sc	Time	Sp/Sc
E-IFDS	1	13.6	1.0/-	235.0	1.0/-	198.4	1.0/-
TaskThrow	1	30.9	0.4/1.0	366.1	0.6/1.0	349.1	0.6/1.0
	2	14.6	0.9/2.1	188.9	1.2/1.9	177.2	1.1/2.0
	4	8.0	1.7/3.9	101.0	2.3/3.6	99.0	2.0/3.5
	8	5.2	2.6/6.0	61.5	3.8/6.0	62.6	3.2/5.6
	16	4.7	2.9/6.6	61.4	3.8/6.0	60.0	3.3/5.8
	32	5.2	2.6/6.0	61.8	3.8/5.9	69.6	2.9/5.0
	64	4.8	2.8/6.4	61.6	3.8/5.9	68.3	2.9/5.1

Table 2 shows the performance of IFDS-A using the priority scheduling policy. Two performance figures for each benchmark/thread count combination. The first, “Sp,” is the speedup obtained relative to the sequential E-IFDS. The second, “Sc,” is the scalability of performance relative to IFDS-A with one thread. Figures 5, 6, and 7 show the speedup graphically. The vertical error bars are the 95% confidence intervals.

When only one thread is available, the additional overhead of IFDS-A puts it at a substantial performance disadvantage. With two threads, however, IFDS-A is largely able to match the performance of E-IFDS, and with more threads it is able to exceed the performance of E-IFDS by a substantial margin. Luindex exhibited the worst speedup of the three, and antlr exhibited the best. Luindex may have a performance disadvantage

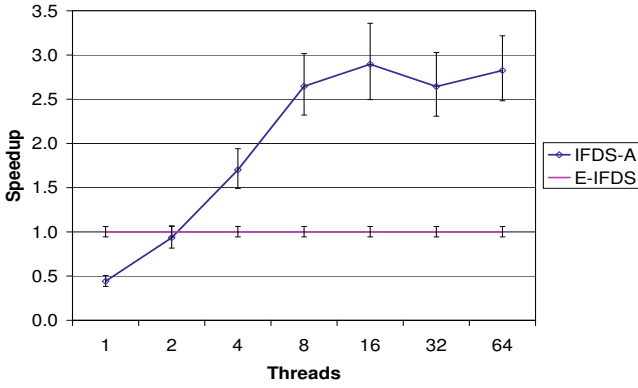


Fig. 5. Performance Chart for LUINDEX Analysis

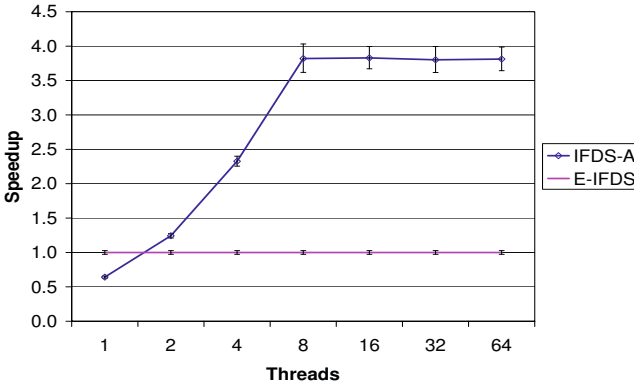


Fig. 6. Performance Chart for ANTLR Analysis

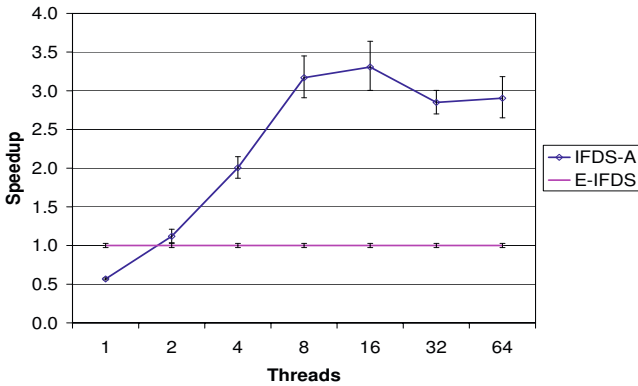


Fig. 7. Performance Chart for JYTHON Analysis

because the parallelism available in the problem is limited. Jython is hindered by message-passing overhead because of the relatively large number of rejected and call-site forwarding messages it generates.

Although our test machine has only eight cores, we also test with 16, 32, and 64 threads to see how our scheduler behaves with thread counts larger than the number of cores. As it turns out, 16 threads provides a small performance improvement over eight threads, indicating that our scheduling algorithm is not making full use of all processing resources available when only given eight threads. At 32 and 64 threads, however, we see performance worsen slightly due to the extra processing overhead incurred by larger numbers of threads.

At peak performance on our eight-core machine, we see IFDS-A reach 2.90x, 3.83x, and 3.31x speedup relative to E-IFDS for luindex, antlr, and jython, respectively, for an average speedup of 3.35x. Maximum scalability of these benchmarks is 6.56x, 5.97x, and 5.82x for an average scalability of 6.12x.

Since the slopes of these performance curves only level off after reaching eight threads on our eight-core machine, we can reasonably expect additional performance improvements on machines with larger numbers of cores.

8 Conclusions

This work began as an effort to see if a computationally expensive context-sensitive dataflow analysis algorithm could be expressed using message-passing, in the hope that some performance gain on multi-core computers would be seen. The resulting algorithm, IFDS-A, yielded performance gains which were significantly better than we initially expected. To the best of our knowledge, IFDS-A is the first implementation of IFDS that uses message-passing to communicate changes in state.

The implementation of IFDS-A performs substantially worse on a single core than the equivalent E-IFDS implementation. With two cores, there is still little reason to use IFDS-A because its performance is not much better E-IFDS (and in some cases is worse). With four or more cores, however, IFDS-A outperforms E-IFDS by a significant margin. On an eight-core computer, IFDS-A is on average 6.12 times as fast as it is with a single core, and 3.35 times as fast as the baseline implementation.

Priority ordering of worklist items was possible with the right scheduling mechanism, but it required implementation of a new scheduler.

There are several directions for possible future work, which include:

- verifying performance against a larger number of benchmarks,
- applying actor-based techniques to other types of analyses,
- reducing overhead by only creating one actor per function or one actor per basic block, and
- experimenting with different scheduling mechanisms to improve performance.

Acknowledgements. This work was supported, in part, by the Natural Sciences and Engineering Research Council of Canada.

References

1. Agha, G.: *Actors: A model of concurrent computation in distributed systems*. MIT Press, Cambridge (1986)
2. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Systems* 34(2), 115–144 (2001)
3. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: *OOPSLA* (2006)
4. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.* 410(2-3), 202–220 (2009)
5. Hardekopf, B., Lin, C.: The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In: *PLDI 2007: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 290–299. ACM, New York (2007)
6. Hewitt, C., Baker, H.: Laws for communicating parallel processes. In: *IFIP* (1977)
7. Kulkarni, M., Burtscher, M., Inkulu, R., Pingali, K., Casçaval, C.: How much parallelism is there in irregular applications? In: *PPoPP* (2009)
8. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. In: *PLDI* (2007)
9. Lee, E.A.: The problem with threads. *Computer* 39(5), 33–42 (2006)
10. Méndez-Lojo, M., Mathew, A., Pingali, K.: Parallel inclusion-based points-to analysis. In: *OOPSLA* (2010)
11. Naeem, N.A., Lhoták, O., Rodríguez, J.: Practical extensions to the IFDS algorithm. In: Gupta, R. (ed.) *CC 2010. LNCS, vol. 6011*, pp. 124–144. Springer, Heidelberg (2010)
12. Odersky, M., Spoon, L., Venners, B.: *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA (2008)
13. Panwar, R., Kim, W., Agha, G.: Parallel implementations of irregular problems using high-level actor language. In: *IPPS* (1996)
14. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: *POPL* (1995)
15. Rodríguez, J.D.: *A Concurrent IFDS Dataflow Analysis Algorithm Using Actors*. Master's thesis, University of Waterloo, Canada (2010)
16. Stein, L.A.: Challenging the computational metaphor: Implications for how we think. *Cybernetics and Systems* 30(6), 473–507 (1999)
17. Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical virtual method call resolution for Java. In: *OOPSLA* (2000)
18. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a Java bytecode optimization framework. In: *CASCON* (1999)
19. Virding, R., Wikström, C., Williams, M.: *Concurrent programming in ERLANG*, 2nd edn. Prentice Hall International (UK) Ltd., Hertfordshire (1996)

Using Disjoint Reachability for Parallelization

James C. Jenista, Yong hun Eom, and Brian Demsky

University of California, Irvine

Abstract. We present a disjoint reachability analysis for Java. Our analysis computes extended points-to graphs annotated with reachability states. Each heap node is annotated with a set of reachability states that abstract the reachability of objects represented by the node. The analysis also includes a global pruning step which analyzes a reachability graph to prune imprecise reachability states that cannot be removed with local reasoning alone. We have implemented the analysis and used it to parallelize 8 benchmarks. Our evaluation shows the analysis results are sufficiently precise to parallelize our benchmarks and achieve an average speedup of $16.9\times$.

1 Introduction

As the number of cores in mainstream processors increases, more software application domains are challenged with developing parallelized implementations in order to harness the available cores. Program analyses such as points-to analysis [1, 2] and shape analysis [3–7] support parallelization tools, particularly in application domains like scientific programs that exhibit highly regular data structures and workloads with much parallelism. In the future, developers will likely need to parallelize programs in the domain of object-oriented desktop and server applications where the program heap is complicated by many levels of abstraction or arbitrary data structure composition.

We believe the key to efficient and general heap analysis in support of parallelizing object-oriented applications is to focus on reachability. Data structures often have a single root object — disjoint reachability analysis can statically extract the property that the objects that comprise a data structure are only reachable from at most one data structure root object. This property is powerful for parallelizing tasks — the combination of (1) the static reachability analysis plus (2) a lightweight dynamic check to determine that the variables live into two tasks reference different root objects suffices to guarantee that the two tasks will not perform conflicting data accesses.

The use of reachability for parallelization is not new; points-to analysis is typically fast and can be scaled to millions of lines of code, but reachability information derived from it is imprecise so it can assist parallelization of only very specific code patterns. A path of edges in a points-to graph can capture that a data structure root object can reach the objects that comprise the data structure, but also admits the possibility that many data structure root objects can reach the same objects.

Reachability from objects is essential and different from the reachability from variables discovered by alias analysis [8–10] — reachability from variables can only express a finite number of disjoint sets. Variable reachability cannot discover that an unbounded set of live data structures do not share objects. This property is often necessary to parallelize computations on data structures.

The strategy of shape analysis is to statically model a potentially unbounded heap with a *shape graph*, where heap objects that share a common pattern of references are summarized with a shape node. While reachability properties can be deduced from a precise shape graph, in the worst case a program's heap shape is difficult to analyze to the effect that the derived reachability properties are much less precise.

Naik and Aiken introduced the concept of disjoint reachability in support of static race detection [11]. We introduce a new approach to disjoint reachability analysis with sufficient precision for parallelizing Java programs with general heap structures. Our analysis compactly represents the reachability of the many objects abstracted by a single node of a points-to graph by annotating each node with a set of reachability states. The reachability of an object is conservatively approximated by one of the reachability states for the corresponding node in the points-to graph. Shape information is not preserved in general; in this regard, shape analysis and disjoint reachability analysis are complementary. For instance, a compiler might generate tree-specific parallel code when shape analysis discovers that a structure is a valid tree.

Existing heap analyses have primarily either used equivalence classes of objects (e.g. TVLA [5], separation logic [12], storage shape graphs [3]) or access paths [4, 10] to reason about heap references. Disjoint reachability analysis combines aspects of both approaches — it combines an abstraction that decomposes the heap into static regions with reachability annotations that provide access path information. Choi et al. used a combination of storage and paths for alias analysis [13], however that approach focuses only on reachability from variables and is subject to the limitations discussed above.

1.1 Basic Approach

Disjoint reachability analysis extends a standard points-to graph with reachability annotations. The nodes in our points-to graph abstract objects and the edges abstract heap references between objects. While the points-to graph captures some reachability information; nodes in the points-to graph by necessity must abstract many objects. In general it is impossible to determine whether a path of edges in a points-to graph from a node n_{src} to a node n_{dst} represents a path of references from one object or many objects abstracted by n_{src} to an object abstracted by n_{dst} . We therefore annotate nodes with sets of reachability states. A reachability state for an object o contains a tuple for each node n that gives an abstract count of how many objects abstracted by node n can reach o .

Using reachability states only on nodes can make it difficult to precisely propagate changes to reachability states. We therefore also annotate edges with sets of reachability states. The reachability state for an edge abstracts the reachability states for the objects that can be reached through that edge. In addition to enabling the analysis to more precisely propagate changes to reachability states, edge reachability annotations can also serve to refine the reachability information for a node based on the reference used to access an object abstracted by the node.

We evaluate the precision of disjoint reachability analysis in support of a task-level parallelizing approach that relies on reachability properties rather than exploiting heap shape. Out-of-order Java (OoOJava) [14, 15] decomposes a sequential program into tasks and discovers which data structure root objects a task uses to obtain references to other objects. OoOJava then queries disjoint reachability analysis to discover whether

the objects reachable by two tasks are disjoint conditionally on whether the root objects of the tasks are distinct at runtime. This information enables parallelization when combined with a constant-time dynamic check that verifies the tasks access distinct root objects. OoJava reports disjoint reachability results back to the developer to identify unintended sharing that prevents parallelization. We also note disjoint reachability analysis is employed by Bamboo [16], another task-based parallel programming model.

Our analysis is demand-driven — it takes as input a set of allocation sites that are of interest to the analysis client. The analysis then computes the reachability only from the objects allocated at the selected allocation sites to all objects in the program.

1.2 Contributions

The paper makes the following contributions:

- **Disjoint Reachability Analysis:** It presents a new demand-driven analysis that discovers precise disjoint reachability properties for a wide variety of programs.
- **Reachability Abstraction:** It extends the points-to graph abstraction with reachability annotations to precisely reason about reachability properties.
- **Global Pruning:** It introduces a global pruning algorithm to improve the precision of reachability states.
- **Experimental Results:** It presents experimental results for several benchmarks. The results show that the analysis successfully discovers disjoint reachability properties and that it is suitable for parallelizing the benchmarks with significant speedups.

The remainder of the paper is organized as follows. Section 2 presents an example that illustrates how the analysis operates. Section 3 presents the program representation and the reachability graph. Section 4 presents the intraprocedural analysis. Section 5 presents the interprocedural analysis. Section 6 evaluates the analysis on several benchmarks. Section 7 presents related work; we conclude in Section 8. The appendix formalizes the reachability abstraction and overviews the correctness of the analysis.

2 Example

Figure 1 presents an example that constructs several graphs and then updates the vertices in those graphs. The `graphLoop` method populates an array with `Graph` objects. The developer uses the `task` keyword to indicate that the annotated block is worth executing in parallel with the current thread if it is safe to do so without violating the program’s sequential semantics. Our analysis will show that each `Vertex` object is reachable from at most one `Graph` object. This information could be used to parallelize the execution of tasks in the loop in Line 11. If a runtime check shows that instances of the task in Line 13 operate on different `Graph` objects, then our static analysis results will imply that task instances operate on disjoint sets of `Vertex` objects. The OoJava compiler [14] therefore *flags* the allocation site on Line 4 to indicate to the analysis that it needs information about the reachability from `Graph` objects to all objects. In this example, the disjoint reachability results allow the OoJava compiler to generate a parallel implementation.

```

1 public void graphLoop(int nGraphs) {
2   Graph [] a=new Graph[nGraphs];
3   for(int i=0; i<nGraphs; i++) {
4     Graph g=new Graph(); /* Analysis client flags this site */
5     Vertex v1=new Vertex();
6     g.vertex=v1;
7     Vertex v2=new Vertex();
8     v2.f=v1; v1.f=v2;
9     a[i]=g; }
10
11  for(int i=0; i<nGraphs; i++) {
12    Graph g=a[i];
13    task { /* This task updates the graph vertices. */
14      Vertex v=g.vertex;
15      while(!v.marked) {
16        v.marked=true;
17        v.updateVertex();
18        v=v.f; } } } }

```

Fig. 1. Graph Example

2.1 Intraprocedural Analysis

We next examine the analysis of the `graphLoop` method. Our analysis computes a reachability graph for each program point. Figure 2(a) presents the analysis results just after the allocation of the `Graph` and `Vertex` objects referenced by variables `g` and `v1`, respectively. The rectangular heap node n_2 abstracts the most recently allocated `Graph` object and is associated with a flagged allocation site; we call such heap nodes *flagged heap nodes* and shade them in all graphs in this paper. The analysis computes reachability only from flagged nodes. Heap nodes are assigned unique identifiers of form n_i , where i is a unique integer. The reachability set $\{\langle n_2, 1 \rangle\}$ on n_2 indicates the object abstracted by that node has the *reachability state* $\langle n_2, 1 \rangle$. The reachability set $\{\langle n_2, 1 \rangle\}$ on the `g` edge indicates that the corresponding heap reference can only reach objects whose reachability state is $\langle n_2, 1 \rangle$.

A reachability state for object o contains a set of *reachability tuples*: a reachability tuple consists of a heap node abstracting objects that may reach o and an arity that indicates how many objects abstracted by that node may reach o . The reachability state $\langle n_2, 1 \rangle$ means that the object with that reachability state is reachable from at most one `Graph` object abstracted by heap node n_2 and no other flagged objects. In Figure 2(a), node n_4 abstracts the most recently allocated `Vertex` object from Line 5 and has the reachability set $\{\langle \rangle\}$, meaning at this program point the object abstracted by n_4 is not reachable from any flagged objects. Appendix A precisely defines the abstraction.

Figure 2(b) presents the analysis results after the `vertex` field of the `Graph` object is updated to reference the `Vertex` object in Line 6. The set of reachability states for n_4 is updated to $\{\langle n_2, 1 \rangle\}$ to reflect that the `Vertex` object is now reachable

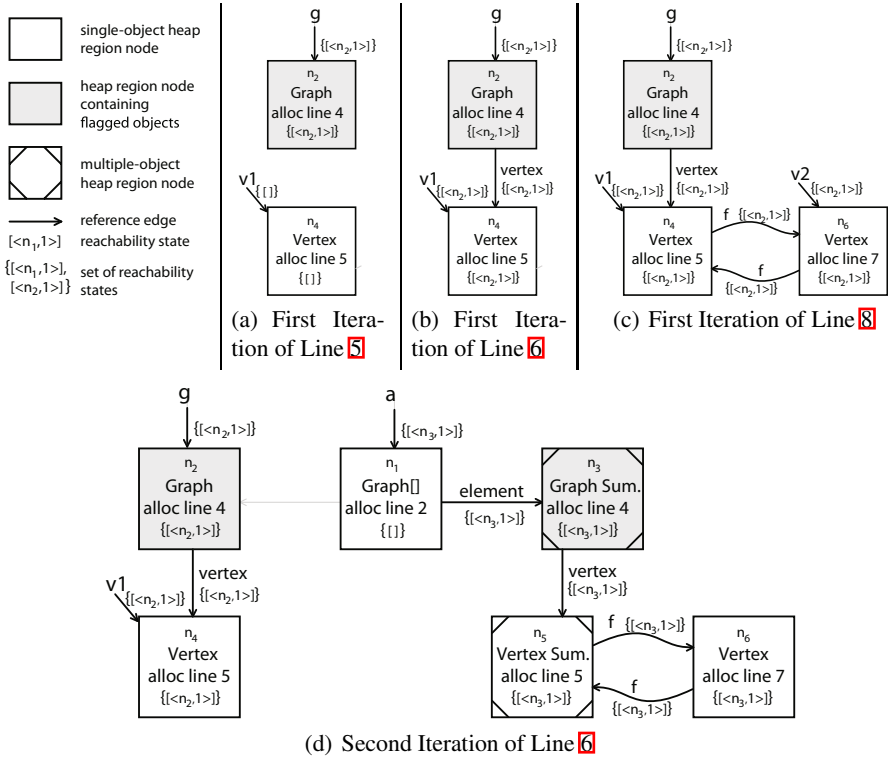


Fig. 2. Intraprocedural reachability graph results for several program points

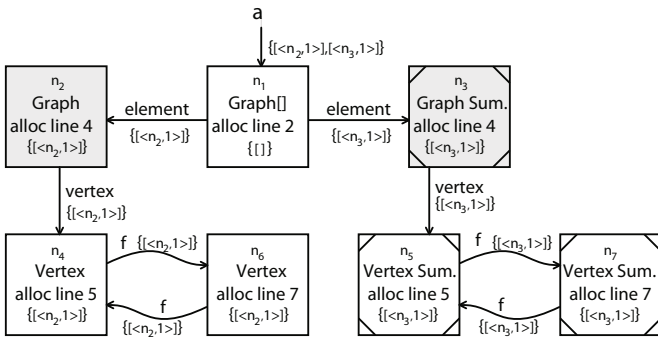


Fig. 3. Analysis result at Line 10 of graphLoop

from at most one Graph object. The newly created edge models the reference from the vertex field of the Graph object. Edges are marked with the field they abstract.

Figure 2(c) presents the analysis results after Line 8. At this point a second Vertex object has been allocated and the two Vertex objects have references to one another. The heap reference from n_4 to n_6 propagated the reachability set $\{ \langle n_2, 1 \rangle \}$ to n_6 .

The intraprocedural analysis continues until it computes a fixed-point solution. Figure 2(d) presents the analysis results after visiting Line 8 a second time. Node n_2 abstracts the most recently allocated `Graph` object while node n_3 summarizes the `Graph` object from the previous loop iteration. We denote summary heap nodes as rectangles with chords across each corner. The reachability state $\langle n_3, 1 \rangle$ has propagated backward across all edges up to the reference from variable `a`. Disjoint reachability analysis maintains the invariant that a reference is annotated with the reachability states of any objects reachable by following that reference. Note we omit the empty reachability state $[\]$ in some reachability sets for brevity.

Figure 3 presents the analysis results at Line 10 of the example program. These results state that any `Vertex` object is reachable from at most one `Graph` object because (1) the analysis client flagged the only allocation site for `Graph` objects and (2) there is no heap node abstracting `Vertex` objects with a reachability state indicating the contrary. Examples of reachability states for a `Vertex` object that is possibly reachable from more than one `Graph` object are $\langle n_2, 1 \rangle$, $\langle n_3, 1 \rangle$ and $\langle n_3, \text{MANY} \rangle$.

Disjoint reachability analysis will determine that a given `Vertex` object is reachable from at most one `Graph` object regardless of the relative shape of `Vertex` objects in the heap. In Section 6 we describe how OoJava queries disjoint reachability results such as this in order to parallelize benchmarks with a variety of heap structures. In this example, the OoJava compiler will use the analysis results to generate a simple dynamic check: if the `Graph` object referenced by variable `g` is different from `Graph` objects referenced in previous iterations, then the task in Line 13 may safely execute in parallel with the current thread.

3 Analysis Abstractions

This section presents the analysis abstractions for the input program, elements of the reachability graph, and reachability annotations that extend reachability graphs.

3.1 Program Representation

The analysis takes as input a standard control flow graph representation of each method. Program statements have been decomposed into statements relevant to the analysis: copy, load, store, object allocation, and call site statements.

3.2 Reachability Graph Elements

Our analysis computes a reachability graph for the exit of each program statement. Reachability graphs extend the standard points-to graph representation to maintain object reachability properties. Heap nodes abstract objects in the heap. There are two heap nodes for each allocation site $m \in M$ in the program — one heap node abstracts the single most recently allocated object at the allocation site, and the other is a summary node that abstracts all other objects allocated at the site¹.

¹ Our implementation generalizes this to support abstracting the k most recently allocated objects from the allocation site with single-object heap nodes.

In general, analysis clients only need to determine reachability from some subset of the objects in the program. The analysis takes as input a set of allocation sites $M_F \subseteq M$ for objects of interest — the analysis then computes for all objects in the program their reachability from objects allocated at those sites.

The reachability graph G has the set of heap nodes $n \in N = M \times \{0, \text{summary}\}$. The analysis client specifies a set of flagged heap nodes $N_F = M_F \times \{0, \text{summary}\} \subseteq N$ that it is interested in determining reachability from.

Graph edges $e \in E$ abstract references $r \in R$ in the concrete heap and are of the form $\langle v, n \rangle$ or $\langle n, f, n' \rangle$. The heap node or variable that edge e originates from is given by $\text{src}(e)$ and the heap node that edge e refers to is given by $\text{dst}(e)$. Every reference edge between heap nodes has an associated field $f \in F = \text{Fields} \cup \text{element}$ ².

The equation $E \subseteq V \times N \cup N \times F \times N$ gives the set of reference edges E in a reachability graph. We define the convenience functions:

$$E_n(v) = \{n \mid \langle v, n \rangle \in E\} \quad (3.1) \quad E_e(n) = \{\langle n, f, n' \rangle \mid \langle n, f, n' \rangle \in E\} \quad (3.4)$$

$$E_e(v) = \{\langle v, n \rangle \mid \langle v, n \rangle \in E\} \quad (3.2) \quad E_n(v, f) = \{n' \mid \langle v, n \rangle, \langle n, f, n' \rangle \in E\} \quad (3.5)$$

$$E_n(n) = \{n' \mid \langle n, f, n' \rangle \in E\} \quad (3.3) \quad E_e(v, f) = \{\langle n, f, n' \rangle \mid \langle v, n \rangle, \langle n, f, n' \rangle \in E\} \quad (3.6)$$

3.3 Reachability Annotations

This section discusses how our analysis augments a points-to graph with reachability annotations; Appendix A formalizes this abstraction. A reachability tuple $\langle n, \mu \rangle$ is a heap node and arity pair where the arity value μ is taken from the set $\{0, 1, \text{MANY}\}$. The arity μ gives the number of objects from the heap node n that can reach the relevant object. The arity 0 means the object is not reachable from any objects in the given heap node, the arity 1 means the object is reachable from at most one object in the given heap node, and the arity MANY means the object is reachable from any number of objects in the given node. The arities have the partial order $0 \sqsubseteq 1 \sqsubseteq \text{MANY}$.

A reachability state $\phi \in \Phi$ contains exactly one reachability tuple for every distinct flagged heap node. For efficiency, our implementation elides arity-0 reachability tuples. When we write reachability states, we use brackets to enclose the reachability tuples to make them visually more clear. For example, the reachability state $\phi_n = [\langle n_3, 1 \rangle] \in \Phi_{n_7}$ that appears on node n_7 in Figure 3 indicates that it is possible for at most one object in heap node n_3 , and zero objects from any other flagged heap nodes (i.e. n_2) to reach an object from heap node n_7 .

The function $\mathcal{A}^N : N \rightarrow \mathcal{P}(\mathcal{P}(M))$ maps a heap node n to a set of reachability states. The reachability of an object abstracted by the heap node n is abstracted by one of the reachability states given by the function \mathcal{A}^N . We represent \mathcal{A}^N as a set of tuples of heap nodes and reachability states and define the helper function:

$$\mathcal{A}^N(n) = \{\phi \mid \langle n, \phi \rangle \in \mathcal{A}^N\}. \quad (3.7)$$

When a new reference is created, the analysis must propagate reachability information. Simply using the graph edges to do this propagation would yield imprecise results. To improve the precision of this propagation step, the analysis maintains for each edge the reachability states of all objects that can be reached from that edge. The function

² The special field `element` abstracts all references from an array's elements.

$\mathcal{A}^E : E \rightarrow \mathcal{P}(\mathcal{P}(M))$ maps a reference edge e to the set of reachability states of all the objects reachable from the references abstracted by e . If there exists a path of heap references $r_1; r_2; r_j$ that leads to an object o with the reachability state ϕ then for each edge e_i that abstracts a reference r_i , $\phi \in \mathcal{A}^E(e_i)$. We represent \mathcal{A}^E as a set of tuples of edges and reachability states and define the helper functions:

$$\mathcal{A}^E(v) = \{\phi \mid \langle\langle v, n \rangle, \phi \rangle \in \mathcal{A}^E\}, \quad (3.8)$$

$$\mathcal{A}^{E^o}(v) = \{\langle\langle v, n \rangle, \phi \rangle \mid \langle\langle v, n \rangle, \phi \rangle \in \mathcal{A}^E\}, \quad (3.9)$$

$$\mathcal{A}^E(e) = \{\phi \mid \langle e, \phi \rangle \in \mathcal{A}^E\}. \quad (3.10)$$

In programs that construct unbounded data structures, heap nodes and reference edges must be summarized in the finite heap abstraction. Summarization in basic points-to analysis rapidly loses precision for reachability properties. Disjoint reachability analysis improves the precision of reachability information with \mathcal{A}^N because it can express that an object is reachable from only a single object abstracted by a given summary node. The abstraction \mathcal{A}^E is instrumental for precisely updating the reachability states of heap nodes because it refines path information present in the points-graph alone and allow the analysis to more precisely propagate changes to reachability states than is possible from the points-to graph alone. Another critical aspect of \mathcal{A}^E is that an edge in effect selects some subset of reachability states of a node — the analysis uses this information to refine the reachability states for a node.

4 Intraprocedural Analysis

We begin by presenting the intraprocedural analysis. Section 5 extends this analysis to support method calls. The analysis is structured as a fixed-point computation.

4.1 Method Entry

The method entry transfer function creates an initial reachability graph to abstract the part of the calling methods' heaps that are reachable from parameters. In the example, the initial reachability graph is empty because the method `graphLoop` does not take parameters. Method context generation is explained in detail in Section 5.1.

4.2 Copy Statement

A copy statement of the form $x=y$ makes the variable x point to the object that y references. The analysis always performs strong updates for variables — it discards all the reference edges from variable x and then copies all the edges along with their reachability states from variable y . Equation 4.1 and Equation 4.2 describe the transformations.

4.3 Load Statement

Load statements of the form $x=y.f$ make the variable x point to the object that $y.f$ references. Existing reference edges for the field are copied to x as described by Equation 4.3. Note that this statement does not change the reachability of any object. The reachability on new edges from x , as described by Equation 4.4, is the intersection of $\mathcal{A}^E(\langle y, n \rangle)$ and $\mathcal{A}^E(\langle n, f, n' \rangle)$, because x can only reach objects that were reachable from both the variable y and a heap reference abstracted by the edge $\langle n, f, n' \rangle$.

<p>(a) Copy Statement</p>	$E' = (E - E_e(x)) \cup (\{x\} \times E_n(y)) \quad (4.1)$ $\mathcal{A}^{E'} = (\mathcal{A}^E - \mathcal{A}^{E^\circ}(x)) \cup \bigcup_{(v,n) \in E_e(y)} (\{x, n\} \times \mathcal{A}^E(y)) \quad (4.2)$
<p>(b) Load Statement</p>	$E' = (E - E_e(x)) \cup (\{x\} \times E_n(y, f)) \quad (4.3)$ $\mathcal{A}^{E'} = (\mathcal{A}^E - \mathcal{A}^{E^\circ}(x)) \cup \bigcup_{(n,f,n') \in E_e(y,f)} (\{x, n'\} \times (\mathcal{A}^E(\langle y, n \rangle) \cap \mathcal{A}^E(\langle n, f, n' \rangle))) \quad (4.4)$
<p>(c) Allocation Statement</p>	$\mathcal{A}^{N'} = \{ \langle \mathcal{S}_N(n, n_{\text{alloc}}), \mathcal{S}_\phi(\phi, n_{\text{alloc}}) \rangle \mid \langle n, \phi \rangle \in \mathcal{A}^N \} \cup \{ \langle n_{\text{alloc}}, \phi_{\text{alloc}} \rangle \} \quad (4.5)$ $\mathcal{A}^{E'} = \{ \langle \mathcal{S}_E(e, n_{\text{alloc}}), \mathcal{S}_\phi(\phi, n_{\text{alloc}}) \rangle \mid \langle e, \phi \rangle \in \mathcal{A}^E \} \cup \{ \langle x, n_{\text{alloc}} \rangle, \phi_{\text{alloc}} \rangle \} \quad (4.6)$ $E' = \{ \mathcal{S}_E(e, n_{\text{alloc}}) \mid e \in E \} \cup \{ x, n_{\text{alloc}} \} \quad (4.7)$ $\phi_{\text{alloc}} = \begin{cases} [\langle n_{\text{alloc}}, 1 \rangle] & \text{if } n_{\text{alloc}} \text{ is flagged} \\ \square & \text{otherwise} \end{cases} \quad (4.8)$

Fig. 4. Transfer Functions for Copy, Load, and Allocation Statements

4.4 Object Allocation Statement

The analysis abstracts the most recently allocated object from an allocation site as a single-object heap node. A summary node for the allocation site abstracts any objects from the allocation site that are older than the most recent.

The object allocation transfer function merges the single-object node n_{alloc} into the site's summary node. The single-object node n_{alloc} is then the target of a variable assignment. Equations 4.5 through 4.7 describe the basic transformation. We define the helper function $\mathcal{S}_N(n, n_{\text{alloc}})$ to return the corresponding summary node for n_{alloc} if $n = n_{\text{alloc}}$ and n otherwise. We define $\mathcal{S}_E(\langle v, n \rangle, n_{\text{alloc}}) = \langle v, \mathcal{S}_N(n, n_{\text{alloc}}) \rangle$ and $\mathcal{S}_E(\langle n, f, n' \rangle, n_{\text{alloc}}) = \langle \mathcal{S}_N(n, n_{\text{alloc}}), f, \mathcal{S}_N(n', n_{\text{alloc}}) \rangle$.

As stated, the single-object node and its reachability information merge with the summary node. We define the helper function $\mathcal{S}_\phi(\phi, n_{\text{alloc}})$ to update a reachability state by rewriting all reachability tuples with n_{alloc} to use the summary node. If both the single-object node and the summary node appeared in the same reachability state before this transform, after rewriting the tuples there will be, conceptually, two summary node tuples in the state. In this case the tuples are combined and the new arity for the summary heap node tuple is computed by $+\Delta$, which is addition in the domain $\{0, 1, \text{MANY}\}$. Note that the reachability annotations enable the analysis to maintain precise reachability information over summarizations.

Finally, if the heap node is flagged, the analysis generates the set of reachability states $\{[\langle n_f, 1 \rangle]\}$, where n_f is the given heap node, for the new object's node and edge. Otherwise, it generates the set $\{[\square]\}$ with the empty state for the node and edge.

4.5 Store Statement

Store statements of the form $x.f=y$ point the f field of the object referenced by x to the object referenced by y . Store statements can change the reachability of objects reachable from y . Equation 4.9 describes how a store changes the edge set.

$$E' = E \cup (E_n(x) \times \{f\} \times E_n(y)) \quad (4.9)$$

Let o_x be the object referenced by x in the concrete heap and o_y be the object referenced by y . The new edge from the object o_x to the object o_y can only add new paths from objects that could previously reach o_x to objects that were reachable from o_y . In the reachability graph, the heap nodes $n_x \in E_n(x)$ abstract o_x and the heap nodes $n_y \in E_n(y)$ abstract o_y . The set of flagged heap nodes containing objects that could potentially reach o_x is given by the set of reachability states:

$$\Psi_x = \mathcal{A}^N(n_x) \cap \mathcal{A}^E(\langle x, n_x \rangle). \quad (4.10)$$

The reachability states of the objects reachable from o_y is

$$\Psi_y = \mathcal{A}^E(\langle y, n_y \rangle). \quad (4.11)$$

We define \cup_Δ to compute the union of two reachability states. When two reachability states are combined, tuples with matching heap nodes merge arity values according to $+\Delta$. We divide updating the reachability graph into the following steps:

1. **Construct the New Graph:** The analysis first constructs the new edge set as described by Equation 4.9.
2. **Update Reachability States of Downstream Heap Nodes:** The reachability of every object o' reachable from o_y is (i) abstracted by some $\psi_y \in \Psi_y$ and (ii) there exist a path of edges from the heap node that abstracts o_y to the heap node that abstracts o' in which each edge has ψ_y in its reachability state. The newly created edge can make the object o' reachable from the objects that can reach o_x — this set of objects is abstracted by some reachability state $\psi_x \in \Psi_x$. Therefore the new reachability state for o' should be $\psi_y \cup_\Delta \psi_x$. We capture this reachability change with the *change tuple set* $C_{n_y} = \{\langle \psi_y, \psi_y \cup_\Delta \psi_x \rangle \mid \psi_y \in \Psi_y, \psi_x \in \Psi_x\}$. Constraints 4.12 and 4.13 express the path constraint (ii). The analysis uses a fixed point to solve these constraints and then uses Equation 4.14 to update the reachability states of downstream nodes.

$$\Lambda^{\text{node}}(n_y) \supseteq C_{n_y} \quad (4.12)$$

$$\Lambda^{\text{node}}(n') \supseteq \{\langle \phi, \phi' \rangle \mid \langle \phi, \phi' \rangle \in \Lambda^{\text{node}}(n), \langle n, f, n' \rangle \in E, \phi \in \mathcal{A}^E(\langle n, f, n' \rangle)\} \quad (4.13)$$

$$\begin{aligned} \mathcal{A}^{N'}(n) &= \{\phi' \mid \phi \in \mathcal{A}^N(n), \langle \phi, \phi' \rangle \in \Lambda^{\text{node}}(n)\} \cup \\ &\quad \{\phi \mid \phi \in \mathcal{A}^N(n), \nexists \phi'. \langle \phi, \phi' \rangle \in \Lambda^{\text{node}}(n)\} \end{aligned} \quad (4.14)$$

3. **Propagate Reachability from Downstream Nodes to Edges:** The analysis must propagate the reachability changes of objects back to any edge that abstracts a reference that can reach the object. Constraint 4.15 ensures that edges contain reachability change tuples that capture the reachability changes in the incident objects.

Constraint [4.16](#) ensures that the change set contains tuples to re-establish the transitive reachability state property.

$$\Lambda^{\text{edge}}(e) \supseteq \{ \langle \phi, \phi' \rangle \mid \langle \phi, \phi' \rangle \in \Lambda^{\text{node}}(\text{dst}(e)), \phi \in \mathcal{A}^N(\text{dst}(e)), \phi \in \mathcal{A}^E(e) \} \quad (4.15)$$

$$\Lambda^{\text{edge}}(e) \supseteq \{ \langle \phi, \phi' \rangle \mid \langle \phi, \phi' \rangle \in \Lambda^{\text{edge}}(e'), \phi \in \mathcal{A}^E(e), \text{dst}(e) = \text{src}(e') \} \quad (4.16)$$

4. Propagate Reachability Changes Upstream of o_x : The reachability states of edges that abstract references that can reach o_x must be updated to reflect the objects they can now reach through the newly created edge. We define the change tuple set $C_{n_x} = \{ \langle \psi_x, \psi_y \cup_{\Delta} \psi_x \rangle \mid \psi_y \in \Psi_y, \psi_x \in \Psi_x \}$ that updates the reachability states of edges that can reach o_x . Constraint [4.17](#) ensures that edges incident to the heap nodes that abstract o_x contain reachability change tuples that capture the reachability states of the newly reachable objects. Constraint [4.18](#) ensures that the change set contains tuples to re-establish the transitive reachability state property.

$$\Upsilon^{\text{edge}}(e) \supseteq \{ \langle \phi, \phi' \rangle \mid \langle \phi, \phi' \rangle \in C_{n_x}, \phi \in \mathcal{A}^E(e), \text{dst}(e) = n_x \} \quad (4.17)$$

$$\Upsilon^{\text{edge}}(e) \supseteq \{ \langle \phi, \phi' \rangle \mid \langle \phi, \phi' \rangle \in \Upsilon^{\text{edge}}(e'), \phi \in \mathcal{A}^E(e), \text{dst}(e) = \text{src}(e') \} \quad (4.18)$$

5. Update Edge Reachability: Finally, the analysis generates the reachability states for the edges in the new graph. Equation [4.19](#) computes the reachability states of all edges that existed before the store operation using the change tuple sets. Equation [4.20](#) computes the reachability for the newly created edges from the reachability of the edges for y with the constraint that every reachability state on the edge must be at least as large as the reachability state for the object o_x . We define $\phi \subseteq_{\Delta} \phi'$ if $\forall \langle n, \mu \rangle \in \phi$ there exists a reachability tuple $\langle n, \mu' \rangle \in \phi'$ such that $\mu \sqsubseteq \mu'$.

$$\begin{aligned} \mathcal{A}^{E'}(e) = & \mathcal{A}^E(e) \cup \{ \phi' \mid \langle \phi, \phi' \rangle \in \Lambda^{\text{edge}}(e), \phi \in \mathcal{A}^E(e) \} \cup \\ & \{ \phi' \mid \langle \phi, \phi' \rangle \in \Upsilon^{\text{edge}}(e), \phi \in \mathcal{A}^E(e) \} \end{aligned} \quad (4.19)$$

$$\mathcal{A}^{E'}(\langle n_x, \mathfrak{f}, n_y \rangle) \subseteq \{ \phi \in \mathcal{A}^{E'}(\langle y, n_y \rangle) \mid \exists \phi' \in \mathcal{A}^{N'}(n_x), \phi' \subseteq_{\Delta} \phi \} \quad (4.20)$$

Strong Updates. While in general the analysis performs *weak updates* that simply add edges, under certain circumstances the analysis can perform *strong updates* that also remove edges to increase the precision of the results. Strong updates are possible under either of two conditions. First, when variable x is the only reference to a heap node n_x . In this case we can destroy all reference edges from n_x with field \mathfrak{f} because no other variables can reach n_x . Second, when the variable x references exactly one heap node n_x and n_x is a single-object heap node. When this is true x definitely refers to the object in n_x and the existing edges with field \mathfrak{f} from n_x can be removed.

For strong updates, the analysis first removes edges that the strong update eliminates. It then performs the normal transform as described in this section. Note that when strong updates remove edges, reachability of graph elements may change if the removed edges provided the reachability path. Therefore, reachability states may become imprecise. After a store transform with a strong update occurs, a global pruning step improves imprecise reachability states. Section [4.9](#) presents the global pruning step.

4.6 Element Load and Store Statements

Our analysis implements the standard pointer analysis treatment of arrays: Array elements are treated as a special field of array objects and always have weak store semantics. The analysis does not differentiate between different indices. This treatment can cause imprecision for operations such as vector removes that move a reference from one array element to another. Our implementation uses a special analysis to identify array store operations that acquire an object reference from an array and then create a reference from a different element of that array to the same object. Because the graph already accounts for this reachability, the effects of such stores can be safely omitted.

4.7 Return Statement

Return statements are of the form `return x` and return the object referenced by `x`. Each reachability graph has a special `Return` variable that is out of program scope. At a method return the transfer function assigns the `Return` variable to the references of variable `x`. We assume without loss of generality that the control flow graph has been modified to merge the control flow for all return statements.

4.8 Control Flow Join Points

To analyze a statement, the analysis first computes the join of the incoming reachability graphs. The operation for merging reachability graphs r_0 and r_1 into r_{out} follows below:

1. The set of variables for r_{out} is the set of live variables into the statement.
2. The set of heap nodes for r_{out} is the union of the heap nodes in the input graphs. The union of the reachability states is taken, $\mathcal{A}_{\text{out}}^N(n) = \mathcal{A}_0^N(n) \cup \mathcal{A}_1^N(n)$.
3. The set of reference edges for r_{out} is the union of the reference edges of the input graphs. Reference edges are unique in a reachability graph with respect to source, field, and destination. For a reference edge e , $\mathcal{A}_{\text{out}}^E(e) = \mathcal{A}_0^E(e) \cup \mathcal{A}_1^E(e)$.

4.9 Global Pruning

When strong updates remove edges, the reachability states may become imprecise. The call site transfer function in Section 5 can also introduce imprecise reachability states. Our analysis includes a global pruning step that uses global reachability constraints to prune imprecise reachability states to improve the precision of the analysis results. The intuition behind global pruning is that multiple abstract states can correspond to the same set of concrete heaps, and the global pruning step generates an equivalent abstraction that locally has more precise reachability states.

Global Reachability Constraints. Reachability information must satisfy two reachability constraints that follow from the discussion in Section 3.3

- **Node reachability constraint:** For each node n , $\forall \phi \in \mathcal{A}^N(n)$, $\forall \langle n', \mu \rangle \in \phi$, if $\mu \in \{1, \text{MANY}\}$ then there must exist a set of edges e_1, \dots, e_m such that $\phi \in \mathcal{A}^E(e_i)$ for all $1 \leq i \leq m$ and the set of edges e_1, \dots, e_m form a path through the reachability graph from n' to n .

- **Edge reachability constraint:** For each edge e , $\forall \phi \in \mathcal{A}^E(e)$ there exists $n \in N$ and $e_1, \dots, e_m \in E$ such that $\phi \in \mathcal{A}^N(n)$; $\phi \in \mathcal{A}^E(e_i)$ for all $1 \leq i \leq m$; and the set of edges e_1, \dots, e_m form a path through the reachability graph from e to n .

The first phase of the algorithm generates a reachability graph with the most precise set of reachability states for the nodes. The second phase of the algorithm generates the most precise set of reachability states for the edges.

1. Improve the precision of the node reachability states: The algorithm first uses the node reachability constraint to prune the reachability states of nodes. This phase uses the existing \mathcal{A}^E to prune reachability tuples from imprecise reachability states to generate a more precise $\mathcal{A}^{N'}$ from the previous \mathcal{A}^N . The algorithm iterates through each flagged node n_f . The function $\mathcal{A}_f^E(e)$ maps the edge $e \in E$ to the set of reachability states Φ for which each $\phi \in \Phi$ (1) includes a non-zero arity reachability tuple with the node n_f and (2) there exist a path from n_f to e for which every edge along the path contains ϕ in its set of reachability states. We compute \mathcal{A}_f^E using a fixed-point algorithm on the following two constraints:

$$\forall e \in E_e(n_f), \mathcal{A}_f^E(e) \supseteq \mathcal{A}^E(e), \quad (4.21)$$

$$\forall e \in E, e' \in E_e(\text{dst}(e)), \mathcal{A}_f^E(e') \supseteq \mathcal{A}^E(e') \cap \mathcal{A}_f^E(e). \quad (4.22)$$

For each node n and each reachability state $\phi \in \mathcal{A}^N(n)$ the analysis shortens ϕ to remove tuples $\langle n_f, 1 \rangle$ or $\langle n_f, \text{MANY} \rangle$ to generate a new reachability state ϕ' if ϕ does not appear in $\mathcal{A}_f^E(e)$ of any edge e incident to n . This step does not prune $\langle n_f, 1 \rangle$ or $\langle n_f, \text{MANY} \rangle$ from the reachability states of flagged nodes n_f . The analysis then propagates these changes to \mathcal{A}^E of the upstream edges using the same propagation procedure described by Equations 4.15 and 4.16 to generate \mathcal{A}_r^E .

2. Improve the precision of the edge reachability states: The algorithm next uses the pruned node reachability states in $\mathcal{A}^{N'}$ and \mathcal{A}_r^E to generate a more precise $\mathcal{A}^{E'}$. The intuition is that an edge can only have a given reachability state if there exists a path from that edge to a node with that reachability state such that all edges along the path contain the reachability state. The analysis starts from every heap node n and propagates the reachability states of $\mathcal{A}^N(n)$ backwards over reference edges. The analysis initializes $\mathcal{A}^{E'} = \{\mathcal{A}_r^E(e) \cap \mathcal{A}^{N'}(n) \mid \forall e \in E, n = \text{dst}(e)\}$. The analysis then propagates reachability information backwards to satisfy the constraint: $\mathcal{A}^{E'}(e) \supseteq \mathcal{A}_r^E(e) \cap \mathcal{A}^{E'}(e')$ for all $e' \in E_e(\text{dst}(e))$. The propagation continues until a fixed-point is reached.

4.10 Static Fields

We have omitted analysis of static fields or globals. We assume that the preprocessing stage creates a special global object that contains all of the static fields and passes it to every call site. Through this semantics-preserving program transformation, static field store and load statements become normal store and load statements, respectively.

5 Interprocedural Analysis

The interprocedural analysis adds a call site transfer function to the intraprocedural analysis. It uses a standard fixed-point algorithm that accommodates recursive call chains and begins by analyzing the `main` method. Our analysis processes each method using one context that summarizes the heaps for all call sites.

We expose only the callee reachable portion of the caller's heap when analyzing a callee method, similar to previous work [17, 18] but with extensions to accommodate reachability properties. A summary of the transform follows:

- i) Compute the portion of the heap that is reachable from the callee.
- ii) Rewrite reachability states to abstract flagged heap nodes that are not in the callee heap with special out-of-context heap nodes.
- iii) Merge this portion of the heap with the callee's current initial graph. If the graph changes, schedule the callee for reanalysis.
- iv) Specialize the callee's current analysis result using the caller context.
- v) Replace the callee reachable part of the caller's heap with the specialized results.
- vi) Merge nodes such that each allocation site has at most one summary heap node and one single object heap node.
- vii) Call the global pruning step (Section 4.9) to improve the precision of the graph.

5.1 Compute Callee Context Subgraph

For each call site, the analysis computes the subgraph $G_{\text{sub}} \subseteq G$ that is reachable from the call site's arguments. For each incoming edge $\langle n, f, n' \rangle \in E$ into G_{sub} where $n \notin G_{\text{sub}}$ and $n' \in G_{\text{sub}}$, the analysis generates a new *placeholder node* n_p and a new edge $e' = \langle n_p, \mathcal{R}(n') \rangle$ where $\mathcal{A}^E(e') = \mathcal{A}^E(e)$. The placeholder node n_p serves as a proxy flagged node for all reachability nodes in $\mathcal{A}^N(n)$ during global pruning in the callee context. For each incoming edge $\langle v, n' \rangle \in E$ into G_{sub} where $n' \in G_{\text{sub}}$, the analysis generates a new placeholder variable v_p and *placeholder edge* $e_p = \langle v_p, n' \rangle$ where $\mathcal{A}^E(e_p) = \mathcal{A}^E(e)$.

5.2 Out-of-Context Reachability

Summarization presents a problem for any out-of-context flagged heap node $n_f \notin G_{\text{sub}}$ that appears in reachability states of an in-context heap node $n \in G_{\text{sub}}$. The interprocedural analysis uses placeholder flagged nodes to rewrite out-of-context flagged heap nodes in reachability states. Each heap node n_f that appears in \mathcal{A}^N of a placeholder node is (1) outside of the graph G_{sub} and (2) abstracts objects that can potentially reach objects abstracted by the subgraph G_{sub} . The analysis replaces all such nodes in all in-context reachability states with special out-of-context heap nodes for the allocation site. There can be up to two out-of-context heap nodes per an allocation site: one is a summary node and one abstracts the most recently allocated object from the allocation site. The purpose of these heap nodes is to allow the analysis of the callee context to summarize in-context, single-object heap nodes without affecting out-of-context flagged heap nodes that can reach objects in the callee's reachability graph.

The analysis maps (1) the newest single-object heap node for an allocation site that is out of the callee’s context to the special single-object out-of-context heap node and (2) all other nodes for the allocation site that are out of the callee’s context for the heap node to the special summary out-of-context heap node. The analysis stores this mapping for use in the splicing step. These special out-of-context nodes serve as placeholders to track changes to the reachability of out-of-context edges.

5.3 Merge Graphs

The analysis merges the subgraphs from all calling contexts using the join operation from Section 4.8 to generate G_{merge} . The analysis of the callee operates on G_{merge} .

5.4 Predicates

The interprocedural analysis extends all nodes, edges, and reachability states in G_{merge} with a set of predicates. These predicates are included to prevent nodes and edges from escaping to the wrong call site and are used to correctly propagate reachability states in the caller. Predicates are comprised of the following atomic predicates, which can be combined with logical disjunction (or):

- Edge e exists with reachability state ϕ in G_{sub} of the caller
- Node n exists with reachability state ϕ in G_{sub} of the caller
- Edge e exists in G_{sub} of caller
- Node n exists in G_{sub} of caller
- *true*

The caller analysis begins by initializing the predicates for all nodes, edges, and reachability states to tautologies. For example, the initial predicate for a node n is that the node n exists in the caller — this prevents node n from escaping to the wrong call site. The initial predicate for a reachability state ϕ on node n is that node n exists in G_{sub} of the caller with reachability state ϕ .

Store operations can change the reachability states of both edges and nodes. When the propagation of a change set creates a new reachability state on a node or an edge, the new state inherits the predicate from the previous state on the node or edge, respectively. Object allocation operations can merge single-object heap nodes into the corresponding summary node. In this case, predicates for the nodes are or’ed together. Likewise, if the operation causes two edges to be merged, their predicates are also or’ed together. Duplicated reachability states may also be merged and their predicates are or’ed together.

Newly created nodes or edges are assigned the *true* predicate.

5.5 Specializing the Graph

The algorithm uses G_{sub} to specialize the callee heap reachability graph G_{callee} . The analysis makes a copy of the heap reachability graph G_{callee} . It then prunes all elements of the graph whose predicates are not satisfied by the caller subgraph G_{sub} . The callee predicates of each heap element in G_{callee} are replaced with the caller predicate for the heap element in G_{sub} that satisfied the callee predicate.

If a reachability state contains out-of-context heap nodes, then the analysis uses the stored mapping to translate the out-of-context heap nodes to caller heap nodes. The stored mapping may map multiple heap nodes to the same out-of-context summary heap node. If the arity of the reachability tuple for an out-of-context heap node was 1, then the analysis generates all permutations of the reachability state using the stored mapping from Section 5.2. If the arity was MANY, the analysis replaces the reachability tuple with a set of reachability tuples that contains one tuple for each heap node that mapped to the out-of-context summary node and that tuple has arity MANY.

5.6 Splice in Subgraph

This step splices the physical graphs together. The placeholder nodes are used to splice references from the caller graph to the callee graph. The placeholder edges are used to splice caller edges into the callee graph.

Finally, the reachability changes are propagated back into the out-of-context heap nodes of the caller reachable portion of the reachability graph. The analysis uses predicates to match the reachability states on the original edges from the out-of-context portion of the caller graph into G_{sub} . The analysis generates a change set for each edge that tracks the out-of-context reachability changes made by the callee. It then solves constraints of the same form as Constraints 4.15 and 4.16 to propagate these changes to upstream portions of the caller graph.

5.7 Merging Heap Nodes

At this point, the graph may have more than one single object heap node or summary heap node for a given allocation site. The algorithm next merges all but the newest single object heap node into the summary heap node. It rewrites all tokens in all reachability states to reflect this merge, and then updates the arities.

5.8 Global Pruning

Finally, the analysis calls the global pruning algorithm to remove imprecision potentially caused by our treatment of reachability from out-of-context heap nodes.

6 Evaluation

We have implemented the analysis in our compiler and analyzed eight OoJava benchmarks [14]. In OoJava the developer annotates code blocks as tasks that the compiler may decouple from the parent thread. OoJava uses disjointness analysis combined with other analyses to generate a set of runtime dependence checks that parallelize the execution only when it can preserve the behavior of the original sequential code — annotations do not affect the correctness of the program. The analysis and benchmarks are available at <http://demsky.eecs.uci.edu/compiler.php>.

Benchmark	Time (s)	Lines	Speedup
Crypt	1.2	2,035	17.7×
KMeans	3.5	3,220	13.8×
Labyrinth	84.8	4,315	11.1×
MolDyn	13.8	2,299	13.8×
MonteCarlo	4.8	5,669	18.7×
Power	6.2	1,846	20.2×
RayTracer	22.1	2,832	20.0×
Tracking	331.2	4,747	20.2×

Fig. 5. Out-of-order Java Results (24 cores)

6.1 Benchmarks

We analyzed and parallelized the following eight benchmarks. From Java Grande [19] we ported all of the large benchmarks: RayTracer, a ray tracer; MonteCarlo, a Monte Carlo simulation; and MolDyn, a molecular dynamics simulation. We also ported Crypt, an IDEA encryption algorithm, from Java Grande. We ported KMeans, a data clustering benchmark, and Labyrinth, a maze-routing benchmark, from STAMP [20]. Power is a power pricing benchmark from JOlden [21] and Tracking is a vision benchmark from the SDVBS [22]. The benchmarks range from 1,846 to 5,669 lines of code, with an average of 96.3 methods per benchmark.

6.2 Disjoint Reachability Analysis Results

We next examine the reachability properties the analysis extracted for our benchmarks. We expect developers might examine the results for particular program points to learn about possible sharing. For instance, disjoint reachability analysis reported that iterations of the main loop in RayTracer reached a common scratchpad object which prevented OoJava from parallelizing the loop. The information allowed the authors to move the scratchpad object into the loop scope to obtain a parallel implementation.

Section 4.6 states that disjoint reachability analysis does not differentiate between array indices; however, it is common programming practice to store a parallelizable workload in an array. Disjoint reachability analysis can provide sufficient precision for parallelization in such a case by examining the disjoint reachability properties of an object just after it is selected from an array, as shown in the example presented in Section 2. We ported the array-based benchmarks using this simple pattern.

Labyrinth allocates `Grid` data structures in a way that makes it difficult for many heap analyses to determine that the inner loop for finding routes may be parallelized. The main complication is that an array of `Grid` data structures is allocated in a separate method and then `Grid` data structures are reused in each iteration of the inner loop for calculating a route. OoJava flags the allocation of `Grid` root objects; disjoint reachability analysis then determines that, at the work division program point, the objects that comprise a `Grid` data structure are only reachable from exactly one `Grid` data structure root object. Determining statically that different tasks access unique `Grid` objects is likely to be challenging for any static analysis and therefore purely static approaches are likely to fail. Instead, OoJava generates a dynamic check that each task has a

unique `Grid` data structure root object which, combined with the static reachability information, guarantees there are no conflicts on the `Grid` data structures.

Power calculates the price of power in a simulated network; each major branch of the network is modeled with a `Lateral` object that references a tree of `Branch` and `Leaf` objects. Disjoint reachability analysis finds that all `Demand` objects written to during the simulation are reachable from at most one `Lateral` object. This allows OoJava to parallelize tasks by verifying they operate on different `Lateral` objects.

MolDyn allocates a collection of scratchpad data structures and gives one scratchpad data structure to each parallel task. In the main loop a second task aggregates the results by reading all scratchpad objects. OoJava flags the scratchpad root object; disjoint reachability analysis concludes that all objects in the scratchpad data structures accessed by a parallel task are reachable from at most one scratchpad root object. OoJava therefore parallelizes the main computation using a dynamic check on the scratchpad data structures and serializes the aggregation steps.

The benchmarks `Tracking`, `KMeans`, `Crypt`, and `Monte` use different data structures but have a common work division pattern: the main loop consists of a parallel phase followed by an aggregation phase. The results of disjoint reachability analysis for each of these benchmarks correctly inform OoJava that the aggregation phases must be serialized because they have actual data structure conflicts.

6.3 Parallelization Speedups

We used OoJava to analyze and parallelize eight benchmarks on a 2.27 GHz Xeon. OoJava makes queries to disjoint reachability analysis when generating a parallel implementation. We then executed our benchmark applications on a 1.9 GHz 24-core AMD Magny-Cour Opteron with 24 cores and 16 GB of memory. Figure 5 presents the speedups and the time column shows the analysis time. The speedups are relative to the single-threaded Java versions compiled to C using the same compiler.

The significant speedups indicate that disjoint reachability analysis extracts reachability properties with sufficient precision for OoJava to generate efficient parallel implementations. The speedups in `Crypt`, `KMeans`, `Labyrinth` and `MolDyn` are limited by significant sequential dependences. `RayTracer`'s and `MonteCarlo`'s speedups are also limited by sequential dependences. We compared our parallel implementations of `KMeans` and `Labyrinth` to the parallelized TL2 versions included in STAMP, with and without the additional overheads from array bounds checks in our compiler. With array bounds checks our versions of `KMeans` and `Labyrinth` ran $1.70\times$ and $1.51\times$ faster than TL2 versions, respectively, and without the checks they ran $2.62\times$ and $2.08\times$ faster.

To quantify the overhead of our research compiler, we compared the generated code against the OpenJDK JVM 14.0-b16 and GCC 4.1.2. The sequential version of `Crypt` compiled with our compiler ran 4.6% faster than on the JVM. We also developed a C++ version compiled with GCC and found our compiler's version ran 25% slower than the C++ version. Our compiler implements array bounds checking; with array bounds checking disabled, the binary from our compiler runs only 5.4% slower than the C++ binary. We used the optimization flag `-O3` for both the C++ version and the C code generated by our compiler. This is in close agreement with more extensive experiments

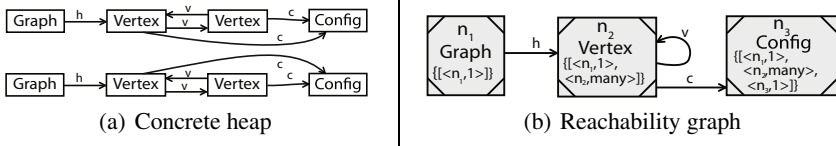


Fig. 6. An example concrete heap with a reachability graph

we performed. Those experiments measured an average overhead for our compiler with array bounds checks disabled of 4.9% relative to GCC.

7 Related Work

We discuss related work in heap analyses, logics, and type systems.

7.1 Shape Analysis

Disjoint reachability analysis discovers properties that are related to but different from those discovered by shape analysis [3–7]. Shape analysis, in general, discovers local properties of heap references and from these properties infers a rich set of derived properties including reachability and disjointness. Where shape analysis can find properties that arise from local invariants, disjoint reachability analysis can find the relative disjointness and reachability properties for any pair of objects. Disjoint reachability analysis complements shape analysis by discovering disjoint reachability properties for arbitrarily complex structures. Calcagno et al. present a shape analysis that focuses on discovering different heap properties [23].

We motivate our discussion of shape analysis with a concrete heap example. Figure 6(a) illustrates a simple concrete heap where a Graph can reach several Vertex objects that all point to a graph-local Config object. We expect that many real programs construct data structures with sharing patterns similar to this example. A possible reachability graph in Figure 6(b) contains enough information to show that Config and Vertex objects are reachable from at most one Graph object. Some shape analyses [3, 4] focus on local shape properties (Is every node in a tree a valid tree node?) and understandably lose precision with the above example or the singleton design pattern. Singleton design patterns include references to globally shared objects. Some parallelizable phases may not even access the shared object, but the presence of a shared object will cause problems for many shape analyses. Our analysis can infer that operations on different graphs that access both Vertex and Config objects may execute in parallel. Note that this result is independent of the relative shape of Vertex objects in the heap.

Marron et al. extend the shape approach for more general heaps with edge-sharing analysis [7, 24]. Their analysis can discover that the Vertex objects from different Graph objects are disjoint. However, their edge-sharing abstraction is localized and thus cannot always resolve non-local reachability properties.

TVLA [5] is a framework for developing shape analysis. Disjointness properties can be written as instrumentation predicates in TVLA, but the system will evaluate them

using the default update rule, providing acceptable results only for trivial examples. To maintain precision, update rules for the disjointness predicates must be supplied, a task that we expect is equivalent in difficulty to disjoint reachability analysis. While TVLA contains reachability predicate update rules, these cannot capture that an object is reachable from exactly one member of a summarized node. Furthermore, the TVLA framework does not scale to the size of our benchmarks.

Separation logic [12] can express that formulas hold on disjoint heap regions. Distefano [25] proposes a shape analysis for linked lists based on separation logic. Raza [26] extends separation logic with assertions to identify statements that can be parallelized. These shape analyses based on separation logic are at an early stage and cannot extract disjoint reachability properties for our examples.

7.2 Alias and Pointer Analysis

Alias analysis [8–10] and pointer analysis [1, 2], like disjoint reachability analysis, analyze source code to discover heap referencing properties. Aiken [11] is similar, but their type system names objects by allocation site and loop iteration. Unlike our analysis, their approach cannot maintain disjointness properties for mutation outside of the allocating loop. Lattner [27] employs a unification-based pointer analysis that scales well, but cannot maintain disjointness properties for data structures that are merged at a later program point. Conditional must not aliasing analysis by Naik and Chatterjee et al. describe a modular points-to analysis that does not extract disjoint reachability properties, but introduces an alternative approach to abstracting caller contexts [28].

7.3 Other Analyses and Type Systems

Sharing analysis [29] computes sharing between variables. Sharing analysis could not determine disjoint reachability properties for the example in Figure 1 of our paper as it would lose information about the relative disjointness of graphs in the array.

Connection analysis discovers which heap-directed pointers may reach a common data structure [30]. There are a finite number of pointers in a program, which implies that connection analysis can only maintain a finite number of disjoint relations. For example, connection analysis cannot determine that all `Graph` objects in our paper’s example reference mutually disjoint sets of `Vertex` objects.

Ownership type systems have been developed to restrict aliasing of heap data structures [31, 32]. Our analysis infers similar properties without requiring annotations.

Craik and Kelly use the property of ownership among objects [33] to discover disjoint ownership contexts, similar to disjoint reachability properties. They do not attempt to track side effects, so their analysis, in comparison to disjoint reachability analysis, can scale better but has significantly less precision.

8 Conclusion

If a compiler can determine that code blocks perform memory accesses that do not conflict, it can safely parallelize them. Traditional pointer analyses have difficulty reasoning about reachability from objects that are abstracted by the same node. We present

disjoint reachability analysis, a new analysis for extracting reachability properties from code. The analysis uses a reachability abstraction to maintain precise reachability information even for multiple objects from the same allocation site. We have implemented the analysis and analyzed eight benchmark programs. The analysis results enabled parallelization of these benchmarks that achieved significant performance improvements.

Acknowledgements

This research was supported by the National Science Foundation under grants CCF-0846195 and CCF-0725350. We thank Mark Marron and the reviewers for feedback.

References

1. Shapiro, M., Horwitz, S.: Fast and accurate flow-insensitive points-to analysis. In: POPL (1997)
2. Landi, W., Ryder, B.G., Zhang, S.: Interprocedural modification side effect analysis with pointer aliasing. In: PLDI (1993)
3. Chase, D.R., Wegman, M., Zadeck, F.K.: Analysis of pointers and structures. In: PLDI (1990)
4. Ghiya, R., Hendren, L.J.: Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In: POPL (1996)
5. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. TOPLAS (2002)
6. McPeak, S., Necula, G.C.: Data structure specifications via local equality axioms. In: Etesami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 476–490. Springer, Heidelberg (2005)
7. Marron, M., Kapur, D., Hermenegildo, M.: Identification of logically related heap regions. In: ISMM (2009)
8. Diwan, A., McKinley, K.S., Moss, J.E.B.: Type-based alias analysis. In: PLDI (1998)
9. Ruf, E.: Partitioning dataflow analyses using types. In: POPL (1997)
10. Deutsch, A.: Interprocedural may-alias analysis for pointers: Beyond k-limiting. In: PLDI (1994)
11. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: POPL (2007)
12. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. LICS (2002)
13. Choi, J.D., Burke, M., Carini, P.: Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In: POPL (1993)
14. Jenista, J.C., Eom, Y., Demsky, B.: OoJava: An out-of-order approach to parallel programming. In: HotPar (2010)
15. Jenista, J.C., Eom, Y., Demsky, B.: OoJava: Software out-of-order execution. In: PPOPP (2011)
16. Zhou, J., Demsky, B.: Bamboo: A data-centric, object-oriented approach to multi-core software. In: PLDI (June 2010)
17. Rinetzky, N., Bauer, J., Reps, T., Sagiv, M., Wilhelm, R.: A semantics for procedure local heaps and its abstractions. In: POPL (2005)
18. Marron, M., Hermenegildo, M., Kapur, D., Stefanovic, D.: Efficient context-sensitive shape analysis with graph based heap models. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 245–259. Springer, Heidelberg (2008)
19. Smith, L.A., Bull, J.M., Obdržálek, J.: A parallel Java Grande benchmark suite. In: SC (2001)
20. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC (2008)

21. Cahoon, B., McKinley, K.S.: Data flow analysis for software prefetching linked data structures in Java. In: PACT (2001)
22. Venkata, S.K., Ahn, I., Jeon, D., Gupta, A., Louie, C., Garcia, S., Belongie, S., Taylor, M.B.: SD-VBS: The San Diego Vision Benchmark Suite. In: IISWC (2009)
23. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL (2009)
24. Marron, M., Méndez-Lojo, M., Hermenegildo, M., Stefanovic, D., Kapur, D.: Sharing analysis of arrays, collections, and recursive structures. In: PASTE (2008)
25. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. LNCS (2006)
26. Raza, M., Calcagno, C., Gardner, P.: Automatic parallelization with separation logic. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 348–362. Springer, Heidelberg (2009)
27. Lattner, C., Adve, V.: Automatic pool allocation: improving performance by controlling data structure layout in the heap. In: PLDI (2005)
28. Chatterjee, R., Ryder, B.G., Landi, W.A.: Relevant context inference. In: POPL (1999)
29. Méndez-Lojo, M., Hermenegildo, M.V.: Precise set sharing analysis for Java-style programs. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 172–187. Springer, Heidelberg (2008)
30. Ghiya, R., Hendren, L.J.: Connection analysis: A practical interprocedural heap analysis for C. IJPP (1996)
31. Clarke, D.G., Drossopoulou, S.: Ownership, Encapsulation and the Disjointness of Type and Effect. In: OOPSLA (2002)
32. Heine, D.L., Lam, M.S.: A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In: PLDI (2003)
33. Craik, A., Kelly, W.: Using ownership to reason about inherent parallelism in object-oriented programs. In: Gupta, R. (ed.) CC 2010. LNCS, vol. 6011, pp. 145–164. Springer, Heidelberg (2010)

Appendix

A Semantics for Intraprocedural Analysis

Define the concrete heap $H = \langle O, R \rangle$ as a set of objects $o \in O$ and a set of references $r \in R \subseteq O \times \{\text{Fields}\} \times O$. We assume a straightforward collecting semantics for the statements in the control flow graph that are relevant to our analysis. The collecting semantics would record the set of concrete heaps that a given statement operates on.

The concrete domain for the abstraction function is a set of concrete heaps $h \in \mathcal{P}(H)$. The abstract domain is defined in Section 3.2. The abstract state is given by the tuple $\langle E, \mathcal{A}^N, \mathcal{A}^E \rangle$, where E is the set of edges, \mathcal{A}^N is the mapping from nodes to their sets of reachability states, and \mathcal{A}^E is the mapping from edges to their sets of reachability states. We next define the lattice for the abstract domain. The bottom element has the empty set of edges E and empty reachability information for both the nodes \mathcal{A}^N and the edges \mathcal{A}^E . The top element for the lattice has (1) all the edges in E that are allowed by type constraints between all reachability nodes, (2) each heap node n has tuples in \mathcal{A}^N for the powerset of all heap nodes that are allowed by types to reach n , and (3) each edge $\langle n, f, n' \rangle \in E$ has the powerset of the maximal set of tuples in \mathcal{A}^E that are allowed by type constraints.

We next define the partial order for the reachability graph lattice. Equation [A.1](#) defines the partial order. The definition for the \sqsubseteq_{Δ} relation between reachability states is given in the Update Edge Reachability step of Section [4.5](#).

$$\langle E, \mathcal{A}^N, \mathcal{A}^E \rangle \sqsubseteq_A \langle E', \mathcal{A}^{N'}, \mathcal{A}^{E'} \rangle \text{ iff } E \subseteq E' \wedge \langle \mathcal{A}^N, \mathcal{A}^E \rangle \sqsubseteq \langle \mathcal{A}^{N'}, \mathcal{A}^{E'} \rangle \quad (\text{A.1})$$

$$\begin{aligned} \langle \mathcal{A}^N, \mathcal{A}^E \rangle \sqsubseteq \langle \mathcal{A}^{N'}, \mathcal{A}^{E'} \rangle &\text{ iff } \forall n \in N, \forall \phi \in \mathcal{A}^N(n), \exists \phi' \in \mathcal{A}^{N'}(n), \\ &\phi \sqsubseteq_{\Delta} \phi' \wedge (\forall \langle n_1, f_1, n_2 \rangle, \dots, \langle n_k, f_k, n \rangle \in E, \\ &\phi \in \mathcal{A}^E(\langle n_1, f_1, n_2 \rangle) \cap \dots \cap \mathcal{A}^E(\langle n_k, f_k, n \rangle) \Rightarrow \\ &\phi' \in \mathcal{A}^{E'}(\langle n_1, f_2, n_2 \rangle) \cap \dots \cap \mathcal{A}^{E'}(\langle n_k, f_k, n \rangle)) \end{aligned} \quad (\text{A.2})$$

The join operation ($\langle E_1, \mathcal{A}^N_1, \mathcal{A}^E_1 \rangle \sqcup \langle E_2, \mathcal{A}^N_2, \mathcal{A}^E_2 \rangle$) on the heap reachability graph lattice simply takes the set unions of the individual components: $\langle E_1 \cup E_2, \mathcal{A}^N_1 \cup \mathcal{A}^N_2, \mathcal{A}^E_1 \cup \mathcal{A}^E_2 \rangle$.

We next define several helper functions. Equation [A.3](#) defines the meaning of the statement that object o is reachable from the object o' in the concrete heap R . We define the object abstraction function $\text{rgn}(o)$ to return the single object heap node for o 's allocation site if o is the most recently allocated object and the allocation site's summary node otherwise. Equation [A.4](#) returns the number of objects abstracted by heap node n_f that can reach the object o . Equation [A.5](#) abstracts the natural numbers into one of three arities. Equation [A.6](#) computes the abstract reachability state for object o in the concrete heap $\langle O, R \rangle$.

$$\begin{aligned} \text{rch}(o', o, R) &= \exists f, o_1, f_1, \dots, o_l, f_l. \langle o', f, o_1 \rangle, \dots, \langle o_i, f_i, o_{i+1} \rangle, \dots, \\ &\quad \langle o_l, f_l, o \rangle \in R \end{aligned} \quad (\text{A.3})$$

$$\text{count}(o, O, R, n_f) = | \{ o' \mid \forall o' \in O. \text{rgn}(o') = n_f, \text{rch}(o', o, R) \} | \quad (\text{A.4})$$

$$\text{abst}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \text{MANY} & \text{otherwise} \end{cases} \quad (\text{A.5})$$

$$\phi(o, O, R) = \{ \langle n_f, \text{abst}(\text{count}(o, O, R, n_f)) \rangle \mid n_f \in N_F \} \quad (\text{A.6})$$

We next define abstraction functions that return the most precise reachability graph for the set of concrete heaps $h \subseteq \mathcal{P}(H)$. We use the standard subset partial ordering relation for our concrete domain of sets of concrete heaps. Equation [A.7](#) generates the edge abstraction, Equation [A.8](#) generates the reachability state abstraction for each node, and Equation [A.9](#) generates the reachability state abstraction for each edge. Note that from the form of the definition of the abstraction function, we can see that it is monotonic. We mechanically synthesize a concretization function $\gamma(\langle E, \mathcal{A}^N, \mathcal{A}^E \rangle) = \sqcup \{ h \mid \alpha(h) \sqsubseteq \langle E, \mathcal{A}^N, \mathcal{A}^E \rangle \}$ to create a Galois connection. The pair α and γ do not form a Galois insertion as two abstract reachability graphs can have the exact same set of concretizations. The global pruning algorithm addresses the practical effects on analysis precision of this issue by converting abstract reachability graphs into equivalent graphs that contain locally more precise reachability states.

$$\alpha_E(h) = \{\langle \text{rgn}(o), f, \text{rgn}(o') \rangle \mid \forall \langle o, f, o' \rangle \in R, \forall \langle O, R \rangle \in h\} \quad (\text{A.7})$$

$$\alpha_{\mathcal{A}^N}(h) = \{\langle \text{rgn}(o), \phi(o, O, R) \rangle \mid \forall o \in O, \forall \langle O, R \rangle \in h\} \quad (\text{A.8})$$

$$\alpha_{\mathcal{A}^E}(h) = \{\langle \langle \text{rgn}(o'), f, \text{rgn}(o'') \rangle, \phi(o, O, R) \rangle \mid \forall o \in O, \forall \langle o', f, o'' \rangle \in R, \forall \langle O, R \rangle \in h.\text{rch}(o'', o, R)\} \quad (\text{A.9})$$

B Termination

Termination of the analysis is straightforward. Reachability graphs form a lattice, and for a given set of allocation sites the lattice has a finite height. All transfer functions in the analysis are monotonic except stores with strong updates and method calls. With a simple modification to enforce monotonicity the analysis will terminate.

Our approach to enforcing monotonicity is to store the latest reachability graph result for every back edge and program point after a method call. The fixed point interprocedural algorithm takes the join of its normal result with these graphs to ensure the local result becomes no smaller.

C Soundness of the Core Intraprocedural Analysis

In this section, we outline the soundness of the core intraprocedural analysis. For all soundness lemmas, we argue $(\alpha \circ f)(h) \sqsubseteq_A (f^\# \circ \alpha)(h)$, where f represents the concrete operation and $f^\#$ is the corresponding transfer function on the abstract domain, to show soundness.

Lemma 1 (Soundness of Copy Statement Transfer Function). *The transfer function for the copy statement $x=y$ is sound with respect to the concrete copy operation.*

Proof Sketch: The soundness of the transfer function for the copy statement $x=y$ is straightforward. After the execution of the copy statement on the concrete heap, the variable x references the object that y referenced before the statement. We note that applying the abstraction function after the concrete copy statement yields the exact same abstract reachability graph as applying the abstraction function followed by the transfer function for the copy statement, therefore the copy transfer function is sound.

Lemma 2 (Soundness of Load Statement Transfer Function). *The transfer function for the load statement $x=y.f$ is sound with respect to the concrete load operation.*

Proof Sketch: The soundness of the transfer function for the load statement $x=y.f$ is also relatively straightforward. After the execution of the load statement on the concrete heap, the variable x references the object referenced by the f field of the object referenced by y . After abstraction, the edge for x would reference the same objects as the f field of the objects referenced by y and have the same reachability set.

The soundness of the edge set transform follows from the definition of α_E — all objects that $y.f$ could possibly reference are included in the set $E_n(y, f)$. Therefore, applying the abstraction function followed by removing the previous edges for x and adding the set of edges $\{x\} \times E_n(y)$ gives an E set that contains all of the edges generated by applying the transfer function and then abstraction function.

From the definition of $\alpha_{\mathcal{A}^E}$ we can determine that for each n that could abstract the object referenced by y and each corresponding n' that could abstract the object referenced by $y.f$, that the reference $y.f$ could only reach objects with reachability states included in the set $\mathcal{A}^E(\langle y, n \rangle) \cap \mathcal{A}^E(\langle n, f, n' \rangle)$. Note the subtle point that the correctness of the intersection operation follows from the edge reachability aspect of the abstraction function definition (and not from the lattice ordering) — there must exist a path through the y reference and $y.f$ to any objects that can be reached by the new x and by the abstraction function both y and $y.f$ will include the reachability states of those objects. Therefore, the application of the abstraction function followed by the transfer function generates a set of reachability states for edges of y that include all of the reachability states generated by applying the concrete load statement followed by the abstraction function.

Lemma 3 (Soundness of Allocation Statement Transfer Function). *The transfer function for the allocation statement $x = \text{new}$ is sound with respect to the concrete allocation operation.*

Proof Sketch: The transfer function for the allocation statement is similarly straightforward. The execution of the allocation statement on the concrete heap followed by the abstraction function yields an abstract reachability graph in which the previous newest allocated object at the site is now mapped to the summary node. The allocation statement transfer function applied to the abstraction function yields the exact same reachability graph and therefore the transfer function is sound.

If the allocation site is flagged, the new heap node has a single reachability state that contains a single reachability token with its own heap node and the arity 1. The variable edge contains the same set of reachability states. If the allocation site is not flagged, the sets of reachability states contains only the empty reachability state.

Lemma 4 (Soundness of Store Statement Transfer Function). *The transfer function for the allocation statement $x.f = y$ is sound with respect to the concrete store operation.*

Proof Sketch: We define o_x to be the concrete object referenced by x and o_y to be the concrete object referenced by y . The store operation can only add new paths in the concrete heap that include the newly created reference $\langle o_x, f, o_y \rangle$. In the abstraction, $E_n(x)$ gives the heap nodes that abstract the objects that x may reference and $E_n(y)$ gives the heap nodes that abstract the objects that y may reference. The concrete operation $x.f = y$ creates a reference from the f field of the object that x references to the object that y references. Applying the abstraction function, the creation of this new reference in all concrete heaps represented by the abstract heap adds a set of edges $E_{\text{new}} \subseteq E_n(x) \times \{f\} \times E_n(y)$ to the abstract heap. Since the application of the transfer function to the initial abstraction adds a larger set of edges, it generates an abstract edge set that is higher in the partial order and therefore our treatment of edges in the store statement is sound.

We next discuss the soundness of the transfer function with respect to the reachability states for nodes. We note that the addition of the concrete reference can only (1) introduce new reachability from objects that could reach o_x to objects that o_y can reach and (2) allow edges that could reach o_x to reach objects that o_y can reach. The set Ψ_x

defined in Equation 4.10 abstracts the reachability states for the objects that can reach o_x by the abstraction function. Similarly, Ψ_y from Equation 4.11 abstracts the allocation sites for the objects that can reach the objects downstream of o_y .

By the abstraction function and the partial order, if an object abstracted by a heap node $n_y \in E_n(y)$ can reach an object abstracted by the heap node n' with the abstract reachability state ϕ , then there must exist a path of edges from n_y to $n' \in N$ in the abstract reachability graph in which every edge along the path has ϕ in its set of reachability states and n' has ϕ in its set of reachability states and $\phi \in \Psi_y$. By the abstraction function, the set of reachability states $\psi_x \in \Psi_x$ for n_x abstract o_x 's reachability from all objects from flagged nodes. Therefore, the constraints given by Equations 4.12 and 4.13 will propagate the correct reachability change set to n' and Equation 4.14 applies these reachability changes to n' . This implies that the set of reachability states for the nodes is higher or equal in the partial order of reachability graphs to the graph generated by applying a concrete operation followed by abstraction and therefore the node reachability states are sound.

We next discuss soundness with respect to edges that are upstream of the objects downstream of o_y in the pre-transformed concrete heap. Consider an object o abstracted by the heap node n that the store operation changed its reachability state from ϕ to ϕ' . By the abstraction function and partial order function, for any reference in the concrete heap, which we abstract by e , that can reach an object abstracted by the heap node n , there must exist a path of edges from e to n in the pre-transformed heap in which ϕ is in the reachability state of each edge along the path. Therefore, Constraints 4.15 and 4.16 propagate the reachability change tuple $\langle \phi, \phi' \rangle$ to e which Equation 4.19 will then apply to e and all edges along the path from e to e' .

Finally, we discuss soundness with respect to edges upstream of o_x that the newly created edge allows to reach objects downstream of o_y . Consider any upstream reference in the concrete heap, which we abstract by the edge e that can reach an object abstracted by the heap node $n_x \in E_n(x)$ — any reachability state it has for the source object of the store must be abstracted by $\phi \in \Psi_x$ in pre-transformed abstract reachability graph and there must exist a path of edges from e to n_x such that ϕ is in the reachability state of every edge along the path. Therefore, Constraints 4.17 and 4.18 propagate the new reachability change tuples $\{\langle \phi, \phi \cup \psi_y \rangle \mid \psi_y \in \Psi_y\}$ to e and Equation 4.19 will then apply the change tuple to e .

At this point, only the new edge remains. Constraint 4.20 simply copies the reachability states from the edge for y whose reachability must be the same. It eliminates reachability states that are smaller in the partial order than any state in the source node as they must be redundant with some larger state. The previous three paragraphs imply that the set of reachability states for edges are higher or equal in the partial order of reachability graphs to the graph generated by applying a concrete operation followed by abstraction and therefore the edge reachability states are sound.

Lemma 5 (Soundness of Global Pruning Transformation). *The global pruning transformation is sound (it generates an abstraction that abstracts the same concrete heaps).*

Proof Sketch: We begin by overviewing the soundness of the first phase of the global pruning algorithm. Consider a flagged heap node n_f and a node n that contains n_f in its

reachability state ϕ with non-0 arity. From the abstraction function and partial order, if there is no path from n_f to n with ϕ in each edge's set of reachability states, then objects in the reachability state ϕ cannot be reachable from objects abstracted by n_f . Therefore, removing n_f from the reachability set ϕ on n and adding this new reachability set to all edges that (1) have ϕ in their reachability state and (2) have a path to n in which all edges along the path have ϕ in their reachability state generates an abstract reachability graph that abstracts the same concrete heaps and therefore the first phase is sound.

We next discuss the soundness of the second phase. Consider an edge e with a reachability state ϕ . If there is no path from edge e to some node n with all edges along the path containing ϕ in their sets of reachability states and node n including ϕ in its set of reachability states, then dropping ϕ from edge e 's set of reachability states yields an abstract state that abstracts the same concrete heaps because if a reference abstracted by e could actually reach an object with the reachability state ϕ then the path would exist. Therefore, the second phase is sound.

D Interprocedural Analysis

We next outline the soundness of the interprocedural analysis. There is a small issue in the interprocedural analysis with the abstraction function for single-object heap nodes. It is possible to have a callee method that only conditionally allocates an object at an allocation site that the caller has a single-object heap node for. The mapping procedure will then merge the caller's single-object heap node into the summary node even though it may abstract the most recently allocated object from the site. One can see that this does not pose a correctness issue through a simple transform of the program that adds a special instruction at each method return that allocates an unreachable object at the given allocation site if the callee did not. It is straightforward to see that such a transform preserves the semantics of the program because it does not change the reachable runtime object graph and after this transform the abstract semantics exactly match the concrete program.

We outline the soundness of the interprocedural analysis by analogy to the intraprocedural analysis with inlining. We note that the callee operates on a graph that is a superset of the callee reachable part of the heap. If we consider only those elements that are in the callee reachable part of the heap, the analysis (1) generates a reachability subgraph that is greater in the partial order than the reachability graph that the inlined version would have and (2) all of those elements get mapped to the caller's heap. We note that reachability state changes on the placeholder edges and edges from placeholder nodes summarize the reachability changes of upstream edges and are sound for the same reasons as the store transfer function.

Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures

Tom Henretty¹, Kevin Stock¹, Louis-Noël Pouchet¹, Franz Franchetti²,
J. Ramanujam³, and P. Sadayappan¹

¹ The Ohio State University

² Carnegie Mellon University

³ Louisiana State University

Abstract. Stencil computations are at the core of applications in many domains such as computational electromagnetics, image processing, and partial differential equation solvers used in a variety of scientific and engineering applications. Short-vector SIMD instruction sets such as SSE and VMX provide a promising and widely available avenue for enhancing performance on modern processors. However a fundamental memory stream alignment issue limits achieved performance with stencil computations on modern short SIMD architectures. In this paper, we propose a novel data layout transformation that avoids the stream alignment conflict, along with a static analysis technique for determining where this transformation is applicable. Significant performance increases are demonstrated for a variety of stencil codes on three modern SIMD-capable processors.

1 Introduction

Short vector SIMD extensions are included in all major high-performance CPUs. While ubiquitous, the ISA and performance characteristics vary from vendor to vendor and across hardware generations. For instance, Intel has introduced SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, and LRBni ISA extensions over the years. With every processor, the latency and throughput numbers of instructions in these extensions change. IBM, Freescale, and Motorola have introduced AltiVec, VMX, VMX128, VSX, Cell SPU, PowerXCell 8i SPU SIMD implementations. In some instances (RoadRunner, BlueGene/L), custom ISA extensions were designed since the supercomputing installation was big enough to warrant such an investment. These extensions provide from 2-way adds and multiplies up to 16-way fused multiply-add operations, promising significant speed-up. It is therefore important to optimize for these extensions.

Stencil computations represent an important class, occurring in many image processing applications, computational electromagnetics and solution of PDEs using finite difference or finite volume discretizations. There has been considerable recent interest in optimization of stencil computations [7], [6], [16], [17], [26], [25], [11], [34], [10], [4], [9], [37], [35], [38], [8], [36], [40], [24], [21], [31]. In this paper, we focus on the problem of optimizing single-core performance on modern processors with SIMD instruction sets. Stencil computations are readily expressible in a form with vectorizable innermost loops where arrays are accessed at unit stride. However, as elaborated upon

later, there is a fundamental performance limiting factor with all current short-vector SIMD instruction sets such as SSE, AVX, VMX, etc. We formalize the problem through the abstraction of stream alignment conflicts. We note that the alignment conflict issue we formulate and solve in this paper pertains to algorithmic alignment constraints and is distinctly different from the previously studied topic of efficient code generation on SIMD architectures with hardware alignment constraints [22,12,41,13]. We address the problem of resolving stream alignment conflict through the novel use of a nonlinear data layout transformation and develop a compiler framework to identify and suitably transform the computations.

<pre> for (t = 0; t < T; ++t) { for (i = 0; i < N; ++i) for (j = 1; j < N+1; ++j) S1: C[i][j] = A[i][j] + A[i][j-1]; for (i = 0; i < N; ++i) for (j = 1; j < N+1; ++j) S2: A[i][j] = C[i][j] + C[i][j-1]; } </pre>	<pre> for (t = 0; t < T; ++t) { for (i = 0; i < N; ++i) for (j = 0; j < N; ++j) S3: C[i][j] = A[i][j] + B[i][j]; for (i = 0; i < N; ++i) for (j = 0; j < N; ++j) S4: A[i][j] = B[i][j] + C[i][j]; } </pre>														
<table border="1"> <tbody> <tr> <td rowspan="3">Performance:</td> <td>AMD Phenom</td> <td>1.2 GFlop/s</td> </tr> <tr> <td>Core2</td> <td>3.5 GFlop/s</td> </tr> <tr> <td>Core i7</td> <td>4.1 GFlop/s</td> </tr> </tbody> </table> <p>(a) Stencil code</p>	Performance:	AMD Phenom	1.2 GFlop/s	Core2	3.5 GFlop/s	Core i7	4.1 GFlop/s	<table border="1"> <tbody> <tr> <td rowspan="3">Performance:</td> <td>AMD Phenom</td> <td>1.9 GFlop/s</td> </tr> <tr> <td>Core2</td> <td>6.0 GFlop/s</td> </tr> <tr> <td>Core i7</td> <td>6.7 GFlop/s</td> </tr> </tbody> </table> <p>(b) Non-Stencil code</p>	Performance:	AMD Phenom	1.9 GFlop/s	Core2	6.0 GFlop/s	Core i7	6.7 GFlop/s
Performance:		AMD Phenom	1.2 GFlop/s												
		Core2	3.5 GFlop/s												
	Core i7	4.1 GFlop/s													
Performance:	AMD Phenom	1.9 GFlop/s													
	Core2	6.0 GFlop/s													
	Core i7	6.7 GFlop/s													

Fig. 1. Example to illustrate addressed problem: The stencil code (a) has much lower performance than the non-stencil code (b) despite accessing 50% fewer data elements

We use a simple example to illustrate the problem addressed. Consider the two sets of loop computations in Fig. 1(a) and Fig. 1(b), respectively. With the code shown in Fig. 1(a), for each of the two statements, $2 \times N^2 + N$ distinct data elements are referenced, with N^2 elements of C and $N^2 + N$ elements of A being referenced in S1, and N^2 elements of A and $N^2 + N$ elements of C being referenced in S2. With the code shown in Fig. 1(b), each of S3 and S4 access $3 \times N^2$ distinct data elements, so that the code accesses around 1.5 times as many data elements as the code in Fig. 1(a). Both codes compute exactly the same number ($2 \times N^2$) of floating point operations. Fig. 1 shows the achieved single-core performance of the two codes on three different architectures, compiled using the latest version of ICC with auto-vectorization enabled. It may be seen that on all systems, the code in Fig. 1(b) achieves significantly higher performance although it requires the access of roughly 50% more data elements.

As explained in the next section, the reason for the lower performance of the stencil code in Fig. 1(a) is that adjacent data elements (stored as adjacent words in memory) from arrays A and C must be added together, while the data elements that are added together in the code of Fig. 1(b) come from independent arrays. In the latter case, we can view the inner loop as representing the addition of corresponding elements from two independent streams $B[i][0:N-1]$ and $C[i][0:N-1]$, but for the former, we are adding shifted versions of data streams: $A[i][0:N-1]$, $A[i][1:N]$, $C[i][0:N-1]$, and $C[i][1:N]$. Loading vector registers in this case requires use of either (a) redundant loads, where a data element is moved with a different load for each distinct vector register position it needs to be used in, or (b) load operations followed by additional

inter- and intra-register movement operations to get each data element into the different vector register slots where it is used. Thus the issue we address is distinctly different from the problem of hardware alignment that has been addressed in a number of previous works. The problem we address manifests itself even on architectures where hardware alignment is not necessary and imposes no significant penalty (as for example the recent Intel Core i7 architecture).

In this paper, we make the following contributions. (1) We identify a fundamental performance bottleneck for stencil computations on all short-vector SIMD architectures and develop a novel approach to overcoming the problem via data layout transformation. (2) We formalize the problem in terms of stream alignment conflicts and present a compile-time approach to identifying vectorizable computations that can benefit from the data layout transformation. (3) We present experimental results on three target platforms that demonstrate the effectiveness of the transformation approach presented in this paper. To the best of our knowledge, this is the first work to identify and formalize this problem with *algorithmic* stream alignment conflicts and provide a solution to it.

The paper is structured as follows. In the next section, we elaborate on the addressed problem. In Sec. 3, we provide an overview of the data layout transformation approach through examples. Sec. 4 presents the formalization and compile-time analysis for detecting stream alignment conflicts. Sec. 5 presents experimental methodology and results. Related work is covered in Sec. 6 and we conclude in Sec. 7.

2 Background and Overview of Approach

2.1 Illustrative Example

The reason for the significant performance difference between the two codes shown in Sec. 1, is that one of them (Fig. 1(a)) exhibits what may be called *stream alignment conflict*, while the other (Fig. 1(b)) is free of such conflicts. When stream alignment conflicts exist, the compiler must generate additional loads or inter-register data movement instructions (such as shuffles) in order to get interacting data elements into the same vector register positions before performing vector arithmetic operations. These additional data movement operations cause the performance degradation seen in Fig. 1. Fig. 2 shows the x86 assembly instructions generated by the Intel ICC compiler on a Core 2 Quad system for code similar to that in Fig. 1(a), along with an illustration of the grouping of elements into vectors. The first four iterations of inner loop j perform addition on the pairs of elements $(B[0], B[1])$, $(B[1], B[2])$, $(B[2], B[3])$, and $(B[3], B[4])$. Four single-precision floating-point additions can be performed by a single SSE SIMD vector instruction (`addps`), but the corresponding data elements to be added must be located at the same position in two vector registers. To accomplish this, ICC first loads vectors $B[0:3]$ and $B[4:7]$ into registers `xmm1` and `xmm2`. Next, a copy of vector $B[4:7]$ is placed into register `xmm3`. Registers `xmm1` and `xmm3` are then combined with the `pslignr` instruction to produce the unaligned vector $B[1:4]$ in register `xmm3`. This value is used by the `addps` instruction to produce an updated value for $A[0:3]$ in `xmm3`. Finally, an aligned store updates $A[0:3]$. The vector $B[4:7]$ in register `xmm2` is ready to be used for the update of $A[4:7]$.

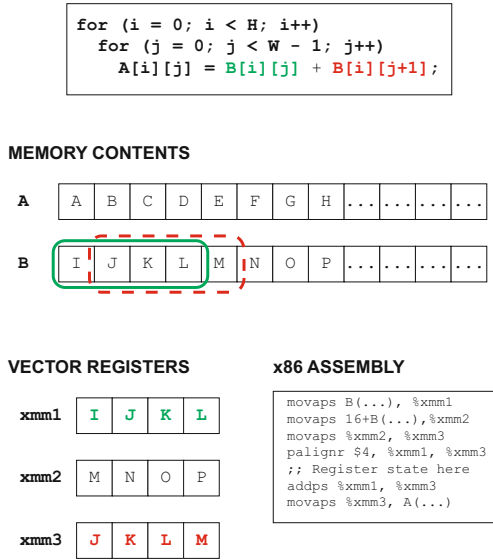


Fig. 2. Illustration of additional data movement for stencil operations

2.2 Stream Alignment Conflict

We now use a number of examples to explain the issue of stream alignment conflicts. Before we proceed, we reiterate that the issue we address is a more fundamental algorithmic data access constraint and is not directly related to the hardware alignment restrictions and penalties on some SIMD instruction set architectures. For example, on IBM Power series architectures using the VMX SIMD ISA, vector loads and stores must be aligned to quadword boundaries. On x86 architectures, unaligned vector loads/stores are permitted but may have a significant performance penalty, as on the Core 2 architecture. On the more recent Core i7 x86 architecture, there is very little penalty for unaligned loads versus aligned loads. The performance difference on the Core i7 shown in Fig. 1, however, provides evidence that the problem we address is a more fundamental algorithmic alignment issue. While hardware alignment restrictions/penalties on a target platform may exacerbate the problem with stream alignment conflicts, the problem exists even if an architecture has absolutely no restrictions/penalties for unaligned loads/stores. The work we present in this paper thus addresses a distinctly different problem than that addressed by many other works on optimizing code generation for SIMD vector architectures with hardware alignment constraints.

Vectorizable computations in innermost loops may be viewed as operations on streams corresponding to contiguous data elements in memory. The computation in the inner loop with statement S1 in Fig. 1(b) performs the computation $C[i][0:N-1] = A[i][0:N-1] + B[i][0:N-1]$, i.e. the stream of N contiguous data elements $A[i][0:N-1]$ is added to the stream of N contiguous elements $B[i][0:N-1]$ and the resulting stream is stored in $C[0][0:N-1]$. In contrast, the inner loop with statement S1 in Fig. 1(a) adds $A[i][1:N]$ and $A[i][0:N-1]$, where these two streams of length

N are subsets of the same stream $A[i][0:N]$ of length $N+1$, but one is shifted with respect to the other by one element. When such shifted streams are computed upon using current short-vector SIMD architectures, although only $N+1$ distinct elements are accessed, essentially $2 \times N$ data moves are required (either through additional explicit loads or inter-register data movement operations like shuffles). The extra moves are required because of the inherent characteristic of short-vector SIMD instructions that only elements in corresponding slots of vector registers can be operated upon.

While in the example of Fig. 3(a) the stream alignment conflict is apparent because the misaligned streams arise from two uses of the same array in the same statement, the same underlying conflict may arise more indirectly, as illustrated in Fig. 3.

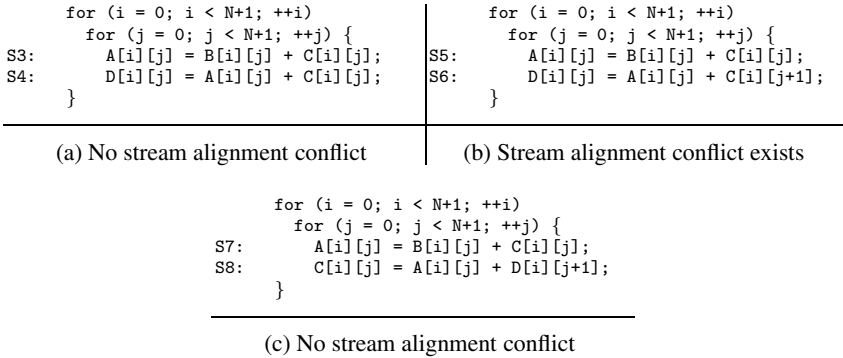


Fig. 3. Illustration of indirect occurrence of stream alignment conflict

In Fig. 3(a), the stream computations corresponding to the inner loop j are $A[i][0:N] = B[i][0:N] + C[i][0:N]$ for S3 and $D[i][0:N] = A[i][0:N] + C[i][0:N]$ for S4. Streams $A[i][0:N]$ and $C[i][0:N]$ are used twice, but all accesses are mutually aligned. With the example shown in Fig. 3(b), however, the stream computations corresponding to the inner j loop are $A[i][0:N] = B[i][0:N] + C[i][0:N]$ for S5 and $D[i][0:N] = A[i][0:N] + C[i][1:N+1]$ for S6. Here we have a fundamental stream alignment conflict. If $A[i][0:N]$ and $C[i][0:N]$ are aligned together for S5, there is a misalignment between $A[i][0:N]$ and $C[i][1:N+1]$ in S6. Finally, there is no stream alignment conflict in Fig. 3(c) since the same alignment of $A[i][0:N]$ with $C[i][0:N]$ is needed for both S7 and S8.

The abstraction of stream alignment conflicts is applicable with more general array indexing expressions, as illustrated by the examples in Fig. 4. In Fig. 4(a), there is no stream alignment conflict. In S9, streams $B[i][i:i+N]$ and $B[i+1][i:i+N]$ (the second and third operands) need to be aligned. Exactly the same alignment is needed between these streams in S10 (the first operand $B[i][i+1:i+N+1]$ and the second operand $B[i+1][i+1:i+N+1]$).

In contrast, in Fig. 4(b) a fundamental algorithmic stream alignment conflict exists with the reused streams in S11 and S12. In S11, the stream $B[i][i:i+N]$ (second operand) must be aligned with $B[i+1][i+1:i+N+1]$ (third operand), but in S12 stream

<pre> for (i = 1; i < N+1; ++i) for (j = 0; j < N+1; ++j) { S9: A[i][j] = B[i-1][i+j] + B[i][i+j] + B[i+1][i+j]; S10: A[i+1][j] = B[i][i+j+1] + B[i+1][i+j+1] + B[i+2][i+j+1]; } </pre>	<pre> for (i = 1; i < N+1; ++i) for (j = 0; j < N+1; ++j) { S11: A[i][j] = B[i-1][i+j+1] + B[i][i+j] + B[i+1][i+j+1]; S12: A[i+1][j] = B[i][i+j+2] + B[i+1][i+j+1] + B[i+2][i+j+2]; } </pre>
(a) No stream alignment conflict	(b) Stream alignment conflict exists

Fig. 4. Illustration of stream alignment conflict with general array indexing expressions

$B[i][i+2:i+N+2]$ (first operand) must be aligned with $B[i+1][i+1:i+N+1]$ (second operand). Thus the required alignments between the reused streams in S11 and S12 are inconsistent — a +1 shift of $B[i][x:y]$ relative to $B[i+1][x:y]$ is needed for S11 but a -1 shift of $B[i][x:y]$ relative to $B[i+1][x:y]$ is needed for S12 i.e., a stream alignment conflict exists.

A formalization and algorithm for compile-time characterization of the occurrences of stream alignment conflicts is developed in Sec. 4. Before providing those details, we present a novel approach through data layout transformation to avoid performance degradation due to stream alignment conflicts.

3 Data Layout Transformation

In this section, we show how the poor vectorization resulting from stream alignment conflicts can be overcome through data layout transformation. As explained in the previous section using Fig. 2, the main problem is that adjacent elements in memory map to adjacent slots in vector registers, so that vector operations upon such adjacent elements cannot possibly be performed without either performing another load operation from memory to vector register or performing some inter-register data movement. The key idea behind the proposed data layout transformation is for potentially interacting data elements to be relocated so as to map to the same vector register slot.

3.1 Dimension-Lifted Transposition

We explain the data layout transformation using the example in Fig. 5. Consider a one dimensional array Y with 24 single-precision floating point data elements, shown in Fig. 5(a), used in a computation with a stream alignment conflict, such as $Z[i] = Y[i-1] + Y[i] + Y[i+1]$.

Fig. 5(b) shows the same set of data elements from a different logical view, as a two-dimensional 4×6 matrix. With row-major ordering used by C, such a 2D matrix will have exactly the same memory layout as the 1D matrix Y . Fig. 5(c) shows a 2D matrix that is the transpose of the 2D matrix in Fig. 5(b), i.e., a dimension-lifted transpose of the original 1D matrix in Fig. 5(a). Finally, Fig. 5(d) shows a 1D view of the 2D matrix in Fig. 5(c).

It can be seen that the data elements A and B , originally located in adjacent memory locations $Y[0]$ and $Y[1]$, are now spaced farther apart in memory, both being in column

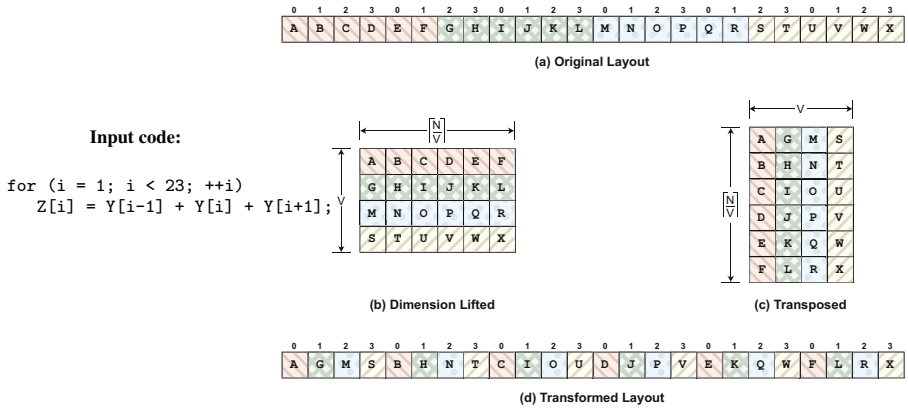


Fig. 5. Data layout transformation for SIMD vector length of 4

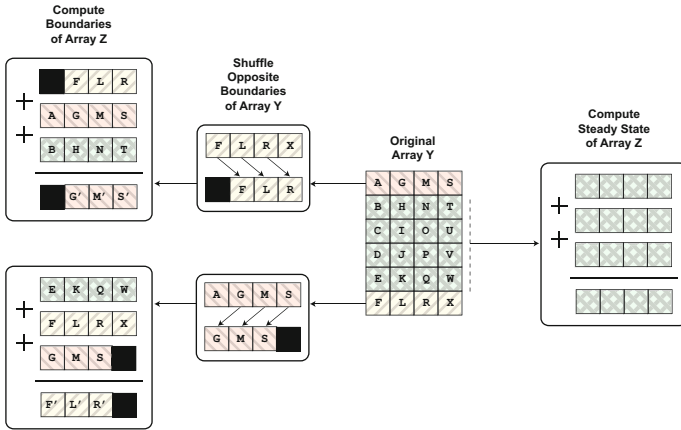


Fig. 6. Illustration of steady state and boundary computation

zero but in different rows of the transposed matrix shown in Fig. 5(c). Similarly, data elements G and H that were adjacent to each other in the original layout are now in the same column but different rows of the dimension-lifted transposed layout. The layout in Fig. 5(c) shows how elements would map to slots in vector registers that hold four elements each. The computation of $A+B$, $G+H$, $M+N$, and $S+T$ can be performed using a vector operation after loading contiguous elements $[A,G,M,S]$ and $[B,H,N,T]$ into vector registers.

3.2 Stencil Computations on Transformed Layout

Fig. 6 provides greater detail on the computation using the transformed layout after a dimension-lifted transposition. Again, consider the following computation:

```

for (i = 1; i < 23; ++i)
    Sb:    Z[i] = Y[i-1] + Y[i] + Y[i+1];
    
```

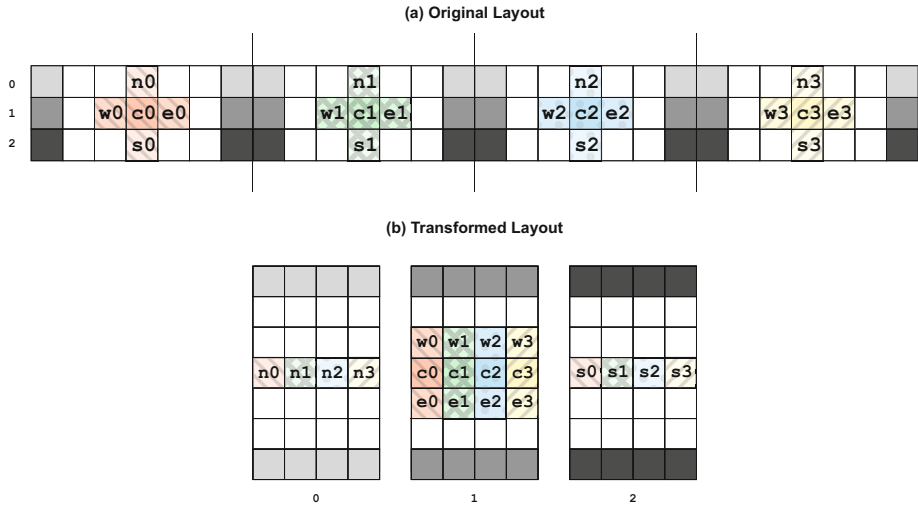


Fig. 7. Illustration for 2D 5-point stencil

Sixteen of the twenty two instances of statement **Sb** can be computed using full vector operations: $[A,G,M,S] + [B,H,N,T] + [C,I,O,U]$; $[B,H,N,T] + [C,I,O,U] + [D,J,P,V]$; $[C,I,O,U] + [D,J,P,V] + [E,K,Q,W]$; and $[D,J,P,V] + [E,K,Q,W] + [F,L,R,X]$. In Fig. 6 these fully vectorized computations are referred to as the “steady state”. Six statement instances (computing $E+F+G$, $F+G+H$, $K+L+M$, $L+M+N$, $Q+R+S$, and $R+S+T$) represent “boundary” cases since the operand sets do not constitute full vectors of size four. One possibility is to perform the operations for these boundary cases as scalar operations, however, it is possible to use masked vector operations to perform them more efficiently. Fig. 6 illustrates the boundary cases corresponding to elements at the top and bottom rows in the transposed layout.

At the top boundary, a “ghost cell” is created by performing a right-shift to the bottom row vector $[F,L,R,X]$ by one position to align $[X,F,L,R]$ above $[A,G,M,S]$. The vector operation $[X,F,L,R]+[A,G,M,S]+[B,H,N,T]$ is then performed and a masked write is performed so that only the last three components of the result vector get written. The bottom boundary is handled similarly, as illustrated in Fig. 6.

Higher order stencils simply result in a larger boundary area. Further, the dimension-lifted transpose layout transformation can be applied in the same manner for multi-dimensional arrays, as illustrated in Fig. 7. The fastest varying dimension (rightmost dimension for C arrays) of a multi-dimensional array is subjected to dimension-lifting and transposition (with padding if the size of that dimension is not a perfect multiple of vector length). Fig. 7 highlights a set of array locations accessed by a 2D 5-point stencil computation on a four wide vector architecture.

In the following section, we present a compiler framework to automatically detect where dimension lifting can be used to overcome the performance loss due to stream alignment conflicts.

4 Framework for Stream Alignment Conflict

Stream alignment conflicts often occur when vectorizing stencil computations, as shown in the examples of Section 2. Our goal in this section is to develop compiler algorithms for detecting stream alignment conflicts. Note that stream alignment conflict is not limited to stencil computations; therefore, dimension-lifted transposition is useful for a more general class of programs. In this paper, we limit the discussion to vectorizable innermost loops. We assume that high-level transformations to expose parallel innermost loops that are good vectorization candidates have already been applied to the code [1]. The first task is to collect all vectorizable innermost loops. Some conditions must hold for our analysis to be performed: (1) all memory references are either scalars or accesses to (multidimensional) arrays; (2) all dependences have been identified; and (3) the innermost loop variable has a unit increment. Note that our framework handles unrolled loops as well; for ease of presentation, we will assume a non-unrolled representation of unrolled loops.

4.1 Program Representation

We first define our representation of candidate loops to be analyzed. The objective is to detect (a) if a loop is vectorizable, and (b) if it has a stream alignment conflict that can be resolved using dimension-lifted transposition of accessed arrays. We operate on an abstract program representation that is independent of the programming language and analyze the memory references of candidate innermost loops.

Innermost loops. Innermost loops are the only candidates we consider for vectorization after dimension-lifted transposition of arrays. As stated earlier, we assume that such loops have a single induction variable that is incremented by one in each iteration of the loop. This is required to ensure the correctness of the analysis we use to characterize the data elements accessed by an innermost loop. We also require the innermost loops to have a single entry and exit point, so that we can precisely determine where to insert the data layout transformation (i.e., dimension-lifted transposition) code.

Memory references. A loop contains a collection of memory references. All data elements accessed by any given iteration of the loop must be statically characterized by an access function f of the enclosing loop indices and other parameters. For a given execution of the loop whose iterator is i , all function parameters beside i must be constant. f can be multidimensional for the case of an array reference, with one dimension in f for each dimension of the array. For instance, for the reference $B[i][alpha+M*j]$ with innermost loop iterator j , the access function is $f_B(j) = (i, alpha + M*j)$.

4.2 Candidate Vector Loops

Dimension-lifted transposition may be useful only for arrays referenced in vectorizable loops. We now provide a simple compiler test to determine which inner loops are candidates for vectorization, based on data dependence analysis of the program. Specifically, we require a candidate innermost loop to meet the following constraints¹: (1) the loop

¹ The case of vectorizing reductions is not addressed here.

does not have any loop-carried dependence; and (2) two consecutive iterations of the loop access either the same memory location or two locations that are contiguous in memory, i.e., the loop has either stride-zero or stride-one accesses.

Loop-Carried Dependences. To determine if a loop is parallel, we rely on the standard concept of dependence distance vectors [19]. There is a dependence between two iterations of a loop if they both access the same memory location and one of these accesses is a write. If there is no loop-carried dependence, then the loop is parallel.

We now define the concept of a *array-distance vector* between two references, restricted to the inner loop (all other variables are assumed constant).

Definition 1 (Array-Distance Vector between two references). Consider two access functions f_A^1 and f_A^2 to the same array A of dimension n . Let ι and ι' be two iterations of the innermost loop. The array-distance vector is defined as the n -dimensional vector $\delta(\iota, \iota')_{f_A^1, f_A^2} = f_A^1(\iota) - f_A^2(\iota')$.

Let us illustrate Definition 1 with two references $A[i][j]$ and $A[i-1][j+1]$, enclosed by loop indices i and j with j as the innermost loop iterator. We have $f_A^1(j) = (i, j)$ and $f_A^2(j) = (i-1, j+1)$. The array-distance vector is

$$\delta(j, j')_{f_A^1, f_A^2} = (1, j - j' - 1)$$

A necessary condition for the existence of a dependence carried by the innermost loop is stated below.

Definition 2 (Loop-carried dependence). There exists a dependence carried by the innermost loop between two references f_A^1 and f_A^2 , at least one of which is a write to memory, if there exists $\iota \neq \iota'$ such that $\delta(\iota, \iota')_{f_A^1, f_A^2} = 0$.

Note that $\delta(\iota, \iota')_{f_A^1, f_A^2}$ requires the values of loop iterators other than the innermost loop to be fixed. Also, when $\delta(\iota, \iota')_{f_A^1, f_A^2}$, the difference between the access functions, still contains symbols beyond the inner-loop iterator (e.g., $\delta(\iota, \iota')_{f_A^1, f_A^2} = (k, \text{alpha})$) we conservatively assume a loop-carried dependence unless the value of the symbols can be determined statically.

Stride-one Memory Accesses. The second condition for a loop to be a candidate for SIMD vectorization is that all memory accesses are such that two consecutive iterations access either consecutive memory locations or an identical location, for all iterations of the innermost loop.

To test for this property, we operate individually on each access function. Memory references that always refer to the same location for all values of the innermost loop are discarded: they are the equivalent to referencing a scalar variable. For each of the remaining references, we form the array-distance vector between two consecutive iterations, and ensure the distance between the two locations is equal to 1. For the case of multidimensional arrays, for the distance in memory to be 1, the array-distance vector must have a zero value in all but the last dimension where it must be one. This is formalized as follows:

Definition 3 (Stride-one memory access for an access function). Consider an access function f_A surrounded by an innermost loop. It has stride-one access if $\forall t, \delta(t, t+1)_{f_A, f_A} = (0, \dots, 0, 1)$.

For example, consider $A[i][j]$ surrounded by the innermost j loop. $f_A = (i, j)$ and $\delta(j, j+1)_{f_A, f_A} = (0, 1)$. Hence the j loop has stride-one access for f_A as above. Let us suppose that for the same f_A , loop i is the innermost loop; in this case, we would have $\delta(i, i+1)_{f_A, f_A} = (1, 0)$ which does not satisfy the condition in Definition 3. Therefore loop i does not have stride-one access with respect to f_A .

4.3 Detection of Stream Alignment Conflict

A stream alignment conflict occurs when a given memory location is required to be mapped to two different vector slots. So, it follows that if a memory element is not used twice or more during the execution of the innermost loop, then a stream alignment conflict cannot occur. If some memory element is reused, one of two cases apply: either the element is reused only during the same iteration of the innermost loop, or it is used by another iteration of the innermost loop. Stream alignment conflict can occur only for the latter; if the element is reused during the same iteration, it is mapped to the same vector slot and there is no stream alignment conflict.

Cross-Iteration Reuse Due to Distinct References. Our principle to detect if a given memory location is being referenced by two different iterations follows the concept of loop-carried dependence, but we extend now to *all pairs* of references and not limit to those containing a write operation.

For a given innermost loop, we thus compute the array-distance vector for all pairs of distinct references f^1 and f^2 to the same array, and see if there exists two distinct iterations such that

$$\delta(t, t')_{f^1, f^2} = 0$$

Consider the example of Figure 8(a). For the case of $f_B^1 = (i, j+1)$ and $f_B^2 = (i, j)$, we can determine if there is cross-iteration reuse by forming the constraint $\delta(j, j')_{f_B^1, f_B^2} = (0, j+1-j')$ and solving the system of constraints

$$S : \begin{cases} j \neq j' \\ 0 = 0 \\ j+1-j' = 0 \end{cases}$$

Since S has a solution, there is cross-iteration reuse in the innermost loop. Technically, the presence of a solution to S is only a necessary condition for cross-iteration reuse. Some cases of cross-iteration reuse may not be detected, typically when distinct variables used in the access functions have the same value at runtime: the *evaluation* of the expression of δ may be 0, but simply doing the difference of the symbols may not give 0. Addressing this issue requires classical data-flow analysis problem for arrays [19] and can be solved with a more complete analysis and under stronger assumptions on the form of the input code. In our framework, the only consequence of not detecting dynamic cross-iteration reuse is that we may not have applied dimension-lifted transposition at places where it could have been useful. Our analysis is conservative in that sense but it requires only minimal assumptions on the input code.

<pre>for (i = lbi; i < ubi; ++i) for (j = lbj; j < ubj; ++j) A[i][j] = B[i][j+1] + B[i][j] + B[i-1][j];</pre>	<pre>for (i = lbi; i < ubi; ++i) for (j = lbj; j < ubj; ++j) { R: A[i][j] += B[i][j-1]; S: C[i][j] += B[i][j]; }</pre>	<pre>for (i = lbi; i < ubi; ++i) { A[i][lbj] += B[i][lbj-1]; for (j = lbj + 1; j < ubj; ++j) { A[i][j] += B[i][j-1]; C[i][j-1] += B[i][j-1]; } C[i][ubj] += B[i][ubj]; }</pre>
(a)	(b)	(c)

Fig. 8. Code examples

Stream Offset. The presence of a cross-iteration reuse does not necessarily imply the presence of a stream alignment conflict. In particular, all cases where cross-iteration reuse can be removed by iteration shifting (that is, a simple offset of the stream) do not correspond to a stream alignment conflict. Thus, in order to detect only the set of arrays to be dimension-lifted and transposed, we need to prune the set of references which do not generate a stream alignment conflict from the set of all references with cross-iteration reuse. .

Consider the example of Figure 8(b) that has cross-iteration reuse. It is possible to modify the innermost loop such that the reuse disappears via *iteration shifting* [19]. Shifting is the iteration space transformation view of manipulating the stream offset, as it influences which specific *instance* of the statements are being executed under the same iteration of loop j . Consider shifting the second statement by 1, the transformed code is shown in Figure 8(c).

Definition 4 (Stream offset via shifting). Consider two statements R and S . Changing the stream offset is performed by shifting the iterations of R with respect to the iterations of S by a constant, scalar factor σ_R . All streams used by R have the same offset σ_R .

There are two important properties of shifting for stream offset. First, as shown above, shifting can make cross-iteration reuse disappear in some cases. It can also introduce new cross-iteration reuse by assigning different offsets to streams associated with the same array.

Second, shifting is not always a legal reordering transformation. It may reorder the statements inside the loop, and change the semantics. This translates to a strong condition on the legality of iteration shifting.

Definition 5 (Legality of iteration shifting). Consider a pair of statements R and S such that they are surrounded by a common vectorizable innermost loop. Let σ_R be (resp. σ_S) the shift factor used to offset the streams of R (resp. S). If there is a dependence between R and S , then to preserve the semantics it is sufficient to set $\sigma_R = \sigma_S$.

To determine if iteration shifting can remove cross-iteration reuse, we first observe that it changes the access functions of a statement R by substituting j with $j - \sigma_R$, given j as the innermost loop iterator. The variable j can be used only in the last dimension of the access function, since the loop is vectorizable with stride-one access. We then formulate a problem similar to that of cross-iteration reuse analysis, with the important difference being that we seek values of σ that make the problem infeasible; indeed if the problem has no solution, then there is no cross-iteration reuse. We also restrict it to the pairs of references such that all but the last dimension of the access functions are

equal, as all cases of reuse require this property. To ease the solution process, we thus reformulate it into a feasibility problem by looking for a solution where $t = t'$ that is independent of the value of t .

Returning to the example in Figure 8(b), we have for B, $f_B^1(j) = (i, j - 1)$ and $f_B^2(j) = (i, j)$. We first integrate the offset factors σ into the access function. As the two statements are not dependent, we have one factor per statement that can be independently computed. The access functions to consider are now $f_B^1(j) = (i, j - \sigma_R - 1)$ and $f_B^2(j) = (i, j - \sigma_S)$. We consider the problem

$$\mathcal{T} : \begin{cases} j = j' \\ j - \sigma_R - 1 - j' + \sigma_S = 0 \end{cases}$$

If \mathcal{T} has a solution for all values of j , that is, a solution independent of j then there is no cross-iteration reuse. For \mathcal{T} , $\sigma_R = 0$ and $\sigma_S = 1$ is a valid solution and this iteration shifting removes all cross-iteration reuse.

In order to find a valid solution for the whole inner-loop, it is necessary to combine all reuse equalities in a single problem: the σ variables are shared for multiple references and must have a unique value. Hence, for code in Figure 8(a), the full system to solve integrates $\delta(j, j')_{f_A^1, f_A^2}$, is augmented with σ_R and σ_S , and is shown below.

$$\mathcal{T} : \begin{cases} j = j' \\ j - \sigma_R - 1 - j' + \sigma_S = 0 & \text{Conditions for B} \\ j - \sigma_R - j' - \sigma_S = 0 & \text{Conditions for A} \end{cases}$$

\mathcal{T} has no solution, showing that there is no possible iteration shifting that can remove all cross-iteration reuse in Figure 8(a). Dimension-lifted transposition is thus required.

Putting It All Together. We now address the general problem of determining if a given innermost loop suffers from a stream alignment conflict that can be solved via dimension-lifted transposition. That is, we look for cross-iteration reuse that cannot be eliminated via iteration shifting.

First, let us step back and precisely define in which cases dimension-lifted transposition can be used to solve a stream alignment conflict. Dimension-lifted transposition in essence spreads out the memory locations of references involved in a stream alignment conflict. In order to ensure there is no conflict remaining, one must precisely know, at compile-time, the distance in memory required to separate all elements. This distance must be a constant along the execution of the innermost loop. This translates to an additional constraint on the cross-iteration reuse that is responsible for the stream alignment conflict: the reuse distance must be a constant. We define the scope of applicability of dimension-lifted transposition as follows.

Definition 6 (Applicability of dimension-lifted transposition). *Consider a collection of statements surrounded by a common vectorizable inner loop. If there exists cross-iteration reuse of a constant distance that cannot be eliminated by iteration shifting, then the stream alignment conflict can be solved with dimension-lifted transposition.*

If an array accessed in a candidate vector inner loop is dimension-lifted-and-transposed, all arrays in the inner loop are also dimension-lifted-and-transposed. The data layout

```

Input P: input program
Output Arrays: the set of arrays to be dimension-lifted

 $\mathcal{L} \leftarrow \emptyset$ 
forall innermost loops  $l$  in  $P$  do
  forall arrays  $A$  referenced in  $l$  do
    /* Check loop-carried dependence */
    forall write references  $f_A^1$  to  $A$  do
      forall references  $f_A^2$  to  $A$  do
         $S \leftarrow \{ \exists (i, i'), \delta(i, i')_{f_A^1, f_A^2} = 0 \}$ 
        if  $S \neq \emptyset$  then goto next loop  $l$ 
    /* Check stride-one */
    forall references  $f_A$  to  $A$  do
       $S \leftarrow \{ \forall i, \delta(i, i + 1)_{f_A, f_A} = (0, \dots, 0, 1) \}$ 
      if  $S = \emptyset$  then goto next loop  $l$ 
    /* Check scalar reuse distance */
    forall pairs of references  $f_A^1, f_A^2$  to  $A$  do
       $S \leftarrow \{ \exists (i, i') \delta(i, i')_{f_A^1, f_A^2} = 0 \}$ 
      if  $S \neq \emptyset$  then
         $S \leftarrow \{ \alpha \in \mathbb{Z}, \delta(i, i)_{f_A^1, f_A^2} = (0, \dots, 0, \alpha) \}$ 
      if  $S = \emptyset$  then goto next loop  $l$ 
   $\mathcal{L} \leftarrow \mathcal{L} \cup l$ 
forall  $l \in \mathcal{L}$  do
  /* Check conflict after iteration shifting */
   $\mathcal{T} \leftarrow \text{createIterationShiftingProblem}(l)$ 
  if  $\mathcal{T} = \emptyset$  then
     $\text{Arrays}(l) \leftarrow$  all arrays in  $l$ 

```

Fig. 9. Algorithm to detect arrays to be dimension-lifted-and-transposed

transformation implies significant changes in the loop control and in the order the data elements are being accessed. All arrays must be dimension-lifted unless some computations simply could not be vectorized anymore. We present in Figure 9 a complete algorithm to detect which arrays are to be transformed by dimension-lifted transposition in a program.

Procedure `createIterationShiftingProblem` creates a system of equalities that integrates the shift factors σ as shown in Section 4.3. If this system has no solution, then at least one cross-iteration reuse remains even after iteration shifting. Since we have prevented the cases where the reuse is not a scalar constant, then the conflict can be solved with dimension lifting. We thus place all arrays of the loop into the list of arrays to be dimension-lifted-and-transposed.

While solving \mathcal{T} , we compute values for σ to remove cross-iteration reuse. When it is not possible to remove all cross-iteration reuses with iteration shifting, we compute values for σ that minimize the distance between two iterations reusing the same element. The largest among all reuse distances in the iteration-shifted program is kept and used during code generation to determine the boundary conditions.

5 Experimental Evaluation

The effectiveness of the dimension-lifting transformation was experimentally evaluated on several hardware platforms using stencil kernels from a variety of application

domains. First, we describe the hardware and compiler infrastructure used for experiments. Next, the stencil kernels used in the experiments are described. Finally, experimental results are presented and analyzed.

5.1 Hardware

We performed experiments on three hardware platforms: AMD Phenom 9850BE, Intel Core 2 Quad Q6600, and Intel Core i7-920. Although all are x86 architectures, as explained below, there are significant differences in performance characteristics for execution of various vector movement and reordering instructions.

Phenom 9850BE. The AMD Phenom 9850BE (*K10h* microarchitecture) is an x86-64 chip clocked at 2.5 GHz. It uses a 128b FP add and 128b FP multiply SIMD units to execute a maximum of 8 single precision FP ops per cycle (20 Gflop/s). The same SIMD units are also used for double precision operations, giving a peak throughput of 10 Gflop/s. Unaligned loads are penalized on this architecture, resulting in half the throughput of aligned loads and an extra cycle of latency. The SSE shuffle instruction `shufps` is used by ICC for single precision inter- and intra-register movement. Double precision stream alignment conflicts are resolved by ICC generating consecutive `movsd` and `movhpd` SSE instructions to load the low and high elements of a vector register.

Core 2 Quad Q6600. The Intel Core 2 Quad Q6600 (*Kentsfield* microarchitecture) is an x86-64 chip running at 2.4 GHz. Like the Phenom, it can issue instructions to two 128-bit add and multiply SIMD units per cycle to compute at a maximum rate of 19.2 single precision GFlop/s (9.6 double precision Gflop/s). The `movups` and `movupd` unaligned load instructions are heavily penalized on this architecture. Aligned load throughput is 1 load/cycle. Unaligned load throughput drops to 5% of peak when the load splits a cache line and 50% of peak in all other cases. ICC generates the `palignr` SSSE3 instruction for single precision inter- and intra-register movement on Core 2 Quad. Double precision shifts are accomplished with consecutive `movsd-movhpd` sequences as previously described.

Core i7-920. The Intel Core i7-920 (*Nehalem* microarchitecture) is an x86-64 chip running at 2.66 GHz. SIMD execution units are configured in the same manner as the previously described x86-64 processors, leading to peak FP throughput of 21.28 single precision GFlop/s and 10.64 double precision Gflop/s. Unaligned loads on this processor are very efficient. Throughput is equal to that of aligned loads at 1 load/cycle in all cases except cache line splits, where it drops to 1 load per 4.5 cycles. Single precision code generated by ICC auto-vectorization uses unaligned loads exclusively to resolve stream alignment conflicts. Double precision code contains a combination of consecutive `movsd-movhd` sequences and unaligned loads.

5.2 Stencil Codes

We evaluated the use of the dimension-lifting layout transformation on seven stencil benchmarks, briefly described below.

Jacobi 1/2/3D. The Jacobi stencil is a symmetric stencil that occurs frequently both in image processing applications as well as with explicit time-stepping schemes in PDE

solvers. We experimented with one-dimensional, 2D, and 3D variants of the Jacobi stencil, and used the same weight for all neighbor points on the stencil and the central point.

In the table of performance data below, the 1D Jacobi variant is referred as J-1D. For the 2D Jacobi stencil, both a five point “star” stencil (J-2D-5pt) and 9 point “box”(J-2D-9pt) stencil were evaluated. A seven point “star” stencil (J-3D-9pt) was used to evaluate performance of Jacobi 3D code.

Heattut 3D. This is a kernel from the Berkeley stencil probe and is based on a discretization of the heat equation PDE. [17].

FDTD 2D. This kernel is the core computation in the widely used Finite Difference Time Domain method in Computational Electromagnetics [31].

Rician Denoise 2D. This application performs noise removal from MRI images and involves an iterative loop that performs a sequence of stencil operations.

Problem Sizes. We assume the original program is tiled such that the footprint of a tile does not exceed the L1 cache size, thus all arrays are sized to fit in the L1 data cache. As is common for stencil codes, for each of the benchmarks, there is an outer loop around the stencil loops, so that any one-time layout transformation cost to copy from an original standard array representation to the transformed representation involves a negligible overhead.

Code versions. For each code, three versions were tested:

- Reference, compiler auto-vectorized
- Layout transformed, compiler auto-vectorized
- Layout transformed, explicitly vectorized with intrinsics

Vector Intrinsic Code Generation. Vector intrinsic code generation is based on the process shown in Figure 6. An outline of the steps in code generation is provided next.

Convert stencil statement(s) into intrinsic equivalents. We convert C statements into vector intrinsic equivalents. For example, consider the following 3 point 1D Jacobi statement:

```
a[i] = b[i-1] + b[i] + b[i+1];
```

This statement can be expressed in SSE intrinsics:

```
ad1t[i] = _mm_add_ps(_mm_add_ps(bd1t[i-1], bd1t[i]), bd1t[i+1]);
```

Note that d1t suffixed arrays have been layout transformed.

Generate boundary code. The reuse distance information obtained with the framework of Section 4 above is used to generate boundary code from the intrinsic statements. This code contains the appropriate shifts and masked stores required to maintain program correctness.

Generate intrinsic steady state code. Again, reuse distance information is used to generate a vector intrinsic inner loop. This loop, along with boundary code, replaces the

		Phenom				Core2 Quad				Core i7			
		SP		DP		SP		DP		SP		DP	
		GF/s	Imp.	GF/s	Imp.	GF/s	Imp.	GF/s	Imp.	GF/s	Imp.	GF/s	Imp.
J-1D	Ref.	4.27	1.00×	3.08	1.00×	3.71	1.00×	2.46	1.00×	8.67	1.00×	3.86	1.00×
	DLT	7.68	1.80×	3.79	1.23×	9.42	2.54×	2.83	1.15×	10.55	1.22×	4.01	1.04×
	DLTi	11.38	2.67×	5.71	1.85×	13.95	3.76×	7.01	2.85×	15.35	1.77×	7.57	1.96×
J-2D-5pt	Ref.	6.96	1.00×	2.71	1.00×	3.33	1.00×	2.94	1.00×	8.98	1.00×	4.54	1.00×
	DLT	9.00	1.29×	3.75	1.38×	8.86	2.66×	4.58	1.56×	10.20	1.14×	5.18	1.14×
	DLTi	11.31	1.63×	5.67	2.09×	11.58	3.48×	5.85	1.99×	13.12	1.46×	6.58	1.45×
J-2D-9pt	Ref.	4.48	1.00×	3.21	1.00×	4.21	1.00×	2.72	1.00×	8.30	1.00×	4.11	1.00×
	DLT	7.71	1.72×	3.81	1.18×	8.04	1.91×	4.08	1.50×	10.23	1.23×	5.23	1.27×
	DLTi	12.26	2.74×	6.11	1.90×	12.01	2.85×	6.03	2.22×	13.62	1.64×	6.80	1.65×
J-3D	Ref.	6.01	1.00×	2.90	1.00×	6.07	1.00×	3.04	1.00×	9.04	1.00×	4.64	1.00×
	DLT	6.84	1.14×	3.73	1.29×	8.07	1.33×	4.25	1.40×	9.46	1.05×	5.02	1.08×
	DLTi	10.08	1.68×	5.36	1.85×	10.36	1.71×	5.31	1.75×	12.02	1.33×	6.04	1.30×
Heatttt-3D	Ref.	6.06	1.00×	3.02	1.00×	6.64	1.00×	3.29	1.00×	8.75	1.00×	4.55	1.00×
	DLT	7.12	1.18×	3.36	1.11×	8.71	1.31×	4.45	1.35×	9.99	1.14×	4.91	1.08×
	DLTi	9.59	1.58×	5.12	1.70×	8.86	1.33×	4.45	1.35×	11.99	1.37×	6.05	1.33×
FDTD-2D	Ref.	5.86	1.00×	3.26	1.00×	6.42	1.00×	3.35	1.00×	8.72	1.00×	4.34	1.00×
	DLT	6.89	1.18×	3.65	1.12×	7.71	1.20×	4.03	1.20×	8.91	1.02×	4.73	1.09×
	DLTi	6.64	1.13×	3.41	1.05×	8.03	1.25×	4.03	1.20×	9.74	1.12×	4.82	1.11×
Rician-2D	Ref.	3.29	1.00×	1.93	1.00×	1.87	1.00×	1.27	1.00×	3.98	1.00×	2.16	1.00×
	DLT	3.46	1.05×	2.40	1.25×	2.59	1.39×	1.27	1.00×	4.13	1.04×	2.23	1.03×
	DLTi	8.09	2.46×	2.56	1.33×	8.50	4.55×	1.27	1.00×	11.31	2.84×	2.23	1.03×

Fig. 10. Summary of experimental results. Ref is the unoptimized, auto-vectorized version. DLT is the layout transformed, auto-vectorized version. DLTi is the layout transformed version implemented with vector intrinsics.

original inner loop. Finally, well-known loop unrolling and register blocking optimizations are performed. It is interesting to note that unrolling the vanilla C versions of the codes did not improve performance (in many cases impacted performance negatively), while unrolled versions of the vector intrinsic code resulted in performance improvement.

5.3 Results

Absolute performance and relative improvement for single and double precision experiments across all platforms and codes are given in Figure 10. Intel C Compiler icc v11.1 with the ‘-fast’ option was used for all machines. Vectorization pragmas were added to the inner loops of reference and layout transformed codes to force ICC auto-vectorization.

Double Precision. Double precision results are shown in columns labeled DP of Figure 10. Significant performance gains are achieved across all platforms and on all benchmarks. ICC auto-vectorized DLT code equaled or improved upon reference code performance in all cases. The harmonic means of relative improvements across all double precision benchmarks on x86-64 were 1.10× (Core i7), 1.22× (Phenom), and 1.28× (Core 2 Quad). Individual benchmark improvements range from, worst case, 1.00× (2D Rician Denoise on Core 2 Quad) to a best case of 1.56× (5 point 2D Jacobi on Core 2 Quad).

The auto-vectorized layout transformed code was fast but certain areas of it were still very inefficient. While ICC automatically unrolled the inner loop of reference code, no

such unrolling was done for the layout transformed code. Further, ICC generated long sequences of scalar code for boundary computations. These deficiencies were addressed in the intrinsic versions of the codes. Scalar boundary code was replaced with much more efficient vector code, and all inner loops were unrolled. Further gains can be also be attributed to register blocking and computation reordering.

Intrinsic codes equaled or improved upon auto-vectorized versions in all cases, with a worst case improvement equal to reference (2D Rician Denoise on Core 2 Quad) and best case of $2.85\times$ (Jacobi 1D on Core 2 Quad). Harmonic means of improvements over reference were $1.35\times$ (Core i7), $1.60\times$ (Phenom), and $1.57\times$ (Core 2 Quad).

Single Precision. While most scientific and engineering codes use double precision for their computations, several image processing stencils use single precision. With the current SSE vector ISA, since only two double precision elements can fit in a vector, acceleration of performance through vectorization is much less than with single precision. However, the increasing vector size of emerging vector ISAs such as AVX and LRBni, imply that the performance improvement currently possible with single precision SSE will be similar to what we can expect for double precision AVX, etc. For these reasons we include single precision performance data for all benchmarks.

Significant single precision performance gains are achieved across all platforms and on all stencils. They are reported in Figure 10 under the SP columns. Layout transformed code auto-vectorized by ICC ran significantly faster than reference code on all platforms. The harmonic means of relative performance improvements across all benchmarks on x86-64 were $1.11\times$ (Core i7), $1.29\times$ (Phenom), and $1.61\times$ (Core 2 Quad). Individual benchmark improvements range from, worst case, $1.02\times$ (2D FDTD on Core i7) to a best case of $2.66\times$ (5 point 2D Jacobi on Core 2 Quad).

Vector intrinsic code optimizations again further increased the performance gains for auto-vectorized layout transformed code. All intrinsic codes were substantially faster than their corresponding auto-vectorized versions. Minimum relative improvement over reference on x86-64 was $1.12\times$ (2D FDTD on Core i7) while maximum relative improvement was $4.55\times$ (2D Rician Denoise on Core 2 Quad). Harmonic means of improvements over reference were $1.53\times$ (Core i7), $1.81\times$ (Phenom), and $2.15\times$ (Core 2 Quad).

Discussion. Performance gains for all x86-64 codes can be attributed to the elimination of costly intra-register movement, shuffle, and unaligned load instructions from inner loop code. The performance gains on Core i7, while significant, were consistently the smallest of any platform tested. This is partly explained by the relatively small performance penalty associated with unaligned loads and shuffle on this CPU. Still, the DLT intrinsic versions achieve a $1.53\times$ average performance improvement for single precision and $1.35\times$ for double precision codes on this platform. In contrast, the *Kentsfield* Core 2 Quad, demonstrates consistently large performance improvements from layout transformation. This can mainly be attributed to poorly performing vector shuffle hardware.

Generally speaking, 1D Jacobi showed both the largest performance gains, and the fastest absolute performance, while higher dimensional stencils showed smaller, but still significant improvement. Higher dimensional stencils have more operands and more intra-stencil dependences. This leads to higher register occupancy, higher load / store

unit utilization, and more pipeline hazards / stalls for these codes. This combination of factors leads to less improvement with respect to the 1D case. General and application-specific optimizations based on the data layout transformation described in this work could likely achieve higher performance through careful instruction scheduling and tuning of register block sizes to address these issues.

6 Related Work

A number of works have addressed optimizations of stencil computations on emerging multicore platforms [6], [26], [25], [11], [34], [4], [9], [37], [35], [38], [36]. In addition, other transformations such as tiling of stencil computations for multicore architectures have been addressed in [40], [24], [21], [31]. Recently, memory customization for stencils has been proposed in [33].

Automatic vectorization has been the subject of extensive study in the literature [19][39]. There has been significant recent work in generating effective code for SIMD vector instruction sets in the presence of hardware alignment and stride constraints as described in [12][28][13]. The difficulties of optimizing for a wide range of SIMD vector architectures are discussed in [27][14]. In addition, several other works have addressed the exploitation of SIMD instruction sets [22][23][29][28]. All of these works only address SIMD hardware alignment issues. The issues of algorithmic stream alignment addressed in this paper are distinctly different from the problem addressed in those works and the dimension-lifted transposition solution that we have developed has a significant impact on performance even on SIMD architectures where hardware misalignment does not significantly degrade performance.

Stream alignment shares a lot similarities with array alignment in data-parallel languages [2][5][20] and several related works. None of these works, however, considered dimension-lifted transposition of accessed arrays. There has been prior work attempting to use static linear data layout optimizations (such as permutations of array dimensions) to improve spatial locality in programs [30][18]. These works do not address dimension-lifted transposition. Rivera and Tseng [32] presented data padding techniques to avoid conflict misses. Recently, linear data layout transformations to improve vector performance have been proposed [15].

To avoid conflict misses and false sharing, Amarasinghe's work [3] maps data accessed by a processor to contiguous memory locations by using strip-mining and permutation of data arrays. In contrast, our approach attempts remap data in order to spread out reuse carrying data in the innermost loops in order to have them map to the same vector register slot; this avoids alignment conflicts and eliminates the need for extra loads or inter- and intra-register data movement.

7 Conclusions

This paper identifies, formalizes and provides an effective solution for a fundamental problem with optimized implementation of stencil computations on short-vector SIMD architectures. The issue of stream alignment conflicts was formalized and a static analysis framework was developed to identify it. A novel nonlinear data layout transformation

was proposed to overcome stream alignment conflicts. Experimental results on multiple targets demonstrate the effectiveness of the approach on a number of stencil kernels.

Acknowledgments

This work was supported in part by the U.S. National Science Foundation through awards 0926127, 0926687, and 0926688, and by the U.S. Army through contract W911NF-10-1-0004. We thank the reviewers, especially “Reviewer 3”, for constructive feedback that helped improve the paper.

References

1. Allen, R., Kennedy, K.: Automatic translation of fortran programs to vector form. *ACM TOPLAS* 9(4) (1987)
2. Amarasinghe, S., Lam, M.: Communication optimization and code generation for distributed memory machines. In: *PLDI* (1993)
3. Anderson, J., Amarasinghe, S., Lam, M.: Data and computation transformations for multiprocessors. In: *PPoPP* (1995)
4. Augustin, W., Heuveline, V., Weiss, J.-P.: Optimized stencil computation using in-place calculation on modern multicore systems. In: Sips, H., Epema, D., Lin, H.-X. (eds.) *Euro-Par 2009*. LNCS, vol. 5704, pp. 772–784. Springer, Heidelberg (2009)
5. Chatterjee, S., Gilbert, J., Schreiber, R., Teng, S.: Automatic array alignment in data-parallel programs. In: *POPL* (1993)
6. Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review* 51(1) (2009)
7. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: *SC 2008*, pp. 1–12 (2008)
8. Datta, K., Williams, S., Volkov, V., Carter, J., Oliker, L., Shalf, J., Yelick, K.: Auto-tuning the 27-point stencil for multicore. In: *iWAPT 2009* (2009)
9. de la Cruz, R., Araya-Polo, M., Cela, J.M.: Introducing the semi-stencil algorithm. In: *PPAM* (1) (2009)
10. Dursun, H., Nomura, K., Wang, W., Kunaseth, M., Peng, L., Seymour, R., Kalia, R., Nakano, A., Vashishta, P.: In-core optimization of high-order stencil computations. In: *PDPTA* (2009)
11. Dursun, H., Nomura, K.-i., Peng, L., Seymour, R., Wang, W., Kalia, R.K., Nakano, A., Vashishta, P.: A multilevel parallelization framework for high-order stencil computations. In: Sips, H., Epema, D., Lin, H.-X. (eds.) *Euro-Par 2009*. LNCS, vol. 5704, pp. 642–653. Springer, Heidelberg (2009)
12. Eichenberger, A., Wu, P., O’Brien, K.: Vectorization for simd architectures with alignment constraints. In: *PLDI* (2004)
13. Fireman, L., Petrank, E., Zaks, A.: New algorithms for SIMD alignment. In: Adsul, B., Vetta, A. (eds.) *CC 2007*. LNCS, vol. 4420, pp. 1–15. Springer, Heidelberg (2007)
14. Hohenauer, M., Engel, F., Leupers, R., Ascheid, G., Meyr, H.: A simd optimization framework for retargetable compilers. *ACM TACO* 6(1) (2009)
15. Jang, B., Mistry, P., Schaa, D., Dominguez, R., Kaeli, D.R.: Data transformations enabling loop vectorization on multithreaded data parallel architectures. In: *PPOPP* (2010)

16. Kamil, S., Datta, K., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Implicit and explicit optimizations for stencil computations. In: MSPC 2006 (2006)
17. Kamil, S., Husbands, P., Oliker, L., Shalf, J., Yelick, K.: Impact of modern memory subsystems on cache optimizations for stencil computations. In: MSP 2005 (2005)
18. Kandemir, M., Choudhary, A., Shenoy, N., Banerjee, P., Ramanujam, J.: A linear algebra framework for automatic determination of optimal data layouts. *IEEE TPDS* 10(2) (1999)
19. Kennedy, K., Allen, J.: *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann, San Francisco (2002)
20. Kennedy, K., Kremer, U.: Automatic data layout for distributed-memory machines. *ACM TOPLAS* 20(4) (1998)
21. Krishnamoorthy, S., Baskaran, M., Bondhugula, U., Ramanujam, J., Rountev, A., Sadayappan, P.: Effective automatic parallelization of stencil computations. In: PLDI (2007)
22. Larsen, S., Amarasinghe, S.P.: Exploiting superword level parallelism with multimedia instruction sets. In: PLDI (2000)
23. Larsen, S., Witchel, E., Amarasinghe, S.P.: Increasing and detecting memory address congruence. In: *IEEE PACT* (2002)
24. Li, Z., Song, Y.: Automatic tiling of iterative stencil loops. *ACM TOPLAS* 26(6) (2004)
25. Meng, J., Skadron, K.: Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In: *ICS* (2009)
26. Micikevicius, P.: 3d finite difference computation on gpus using cuda. In: *GPGPU-2* (2009)
27. Nuzman, D., Henderson, R.: Multi-platform auto-vectorization. In: *CGO* (2006)
28. Nuzman, D., Rosen, I., Zaks, A.: Auto-vectorization of interleaved data for simd. In: PLDI (2006)
29. Nuzman, D., Zaks, A.: Outer-loop vectorization: revisited for short simd architectures. In: *PACT* (2008)
30. O'Boyle, M., Knijnenburg, P.: Nonsingular data transformations: Definition, validity, and applications. *IJPP* 27(3) (1999)
31. Orozco, D., Gao, G.R.: Mapping the FDTD Application to Many-Core Chip Architectures. In: *ICPP* (2009)
32. Rivera, G., Tseng, C.-W.: Data transformations for eliminating conflict misses. In: PLDI (1998)
33. Shafiq, M., Pericas, M., de la Cruz, R., Araya-Polo, M., Navarro, N., Ayguade, E.: Exploiting memory customization in fpga for 3d stencil computations. In: *FPT* (2009)
34. Solar-Lezama, A., Arnold, G., Tancau, L., Bodik, R., Saraswat, V., Seshia, S.: Sketching stencils. In: PLDI (2007)
35. Treibig, J., Wellein, G., Hager, G.: Efficient multicore-aware parallelization strategies for iterative stencil computations. *CoRR*, abs/1004.1741 (2010)
36. Venkatasubramanian, S., Vuduc, R.: Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems. In: *ICS* (2009)
37. Wellein, G., Hager, G., Zeiser, T., Wittmann, M., Fehske, H.: Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In: *COMPSAC* (2009)
38. Wittmann, M., Hager, G., Treibig, J., Wellein, G.: Leveraging shared caches for parallel temporal blocking of stencil codes on multicore processors and clusters. *CoRR*, abs/1006.3148 (2010)
39. Wolfe, M.J.: *High Performance Compilers For Parallel Computing*. Addison-Wesley, Reading (1996)
40. Wonnacott, D.: Achieving scalable locality with time skewing. *IJPP* 30(3) (2002)
41. Wu, P., Eichenberger, A.E., Wang, A.: Efficient SIMD Code Generation for Runtime Alignment and Length Conversion. In: *CGO* (2005)

Subregion Analysis and Bounds Check Elimination for High Level Arrays

Mackale Joyner¹, Zoran Budimčić², and Vivek Sarkar²

¹ Texas Instruments, Dallas, TX 75243

² Rice University, Houston, TX 77054

Abstract. For decades, the design and implementation of arrays in programming languages has reflected a natural tension between productivity and performance. Recently introduced HPCS languages (Chapel, Fortress and X10) advocate the use of *high-level arrays* for improved productivity. For example, high-level arrays in the X10 language support rank-independent specification of multidimensional loop and array computations using *regions* and *points*. Three aspects of X10 high-level arrays are important for productivity but pose significant performance challenges: high-level accesses are performed through point objects rather than integer indices, variables containing references to arrays are rank-independent, and all subscripts in a high-level array access must be checked for bounds violations.

The first two challenges have been addressed in past work. In this paper, we address the third challenge of optimizing the overhead of array bounds checks by developing a novel *region-based* interprocedural array bounds analysis to automatically identify redundant checks. Elimination of redundant checks reduces the runtime overhead of bounds checks, and also enables further optimization by removing constraints that arise from precise exception semantics. We have implemented an array bounds check elimination algorithm that inserts special annotations that are recognized by a modified JVM.

We also introduce *array views*, a high-level construct that improves productivity by allowing the programmer to access the underlying array through multiple views. We describe a technique for optimizing away the overhead of many common cases of array views in X10. Our experiments show that eliminating bounds checks using the results of the analysis described in this paper improves the performance of our benchmarks by up to 22% over JIT compilation.

1 Introduction

Since the dawn of computing, arrays have played an important role in programming languages as a central data structure used by application and library developers. However, the design and implementation of array operations have been subject to a natural tension between productivity and performance. Languages such as APL and MATLAB have demonstrated the productivity benefits of *high-level arrays* that support a powerful set of array operations, but the usage of high-level arrays is typically restricted to prototyping languages because it has proved very challenging to deliver production-strength performance for these operations. Conversely, languages such as C include *low-level arrays* with a very restricted set of array operations (primarily, subscripting to access

individual array elements) that are amenable to efficient production-strength implementations. Some languages, such as FORTRAN 90 and its successors, augment low-level arrays with a few high-level operations that operate on entire arrays; in some cases, efficient code can be generated through optimized *scalarization* [30] for these high-level operations, but it is often necessary for users of these languages to replace high-level array operations by low-level equivalents when hand-tuning their code.

Chapel, Fortress and X10, initially developed within DARPA's High Productivity Computing System (HPCS) program, are all parallel high-level object-oriented languages designed to deliver both high productivity and high performance. These languages offer abstractions that enable programmers to develop applications for parallel environments without having to explicitly manage many of the details encountered in low level parallel programming. Unfortunately, runtime performance usually suffers when programmers use early implementations of these languages. Compiler optimizations are crucial to reducing performance penalties resulting from their abstractions.

For example, high-level arrays in the X10 language support rank-independent specification of multidimensional loop and array computations using *regions* and *points*. Three aspects of X10 high-level arrays are important for productivity but also pose significant performance challenges: high-level accesses are performed through point objects instead of integer indices, variables containing references to arrays are rank-independent, and all high-level array accesses must be checked for bounds violations.

The first two challenges have been addressed in the past [17,18]. In this paper, we address the optimizing the overhead of array bounds checks by developing a novel *region-based* interprocedural array bounds analysis to automatically identify redundant checks. Elimination of redundant checks reduces the runtime overhead of bounds checks, and also enables further optimization by removing constraints that arise from precise exception semantics. A single high-level array access may result in multiple bounds checks, one per dimension in general. We have implemented an array bounds check elimination algorithm that eliminates these per-dimension checks from the X10 program when legal to do so, and inserts special annotations to enable the underlying JVM to eliminate the last remaining bounds check for the underlying linearized array. An unusual aspect of our analysis is the use of *must* information (the sub-region relationship) to eliminate bounds checks, while most of the traditional analysis use *may* information for bounds check elimination.

We also introduce *array views*, a high-level construct that improves productivity by allowing the programmer to access the underlying array through multiple views. We present a method for optimizing away the overhead of some common cases of array views in X10. Our experiments show that a bounds check elimination optimization alone using the results of the analysis described in this paper improves the performance of our benchmarks by up to 22% over JIT compilation and approaches the performance of the code in which all runtime checks (bounds checks, null checks, cast checks, etc...) are turned off.

2 Points, Regions and Arrays

High level arrays are embodied in a number of languages including the recent HPCS languages and earlier languages such as Titanium [29]. Both Chapel and X10 build

on ZPL’s foundational concepts of *points* and *regions* [24]. In this paper, we focus on X10’s embodiment of high level arrays.

A *point* is an element of an n -dimensional Cartesian space ($n \geq 1$) with integer-valued coordinates, where n is the *rank* of the point. A *region* is a set of points, and can be used to specify an array allocation or iteration constructs such as point-wise sequential and parallel loops. For instance, the region `[0:200, 1:100]` specifies a collection of two-dimensional points (i, j) with i ranging from 0 to 200 and j ranging from 1 to 100.

X10’s high level arrays can be used to express rank-independent loop and array computations as shown in Figure 1. For simplicity, an additional loop is introduced to compute the weighted sum using the elements in the stencil, but this loop could be replaced by a high level array `sum()` operation as well. Note that the code in this example can be executed on arrays with different ranks, by setting `R_inner` and `stencil` appropriately.

```

region R_inner = ... ; // Inner region
region stencil = ... ; // Set of points in stencil
double omega_factor = ... ; // Weight used for stencil points
for (int p=0; p<num_iterations; p++) {
  for (point t : R_inner) { //
    double sum = one_minus_omega * G[t];
    for (point s : stencil) sum += omega_factor * G[t+s];
    G[t] = sum;
  }
}

```

Fig. 1. Rank-independent version of Java Grande SOR benchmark

Points and regions are first-class value types — a programmer can declare variables and create expressions of these types using the operations listed in Figure 2 — in X10 [8]. In addition, X10 supports a special syntax for point construction — the expression, “[a, b, c]”, is implicit syntax for a call to a three-dimensional point constructor, “`point.factory(a, b, c)`” — and also for variable declarations. The declaration, “`point p[i, j]`” is exploded syntax for declaring a two-dimensional point variable `p` along with integer variables i and j which correspond to the first and second elements of `p`. Further, by requiring that points and regions be value types, the X10 language ensures that individual elements of a point or a region cannot be modified after construction.

A summary of array operations in X10 can be found in Figure 2. Note that the X10 array allocation expression, “`new double[R]`”, directly allocates a multi-dimensional array specified by region `R`. The region associated with an array is available through the `.region` field. In its full generality, an array allocation expression in X10 takes a *distribution* instead of *region*. However, we will ignore distributions in this paper and limit our attention to single-place executions although it is straightforward to extend the subregion analysis and bounds check elimination algorithms in this paper to handle distributed arrays.

¹ This paper uses the old Java-based syntax from X10 v1.5 [8]. The latest version of X10 has a different syntax and an updated type system, but retains the basic structure of high level arrays from v1.5.

Region operations:

```

R.rank ::= # dimensions in region;
R.size() ::= # points in region
R.contains(P) ::= predicate if region R contains point P
R.contains(S) ::= predicate if region R contains region S
R.equal(S) ::= true if region R and S contain same set of points
R.rank(i) ::= projection of region R on dimension i
R.rank(i).low() ::= lower bound of i-th dimension of region R
R.rank(i).high() ::= upper bound of i-th dimension of region R
R.ordinal(P) ::= ordinal value of point P in region R
R.coord(N) ::= point in region R with ordinal value = N
R1 && R2 ::= region intersection
R1 || R2 ::= union of regions R1 and R2
R1 - R2 ::= region difference

```

Array operations:

```

A.rank ::= # dimensions in array
A.region ::= index region (domain) of array
A[P] ::= element at point P, where P belongs to A.region
A | R ::= restriction of array onto region R
A.sum(), A.max() ::= sum/max of elements in array
A1 <op> A2 ::= result of applying point-wise op on A1 and A2,
              when A1.region = A2.region
              (<op> can include +, -, *, and / )
A1 || A2 ::= disjoint union of arrays A1 and A2
              (A1.region and A2.region must be disjoint)
A1.overlay(A2) ::= array with region, A1.region || A2.region,
              with element value A2[P] for all points P in
              A2.region and A1[P] otherwise.

```

Fig. 2. Region and array operations in X10

3 Region Analysis

3.1 Intraprocedural Region Analysis

Our static bounds analysis first runs a local pass over each method after we translate the code into a static single assignment (SSA) form. Using a dominator based value numbering technique [7], we assign value numbers to each point, region, array, and array access inside the method body. These value numbers represent region association. Upon completion of local bounds analysis, we map region value numbers back to the source using the source code position as the unique id. Algorithm 1 shows the algorithm for the intraprocedural region analysis.

To perform the analysis and transformation techniques described above, we use the Matlab D framework developed at Rice University [9, 11]. We generate an XML file from the AST of the X10 program, then read this AST within the Matlab D compiler, convert it into SSA, perform the value numbering based algorithms presented in this chapter to infer the regions associated with arrays, points and regions in the program, then use the unique source code position to map the analysis information back into the X10 compiler.

We build both array region and value region relationships during the local analysis pass. Discovering an array's range of values exposes additional code optimization

opportunities. Barik and Sarkar’s [4] enhanced bit-aware register allocation strategy uses array value ranges to precisely determine how many bits a scalar variable requires when it is assigned the value of an array element. In the absence of sparse data structures [19], sparse matrices in languages like Fortran, C, and Java are often represented by a set of 1-D arrays that identify the indices of non-zero values in the matrix. This representation usually inhibits standard array bounds elimination analysis because array accesses often appear in the code with subscripts that are themselves array accesses. We employ value range analysis to infer value ranges for arrays. Specifically, our array value range analysis tracks all assignments to array elements. We ascertain that when program execution assigns an array’s element a value using the *mod* function, a loop induction variable, a constant, or array element value, we can analyze the assignment and establish the bounds for the array’s element value range. [8](#)

In Figure [3](#), assuming that the assignment of array values for *row* is the only *row* update, analysis will conclude that *row*’s value region is *reg1*. Our static bounds analysis establishes this value region relationship because the *mod* function inherently builds the region $[0:reg1.high())$. Figure [4](#) shows this analysis code view update for array element assignments to *row* and *col*.

```
//code fragment is used to highlight
//interprocedural array element value
//range analysis
...
region reg1 = [0:dm[size]-1];
region reg2 = [0:dm[size]-1];
region reg3 = [0:dp[size]-1];
double[,] x = randVec(reg2);
double[,] y = new double[reg1];
int[,] val = new double[reg3];
int[,] col = new double[reg3];
int[,] row = new double[reg3];
Random R;...
for (point p1 : reg3) {
    //array row has index set in reg3 and value range in reg1
    row[p1] = Math.abs(R.Int()) % (reg1.high()+1);
    col[p1] = Math.abs(R.Int()) % (reg2.high()+1);...
}
kernel(x,y,val,col,row,..);

double[,] randVec(region r1){
    double[,] a = new double[r1];
    for (point p2: r1)
        a[p2] = R.double();
    return a;
}

kernel(double[,]x,double[,]y,int[,]val,int[,]col,int[,]row,..){
    for (point p3 : col)
        y[row[p3]]+= x[col[p3]]*val[p3];
}

```

Fig. 3. Java Grande Sparse Matrix Multiplication (source view)

² Note: when array *a1* is an alias of array *a2* (e.g. via an array assignment), we assign both *a1* and *a2* a value range of \perp , even if *a1* and *a2* share the same value range, in order to eliminate the need for interprocedural alias analysis. In the future, value range alias analysis can be added to handle this case.

```

//code fragment is used to highlight
//interprocedural array element value
//range analysis
...
reg1 = [0:dm[size]-1];
reg2 = [0:dn[size]-1];
reg3 = [0:dp[size]-1];
x = reg2; //replaced call with region argument reg2
y = reg1;
col = reg3;
row = reg3;
Random R;...
p1 = reg3; //replaced \Xten{} loop with assignment to p1
row[p1] = [0:reg1.high()]; //followed by loop body
col[p1] = [0:reg2.high()]; //created value range from mod
kernel(x,y,col,row,...);
...
region randVec(region r1){
  a = r1;
  p2 = r1; //replaced \Xten{} loop with assignment to p2
  a[p2] = R.double(); //followed by loop body
  return r1; //returns formal parameter
}
kernel(double[.]x,double[.]y,int[.]col,int[.] row,...){
  p3 = col; //replaced \Xten{} loop with assignment to p3
  y[row[p3]]+= x[col[p3]]*val[p3]; //followed by loop body
}

```

Fig. 4. Java Grande Sparse Matrix Multiplication (analysis view)

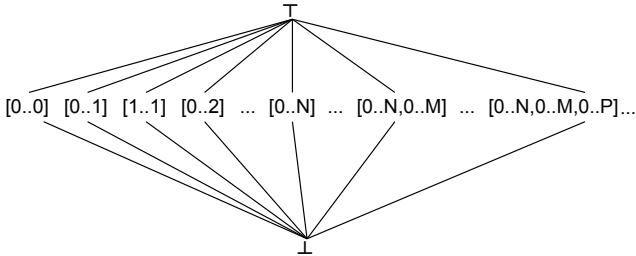


Fig. 5. Type lattice for region equivalence

We use an implicit, infinitely wide type lattice to propagate the values of the regions through the program. The lattice is shown on Figure 5. In the Matlab D compiler [11], a ϕ function performs a *meet* operation (\wedge) of all its arguments, where each argument is a lattice value, and assigns the result to the target of the assignment.

3.2 Interprocedural Region Analysis

If a method returns an expression with a value number that is the same as a formal parameter value number, analysis will give the array assigned the result of the method call the value number of the corresponding actual argument at the call site.

Static interprocedural analysis commences once intraprocedural analysis completes. During program analysis, we work over two different views of the code. The first is the standard source view which affects code generation. The second is the analysis view. Changes to the analysis view of the code do not impact code generation. In Figure 3,

Algorithm 1. Intraprocedural region analysis algorithm builds local region relationships

Input: CFG of X10 program

Output: $regmap$, a local mapping of each variable with type X10 array, region or point to its region value number

```

begin
  // initialization
  foreach  $n \in Region, Point, Array$  do
     $regmap(n) = \perp$ 
  // infer X10 region mapping
  foreach  $a \in assign$  do
    if  $a \in \phi$  function then
       $regmap(a.def) \leftarrow \bigwedge_{i=0}^{a.numargs} regmap(a.arg(i))$ 
    else if  $a.rhs \in constant$  then
       $regmap(a.lhs) = a.rhs$ 
    else
       $regmap(a.lhs) = regmap(a.rhs)$ 
  end

```

program execution assigns array x the result of invoking method *RandomVector*. Because our analysis determines that the method will return the region the program passes as an argument (assuming region has a lower bound of 0), we will modify the analysis view by replacing the method call with an assignment to the argument ($reg2$ in our example). Figure 4 shows this update. When encountering method calls which our intraprocedural regions analysis is not currently analyzing, we assign each formal argument to the actual argument if and only if the actual argument has a region association. Each actual argument can have one of the following three region states:

- If the method argument is a X10 array, region, or point, then the argument will be in the full region state.
- The method argument has a partial region state when it represents the high or low bound of a linear region.
- If the method argument does not fall within the first two cases, then we assign \perp to the argument (no region association). This distinction minimizes the number of variables that we need to track during region analysis.

In addition to analyzing the code to detect region equivalence, we augment the analysis with extensions to support sub-region relationships. Inferring sub-region relationships between arrays, regions and points is similar in structure to region equivalence inference analysis, but is different enough to warrant a separate discussion. As with the interprocedural region equivalence analysis, there is an implicit type lattice, but this time the lattice is unbounded in height as well as in width. The lattice is defined as follows:

- There \exists an edge between regions A and B in the lattice L iff the two regions are of the same dimensionality and region A is completely contained within region B .

- The lattice L meet operator \wedge is idempotent, commutative, and associative for all $i \in L$.
- Given the lattice L and $i, j \in L$, $i < j$ iff $i \wedge j = i$ and $i \neq j$

During our analysis, we compute on demand an approximation of the sub-region lattice relation — if we cannot prove that a region A is a sub-region of region B , then $A \wedge B = \perp$. It is important to point out that, even though the subregion lattice is unbounded in height and width, this does not adversely affect the running time or correctness of our algorithm. The lattice is implicit — it is never completely computed or traversed in the algorithm. Checking whether there is a sub-region relation between two regions takes constant amount of time, regardless of their position in the lattice.

In addition, our analysis is flow-insensitive for global variables. When static analysis determines that a global variable might be involved in multiple region assignments involving different regions, the region for the variable becomes \perp . In the future, we can extend the algorithm to assign the variable the region intersection instead of \perp . Algorithm 2 presents pseudo code for the static interprocedural region analysis algorithm. The interprocedural region analysis algorithm can be implemented to run in $O(|V| + |E|)$ time for a graph G , where V is the number of array, point, and region variables in the whole program and E is the number of edges between them. An edge exists between two variables if one defines the other. Theorem 1 shows that this algorithm has complexity $O(|V| + |E|)$ and preserves program correctness:

Definition 1. Given a program P , let T be the set containing point, region and array types in P and N be the set of variables in P with type $t \in T$ such that for all $m \in N$:

(1) $DEF(m)$ is the set of variables in P defined by m .

(2) $REG(i)$ is the region associated with i . There \exists precise region for i iff $i \in V$ and $REG(i) \neq \top$ or \perp .

Definition 2. Given a directed graph G where V is the set of program variables of type array or region, there exists an edge E between $i, j \in V$ where i is the source and j is the sink iff $j \in DEF(i)$.

Theorem 1. The region analysis algorithm runs in time $O(V+E)$ and preserves program correctness.

Proof. Initially each node $n \in V$ is placed on the worklist with lattice value \top . Once node n is taken off the worklist, n can only be put back on the list iff $n \in DEF(m)$ and $m < n$ or there \exists precise regions for both n and m and $REG(n) \neq REG(m)$. In the latter case $n \leftarrow \perp$ before we place n back on the worklist. Since the lattice is bounded, all regions in the lattice have finite size k , and a node n can only have its lattice value lowered, each node can only be placed on the worklist a maximum of $k+2$ times. Because we traverse source node edges when lattice value changes, each edge will be traversed a maximum of $k+1$ times. Therefore, because V is a finite set of nodes, the algorithm must eventually halt. Since each node n is placed on the worklist a maximum of $k+2$ times and its edges are traversed a maximum of $k+1$ times, the complexity is $O(V+E)$. Assuming the whole

program is available to the region analysis algorithm, the algorithm preserves program correctness. The region algorithm will produce an incorrect program iff the algorithm assigns an incorrect precise region to a program variable with type array or region. This would only occur when the variable can have multiple regions. However, when a variable has multiple regions, the region analysis algorithm assigns the variable \perp . Therefore, the region analysis algorithm produces a correct program.

Algorithm 2. Interprocedural region analysis, maps variables of type X10 array, point, and region to a concrete region

```

Input: X10 program
Output: regmap, a mapping of each variable with type X10 array, region or point
           to its region
begin
  // initialization
  worklist =  $\emptyset$ , def =  $\emptyset$ 
  foreach  $n \in \text{Region, Point, Array}$  do
    | regmap( $n$ ) =  $\top$ 
    | worklist = worklist +  $n$ 
  foreach assign  $a$  do
    | if  $a.rhs \in \text{constant}$  then
    | | regmap( $a.lhs$ ) =  $a.rhs$ 
    | | use( $a.rhs$ ) = use( $a.rhs$ )  $\cup$   $a.lhs$ 
  foreach call arg  $c \rightarrow \text{param } f$  do
    | if  $c \in \text{constant}$  then
    | | regmap( $f$ ) =  $c$ 
    | | use( $c$ ) = use( $c$ )  $\cup$   $f$ 
  // infer X10 region mapping
  while worklist  $\neq \emptyset$  do
    | worklist = worklist -  $n$ 
    | foreach  $v \in \text{use}(n)$  do
    | | if  $\text{regmap}(n) < \text{regmap}(v)$  in lattice then
    | | | regmap( $v$ ) = regmap( $n$ )
    | | | worklist = worklist +  $v$ 
    | | else if  $\text{regmap}(n) \not\leq \text{regmap}(v)$  in lattice then
    | | | regmap( $v$ ) =  $\perp$ 
    | | | worklist = worklist +  $v$ 
  end
end

```

3.3 Rectangular Region Algebra

In this paper, we refer to sub-arrays (products of intervals) as *regions*, as it has been the common practice in recent literature. This is in contrast to the *region* concept originally introduced by Triolet et al. [27] to describe convex sets of array elements, which are strictly more powerful.

Often in scientific codes, loops iterate over the interior points of an array. If through static analysis we can prove that loops are iterating over sub-regions of an array, we can identify the bounds checks for those array references as superfluous. We use the example on Figure 6 to highlight the benefits of employing region algebra to build variable region relationships. Algorithm 3 shows the algorithm for region algebra analysis.

When our static region analysis encounters the *dgefa* method call with a region high bound argument in Figure 6, analysis will assign *dgefa*'s formal parameter *n* the high bound of *region1*'s second dimension and *a* the region *region1*. We shall henceforth refer to the region representing *region1*'s second dimension as *region1_2dim*. Inside *dgefa*'s method body, analysis will categorize *nm1* as a region bound and *region3* as a sub-region of *region1_2dim* when inserting it in the region tree.

Next, we assign array *col_k* the region *region1_2dim* and categorize *kp1* as a sub-region of *region1_2dim*. When static region analysis examines the binary expression *n-kp1* on the right hand side of the assignment to *var1*, it discovers that the *n* is *region1_2dim.hbound()* and *kp1* is a sub region of *region1_2dim*. As a result, we can use region algebra to prove that this region operation will return a region *r* where: $r.lbound() \geq region1_2dim.lbound()$ and $r.bound() \leq region1_2dim.hbound()$. Consequently, *var1* will be assigned *region1_2dim*.

Finally, analysis determines that *var2*'s region is a sub-region of *region1_2dim*. As a result, when analysis encounters the *daxpy* call it will assign *daxpy* formal parameter *dx* the region *region1_2dim* and formal parameter *dax_reg* the same region as *var2* enabling us to prove and signal to the VM that the bounds check for the array access *dx[p2]* in *daxpy*'s method body is unnecessary.

```
//code fragment is used to highlight
//interprocedural region analysis using region algebra
int n = dsizes[size];
int ldaa = n;
int lda = ldaa + 1;
...
region region1 = [0:lda-1,0:lda-1];...
double[] a = new double[region1]...
info = dgefa(a, region1.rank(1).high(), ipvt);
//dgefa method, lufact kernel
int dgefa(double[] a, int n, int[] ipvt){...
    nm1 = n - 1;...
    region region3 = [0:nm1-1];...
    for (point p1[k] : region3) {
        col_k = RowView(a,k);...
        kp1 = k + 1...
        int var1 = n-kp1;
        region var2 = [kp1:n];...
        daxpy(var1,col_k,kp1,var2,...);...
    }
}
...
//daxpy method
void daxpy(int n,double[] dx,int dx_off,region dax_reg,...){...
    for (point p2 : dax_reg)
        dy[p2]+= da*dx[p2];...
}
```

Fig. 6. Java Grande LU factorization kernel

Algorithm 3. Region algebra algorithm discovers integers and points that have a region association

Input: X10 program

Output: $regAssoc$, a mapping of each variable of type X10 array, region, point or int to its region association

```

begin
  // initialization
  worklist =  $\emptyset$ , def =  $\emptyset$ 
  foreach  $n \in Region, Point, Array, int$  do
     $regAssoc(n) = \top$ 
    worklist = worklist +  $n$ 
  foreach assign  $a$  do
    if  $a.rhs \in constant \vee bound$  then
       $regAssoc(a.lhs) = a.rhs$ 
      use( $a.rhs$ ) = use( $a.rhs$ )  $\cup$   $a.lhs$ 
    foreach call arg  $c \rightarrow param f$  do
      if  $c \in constant \vee bound$  then
         $regAssoc(f) = a.rhs$ 
        use( $c$ ) = use( $c$ )  $\cup$   $f$ 
  // infer X10 region mapping
  while worklist  $\neq \emptyset$  do
    worklist = worklist -  $n$ 
    foreach  $v \in use(n)$  do
      if  $regAssoc(n) < regAssoc(v)$  in lattice then
         $regAssoc(v) = regAssoc(n)$ 
        worklist = worklist +  $v$ 
      else if  $regAssoc(n) \not\leq regAssoc(v)$  in lattice then
         $regAssoc(v) = \perp$ 
        worklist = worklist +  $v$ 
end

```

3.4 Interprocedural Linearized Array Bounds Analysis

Our array bounds analysis algorithm as described in Section 3.1 and Section 3.2 makes heavy use of X10 points and regions to discover when bounds checks are superfluous. In general, the programmer iterates through the elements in an array by implementing an X10 *for* loop whose header contains both a point declaration $p1$ and the region $r1$ containing the set of points defining $p1$. As a result, when encountering an array access with subscript $p1$, if our array bounds analysis can establish a subset relationship between the array's region and region $r1$, our analysis can signal the VM that a bounds check for this array access is superfluous.

Figure 7 illustrates an MG code fragment where the application developer linearizes a 3-dimensional array to boost runtime performance. This example shows why our current array bounds analysis cannot rely on the compiler automatically converting linearized arrays to X10 multi-dimensional arrays because the range for each dimension in this case cannot be established. As a result, our bounds analysis must be extended

if we want to analyze linearized array accesses to discover useless bound checks. Figure 7 highlights another extension to the array bounds analysis we previously described in Section 3.1 and Section 3.2. Studying the MG code fragment reveals that all the accesses to array r inside method $psinv$ are redundant. Our array bounds analysis adds the following requirements to prove that r 's bounds checks are redundant:

- The array region summary for $psinv$'s formal parameter r is a subset of the region summary for $zero3$'s formal parameter z . The region summary for a given array and procedure defines the valid array index space inside the procedure for which a bounds check is useless. The region summary contains only an index set that must execute when the programmer invokes this method. We do not include array accesses occurring inside conditional statements in the region summary.
- The region representing the actual argument of $psinv$'s formal parameter r is a subset of the region representing the actual argument for $zero3$'s formal parameter z .
- The program must call $zero3$ before calling $psinv$.
- Since our analysis modifies $psinv$'s actual method body, the previous requirements must hold on all calls to $psinv$.

These requirements enable our interprocedural region analysis to de-linearize array accesses into region summaries and to propagate the region summary information to discover redundant bounds checks. Note: The algorithm does not reanalyze recursive functions in a call chain. The algorithm can be extended to take advantage of recursion by splitting the recursive function A into A and A' and subsequently eliminating the checks in A' .

```
//MG code fragment highlights opportunity to eliminate
//bound checks with procedure array bound summaries
int nm = 2+(1<<lm);
int nv = (2+(1<<ndim1))*(2+(1<<ndim2))*(2+(1<<ndim3));
int nr = (8*(nv+nm*nm+5*nm+7*lm))/7;
region reg_nr = {0:nr-1};
double[]u=new double[reg_nr]; //create linearized array
zero3(u, 0, n1, n2, n3);
psinv(u, 0, n1, n2, n3);
...
void zero3(double[] z, int off, int n1, int n2, int n3) {
    for (point p1[i3,i2,i1]: {0:n3-1,0:n2-1,0:n1-1})
        z[off+i1+n1*(i2+n2*i3)] = 0.0;
}...
void psinv(double[] r, int roff, int n1, int n2, int n3) {...
    for (point p40[i3,i2]: {1:n3-2,1:n2-2}) {
        for (point p41[i1] : {0:n1-1}) {
            r1[p41] = r[roff+i1+n1*(i2-1+n2*i3)]
                + r[roff+i1+n1*(i2+1+n2*i3)]
                + r[roff+i1+n1*(i2+n2*(i3-1))]
                + r[roff+i1+n1*(i2+n2*(i3+1))];
            r2[p41] = r[roff+i1+n1*(i2-1+n2*(i3-1))]
                + r[roff+i1+n1*(i2+1+n2*(i3-1))]
                + r[roff+i1+n1*(i2-1+n2*(i3+1))]
                + r[roff+i1+n1*(i2+1+n2*(i3+1))];
        }...
    }
}
```

Fig. 7. This MG code fragment shows an opportunity to remove all array r bounds checks inside the $psinv$ method because those checks are all redundant since the programmer must invoke method $zero3$ prior to method $psinv$

4 Bounds Check Elimination

The results of the interprocedural region analysis described in Section 3 can be used in different contexts in an optimizing compiler. In this section, we describe how the subregion relationship computed in Section 3 can be used to eliminate unnecessary array bounds checks in an X10 program.

Many high-level languages perform automatic array bounds checking to improve both safety and correctness of the code, by eliminating the possibility of an incorrect (or malicious) code randomly “poking” into memory through an out of bounds array access or buffer overflow. While these checks are beneficial for safety and correctness, performing them at run time can significantly degrade performance especially in array-intensive codes. Two main ways that bounds checks can affect performance are:

1. *The Cost of Checks*. The runtime may need to check the array bounds when program execution encounters an array access.
2. *Constraining Optimizations*. The compiler may be forced to constrain or disable code optimizations in code region containing checks, in the presence of precise exception semantics.

In our approach, we insert a special *noBoundsCheck* annotation³ around an array subscript to signal to a modified version of the IBM J9 Java Virtual Machine⁴ that it can skip the array bounds check for that particular array access. These annotations can be inserted if the compiler can establish one of the following properties:

1. *Array Subscript within Region Bound*. If the array subscript is a point that the programmer is using to iterate through region $r1$ and $r1$ is a subregion of the array’s region, then the bounds check is unnecessary.
2. *Subscript Equivalence*. Given two array accesses, one with array $a1$ and subscript $s1$ and the second with array $a2$ and subscript $s2$: if subscript $s1$ has the same value number as $s2$, $s1$ executes before subscript $s2$ and array $a1$ ’s region is a subregion of $a2$ ’s region, then the bounds check for $a2[s2]$ is unnecessary.

We use a dominator-based value numbering technique [7] to find redundant array accesses. We annotate each array access in the source code with two value numbers. The first value number represents a value number for the array access. We derive a value number for the array access by combining the value numbers of the array reference and the subscript. The second value number represents the array’s element value range. By maintaining a history of these array access value numbers we can discover redundant array accesses and eliminate them.

5 Array Views

Though multidimensional high-level arrays provide significant productivity benefits compared to their low-level counterparts, there are many cases when a programmer

³ The “annotation” is simply a call to an empty *noBoundsCheck* method that is recognized by the JVM as a *NOP* and causes the JVM to skip the bounds check for the enclosed subscript.

⁴ Any JVM can be extended to recognize the *noBoundsCheck* annotation, but in this paper we report our experiences with a version of the IBM J9 JVM that was modified with this capability.

wishes to “view” a subset of array elements using a different index set from that of the original array *e.g.*, when a subarray needs to be passed by reference to a procedure or when a multi-dimensional array needs to be accessed as a one-dimensional array. In contrast, languages such as APL and Matlab provide restructuring operations that are value-oriented with copying semantics by default.

In this paper, we introduce *array views* for high-level arrays to address this limitation. Array views give the programmer the opportunity to work with multiple views of an array, with well defined bounds checks that are performed on the region associated with the view. A programmer can exploit the array’s view to traverse an alternative representation of the array. Prevalent in scientific codes is the expression of the form $a = b[i]$ which often assigns the variable a row i of array b when b is a two-dimensional array. Array views can extend this idea by providing an alternate view for the entire array. The following code snippet shows an array view example:

```
double[.] ia = new double[[1:10,1:10]];
double[.] v = ia.view([10,10],[1:1]);
v[1] = 42;
print(ia[10,10]);
```

The programmer declares array ia to be a 2-dimensional array. Next, the programmer creates the array view v to represent a view of array ia , with starting point $[10, 10]$ and region $[1 : 1]$. This essentially introduces a pointer to element $ia[10,10]$. Subsequently, when the programmer modifies the array v , array ia is also modified resulting in the print statement yielding the value 42. We will use a hexahedral cells code [12] as a running example to illustrate the productivity benefits of using array views in practice.

```
//code fragment highlights array view productivity benefit
region reg_mex = [0:MESH_EXT-1];
region reg_mex_linear=[0:MESH_EXT*MESH_EXT*MESH_EXT-1];
double[.] x = new double[reg_mex_linear-];
double[.] y = new double[reg_mex_linear];
double[.] z = new double[reg_mex_linear];...
for (point pt3[pz] : reg_mex) {...
  for (point pt2[py] : reg_mex) {...
    for (point pt1[px] : reg_mex) {
      //using less productive linearized array access
      x[px+MESH_EXT*(py + MESH_EXT*pz)] = tx;
      y[px+MESH_EXT*(py + MESH_EXT*pz)] = ty;
      z[px+MESH_EXT*(py + MESH_EXT*pz)] = tz;
      tx += ds;
    }
    ty += ds;
  }
  tz += ds;
}...
region reg_br = [0:MESH_EXT-2];
region reg_br_3D = [reg_br, reg_br, reg_br];
int[.] p1,p2 = new int[reg_br_3D];...
//would be invalid if x, y, and z were 3-D arrays
for (point pt7 : reg_br_3D) {
  ux = x[p2[pt7]] - x[p1[pt7]];
  uy = y[p2[pt7]] - y[p1[pt7]];
  uz = z[p2[pt7]] - z[p1[pt7]]; ...
}
```

Fig. 8. Hexahedral cells code showing that problems arise when representing arrays x , y , and z as 3-dimensional arrays due to programmers indexing into these arrays using an array access returning integer value instead of a triplet

Figure 8 illustrates one problem that arises when programmers utilize an array access as a multi-dimensional array subscript. Since the subscript returns an integer, the developer cannot use the subscript for multi-dimensional arrays. As a result, the programmer must rewrite this code fragment by first replacing the 3-dimensional arrays x , y and z with linearized array representations. Subsequently, the developer needs to modify the array subscripts inside the innermost loop of Figure 8 with the more complex subscript expression for the linearized arrays. While this solution is correct, we can implement a more productive solution using X10 array views as shown in Figure 9. This solution enables programmers to develop scientific applications with multi-dimensional array computations in the presence of subscript expressions returning non-tuple values.

Figure 9 shows the result of converting the 3-D X10 arrays into 3-D Java arrays when analysis determines it is safe to do so. This compilation pass does not transform the X10 arrays x , y , z , xv , yv , and zv because of their involvement in the X10 `array.view()` method call. There is not a semantically-equivalent Java method counterpart for the X10 `array.view()` method. One drawback of array views as presented is that safety analysis marks the view's target array as unsafe to transform. Our compiler does convert the X10 general arrays $p1$ and $p2$ in Figure 9 into 3-D Java arrays. Although 3-D array accesses in Java are inefficient, this transformation still delivers more than a factor of 3 speedup over the code version with only X10 general arrays. Finally, we can achieve even better performance by linearizing the 3-D Java arrays, and optimizing away the array views by replacing them by assignments to the whole array. Figure 10 provides the final source output for the hexahedral cells code fragment.

```
//code fragment highlights \Xten{} to Java array translation
region reg_mex = [0:MESH_EXT-1];
region reg_mex_3D = [reg_mex,reg_mex,reg_mex];
double[,] x,y,z = new double[reg_mex_3D];...
for (point pt3[pz] : reg_mex) {...
  for (point pt2[py] : reg_mex) {...
    for (point pt1[px] : reg_mex) {
      x[pz,py,px] = tx; //use productive multi-D
      y[pz,py,px] = ty; //access with array views
      z[pz,py,px] = tz;
      tx += ds;
    }
    ty += ds;
  }
  tz += ds;
}...
region reg_br = [0:MESH_EXT-2];
region reg_br_3D = [reg_br, reg_br, reg_br];
int[][][] p1,p2 = new int[reg_br_3D];...
region reg_linear=[0:MESH_EXT*MESH_EXT*MESH_EXT-1];
double[,] xv = x.view([0,0],[0:reg_linear];
double[,] yv = y.view([0,0],[0:reg_linear];
double[,] zv = z.view([0,0],[0:reg_linear];
for (point pt7[i,j,k]: reg_br_3D) {
  ux = xv[p2[i][j][k]] - xv[p1[i][j][k]] ;
  uy = yv[p2[i][j][k]] - yv[p1[i][j][k]] ;
  uz = zv[p2[i][j][k]] - zv[p1[i][j][k]] ;...
}
```

Fig. 9. Array views xv , yv , and zv enable the programmer to productively implement 3-dimensional array computations inside the innermost loop. We highlight the array transformation of X10 arrays into Java arrays to boost runtime performance. In this hexahedral cells volume calculation code fragment, our compiler could not transform X10 arrays x , y , z , xv , yv , zv into Java arrays because the Java language doesn't have an equivalent array view operation.


```

//code fragment shows opt with array linearization
region reg_mex = [0:MESH_EXT-1];
region reg_mex_3D = [reg_mex,reg_mex,reg_mex];
double[,] x,y,z = new double[LinearViewAuto(reg_mex_3D)];...
for (point pt3[pz] : reg_mex) {...
    for (point pt2[py] : reg_mex) {...
        for (point pt1[px] : reg_mex) {
            x[pz,py,px] = tx; //use productive multi-D
            y[pz,py,px] = ty; //access with array views
            z[pz,py,px] = tz;
            tx += ds;
        }
        ty += ds;
    }
    tz += ds;
}...
region reg_br = [0:MESH_EXT-2];
region reg_br_3D = [reg_br, reg_br, reg_br];
int[] p1,p2 = new int[LinearViewAuto(reg_br_3D)];...
region reg_linear=[0:MESH_EXT*MESH_EXT*MESH_EXT-1];
double[] xv = x;
double[] yv = y;
double[] zv = z;
for (point pt7[i,j,k]: reg_br_3D) { //sub M for MESH_EXT
    ux=xv[p2[k+(M-1)(j+(M-1)*i)]-xv[p1[k+(M-1)(j+(M-1)*i)]];
    uy=yv[p2[k+(M-1)(j+(M-1)*i)]-yv[p1[k+(M-1)(j+(M-1)*i)]];
    uz=zv[p2[k+(M-1)(j+(M-1)*i)]-zv[p1[k+(M-1)(j+(M-1)*i)]];...
}

```

Fig. 10. We show the final version for the Hexahedral cells code which demonstrates the compiler’s ability to translate X10 arrays into Java arrays in the presence of array views

6 Experimental Results

We performed our experiments on a single node of an IBM 16-way 4.7 GHz Power6 SMP with 186 GB main memory. The Java runtime environment used is the IBM J9 virtual machine (build 2.4, J2RE 1.6.0) which includes the IBM Testarossa (TR) Just-in-Time (JIT) compiler [25].

It is important to note that all our experiments are performed on a JVM that already performs dynamic bounds check elimination within a JIT compiler. Our improvements are relative to a state-of-the art dynamic compilation system that already removes as many runtime checks as it can.

We studied the X10 sequential ports of benchmarks from the Java Grande [16] suite. We compare two versions of each benchmark. The first version is the ported code. The second version, through static analysis, inserts *noBoundsCheck* calls around an array index when the bounds check is unnecessary.

In columns 2,3 and 4 of Table 1, we report the dynamic counts for the Java Grande, *hexahedral*, and 2 NAS parallel (cg, mg) X10 benchmarks. We compare dynamic counts for potential general X10 array bounds checks against omitted general X10 array bounds checks using our static analysis techniques. We use the term ”general X10 array” to refer to arrays the programmer declares using X10 regions. In several cases our static bounds analysis removes over 99% of potential bound checks.

In columns 5,6 and 8 of Figure 1, we report the execution times for the Java Grande, *hexahedral*, and 2 NAS parallel (cg, mg) X10 benchmarks. Column 5 shows the execution times of the baseline unoptimized code that is executed on a JIT with dynamic bounds check elimination. Column 6 shows the execution times with automatically

generated *noBoundsCheck* annotations with runtime checks enabled. These annotations alert the IBM J9 VM when array bounds checking for an array access is unnecessary. Performing static array bounds analysis and subsequent automatic program transformation, we improve runtime performance by up to 22.3%. These results demonstrate that our static no bounds check analysis helps reduce the performance impact of programmers developing applications in type-safe languages. We experience a -2.4% decrease in performance for *raytracer* because the *noBoundsCheck* annotation cost was not amortized over all bounds checks along the "hot" path. This occurs when the "hot" path does not involve loops with *noBoundsCheck* annotations (hoisted out of loop). Column 8 of Table 1 shows the execution times that are obtained by running a version of the code with *all* runtime checks turned off (including bounds checks, null checks and cast checks) and illustrates how close does our optimization come to the theoretical limit on runtime checks elimination. We can conclude that we still may further improve runtime performance in some cases by eliminating other types of runtime checks such as null checks or cast checks. One interesting result to point out is the *mg* benchmark, where even though we eliminate only 5.8% of the total number of bounds checks, that does not have a large effect on performance, since only 6.1% of the execution time is spent on all runtime checks.

Table 1. Dynamic counts for Array Bounds Checks and execution times for unoptimized and optimized versions of the code

Benchmarks	Array Bounds Checks Dynamic Counts			Sequential Runtime Performance in seconds				
	total X10 ABCs	total X10 ABCs eliminated	percent eliminated	baseline	optimized baseline	runtime improv.	all checks removed	max. theor. improv.
sparsemm	2.51×10^9	2.51×10^9	100.0%	34.46	27.02	21.6%	24.01	30.3%
crypt	1.0×10^9	1.0×10^8	10.0%	9.11	9.1	0.1%	8.79	3.5%
lufact	5.43×10^9	5.37×10^9	99.1%	46.86	40.43	13.7%	39.59	15.5%
sor	4.81×10^9	4.80×10^9	99.8%	3.67	3.66	0.2%	3.66	0.2%
series	4.0×10^6	4.0×10^6	99.9%	1233.77	1226.61	0.6%	1218.39	1.2%
moldyn	5.95×10^9	4.02×10^9	67.6%	89.98	88.65	1.5%	75.21	16.4%
montecarlo	7.80×10^8	4.20×10^8	53.8%	24.64	24.41	0.9%	24.19	1.8%
raytracer	1.18×10^9	1.18×10^9	100.0%	34.79	35.73	-2.4%	33.11	4.8%
hexahedral	3.59×10^{10}	3.21×10^{10}	89.4%	15.31	12.03	22.3%	10.38	32.2%
cg	3.04×10^9	1.53×10^9	50.4%	9.73	9.34	4.0%	9.04	7.1%
mg	6.61×10^9	3.83×10^8	5.8%	31.3	30.4	2.9%	29.39	6.1%

Finally, in Table 2 we compare Fortran, Unoptimized X10, Optimized X10, and Java execution times for the 2 NAS parallel (cg, mg) benchmarks. The Optimized X10 significantly reduces the slowdown factor that results from comparing Unoptimized X10 with Fortran. These results were obtained on the IBM 16-way SMP. Note: the 3.0 NAS Java mg version was run on a 2.16 GHz Intel Core 2 Duo with 2GB of memory due to a J9 JIT compilation problem with this code version. In the future, we will continue to extend our optimizations to further reduce the overhead of using high-level X10 array computations.

Table 2. Fortran, Unoptimized X10, Optimized X10, and Java raw sequential runtime performance comparison (in seconds) for 2 NAS Parallel benchmarks, obtained on the IBM 16-way SMP machine

Benchmark	Sequential Runtime Performance			
	Fortran Version	Unoptimized X10 Slowdown	Optimized X10 Slowdown	Java Slowdown
cg	2.58	10.43×	3.31×	1.60×
mg	2.02	46.72×	13.66×	9.53×

7 Related Work

Bodík et al. [6] reduce array bounds checks in the context of dynamic compilation in a JVM. They focus their optimization on program hot spots to maximize benefits and to amortize the cost of performing the analysis on a lightweight inequality graph. Though it was limited to the intraprocedural context, a dynamic interprocedural analysis framework such as [21] can be used to extend their work to the interprocedural context. Rather than modifying the JVM, our strategy is that the compiler communicates to the JVM the results of the static analysis. Suzuki and Ishihata [26] provide an intraprocedural array bounds checking algorithm based on theorem proving which can be prohibitively costly. Most JIT compilers also perform array bounds analysis to eliminate bounds checks. However, the analysis is generally intraprocedural, limiting the effectiveness. A key difference between our work and [6,26] is that our work targets high level arrays and leverages subregion analysis for enhanced bounds check elimination.

Aggarwal and Randall [1] use related field analysis to eliminate bounds checks. They observe that an array a and an integer b may have an invariant relationship where $0 \leq b < a.length$ for every instance of class c . To find related fields, they analyze every pair $[a, b]$ where a is a field with type array(1-Dimensional) and b is a field with type integer in class c . Heffner et al. [14] extend this by addressing the overhead required to prove program invariants for field relations at each point in the program. By contrast, we examine every array, region, point, and integer variable. As a result, we can eliminate bound checks for multi-dimensional arrays that Aggarwal and Randall would miss.

Gupta [13] uses a data-flow analysis technique to eliminate both identical and subsumed bounds checks. Ishizaki et al. [15] extends Gupta’s work by showing when bounds checks with constant index expressions can be eliminated. This algorithm relies on the assumption that all arrays have a lower bound of 0, which is generally not the case in X10. FALCON [22], is a compiler for translating MATLAB programs into Fortran 90, that performs both static and dynamic inference of scalar (e.g. real, complex) or fixed array types. MAJIC [2], a MATLAB just-in-time compiler, compiles code ahead of time using speculation. Both the FALCON and MAJIC type inference schemes are limited compared to our precise type inference with type jump functions since neither uses symbolic variables to resolve types.

Some research in the verification community, such as the ASTREÉ static analyzer [5] and the memory safety analysis of OpenSSH [10] has focused on proving safety of array

accesses, which could also be used for optimization. These analyses are much heavier weight than what we present in this paper.

The use of equivalence sets in our type analysis algorithm builds on past work on equivalence analysis [3] and constant propagation [28]. As in constant propagation, we have a lattice of height ≤ 3 . By computing the meet-over-all-paths, our type inference may be more conservative than Sagiv's [23] algorithm for finding the meet-over-all-valid-paths solution. The idea of creating specialized method variants based on the calling context is related to specialized library variant generation derived from type jump functions [9]. McCosh's [20] type inference generates pre-compiled specialized variants for MATLAB. The context in which we apply our algorithm differs from McCosh since we perform type inference in an object-oriented environment on rank-independent type variables that must be mapped to rank-specific types. Without function cloning during rank analysis, formal parameters with multiple ranks resolve to \perp .

8 Conclusions

In this paper, we addressed the problem of subregion analysis and bounds check elimination for high level arrays. We describe a novel analysis technique that computes the subregion relation between arrays, regions and points. We used this analysis to implement an array bounds elimination optimization, which improves the performance of our benchmarks by up to 22% over JIT compilation and is very close to the performance of the code where *none* of the runtime checks are performed. We also introduced *array views*, a high-level construct that improves productivity by allowing the programmer to access the underlying array through multiple views, and described a technique for optimizing away the overhead resulting from this abstraction in some common cases.

References

1. Aggarwal, A., Randall, K.H.: Related field analysis. In: PLDI 2001, pp. 214–220 (2001)
2. Almási, G., Padua, D.: MaJIC: compiling MATLAB for speed and responsiveness. In: PLDI 2002, pp. 294–303 (2002)
3. Alpern, B., Wegman, M.N., Zadeck, F.K.: Detecting equality of variables in programs. In: POPL 1988, pp. 1–11. ACM Press, New York (1988)
4. Barik, R., Sarkar, V.: Enhanced bitwidth-aware register allocation. In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 263–276. Springer, Heidelberg (2006)
5. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI 2003), San Diego, California, USA, June 7–14, pp. 196–207. ACM Press, New York (2003)
6. Bodík, R., Gupta, R., Sarkar, V.: ABCD: Eliminating Array Bounds Checks on Demand. In: PLDI 2000, pp. 321–333. ACM Press, New York (2000)
7. Briggs, P., Cooper, K., Simpson, T.L.: Value numbering. *Software Practice and Experience* 27(6), 701–724 (1997)
8. Charles, P., Donawa, C., Ebcioğlu, K., Grothoff, C., Kielstra, A., von Praun, C., Saraswat, V., Sarkar, V.: X10: An object-oriented approach to non-uniform cluster computing. In: OOPSLA 2005 Onward! Track (2005)

9. Chauhan, A., McCosh, C., Kennedy, K., Hanson, R.: Automatic type-driven library generation for telescoping languages. In: Supercomputing 2003, Washington, DC (2003)
10. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: Beyond strong vs. Weak updates. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 246–266. Springer, Heidelberg (2010)
11. Fletcher, M., McCosh, C., Jin, G., Kennedy, K.: Compiling Parallel Matlab for General Distributions Using Telescoping Languages. In: ICASSP: Proceedings of the 2007 International Conference on Acoustics, Speech and Signal Processing, Honolulu, Hawai'i, USA (2007)
12. Grandy, J.: Efficient computation of volume of hexahedral cells. Technical Report UCRL-ID-128886, Lawrence Livermore National Laboratory (October 1997)
13. Gupta, R.: Optimizing array bound checks using flow analysis. *ACM Lett. Program. Lang. Syst.* 2(1–4), 135–150 (1993)
14. Heffner, K., Tarditi, D., Smith, M.D.: Extending object-oriented optimizations for concurrent programs. In: PACT 2007: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007), pp. 119–129 (2007)
15. Ishizaki, K., et al.: Design, implementation, and evaluation of optimizations in a just-in-time compiler. In: Proceedings of the ACM 1999 Conference on Java Grande, pp. 119–128 (1999)
16. The Java Grande forum benchmark suite, <http://www.epcc.ed.ac.uk/javagrande>
17. Joyner, M., Budimlić, Z., Sarkar, V.: Optimizing array accesses in high productivity languages. In: Proceedings of the High Performance Computation Conference (HPCC), Houston, Texas (September 2007)
18. Joyner, M., Budimlić, Z., Sarkar, V., Zhang, R.: Array optimizations for parallel implementations of high productivity languages. In: Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL), Miami, Florida (April 2008)
19. Mateev, N., Pingali, K., Stodghill, P., Kotlyar, V.: Next-generation generic programming and its application to sparse matrix computations. In: ICS 2000: Proceedings of the 14th International Conference on Supercomputing, pp. 88–99 (2000)
20. McCosh, C.: Type-Based Specialization in a Telescoping Compiler for ARPACK. Master's thesis, Rice University, Houston, Texas (2002)
21. Pechtchanski, I., Sarkar, V.: Dynamic optimistic interprocedural analysis: a framework and an application. In: Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA 2001, pp. 195–210. ACM, New York (2001)
22. Rose, L.D., Padua, D.: Techniques for the translation of MATLAB programs into Fortran 90. *ACM Trans. Program. Lang. Syst.* 21(2), 286–323 (1999)
23. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.* 167(1–2), 131–170 (1996)
24. Snyder, L.: *A Programmer's Guide to ZPL*. MIT Press, Cambridge (1999)
25. Sundaresan, V., et al.: Experiences with multi-threading and dynamic class loading in a java just-in-time compiler. In: CGO 2006, Washington, DC, USA, pp. 87–97 (2006)
26. Suzuki, N., Ishihata, K.: Implementation of an Array Bound Checker. In: POPL 1977, pp. 132–143 (1977)
27. Triolet, R., Irigoin, F., Feautrier, P.: Direct parallelization of call statements. *SIGPLAN Not.* 21, 176–185 (1986)
28. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.* 13(2), 181–210 (1991)
29. Yelick, K., et al.: Titanium: A High-Performance Java Dialect. *Concurrency: Practice and Experience* 10(11) (September 1998)
30. Zhao, Y., Kennedy, K.: Scalarizing Fortran 90 Array Syntax. Technical Report TR01-373, Department of Computer Science, Rice University (2001)

Practical Loop Transformations for Tensor Contraction Expressions on Multi-level Memory Hierarchies

Wenjing Ma¹, Sriram Krishnamoorthy², and Gagan Agrawal¹

¹ The Ohio State University, Columbus, OH, 43210, USA
`{mawe, agrawal}@cse.ohio-state.edu`

² Pacific Northwest National Lab, Richland, WA, 99352, USA
`sriram@pnl.gov`

Abstract. Modern architectures are characterized by deeper levels of memory hierarchy, often explicitly addressable. Optimizing applications for such architectures requires careful management of the data movement across all these levels. In this paper, we focus on the problem of mapping tensor contractions to memory hierarchies with more than two levels, specifically addressing placement of memory allocation and data movement statements, choice of loop fusions, and tile size selection. Existing algorithms to find an integrated solution to this problem even for two-level memory hierarchies have been shown to be expensive. We improve upon this work by focusing on the first-order cost components, simplifying the analysis required and reducing the number of candidates to be evaluated. We have evaluated our framework on a cluster of GPUs. Using five candidate tensor contraction expressions, we show that fusion at multiple levels improves performance, and our framework is effective in determining profitable transformations.

1 Introduction

Starting from the last 4-5 years, chip designers cannot enable sequential code to be executed faster through increases in clock rates and/or instruction-level parallelism. This paradigm shift is impacting high performance computing as well, as clusters of uniprocessors are no longer the dominant HPC architectures. Instead, clusters of multi-core and/or many-core architectures are becoming extremely common, and overall, architectures are becoming increasingly heterogeneous and complex. One emerging trend in HPC is towards clusters of GPUs. In the list of top 500 supercomputers released in May 2010, two of the top seven fastest supercomputers are based on GPUs. Because of the overall popularity and cost-effectiveness of GPUs, this trend can be expected to continue.

Developing applications on these emerging systems involves complex optimizations. Among other issues, orchestrating data movement across multiple levels of hierarchy, and restructuring the computation to reduce the cost of such data movement, are both challenging problems. As a specific example, consider processing of a disk-resident dataset on a machine with one or more GPUs. If

the size of the main memory is smaller than the size of the dataset, there will be significant data movement costs between the disk and main memory. Similarly, it is likely that the GPU device memory is smaller than the main memory. Even though GPU memory capacities have increased rapidly, high-end GPUs today have 3-6 GB memory, and cheaper or older GPUs have much less memory. In comparison, servers today easily have 12 or more GB of main memory. Thus, data movements between main memory and device memory also need to be carefully planned. Moreover, when considering a cluster of GPUs and/or a single node connected with multiple GPUs, the data movement and optimization problems are even more challenging.

Optimizations such as tiling and loop fusion can enable better data reuse and reduce the cost of these data movements. However, what we need is an integrated framework to enable these transformations across more than 2 levels of memory hierarchy. There has been a considerable amount of work on compiler optimizations for out-of-core programs [6,18,17,5,36], but the solutions do not directly extend when data movements for another level need to be considered. Similarly, there has been some work on optimizing data movements from main memory to device memory [35,34,22]. As multi-level processor caches became very common in mid-nineties, several compiler efforts considered optimizations for them [25,32,30].

This paper presents an optimization framework for systems requiring data movement across multiple memory hierarchy levels. Our work is specifically in the context of tensor contractions [33,14], which can be viewed as generalized matrix products. Sequences of tensor contractions arise in several domains, particularly, in *ab initio* computational models in quantum chemistry. Such contractions are often performed over several large multi-dimensional arrays, making it very likely that input data does not fit in main memory. At the same time, these applications are compute-intensive, and can benefit from GPUs.

We present an extensive framework for optimizing tensor contractions on a system requiring explicit data movement across three memory hierarchy levels. We address the following challenges: 1) determining the loop structure, which comprises of fused and tiled loops, 2) the placement of memory allocation and data movement statements for each of the memory hierarchy levels, and 3) the tile sizes. While we specifically target machines with GPUs and problems with out-of-core data structures, our framework can be applied on any system with more than 2 explicitly managed memory hierarchy levels.

2 Background

The tensor contraction expressions we consider in this paper arise, for example, in the context of the Coupled Cluster (CC) theory [3], a widely used method for solving the electronic Schrödinger equation. A tensor contraction expression is comprised of a collection of multi-dimensional summations of product of several input tensors or arrays. Consider the following contraction:

$$B(a, b, c, d) = \sum_{p, q, r, s} C1(s, d) \times C2(r, c) \times C3(q, b) \times C4(p, a) \times A(p, q, r, s)$$

This contraction is referred to as a *four-index transform*. Here, $A(p, q, r, s)$ is a four-dimensional input tensor and $B(a, b, c, d)$ is the transformed (output) tensor. $C1$, $C2$, $C3$, and $C4$ are the *transformation* tensors. The indices a, b, \dots denote the dimensionality of the tensors and the relationship between them across tensors. Minimizing the operation count results in the following sequence of binary tensor contractions:

$$T3(a, q, r, s) = \sum_p C4(p, a) \times A(p, q, r, s); T2(a, b, r, s) = \sum_q C3(q, b) \times T3(a, q, r, s)$$

$$T1(a, b, c, s) = \sum_r C2(r, c) \times T2(a, b, r, s); B(a, b, c, d) = \sum_s C1(s, d) \times T1(a, b, c, s)$$

where $T1$, $T2$, and $T3$ are *intermediate* tensors. In each contraction, a set of indices are summed over, or *contracted*, and are referred to as the *summation (or contracted) indices*. These indices occur in both inputs of a binary contraction and are also referred to as *common* indices. This is a generalization of matrix-matrix multiplication with individual indices replaced by sets of indices.

The size of each dimension (or loop index) varies from 100 to a few thousand. As a result, the input, output, and the intermediate tensors often do not fit into the main memory of a single node. This, together with the high floating-point intensity of these expressions, lends themselves to execution on a parallel system, and considerable effort has been put on parallelization of these methods [15,2].

Effective execution of tensor contraction expressions on parallel systems with multi-level memory hierarchies necessitates several transformations. In particular, the allocation and freeing of memories in the the multi-level hierarchy together with the associated data transfer needs to be carefully orchestrated. While the execution is dominated by the floating point operations, data movement can quickly overwhelm the total execution time if not effectively managed. Tiling directly benefits these calculations, analogous to its applicability to matrix-matrix multiplication implementations. While much work on matrix multiplication focuses on square matrices, tensor contractions often involve highly rectangular matrices, exacerbated by the higher dimensionality involved. In particular, it is not uncommon for one of the arrays involved in a contraction to be small enough to fit at some higher-level of the memory hierarchy, such as in the example above.

Loop fusion is an integral part of effective data locality management in such multi-level memory hierarchies. Much of the early work on loop fusion either focused on identifying the feasibility of fusing two loops, fusing the identified loops [23,24], possibly with other transformations that enable loop fusion [38], or a general transformation framework that encompasses loop permutation, fusion, and other transformations under an abstract cost model [21,11,19].

In this paper, we focus on identifying effective fusions for tensor contractions with as concrete a cost model as possible. Specifically, we consider data movement costs under memory constraints. For systems with only two levels of memory hierarchy, this problem has been addressed before by Sahoo *et al.* [33].

We note that, unlike more general application classes, tensor contractions, which consist of fully permutable loops, can be fused without requiring enabling loop transformations.

3 Problem Statement and Notation

Given a sequence of binary contractions, such as the one in Section 2, efficient code tailored to the specific memory hierarchy needs to be generated. We focus on machines comprising more than two levels in the memory hierarchies. The key decisions to be made are 1) determining the loop structure, comprising of fused and tiled loops, 2) the placement of memory allocation and data movement statements for each of the memory hierarchy levels, and 3) the tile sizes. It has been shown previously that the the space of possible choices is exponentially large [33]. Besides considering additional levels in the memory hierarchy in determining fusion structures, we focus on the prescriptive approaches to choosing candidate loop fusions without requiring expensive optimization procedures.

In our presentation, N_i, N_j, \dots and T_i, T_j, \dots denote sizes of full dimensions and individual tiles along dimensions i, j, \dots , respectively. Lower case letters denote loop and array reference indices, or sets of indices. $A[a, b]$ denotes a tensor indexed by a and b . In addition to identifying the dimensionality of the tensor, the labels (here a and b) identify the relationship between the indices of the tensors participating in a tensor contraction. The memory levels are denoted by α, β, \dots , with α being the slowest memory hierarchy level. The size of memory available at each level is denoted as M_α, M_β, \dots , and the time to move one element of the data between adjacent levels is $\Lambda_{\alpha\beta}, \Lambda_{\beta\gamma}, \dots$. We assume that data movement is always performed between adjacent levels of the memory hierarchy. The cost to perform one double precision floating point operation is denoted by C_{flop} .

4 Single Contraction Optimization

In this section, we determine the loop structure to map a single tensor contraction to a multi-level memory hierarchy. We first summarize existing results for the case when data movement is considered at only a single level (e.g. from disks to main memory). A detailed description is available from an earlier publication [33]. In a tensor contraction, the indices can be grouped into those that are contracted and the remaining indices in the two input tensors. Given that the loops are fully permutable and we reason about the total data movement volume, ignoring issues such as stride of access, the indices can be grouped into three composite indices. In the ensuing discussion, we refer to the contracted indices as N_k , and the remaining indices in the two inputs as N_i and N_j .

One Array Fits in Memory: Consider the three tensors $X[x, y]$, $Y[y, z]$, and $Z[x, z]$ in a tensor contraction, where X , Y , and Z correspond to the three tensors participating in a binary tensor contraction, one of them being the output tensor. Without loss of generality, assume that X fits in memory. Let the size along the

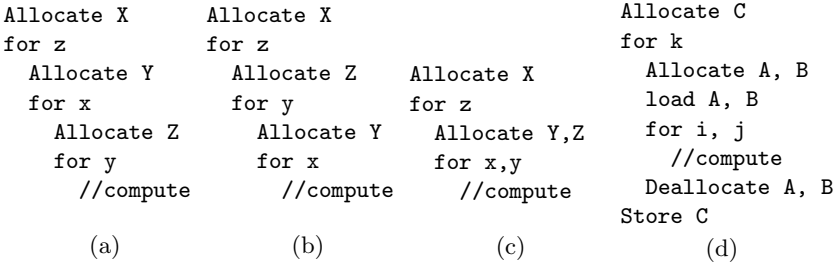


Fig. 1. Loop structures for a single contraction on a two-level memory hierarchy. (a) Loop structure if $N_y \leq N_x$ (b) Loop structure if $N_x \leq N_y$ (c) Simplified loop structure for discussion (d) Example code for the contraction $C = A \times B$.

dimensions $x, y,$ and z be $N_x, N_y,$ and $N_z,$ respectively. It can be shown that to minimize the data movement costs, the smallest array is retained in memory, with the other two arrays being streamed. This results in a memory cost of [33]:

$$N_x \times N_y + \min(N_x, N_y) + 1 \leq M_\beta \tag{4.1}$$

We approximate Equation 4.1 by replacing the left hand side with $N_x \times N_y.$ For values of M_β encountered in practice, the value of the min term in the worst case ($N_x = N_y = \sqrt{M_\beta}$) is much smaller than the memory size, and can be ignored.

The loop structures for the cases where one of the arrays fits in the memory are shown in Figure 1, where X, Y, Z represent any of the three tensors. Note that none of the tensors are surrounded by loops that do not index it. This ensures that there is no redundant data movement for any of the arrays, achieving the minimum possible data movement cost, i.e. we read each input array and write the output array exactly once [33]. The position of a tensor in the loop structure identifies both data movement and memory management steps. Viewing the loop structure as a loop nest tree, a tensor label, denoted by an allocate statement for a tensor, identifies its position in the loop nest tree. The memory allocation and read statements (if the array is read) are inserted into the structure before the code corresponding to the nested statements are generated. After the processing of all nested statements, the deallocation and write statement (if it is an output array) are generated. While the optimal choice of loop structure that minimizes the memory usage is given by Figure 1(a) or Figure 1(b), the difference in practice is a small reduction in memory overhead, not the dominant memory overhead or data movement cost. We, therefore, consider the simplified structure shown in Figure 1(c) for further analysis. Figure 1(d) illustrates the insertion of memory allocation and data movement statements for the abstract structure shown in Figure 1(c) for the contraction $C = A \times B.$

No Arrays Fit in Memory: For a two-level memory hierarchy, say the $\alpha \leftrightarrow \beta$ levels, it has been shown that the optimal loop structure involves the data

transfer for the output array being higher in the loop structure than that for the other two arrays. Recalling that tile sizes along X and Y dimensions are T_i and T_j , respectively, the data movement and memory costs are now given by:

$$Data_{\alpha\beta} = A_{\alpha\beta} \times \left(N_i \times N_j + N_i \times N_j \times N_k \times \left(\frac{1}{T_i} + \frac{1}{T_j} \right) \right) \quad (4.2)$$

$$Memory_{\beta} = T_i \times T_j + T_i + T_j < M_{\beta} \quad (4.3)$$

Minimizing the data movement under the memory constraint results in $T_i = T_j = T \approx \sqrt{M_{\beta}}$. Intuitively, optimized mapping of a tensor contraction to a two-level memory hierarchy requires maximizing the tile size for the output array with a small amount of data for the input arrays brought in at a time.

4.1 Multi-level Memory Hierarchies

We now focus on extending the existing work to the systems where data movement is required across more than two levels in the memory hierarchy. Specifically, consider a memory hierarchy consisting of three levels, namely α , β , and γ . The total data movement cost is the sum of costs of the two $\alpha \leftrightarrow \beta$ and $\beta \leftrightarrow \gamma$ data transfers. We shall assume in this discussion that levels cannot be bypassed, with data movement always occurring between adjacent levels. We also assume that the memories get faster and smaller as we move along the memory hierarchies, with α being the slowest level with infinite capacity.

First, if one of the tensors fits into γ , minimal data movement can be achieved by placing the array in γ , or reading directly into γ , and streaming the other two arrays through all levels of memory hierarchy using the loop structure shown in Figure 1. Each tensor label in such a schedule corresponds to data transfer across all memory hierarchy levels.

Consider the data movement order with all $\alpha\beta$ transfers grouped to nest the $\beta\gamma$ transfers, with $C_{\alpha\beta}$ being the outermost data placement and the data transfer for the A and B arrays at the same loop nesting. As explained earlier, we ignore the nesting between A and B given the limited memory usage improvement with no change in data movement cost. The costs for this data placement are given by:

$$Data_{\alpha\beta} = A_{\alpha\beta} \left(N_i \times N_j + N_i \times N_j \times N_k \left(\frac{1}{T_i} + \frac{1}{T_j} \right) \right) \quad (4.4)$$

$$Data_{\beta\gamma} = A_{\beta\gamma} \times N_i \times N_j \times N_k \left(\frac{1}{T'_i} + \frac{1}{T'_j} + \frac{q_k}{T'_k} \right) \quad (4.5)$$

$$Memory_{\beta} = T_i \times T_j + T_i \times T_k + T_j \times T_k \quad (4.6)$$

$$Memory_{\gamma} = T'_i \times T'_j + T'_i \times T'_k + T'_j \times T'_k \quad (4.7)$$

Here, T'_i , T'_j , and T'_k are the tile sizes at the γ level, and q_k is 1 if $T_k = N_k$, 2 otherwise. This is because when $T_k = N_k$, the values of C could be written to the output without having to be loaded and accumulated. The tile sizes for the data movement placement can be determined by solving this non-linear constrained optimization problem. However, the solution requires additional information on the problem size, which is not always available to a compiler. Thus, here our focus

is on deriving prescriptive solutions that do not require precise information on the problem size. Where the problem size is needed, we reduced the determination to a quick runtime decision. In the process, we demonstrate the tile sizes for optimal data movement in a single-level memory hierarchy need not match those of a multi-level memory hierarchy.

Note that the data transfer cost at the $\beta\gamma$ level depends on the tile size T_k determined for the $\alpha\beta$ data transfer. The optimal data transfer for the single-level memory hierarchy, when no array fits in β , involves $T_i = T_j = \sqrt{M_\beta}$ and $T_k = 1$. Even under the assumption that $A_{\alpha\beta}$ is much higher than $A_{\beta\gamma}$, for large values of N_k , the data transfer cost is dominated by $Data_{\alpha\beta} + A_{\beta\gamma}N_i \times N_j \times N_k$. We thus consider another alternative, with $T_i = T_j = T_k = \sqrt{M_\beta/3}$. This corresponds to equal amounts of all three arrays being stored in memory at any time, a less than optimal choice for a two-level memory hierarchy. The total data movement cost (with $T'_i = T'_j = \sqrt{M_\gamma}$) is given by:

$$N_i \times N_j \times \left[A_{\alpha\beta} \times \left(1 + N_k \times \frac{2\sqrt{3}}{\sqrt{M_\beta}} \right) + A_{\beta\gamma}N_k \times \left(\frac{2}{\sqrt{M_\gamma}} + \frac{2\sqrt{3}}{\sqrt{M_\beta}} \right) \right] \quad (4.8)$$

Comparing this cost with Equations 4.4 and 4.5, we observe that equal tile sizes are better when:

$$\frac{2}{\sqrt{M_\beta}}A_{\alpha\beta} + 2A_{\beta\gamma} \geq \frac{2\sqrt{3}}{\sqrt{M_\beta}}(A_{\alpha\beta} + A_{\beta\gamma}) \implies M_\beta \geq \left(\frac{(\sqrt{3} - 1)A_{\alpha\beta}}{A_{\beta\gamma}} + \sqrt{3} \right) \quad (4.9)$$

This condition is evaluated with $q_k = 2$ in the above determination. It can be quickly evaluated at install time to choose the best tile size. Intuitively, the data transfer between the $\alpha\beta$ memory levels is increased to significantly reduce the cost of transfer between the $\beta\gamma$ levels, keeping it relatively small. While globally optimizing the objective function yields the best solution, the prescriptive solution is independent of the problem size while achieving similar ends.

When one of the arrays fits in β , the optimal single-level solution involves fitting the array in β and streaming the other arrays. When it fits within $(M_\beta/3)$, both schemes result in similar cost functions and the tile sizes chosen for the single-level optimal solution are chosen. However, for the smallest array size between $(M_\beta/3)$ and M_β , we have a choice between the two tiling alternatives. With the streaming scheme, without loss of generality, we assume the smallest matrix is A , then by keeping the smallest matrix in β , the tile sizes are $T_i = N_i$, $T_k = N_k$, and $T_j \leq \frac{M_\beta - N_iN_k}{N_i + N_k}$. The data movement at the $\alpha\beta$ level will be

$$A_{\alpha\beta}(N_iN_k + N_jN_k + N_iN_j) \quad (4.10)$$

This tiling decision at the $\alpha\beta$ level results in tensors of sizes T_j , T_jN_k , and T_jN_i residing in β , which need to be scheduled for data movement across $\beta\gamma$. Since N_iN_k is larger than $(M_\beta/3)$, we assume it does not fit in γ . However, because of possibly smaller T_j , T_jN_k , or T_jN_i might fit γ . In this case, streaming is done in a way similar to β level. Suppose the smallest tile at β level is from B , which is of the size T_jT_k , then the total data movement at $\beta\gamma$ is

$$A_{\beta\gamma} \left(N_jN_k + N_iN_j + \frac{N_iN_kN_j}{T_j} \right) = A_{\beta\gamma}N_j(N_k + N_i) \left(1 + \frac{N_iN_k}{M_\beta - N_iN_k} \right) \quad (4.11)$$

When none of the data tiles fit in γ , a tiling solution analogous to the single-level tiling presented in Section 4 is employed. This results in the total data movement cost at $\beta\gamma$ being:

$$A_{\beta\gamma} \left(\frac{N_j N_k}{\sqrt{M_\gamma}} + \frac{N_i N_j}{\sqrt{M_\gamma}} + \frac{N_i N_k N_j}{T_j} \right) = A_{\beta\gamma} N_j (N_i + N_k) \left(\frac{2}{\sqrt{M_\gamma}} + \frac{N_i N_k}{M_\beta - N_i N_k} \right) \tag{4.12}$$

The total data movement cost is computed from the $\alpha\beta$ cost (given by Equation 4.12) together with the $\beta\gamma$ cost. The latter is determined using either of the Equations 4.11 and 4.12, based on the problem size. By comparing this overhead with the data movement using equal tile sizes, whose cost is given by the Equation 4.8, the tiling decision could be made at runtime. The decision could also be made at installation time for different matrix shapes, which impact the $(N_i + N_k)$ term.

5 Fusion for Tensor Contraction Sequences

In this section, we describe our approach to determining fused loop structures for multi-level memory hierarchies. We begin with an analytical approach to identify profitable loop fusions for tensor contractions. Intuitively, we observe that loop fusion can result in substantial performance gains only if the overall execution time is bound by data movement, and not computation. Consider a series of loops, which correspond to the following series of tensor contraction expressions.

$$\begin{aligned} I_1(d, c_2, \dots, c_n) &= I_0(d, c_1, \dots, c_n) \times B_0(d, c_1, \dots, c_n) \\ I_2(d, c_3, \dots, c_n) &= I_1(d, c_2, \dots, c_n) \times B_1(d, c_2, \dots, c_n) \\ &\dots \\ I_n(d) &= I_{n-1}(d, c_n) \times B_{n-1}(d, c_n) \end{aligned}$$

The tensors B_0, \dots, B_{n-1} and I_0 are the *input* tensors and I_n is the *output* tensor. I_1, \dots, I_{n-1} are the *intermediate tensors*. Data movement costs for such intermediate tensors can potentially be eliminated through fusion. c_j corresponds to the indices contracted in the production of I_j . The various d indices are never contracted out and contribute to the indices in I_n . $I_i(d)$ represents the set of indices I_i contributes to the final output. $I_i(c_j)$ denotes the indices in I_i that are a subset of the contracted indices in contraction j . I_i denotes both the tensor produced by contraction i and set of all indices that constitute that tensor. The reference shall be clear from the context. $|I_j(d)|$, and similar usage, denotes the total size of the indices in tensor I_j that contribute to the final output. The relationship between the indices is defined as:

$$\begin{aligned} 1 \leq i \leq n : I_i(d) &= I_{i-1}(d) \cup B_{i-1}(d) \\ 1 \leq i \leq n, 1 \leq j \leq i : I_i(c_j) &= \emptyset \\ 1 \leq i \leq n, i + 1 \leq j \leq n : I_i(c_j) &= I_{i-1}(c_j) \cup B_{i-1}(c_j) \\ 0 \leq i \leq n - 1 : I_i(c_{i+1}) &\equiv B_i(c_{i+1}) \end{aligned}$$

The fusion of the loops corresponding to the above tensor expressions is beneficial only if the data movement cost for the intermediate tensors (I_1 through I_{n-1}) dominates, or is at least comparable to, the data movement cost for the input and the output tensors and the computation cost.

Consider the fusion of the first two contractions. The computation cost of the first contraction is $C_{flop} \left(|I_0(d)| \times \prod_{j=1}^n |I_0(c_j)| \times |B_0(d)| \times \prod_{j=2}^n |B_0(c_j)| \right)$. The minimum data movement cost for I_1 that would be eliminated through loop fusion for the $\alpha\beta$ levels is given by: $2 \times \Lambda_{\alpha\beta} \times \left(|I_1(d)| \times \prod_{j=2}^n |I_1(c_j)| \right)$, the cost of reading and writing the tensor once for each of the contractions. Under the requirement that this is greater than the fraction, $0 < frac \leq 1$, of the computation cost of the two relevant contractions, we have:

$$2 \times \Lambda_{\alpha\beta} \geq frac \times C_{flop} \times |I_0(c_1)|$$

leading to

$$|I_0(c_1)| \leq \frac{2 \times \Lambda_{\alpha\beta}}{frac \times C_{flop}} \quad (5.1)$$

For the second contraction, the computation cost is given by

$C_{flop} \times \left(|I_1(d)| \times \prod_{j=2}^n |I_1(c_j)| \times |B_1(d)| \times \prod_{j=3}^n |B_1(c_j)| \right)$. If we want the execution to be dominated by the movement of I_1 , the condition now becomes

$$2\Lambda_{\alpha\beta} \geq frac \times C_{flop} \prod_{j=3}^n |B_1(c_j)| |B_1(d)| \implies \prod_{j=3}^n |B_1(c_j)| |B_1(d)| \leq \frac{2 \times \Lambda_{\alpha\beta}}{frac \times C_{flop}} \quad (5.2)$$

Therefore, for a contraction to be fused with both its previous and the next contraction, the input B_i has to satisfy both of the following requirements:

$$\begin{aligned} |I_i(c_{i+1})| &\leq \frac{2 \times \Lambda_{\alpha\beta}}{frac \times C_{flop}} \\ \prod_{j=i+2}^n |B_i(c_j)| \times |B_i(d)| &\leq \frac{2 \times \Lambda_{\alpha\beta}}{frac \times C_{flop}} \end{aligned} \quad (5.3)$$

Combining the above two expressions, together with the fact that $I_i(c_{i+1}) = B_i(c_{i+1})$, the size of B_i should be less than $\left(\frac{2 \times \Lambda_{\alpha\beta}}{frac \times C_{flop}} \right)^2$. In current systems, this number is typically much smaller than the memory size. We therefore assume that the B tensors in all contractions in the sequence considered, except the first and last contraction, fit in memory.

The above evaluation does not preclude the B arrays in the first or last contraction in a fusible list, B_0 and B_{n-1} in the above example, from being too large to fit in memory. In these cases, fusion eliminates the data movement cost to read and write the intermediate tensor. On the other hand, fusion requires this contraction to share the available memory with other contractions, potentially increasing the cost incurred due to duplicated data movement.

Consider B_0 being large, with qM ($0 \leq q \leq 1$, M_β is the memory limit of level β) being the amount of memory required for fused execution of contractions

```

S1=Ii∩ Ii+1∩ Ii+2
S2=Ii ∩ Ii+1, S3 = Ii+2
S4 = Ii
for sx ∈ S1 do
    {Allocate Ii+1[sx]};
    for sy ∈ S2 − S1 do
        {Allocate Ii[sy]};
        for sz ∈ S4 − S2 do
            {Produce Ii[sz]};
        end for
        {Update Ii+1[sy]};
    end for
    for sw ∈ S3 − S1 do
        {Allocate Ii+2[sw]};
        {Produce Ii+2[sw]};
    end for
end for
    
```

Algorithm 1. Sample loop structure with allocation for I_i, I_{i+1} , and I_{i+2}

producing tensors I_2 through I_n in the running example. Comparing the two alternatives discussed above, fusing the first contraction is beneficial when the condition below is true:

$$\begin{aligned}
 \frac{2|I_1||I_0(c_1)|}{\sqrt{M_\beta}} + 2|I_1| &\geq \frac{2|I_1||I_0(c_1)|}{\sqrt{(1-q)M_\beta}} \implies \frac{|I_0(c_1)|}{\sqrt{M_\beta}} + 1 \geq \frac{|I_0(c_1)|}{\sqrt{M_\beta}\sqrt{1-q}} \\
 &\implies \frac{\sqrt{1-q}}{1-\sqrt{1-q}} \geq \frac{|I_0(c_1)|}{\sqrt{M_\beta}} \quad (5.4)
 \end{aligned}$$

As shown in equation 5.1, $|I_0(c_1)|$ is small, usually much smaller than the memory available. The expression on the left hand side of last inequality above is greater than 1 for values of $q \leq 0.7$. Thus fusion is beneficial despite duplicated data movement for one of the arrays even when up to 70% of the memory is consumed by the remaining contractions. Note that this condition can be quickly verified at runtime once the problem size is known.

5.1 Two-Level Memory Hierarchy

Based on the discussion above, we assume that for a given sequence of tensor contractions to be fused, the non-intermediate arrays in all contractions, except for the first and last contraction, must fit in the memory. For simplicity, we further assume that all such arrays together fit in the available memory. Based on this assumption, we will first present a solution to the problem of identifying a loop structure, with placement of data movement statements for all the intermediate arrays and one of the input arrays in the first contraction and the output array in the last contraction, assuming the remaining arrays fit in memory. This will later be extended to support the scenario in which the first or the last contraction require redundant data movement.

We simplify our presentation by assuming that all dimensions are similar in size. Consider a contraction list, referred to as a *fusable list*, in which all contractions are *fusable* according to the analysis presented in the previous subsection. We now determine the actual fusion chain and locations of the data allocation and movement statements. Let the fusable list be I_0, \dots, I_n . For any three tensors I_i, I_{i+1} , and I_{i+2} , a sample loop structure is as shown in Algorithm 1, where the intersection operation denotes the intersection of indices of two tensors. The key observation here is that the loop nest producing I_{i+1} must be enclosed by a memory allocation for I_i , since the loop nest consumes I_i . However, the allocation statements for tensors I_i and I_{i+2} need not enclose each other, since they do not have an immediate producer-consumer relationship. The data movement statements can be in arbitrary order as long as this condition is satisfied.

In this sample loop structure, the total memory cost is given by $mem(I_{i+1}) + \max(mem(I_i), mem(I_{i+2}))$. Since the unfused loop nest for each contraction, with one of the arrays fitting in memory, does not incur any redundant data movement, we consider only loop nests with no redundant data movement. The total memory cost can be determined through the recurrence relation shown below:

$$\begin{aligned} f(i, j) &= 0 \quad , \quad \text{if } j < i \\ &= \min_{k=i}^j \frac{|I_k|}{|Common_{i,j}|} + \max(f(i, k-1), f(k+1, j)) \quad , \quad \text{otherwise} \\ Common_{i,j} &= \cap_{k=i}^j I_k \end{aligned}$$

The recurrence is evaluated as $f(1, n)$ to compute the values of all $f(i, j)$ through dynamic programming. $Common_{i,j}$ denotes all the loops that are common to tensor contractions i through j . The procedure to determine the loop structure first involves computing the memory required by arbitrary sub-chains of the “fusable list” memoized in a matrix. The function computes the data movement cost, in terms of volume, of executing a given chain of tensor contractions, ignoring the data movement for the first input and the last output. If the entire chain can be fused with the memory available, the data movement cost is evaluated to be zero. If not, various choices of splitting the chain into sub-chains are considered, and the choice that minimizes the data movement cost is determined recursively. For each split the array at the split incurs the data movement cost equal to twice its size, to write it as output from the first sub-chain and read it back in the next sub-chain. Note that the algorithms shown only compute the memory and data movement costs. Determining the actual loop structure including the loop nesting to obtain the memory cost and splits to achieve the data movement cost can be determined by keeping track of the choices made while minimizing the cost function. The procedure is similar to that employed in the dynamic programming solution to matrix-chain multiplication [9].

When either the first or the last contraction involves large tensors necessitating duplicated data movement, we determine the optimal fusion structures for the fusable list excluding the contractions requiring such duplicated data movement. Let us consider the case with I_0, B_0 , and I_1 being too large to fit in

memory. As discussed in Section 5.1, the loops in the second contraction consuming the output of the first contraction should be enclosed by the memory allocation statement for the output tensor of the first contraction. In the running example, the allocation statement for I_1 must enclose the production of I_2 . Positions in the loop structure determined for contractions 2 through n that satisfy this requirement are determined. If the memory consumed by all live tensors at any of these positions is small enough to enable profitable fusion, as expressed by the condition 5.4, the first contraction is fused. This procedure is repeated for the last contraction in the list when it incurs duplicated data movement.

5.2 Multi-level Memory Hierarchies

In the previous section, we presented the steps to determine the fusion structure that minimizes the data movement cost given a memory constraint. Here we show how the procedure naturally extends to handle multi-level memory hierarchies. We have considered fusion as a transformation that combines the loop corresponding to the producer and consumer such that the data movement cost for the associated intermediate tensor is eliminated. Effective fusion of a sequence of contractions for data movement between the $\beta\gamma$ levels implies that all contractions in the chain can be executed without intervening $\beta\gamma$ data transfer, with sufficient space in γ to fit all necessary intermediates. A sub-chain can be fused at the γ level if the memory required by it, as computed by this expression, is less than M_γ . Given the memory cost for each sub-chain at γ , the memory cost at the β level is calculated using the following modified recurrence expression:

$$\begin{aligned}
 f(i, j) &= 0, \text{ if } j < i \\
 &= \frac{|I_i| + |I_j|}{|I_i \cap I_j|} \text{ if } \text{memory}_\gamma(i, j) \leq M_\gamma \\
 &= \min_{k=i}^j \frac{|I_k|}{|\text{Common}_{i,j}|} + \max(f(i, k - 1), f(k + 1, j)), \text{ otherwise} \\
 \text{Common}_{i,j} &= \cap_{k=i}^j I_k
 \end{aligned}$$

This expression takes into account the fact that when a sub-chain is fused at the γ level, only the first and last tensor in that sub-chain consume memory at the β level. The total data movement cost for fusion in the presence of a multi-level memory hierarchy is computed using a modified version of procedure employed for a two-level memory hierarchy. If the memory required at the β level to execute a sub-chain is less than M_β , the corresponding data movement cost is determined. The data movement cost for a sub-chain fusable at the γ level is given as the cost to transfer the first and last tensor in the chain between the $\beta\gamma$ levels. We assume that despite fusion of some of the loops, the dimensions of the remaining loops are large enough for tiling all dimensions at the γ level. When a sub-chain involves a single contraction, due to the need for redundant data movement, its total data movement cost is computed as discussed in Section 4.1.

Expression A (Small Input):

$$C[h1,h2,h3,h4,h5,h6] = A[h1,h2,h3,h7] \times B[h4,h5,h6,h7]$$

$$I[h1,h2,h3,h4,h5,h8] = B2[h8,h6] \times C[h1,h2,h3,h4,h5,h6]$$

Expression B (Large Common Index):

$$C[h6,h5,h4,h3,h2,h1] = A[h7,h5,h4,h3,h2,h1] \times B[h7,h6]$$

$$I[h8,h2,h1] = B2[h8,h6,h5,h4,h3] \times C[h6,h5,h4,h3,h2,h1]$$

Expression C (Large Input):

$$C[h8, h7, h6,h5,h4,h3,h2,h1] = A[h9, h4,h3,h2,h1] \times B[h9, h8, h7,h6,h5]$$

$$I[h10,h7,h6,h5,h4,h3,h2,h1] = B2[h10,h8] \times C[h8, h7, h6,h5,h4,h3,h2,h1]$$

Expression D (Long Chain):

$$M1[h1,h2,h3,h4,h5,h7] = M0[h1,h2,h3,h4,h5,h6] \times B1[h6,h7]$$

$$M2[h1,h2,h3,h4,h5,h8] = M1[h1,h2,h3,h4,h5,h7] \times B2[h5,h8]$$

$$M3[h1,h2,h3,h9,h7,h8] = M2[h1,h2,h3,h4,h5,h8] \times B3[h4,h9]$$

$$M4[h1,h2,h10,h9,h7,h8] = M3[h1,h2,h3,h9,h7,h8] \times B4[h3,h10]$$

$$M5[h1,h11,h10,h9,h7,h8] = M4[h1,h2,h10,h9,h7,h8] \times B5[h2,h11]$$

Expression E (Fewer Dimensions and Large Dimension Size):

$$C[h1,h2,h3,h4] = B[h3,h5] \times A[h1,h2,h4,h5]$$

$$I[h1,h2,h3,h6] = B2[h6,h4] \times C[h1,h2,h3,h4]$$

Fig. 2. Tensor contraction expressions used as benchmarks

6 Experimental Evaluation

We evaluate our methods by executing tensor contractions on a system with an explicitly managed, multi-level, memory hierarchy. Particularly, the system consisted of the following three levels in the memory hierarchy: disk, global memory, and the *local* or *device* memory. The global memory corresponds to the use of *global arrays* [27] on the cluster. The device memory is the memory on the GPU, which had a similar size as the local memory on each node. Since all processing is done on the GPU, processor cache is not an important component of the memory hierarchy. Within the GPU, the programmable cache or the *shared memory* reflects another level in the hierarchy, but did not turn out to be important for our target class of applications. In particular, the shared memory in the GPUs we used was too small to enable fusion. The impact of tiling on kernels such as matrix-matrix multiplication has been extensively evaluated in a variety of contexts [25,31,16,8]. While the tile sizes are derived as discussed earlier, we focus on demonstrating the impact of our approach to fusion.

The specific configuration used was a cluster environment where each computing node has two quad-core Intel Xeon X5560 CPUs, running at 2.80 GHz. Every pair of nodes shares a Tesla 1070 GPU box, which means every node has two GPUs available as accelerators. For simplicity, in our experiments, we launch two processes on each node, so that every process can use one GPU for acceleration. Our analytical models assume that there is a 32 GB global memory and 1 GB local memory for the application's data structures. In all the experiments, the input data and the final output are stored on the disk. Input files are

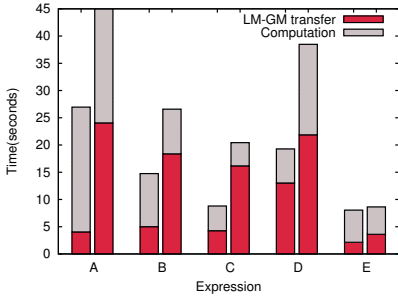


Fig. 3. Execution time with (left) and without (right) fusion at global memory level

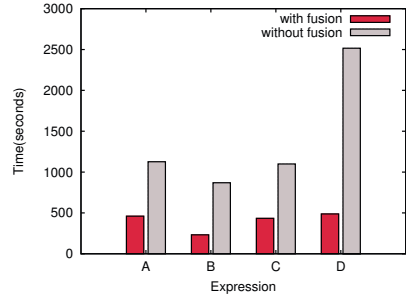


Fig. 4. Disk I/O time with (left) and without (right) fusion at disk level

first loaded into the global memory, then moved to the local memory. Storage of output file follows the reverse order. Communication between disk and global memory is *collective*, whereas, the data movement between the global memory and the local memory is *one-sided* (each processor transfers data independently).

Our evaluation was based on five candidate sets of contractions, chosen to reflect the nature of the contractions in quantum chemistry. These benchmark expressions are listed in Figure 2. For each expression, we evaluate the benefit of fusion at both the global memory level and the disk level. Fusion at the global memory level implies that the intermediate data resides in the device memory, whereas fusion at the disk level implies that the intermediate data is kept in the global memory. As a baseline, we also created a non-fused version which refers to the case when all intermediate results are copied to the disk and need to be loaded for the next contraction. Our approach predicts that fusion is beneficial at the disk level for all five expressions, while fusion at the global memory level is beneficial for expressions *A*, *B*, *C*, and *D*.

6.1 Effectiveness on Different Expressions

We comparison the fused and non-fused versions in terms of computation cost, the transfer time between global and local memory levels, and disk I/O cost. In all the charts, LM-GM implies data transfer time between the local memory and the global memory. In Figures 3, 5, and 7, the left bar in each *bar cluster* denotes the time with fusion, and the right bar denotes the time without fusion. **Computation** time reported here includes time spent in floating point operations and data movement between local memory and the GPU device memory. Therefore, in some cases, the computation time reported is shorter in the version with fusion. The disk I/O times, which could be much higher than the other components, are shown in Figures 4, 6, and 8.

Expression A – Fusion with small input matrices: The Expression A involves two contractions, where all input matrices fit in local memory. Data transfers for the matrices *C* and *I* are the dominating factors in this computation,

and fusion eliminates the overhead of moving C . The first cluster of bars in Figure 3 and Figure 4 show the benefit achieved by fusion at the disk and global memory levels, respectively. The experiments were done with the dimension size 40, on four computing nodes. We can see that the fusion at the GM level reduces the sum of computation and the local memory transfer time by about 40%. On the other hand, fusion at disk to global memory level is playing an even more important role, since the execution is dominated by this part. From Figure 4, we can see that fusion results in speedup of about 2.4.

Expression B – Fusion with large common index in the last contraction: In this expression, the input $B2$ in the second contraction is large, so tiling is done on the dimensions $h5$, $h4$, $h3$, and $h2$. Though it introduced replicated accumulation for I , the overhead is still much smaller than the overhead of communication for C and $B2$. Fusion turns out to be beneficial as it removes communication for C . The performance with and without fusion at the two levels is shown in the second clusters from the left in Figures 3 and 4. We have experimented with the size of each dimension being 40. For the sum of computation and local memory transfer times, fusion resulted in a 45% reduction. At the disk level, the speedup by fusion is about 3.7, because the large amount of time spent in writing C is avoided.

Expression C – Fusion for large inputs in the first contraction: As stated in Section 5.1, when the two input matrices in the first contraction are large enough to exceed the memory limit at a certain level of memory, fusion is still beneficial because the intermediate result is still the dominant part. Expression C represents this case, under the assumption that the matrix C is very large and does not fit in the local memory. Therefore, tiling has to be performed on the *non-common index* of the input matrices A and B .

We experimented with the dimension size being 16. From the third bar cluster in Figure 3, we can see that fusion saved 75% of the sum of computation and the local memory transfer times. Again, the benefit of fusion at the disk level is more significant. By fusion, the total disk I/O has a speedup of about 2.5 by removing matrix C , the dominating factor in the non-fused version.

Expression D – Fusion for a list of five contractions: The expression D was particularly chosen to test the fusion algorithm we had presented for a *fusion chain*. Every contraction here has one small input ($B1$, $B2$, $B3$, $B4$, and $B5$), which could reside in the local memory. Fusion of all five contractions requires memory size N^5 , where N is the size of one dimension. Therefore, in the case where the size of the local memory is 1 GB, we cannot fuse all the five contractions at global memory level when tile size is 40 or more. According to our fusion algorithm, in the fused version at the global memory level, if the tile size is smaller than 40, we fuse all the five contractions. However, when tile size is larger, we break the fusion chain and fuse the two shorter chains separately.

The performance for this expression with the dimension size 40 is shown as the fourth bar cluster in Figures 3 and 4. Moreover, as we have analyzed

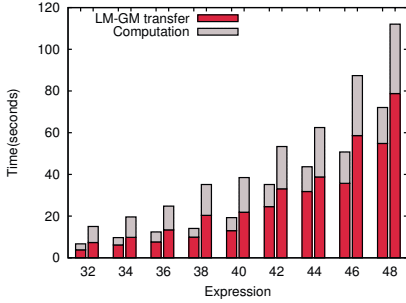


Fig. 5. Execution time with (left) and without (right) fusion at global memory level for expression D - different dimension sizes

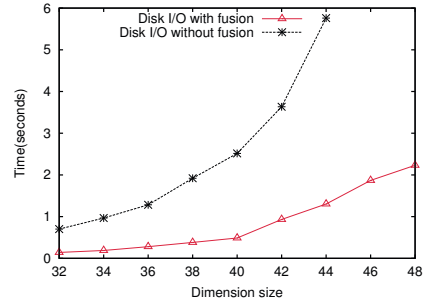


Fig. 6. Disk I/O time with and without fusion at disk level for expression D - different dimension sizes

above, this expression exhibits interesting behavior as problem size is changed. Therefore, we analyzed the performance with varying dimension sizes, and the results are shown in Figures 5 and 6. For dimension size smaller than 40, the speedup of fusion is about 2~2.5, and drops to 1.4~1.7 when $N \geq 40$. At the disk level, fusion of five contractions is always feasible with problem sizes we have experimented with, resulting in a speedup of 4~5 due to fusion.

Expression E – A case for which fusion does not have significant benefits: Our fusion condition predicts that fusion does not have significant benefit under certain conditions. The right most bar cluster in Figure 3 shows the performance with and without fusion at global memory level for the Expression E. We used a dimension size of 256. The execution time here is dominated by the computation time. Speedup by fusion is only about 1.02~1.09, which is consistent with our prediction. This shows that our models are capable of correctly predicting when fusion is not beneficial, or result in small improvements.

6.2 Scalability Study

We studied the benefit of our optimizations as the number of nodes in the cluster increases. We only present detailed results from the Expression A – the results from other cases are similar. Figures 7 and 8 show the performance on a relatively large problem size ($N = 48$), with different numbers of nodes. For the sum of computation and local memory transfer times, the speedup is about 1.8~2.3. At disk level, the speedup due to fusion ranges between 2.3 and 2.5. It can be seen that the computation time decreases with increasing number of nodes, i.e., the computation is being parallelized effectively. However, the communication time becomes relatively large, as data may be retrieved from different parts of the global memory. Thus, fusion becomes increasingly important at larger node counts. At the disk level, the I/O time is relatively independent of the number of nodes, resulting in relatively stable performance improvement from fusion.

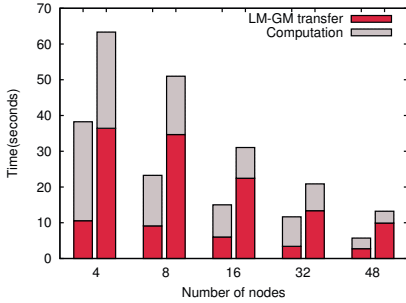


Fig. 7. Execution time with (left) and without (right) fusion at global memory for expression A on different numbers of nodes

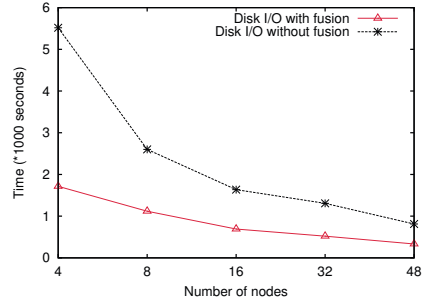


Fig. 8. Disk I/O time with and without fusion for expression A on different numbers of nodes

7 Related Work

The previous work on optimizing tensor contractions focused on efficient exploration of the search space of memory hierarchy transformations [33,14]. Sahoo *et al.* pruned the search space while combining permutation, loop fusion, tiling, and placement of disk I/O operations [33]. Gao *et al.* extended this work further to minimize disk I/O costs [14]. This work only considered data movement across one level of memory hierarchy. In comparison, our work is the first in offering solutions with data movement across more than two levels. We have also presented more efficient algorithms for determining candidates for loop fusion.

Outside the specific context of tensor contractions, there is a very large volume of work on optimizing for memory hierarchies [23,24]. Most of this work has focused on improving the utilization of a processor cache, which is not explicitly programmer controlled. One key difference between the work in this area and our work is in the metrics used for transformations. Cache transformations are based on metrics like reuse distance [12] or stack distance [7]. In comparison, we target programmer controlled memory hierarchy levels, and applications where we primarily see capacity misses. Also, because we are focusing on a limited class of applications, it is possible to develop a more accurate cost model. Darte has studied the general complexity of loop fusion [10] and has presented an approach for optimized memory allocation [11].

Significant work has been done on optimizing out-of-core programs. Particularly, Brown *et al.* have used compiler analysis to optimize prefetching of data in out-of-core programs [6]. Kandemir *et al.* have considered several compiler optimizations for out-of-core parallel programs [18] and other I/O intensive applications [17], building on earlier work at Syracuse/Northwestern and Rice [5,36]. The key difference in our work is the consideration of data movement across more than two levels of programmer controlled memory hierarchy.

Prior to interest in accelerating computations using GPUs, several researchers have studied deeper memory hierarchies. Mitchell *et al.* considered *multi-level*

caches, and further included considerations for instruction-level parallelism and TLB in their work [25]. For multi-level caches, they observed that single-level tiling is sufficient. We have considered a different architecture, and our conclusions are different. Rivera and Tseng examined loop transformations for multi-level caches, and finding that all performance gains can be achieved by simply focusing on L1 cache [32]. Clearly, as we have considered architectures with a different type of multi-level memory hierarchy, our conclusions are different. More recent work has considered multi-level tiling, but specific to stencil computations [30]. We perform tiling for a different class of applications. Qaseem *et al.* have used complex modeling and a direct search to find transformation parameters [28]. Ren *et al.* presented an auto-tuning framework for software-managed memory hierarchies [29]. We, in comparison, employ an analytical approach.

With recent interest in GPUs, there has been some work on optimizing data movements from main memory to device memory [35,34,22]. However, we are not aware of any extensive code restructuring framework to optimize these costs. Within the context of GPUs, another research direction involves optimization of shared memory use in GPUs, which are also a form of application-controlled cache. In this area, Baskaran *et al.* have provided an approach for automatically arranging shared memory on NVIDIA GPU by using the polyhedral model for affine loops [4]. Moazeni *et al.* have adapted approaches for register allocation, particularly those based on graph coloring, to manage shared memory on GPU [26]. A very similar problem is optimization for *scratch-pad* memory in embedded systems. Diouf *et al.* used integer linear programming to solve the problem [13]. Li *et al.* proposed interval coloring approach for arranging scratch-pad memory, utilizing the observation that live ranges of two arrays should not interfere or one should not contain the other [20]. Udayakumaran *et al.* address the scratch-pad memory allocation in a dynamic way, using a greedy heuristic as the cost model [37]. None of these efforts have considered multi-level data movement. Most of these efforts are also not suitable for applications like tensor contractions, where only a fraction of a single array can fit in the memory.

8 Conclusions

Emerging high-end architectures are bringing new challenges for compilers and code generation systems. Particularly, deeper, explicitly controlled memory hierarchies are becoming common and need to be optimized for. This paper has considered tiling, loop fusion, and placement of data movement operations for tensor contractions, which is an important class of scientific applications, for systems with more than two levels in the memory hierarchy. We have developed practical techniques focusing on the dominant cost factors. Experimental evaluation has shown that loop fusion for multiple levels improves performance and our methods can correctly predict cases where loop fusion is not advantageous. In addition to being effective, we believe these low complexity approaches serve to reduce the search space and provide effective starting points for empirical search in the “neighborhood” of the candidate loop structures chosen.

References

1. Ahmed, N., Mateev, N., Pingali, K.: Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. *International Journal of Parallel Programming* 29(5), 493–544 (2001)
2. Aprà, E., Rendell, A.P., Harrison, R.J., Tipparaju, V., deJong, W.A., Xantheas, S.S.: Liquid water: obtaining the right answer for the right reasons. In: *SC (2009)*
3. Bartlett, R.J., Musiał, M.: Coupled-cluster Theory in Quantum Chemistry. *Rev. Mod. Phys.* 79(1), 291–352 (2007)
4. Baskaran, M.M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In: *PPoPP*, pp. 1–10 (2008)
5. Bordawekar, R., Choudhary, A., Kennedy, K., Koelbel, C., Paleczny, M.: A model and compilation strategy for out-of-core data parallel programs. In: *PPoPP*, pp. 1–10 (July 1995)
6. Brown, A.D., Mowry, T.C., Krieger, O.: Compiler-based i/o prefetching for out-of-core applications. *ACM Trans. Comput. Syst.* 19(2), 111–170 (2001)
7. Cascaval, C., Padua, D.A.: Estimating cache misses and locality using stack distances. In: *ICS*, pp. 150–159 (2003)
8. Coleman, S., McKinley, K.S.: Tile size selection using cache organization and data layout. In: *PLDI*, pp. 279–290 (1995)
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: *Introduction to algorithms*. MIT Press, Cambridge (2001)
10. Darte, A.: On the complexity of loop fusion. *Parallel Computing* 26(9), 1175–1193 (2000)
11. Darte, A., Schreiber, R., Villard, G.: Lattice-based memory allocation. *IEEE Trans. Computers* 54(10), 1242–1257 (2005)
12. Ding, C., Zhong, Y.: Predicting whole-program locality through reuse distance analysis. In: *PLDI*, pp. 245–257. ACM, New York (2003)
13. Diouf, B., Ozturk, O., Cohen, A.: Optimizing local memory allocation and assignment through a decoupled approach. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) *LCPC 2009*. LNCS, vol. 5898, pp. 408–415. Springer, Heidelberg (2010)
14. Gao, X., Krishnamoorthy, S., Sahoo, S.K., Lam, C.-C., Baumgartner, G., Ramanujam, J., Sadayappan, P.: Efficient search-space pruning for integrated fusion and tiling transformations. *Concurrency and Computation: Practice and Experience* 19(18), 2425–2443 (2007)
15. Hirata, S.: Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *Journal of Physical Chemistry A* 107(46), 9887–9897 (2003)
16. Hsu, C.-h., Kremer, U.: A quantitative analysis of tile size selection algorithms. *J. Supercomput.* 27(3), 279–294 (2004)
17. Kandemir, M., Choudhary, A., Choudhary, A.: Compiler optimizations for i/o intensive computations. In: *Proceedings of International Conference on Parallel Processing (September 1999)*
18. Kandemir, M., Choudhary, A., Ramanujam, J., Bordawekar, R.: Compilation techniques for out-of-core parallel computations. *Parallel Computing* 24(3-4), 597–628 (1998)
19. Kelly, W., Pugh, W.: Finding legal reordering transformations using mappings. In: Pingali, K.K., Gelernter, D., Padua, D.A., Banerjee, U., Nicolau, A. (eds.) *LCPC 1994*. LNCS, vol. 892. Springer, Heidelberg (1995)

20. Li, L., Nguyen, Q.H., Xue, J.: Scratchpad allocation for data aggregates in super-perfect graphs. In: LCTES 2007: Proceedings of Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 207–216 (2007)
21. Lim, A.W., Cheong, G.I., Lam, M.S.: An affine partitioning algorithm to maximize parallelism and minimize communication. In: International Conference on Supercomputing, pp. 228–237 (1999)
22. Ma, W., Agrawal, G.: A Translation System for Enabling Data Mining Applications on GPUs. In: Proceedings of International Conference on Supercomputing (ICS) (June 2009)
23. McKinley, K.S., Carr, S., Tseng, C.-W.: Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems* 18(4), 424–453 (1996)
24. McKinley, K.S., Temam, O.: Quantifying loop nest locality using spec'95 and the perfect benchmarks. *ACM Trans. Comput. Syst.* 17(4), 288–336 (1999)
25. Mitchell, N., Högstedt, K., Carter, L., Ferrante, J.: Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming* 26(6), 641–670 (1998)
26. Moazeni, M., Bui, A., Sarrafzadeh, M.: A Memory Optimization Technique for Software-Managed Scratchpad Memory in GPUs (July 2009), <http://www.sasp-conference.org/index.html>
27. Nieplocha, J., Harrison, R.J., Littlefield, R.J.: Global arrays: A nonuniform memory access programming model for high-performance computers. *Journal of Supercomputing* 10(2), 169–189 (1996)
28. Qasem, A., Kennedy, K., Mellor-Crummey, J.M.: Automatic tuning of whole applications using direct search and a performance-based transformation system. *The Journal of Supercomputing* 36(2), 183–196 (2006)
29. Ren, M., Park, J.Y., Houston, M., Aiken, A., Dally, W.J.: A tuning framework for software-managed memory hierarchies. In: PACT, pp. 280–291 (2008)
30. Renganarayana, L., Harthikote-matha, M., Dewri, R., Rajopadhye, S.: Towards optimal multi-level tiling for stencil computations. In: IPDPS (2007)
31. Renganarayana, L., Rajopadhye, S.: Positivity, posynomials and tile size selection. In: SC, pp. 1–12 (2008)
32. Rivera, G., wen Tseng, C.: Locality Optimizations for Multi-level Caches. In: Proceedings of the SC 1999 (November 1999)
33. Sahoo, S.K., Krishnamoorthy, S., Panuganti, R., Sadayappan, P.: Integrated loop optimizations for data locality enhancement of tensor contraction expressions. In: SC, p. 13. IEEE Computer Society, Los Alamitos (2005)
34. Sundaram, N., Raghunathan, A., Chakradhar, S.: A framework for efficient and scalable execution of domain-specific templates on GPUs. In: IPDPS (2009)
35. Tarditi, D., Puri, S., Oglesby, J.: Accelerator: using data parallelism to program gpus for general-purpose uses. In: ASPLOS, pp. 325–335 (2006)
36. Thakur, R., Bordawekar, R., Choudhary, A.: Compilation of out-of-core data parallel programs for distributed memory machines. In: Second Annual Workshop on Input/Output in Parallel Computer Systems (IPPS), pp. 54–72 (April 1994)
37. Udayakumar, S., Dominguez, A., Barua, R.: Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.* 5(2), 472–511 (2006)
38. Yi, Q., Kennedy, K., Adve, V.: Transforming complex loop nests for locality. *J. Supercomput.* 27(3), 219–264 (2004)

A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL

Dominik Grewe and Michael F.P. O’Boyle

School of Informatics, The University of Edinburgh, UK
dominik.grewe@ed.ac.uk, mob@inf.ed.ac.uk

Abstract. Heterogeneous multi-core platforms are increasingly prevalent due to their perceived superior performance over homogeneous systems. The best performance, however, can only be achieved if tasks are accurately mapped to the right processors. OpenCL programs can be partitioned to take advantage of all the available processors in a system. However, finding the best partitioning for any heterogeneous system is difficult and depends on the hardware and software implementation.

We propose a portable partitioning scheme for OpenCL programs on heterogeneous CPU-GPU systems. We develop a purely static approach based on predictive modelling and program features. When evaluated over a suite of 47 benchmarks, our model achieves a speedup of 1.57 over a state-of-the-art dynamic run-time approach, a speedup of 3.02 over a purely multi-core approach and 1.55 over the performance achieved by using just the GPU.

Keywords: Heterogeneous programming, task partitioning, OpenCL, parallel programming, static code analysis.

1 Introduction

Heterogeneous computing systems promise to deliver high performance at relatively low energy costs [15,18]. By having processing units with different characteristics, computation can be mapped to specialised devices that perform a specific type of task more efficiently than other devices. In embedded systems this has been the case for many years with specialised DSP units for instance [15]. This trend has spread to the desktop, where the high-end relies on accelerator devices for increased performance. With the rise of GPGPU (general-purpose computing on GPUs), heterogeneous computing has become increasingly prevalent and attractive for more mainstream programming [20,24,27].

The most widely adapted framework for heterogeneous computing is OpenCL [16], an open standard for parallel programming of heterogeneous systems supported by many hardware vendors such as AMD, NVIDIA, Intel and IBM. OpenCL can be used for programming multiple different devices, e.g. CPUs and GPUs, from within a single framework. It is, however, fairly low-level, requiring the programmer to tune a program for specific platforms in order to get the optimal performance. This, in particular, includes the mapping of tasks to devices,

i.e. what part of the computation is performed on which device. As processors in a heterogeneous system are often based on entirely different architectures, the right mapping can be crucial in achieving good performance as shown in section 3. Heterogeneous platforms continue to evolve with increased numbers of cores and more powerful accelerators, the consequence being that the best partitioning will also change. Furthermore as OpenCL is a relatively new API, it is likely that each new implementation release will change the relative performance between different types of cores again affecting the best partitioning. Finally, it is likely that OpenCL will increasingly be used as a target language for high-level compilers (e.g. CAPS HMPP [11] or PGI [28]), making automatic mapping of tasks desirable. Ideally, we would like an approach that can adapt to architecture and implementation evolution without requiring repeated compiler-expert intervention.

GPUs are specifically suited for data-parallelism, because they comprise of groups of processing cores that work in a SIMD manner. Data-parallel tasks can often be easily split into smaller sub-tasks and distributed across multiple devices. Finding the best partitioning to achieve the best performance on a particular systems is non-trivial, however. Several efforts have been made to automate this process: Qilin [20] relies on extensive off-line profiling to create a performance model for each task on each device. This information is used to calculate a good partitioning across the devices. However, the initial profiling phase can be prohibitive in many situations. Ravi et al. [24] develop a purely dynamic approach that divides a task into chunks that are distributed across processors in a task-farm manner. While this eliminates profiling, it incurs communication and other overheads.

The approach to partitioning data-parallel OpenCL tasks described in this paper is a *purely static* approach. There is no profiling of the target program and the run-time overhead of dynamic schemes is avoided. In our method static analysis is used to extract code features from OpenCL programs. Given this information, our system determines the best partitioning across the processors in a system and divides the task into as many chunks as there are processors with each processor receiving the appropriate chunk. Deriving the optimal partitioning from a program's features is a difficult problem and depends heavily on the characteristics of the system. We therefore rely on machine-learning techniques to *automatically* build a model that maps code features to partitions. Because the process is entirely automatic, it is easily *portable* across different systems and implementations. When either change, we simply rerun the learning procedure without human intervention in building the model. We focus on CPU-GPU systems as this is arguably the most common form of heterogeneous systems in the desktop and high-performance computing domain.

The contributions of this paper are as follows:

- We develop a machine-learning based compiler model that accurately predicts the best partitioning of a task given only static code features.
- We show that our approach works well across a number of applications and outperforms existing approaches.

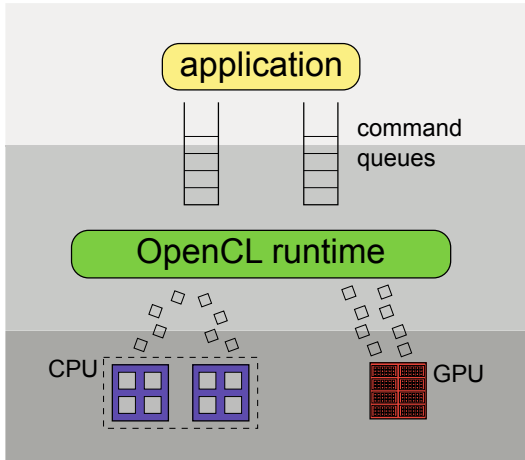


Fig. 1. OpenCL in a heterogeneous environment. The user schedules task to command queues of which there is one for each device. The OpenCL run-time then breaks data-parallel task into chunks and sends them to the processing elements in the device.

The rest of this paper is structured as follows. Section 2 gives a brief introduction to OpenCL mapping and is followed in section 3 by a short example illustrating the performance impact of mapping decisions. Section 4 describes how we automatically build a partitioning model which is then evaluated in sections 5 and 6. Section 7 describes related work and is followed by some concluding remarks.

2 The OpenCL Programming Framework

Recent advances in the programmability of graphics cards have sparked a huge interest in what is now called general-purpose computing on graphics processing units or GPGPU. Several proprietary solutions like Brook [7] or NVIDIA CUDA [22] have been proposed, out of which the latter has arguably the greatest following. OpenCL is an attempt to develop an *open alternative* to these frameworks and is now being supported by most major hardware manufacturers. Furthermore, OpenCL not only targets GPUs, but entire heterogeneous systems including GPUs, CPUs and the Cell architecture.

Due to its success, OpenCL’s programming model is similar to CUDA, focusing on data-parallelism. Data-parallel tasks are suitable for GPUs, in which groups of processing cores work in a SIMD fashion. In OpenCL, a data-parallel task is expressed as a *kernel* that describes the computation of a single *work-item*¹. During program execution, a user-specified number of work-items is launched to execute in parallel. These work-items are organized in a multi-dimensional grid and subsets of work-items are grouped together to form *work-groups*, which allow work-items to interact. Each work-item can query its position in the grid by calling certain built-in functions from within the kernel code.

¹ A work-item is equivalent to a thread in CUDA.

Despite OpenCL’s focus on data-parallelism, task-parallelism is also supported in the framework to allow execution on multiple devices, e.g. multiple GPUs or CPUs and GPUs.² For each device, the user can create a *command queue* to which (data-parallel) tasks can be submitted (see figure 1). This not only allows the user to execute different tasks in parallel, but also enables decomposition of data-parallel tasks into sub-tasks that are distributed across multiple devices. Because OpenCL supports a variety of processing devices, all this can be achieved with just a single implementation of each task. Using CUDA a separate implementation for other devices, such as CPUs, would be needed.

OpenCL’s memory model reflects the memory hierarchy on graphics cards. There is a global memory that is accessible by all work-items. There is also a small local memory for each work-group that can only be accessed by work-items from that particular work-group. Additionally, there is a constant memory which is read-only and can be used to store look-up tables, etc. This memory model is general enough to be mapped to many devices. Some processing devices, for example GPUs, have a global memory that is separate from the computer’s main memory. In this case, any data needs to be copied to the device and back to main memory before and after task execution, and can be a considerable overhead.

3 Motivation

Determining the right mapping for a task is crucial to achieve good performance on heterogeneous architectures. This section illustrates this point by examining the performance of three OpenCL programs, each of which needs a different partitioning to achieve its best performance.

Figure 2 shows the speedup of three OpenCL programs with different mapping over single-core execution on a CPU. The specification of our system is provided in section 5.1. The x-axis shows how much of the program’s workload is executed on each device, i.e. the leftmost bar shows the speedup of GPU-only execution, one bar to the right shows the execution with 90% of work on the GPU and 10% on the CPUs and so on.

For the *coulombic potential* program (figure 2a), a GPU-only execution achieves by far the best performance. Scheduling an amount of work as small as 10% to the CPUs leads to a slow-down of more than 5 times and this value increases if more work is mapped to the CPUs. For these types of programs it is absolutely vital to know ahead of time what the optimal mapping is, because a small mistake is going to be very costly.

The *matrix-vector multiplication* program exhibits an entirely different behaviour (see figure 2b). The highest speedup is observed when 90% of the work is scheduled to the CPUs and only 10% to the GPU. The amount of computation per data item is fairly small and therefore the overhead of transferring data between main memory and the GPU’s memory is not worthwhile for large parts of the computation. The *convolution* program in figure 2c shows yet another

² In OpenCL all CPU cores (even across multiple chips) are viewed as a single device.

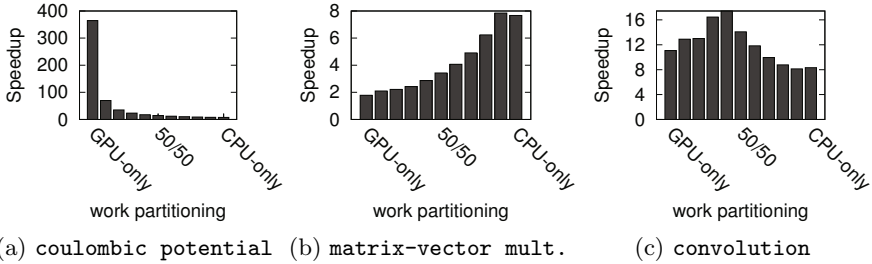


Fig. 2. The speedup over single-core performance of three OpenCL programs with different partitions. The significant variations demonstrate the need for program-specific mappings.

different behaviour: A roughly even partitioning of work between CPUs and GPU leads to the best speedup. Unlike the other cases, neither a GPU-only nor a CPU-only execution would achieve good performance.

As these programs have shown, a partitioning scheme that takes program characteristics into account is necessary to achieve good performance on heterogeneous systems. Different programs need different mappings and for some of them making a small mistake means that large potential speedups are missed. As OpenCL is a fairly new framework, compilers are likely to improve in the near future. The CPU implementation, in particular, seems to have much room for improvement.

The next section describes our static partitioning approach based on static program code structure and machine-learning. By using machine-learning methods, our approach is *portable* across systems as well as implementations; a highly desirable property as program performance is likely to change across heterogeneous architectures and as OpenCL tools mature.

4 Partitioning Data-Parallel Tasks

Our approach uses machine-learning to predict the optimal partitioning for an OpenCL program solely based on compiler analysis of the program structure. The static analysis characterises a program as a fixed vector of real values, commonly known as *features*. We wish to learn a function f that maps a vector of program code features \mathbf{c} to the optimal partitioning of this program, i.e. $f(\mathbf{c}) = p$ where p is as near as possible to the optimal partitioning.

In order to map a task to the hardware without executing it, we need to analyse the code and extract code features. This section describes the static analysis framework used to extract code features at compile time. It also describes how a machine-learning based model is built and then used to predict the optimal partitioning for any OpenCL program. Instead of relying on a single prediction model, we use hierarchical classification [5] where a hierarchy of models is evaluated to find the answer. As the models used are instances of support vector machines (SVMs) [8], we will also provide a brief introduction to SVMs.

4.1 Static Code Feature Extraction

Our partitioning method is entirely based on static code features eliminating the need for expensive off-line profiling [20] and also avoiding the pitfalls of dynamic techniques [24]. However, the features need to carry enough information to characterize the behaviour of OpenCL programs. In the following paragraphs we explain the feature extraction framework and describe the program code features used in the partitioning scheme.

The compiler analysis is implemented in Clang [1], a C-language front-end for LLVM. The OpenCL program is read in by Clang which builds an abstract syntax tree. The analysis is based on a traversal of this tree, extracting code features such as the number of floating point instructions or the number of memory accesses in the program. Because many programs contain loops, we perform a value analysis to determine loop bounds (if possible). The value analysis is also used to analyze memory access patterns, which have a significant impact on performance on GPUs [26].

Memory accesses are called *coalesced* if adjacent work-items access adjacent memory locations. In this case multiple memory transfers can be coalesced into a single access increasing the overall memory bandwidth. This needs to be taken into account when mapping programs as it has a considerable impact on performance.

The full list of static code features is shown in table 1. As indicated in the table, features describing absolute values are normalized. By multiplying the value by the number of work-items we compute the *total* number of operations for this program execution. Since a machine-learning model will not be able to relate two similar programs that have been executed with different input sizes, the total number of operations is divided by the data transfer size, to compute the number of operations *per data item*. In other words the normalized features are computed as

$$\text{operations in program code} \times \frac{\text{number of work-items}}{\text{data transfer size}}$$

Before the features are passed to our model, we apply principal component analysis (PCA) [5] to reduce the dimensionality of the feature space and normalize the data (see section 4.2 for details).

Our features describe both the computation and the memory operations of the kernel. First, it is important to describe the type and amount of computations (features 1-5). Some math operations, such as sine and cosine, can be mapped to special function units on some GPUs but may need to be emulated on CPUs, for example. Barriers (feature 6) may also cause different costs on different architectures.

Second, memory operations (features 7-10) are important to consider. Depending on the architecture and type of memory the cost of memory accesses may vary. Accessing local memory on GPUs, for example, is cheap because it is mapped to small on-chip memory. On CPUs, however, local and global memory both get mapped to the same memory space.

Table 1. List of static code features used to characterize OpenCL programs and the corresponding values of the three example programs

	Static Code Feature	cp	mvm	conv
1	int operations (norm.)	31.6	0.6	28.8
2	int4 operations (norm.)	0	0	0
3	float operations (norm.)	1593.5	0.5	4.25
4	float4 operations (norm.)	0	0	0
5	intrinsic math operations (norm.)	249.9	0	0
6	barriers (norm.)	0	0.012	0.031
7	memory accesses (norm.)	0.125	0.5	2.6
8	percentage of local memory accesses	0	0.04	0.88
9	percentage of coalesced memory accesses	1	0.996	0
10	compute-memory ratio	15004	2.1	105.9
11	data transfer size	134249726	67141632	134217796
12	computation per data transfer	1875	1.6	35.7
13	number of work-items	2097152	4096	4194304

GPUs have a physically separate memory space and any data used during program execution needs to be copied to the GPU. The cost of data transfers between the memories is thus important. Features 11 and 12 capture the amount of memory to be transferred and how it compares to the amount of computation performed on the data. Lastly, feature 13 captures the overall size of the problem.

Examples. Table 1 shows the feature vectors for the example benchmarks introduced in section 3. The “computation to data transfer” ratio (feature 12), for example, is 1875 for the `coulombic potential` program, which is significantly higher than the value for `convolution` (35.7) and `matrix-vector multiplication` (1.6). The number of compute operations (features 1-5) and the ratio between compute- and memory-operations (feature 10) is also higher for `coulombic potential` compared to the others.

These differences in input features reflect the different behaviours shown in figure 2. The optimal performance for the `coulombic potential` benchmark is achieved with GPU-only execution, because of the large number of compute-operations and the relatively small data transfer overhead. For the `matrix-vector multiplication`, on the other hand, very few operations are performed for each data item and the data transfer costs undo any potential speedups the GPU may offer. The feature values of the `convolution` benchmark are in-between the values of the other two programs. This explains why a more balanced work distribution is beneficial.

4.2 Building the Predictor

Building a machine-learning based model involves the collection of training data which is used to fit the model to the problem at hand. In our case, the training data consists of static code features of other OpenCL programs and the optimal

partitioning of the corresponding program. This enables us to create a model that maps program code features to the program’s optimal partitioning. Rather than relying on an individual predictor, we combine several models to form a hierarchical classification model [5].

Collecting Training Data. The training data for our model is divided into static code features (as described in section 4.1) and the optimal partitioning for the corresponding OpenCL program. The former will be the input for our model, whereas the latter is the output (or target) of our model.

Each program is run with varying partitionings, namely all work on the CPU, 90% of work on the CPU and the remaining 10% on the GPU, 80% on the CPU and 20% on the GPU and so on. The partitioning with the shortest run-time is selected as an estimate of the optimal work partitioning for the program.

Two-Level Predictor. As was shown in section 3 OpenCL programs can be loosely divided into three categories, namely programs that achieve the best performance when

- (1) executed on GPU only
- (2) executed on CPUs only
- (3) partitioned and distributed over GPU and CPUs

Getting the partitioning right is especially vital for programs in category 1. Not mapping all the work on the GPU leads to significant slowdowns (see Fig. 2a). Similarly (even though less drastic) for programs in category 2: If it is not worth copying the data back and forth to the GPU, one needs to make sure that the work is mapped to the CPU and any data transfer overhead is avoided.

We therefore develop a prediction mechanism utilizing a *hierarchy* of predictors. This approach is known as hierarchical classification [5]. In the first level, programs from categories 1 and 2 are filtered out and mapped to the GPU or the CPUs, respectively. The remaining programs are mapped according to a third predictor in level 2 (see Fig. 3). The kernel features are reduced to two and eleven principal components using PCA for the first- and second-level predictors, respectively.

Formally, we are mapping input features to one out of 11 classes, where class 0 represents GPU-only execution and class 10 CPU-only execution. Let *gpu* and *cpu* be the first-level predictors and *mix* the second-level predictor. The hierarchical model can be described as

$$prediction(x) = \begin{cases} 0 & \text{if } gpu(x) \text{ and } \neg cpu(x) \\ 10 & \text{if } cpu(x) \text{ and } \neg gpu(x) \\ mix(x) & \text{otherwise} \end{cases}$$

Level 1 predictors. Focusing only on the extremes of the partitioning spectrum, the models in the first stage of the prediction are simple, but highly accurate

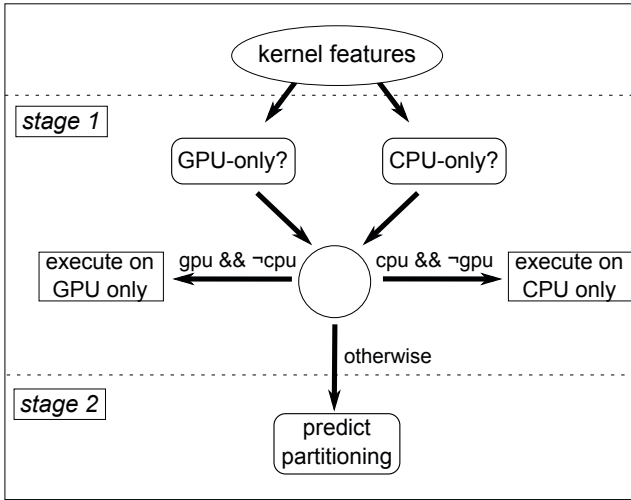


Fig. 3. Overview of our prediction approach. Programs that should be entirely mapped to the GPU or to the CPUs are filtered out in the first level, while the second level handles programs that cannot be classified in level 1.

(see section 6.2). One model predicts whether or not a program should be executed on the GPU only (category 1), while the other determines if a task is entirely mapped to CPUs (category 2). These “one-vs-all” classifiers [25] are implemented as binary classifiers based on a support vector machine (SVM) with a linear kernel (see section 4.4).

Level 2 predictor. If the first level of predictors does not lead to a conclusion, the program is passed on to another predictor. This one is more complex, because it needs to map a program to one out of the 11 classes determined during training. Again, we use an SVM-based model, but this time a radial basis function [5] kernel is deployed to account for the increased complexity of this problem.

Whereas for the stage-1 models we use all of the available training data, we only use data from category 3 programs to train our model in the second level. This allows for the predictor to focus on programs whose optimal partitioning is likely to be neither CPU- nor GPU-only.

4.3 Deployment

At compile time, the program code is analyzed and code features are extracted. Because our model’s input features incorporate the task’s input size, the prediction cannot be made just yet. However, at run-time the input size is known and together with the previously extracted code features is passed to the model. The model’s output is the optimal partitioning for this program and input size which is used to partition the program between multiple devices. In OpenCL this can

be easily done by setting a few variables. Although the prediction is done at run-time, the overhead is negligible as it only takes in the order of microseconds to evaluate our models; the cost of which is included in our later results.

Examples. Passing the features for the `coulombic potential` program as shown in figure 2a into our first-level predictors, we get a positive classification from the “GPU-only” predictor and a negative one from the “CPU-only” model. We therefore immediately map all of the computation to the GPU without evaluating the second level predictor. This leads to the optimal performance for this program.

For the `matrix-vector multiplication` program, it is the other way around, i.e. we map the computation to the CPU only. Looking at figure 2b shows that while this is not the optimal partitioning we still achieve 98% of the optimum.

With the `convolution` program, both first-level predictors say “no” and we move on to the second level predictor. Given the input features this model predicts that we should map 60% of the work to the GPU and the remaining 40% to the CPUs. According to figure 2c this leads to the optimal performance.

4.4 Support Vector Machines

Support Vector Machines (SVMs) [5] belong to the class of supervised learning methods and can be used for both classification and regression. The idea behind SVMs is to map the input feature space into a higher-dimensional space and then find hyperplanes that separate the training points from different classes. In the original feature space a linear separation may not be possible, but in a higher-dimensional space it is often easier to find such a separation. A new input feature vector is projected to the higher-dimensional space and a prediction is made depending on which side of the hyperplanes the projection is located. The projection into a higher-dimensional space is done using *kernel functions*. These include linear kernels or radial basis function kernels. Depending on the nature of the problem, some kernels perform better than others. A more detailed description of SVMs can be found in [5].

5 Methodology

5.1 Experimental Setup

All experiments were carried out on a heterogeneous computer comprising two quad-core CPUs with Intel HyperThreading and an ATI Radeon HD 5970 GPU. Table 2 shows more detailed information on our system as well as the software used. When the GPU was used, one CPU core was dedicated to managing the GPU. This has shown to be beneficial and is in line with observations made by Luk et al. [20]. Each experiment was repeated 20 times and the average execution time was recorded.

Table 2. Experimental Setup

	CPU	GPU
Architecture	2x Intel Xeon E5530	ATI Radeon HD 5970
Core Clock	2.4 GHz	725 MHz
Core Count	8 (16 w/ HyperThreading)	1600
Memory Size	24 GB	1 GB
Compiler	GCC 4.4.1 w/ "-O3"	
OS	Ubuntu 9.10 64-bit	
OpenCL	ATI Stream SDK v2.01	

In total we used 47 different OpenCL programs, collected from various benchmark suites: SHOC [9], Parboil³ [23], NVIDIA CUDA SDK [22] and ATI Stream SDK [2]. By varying the programs’ input sizes, we conducted a total of 220 experiments. We used the standard approach of cross-validation which has the critical property that when evaluating the model on a certain program, *no training data from this program* was used to build the model.

5.2 Evaluation Methodology

We compare our approach to an “oracle”, which provides an estimate of the *upper bound* performance. To find the oracle we tried all 11 partitions on the target program and selected the one with the lowest execution time. It may be possible that a partition that is not a multiple of 10% gives an even better speedup and thus our oracle is only an approximation.

We further evaluate two default strategies: “CPU-only” and “GPU-only” simply map the entire work to the CPUs or to the GPU, respectively. These are two very primitive methods that serve as a *lower bound* in the sense that any partitioning scheme should (on average) beat them to prove itself useful.

The fourth method we compare our approach against is a dynamic mapping scheme similar to what is presented by Ravi et al. [24], where the work of a kernel is broken into a certain number of chunks. Initially, one chunk is sent to the GPU and one to the CPUs. When either of them finishes, a new chunk is requested until the work is complete. We searched for the best number of chunks to divide the work into and found that using 8 chunks leads to the best overall performance on the set of kernels used in this paper, providing the right balance between scheduling overhead and flexibility.

The performance of each mapping technique is evaluated by computing the speedup over single-core execution. To collect the single-core run-times we used the same OpenCL code, but instructed the OpenCL run-time to only use one CPU core. This may underestimate the performance of a sequential version as it

³ The Parboil benchmark suite only contains CUDA programs. We therefore translated the benchmarks from CUDA to OpenCL. The OpenCL source code can be found at <http://homepages.inf.ed.ac.uk/s0898672/openc1/benchmarks.html>

is likely to be faster than the parallel OpenCL code on a single core. However, as this value is used solely as a baseline to compare speedups of the competing techniques, it is appropriate for our purposes.

6 Results

In this section we show the performance of various mapping techniques. We compare our new approach with two static default strategies, namely “CPU-only” and “GPU-only”, as well as with the dynamic scheduling method described in section 5.2. The performance achievable with an optimal partitioning is also presented to compare the approaches to the maximal available speedups. This is followed by an evaluation of the accuracy of the various predictors in our model.

6.1 Speedup Over Single-Core Performance

We compare the run-times achieved with different partitioning schemes to the run-time on a single CPU core. Because of the large number of experiments we divided our programs according to the three categories described in section 4.2. Figure 4 shows programs where a “GPU-only” mapping achieves more than 90% of the optimal performance, whereas figure 5 shows programs where “CPU-only” achieves more than 80% of the optimum. The remaining programs are shown in figure 6. This not only helps understanding the results but also improves readability, as programs from different categories often have huge differences in speedup.

GPU-friendly benchmarks. Figure 4 shows the performance of the various techniques on OpenCL programs of category 1, i.e. programs that achieve the best speedups when executed on the GPU only. Unsurprisingly, the static “GPU-only” policy achieves near-optimal performance (compare to the right-most “oracle” bars). On these benchmarks this leads to an average speedup of 112 over single-core performance. Similarly obvious is that the “CPU-only” method clearly loses on these benchmarks, only achieving an average speedup of 8. The results for the dynamic approach are slightly better: Because the GPU is much faster than the CPUs on these programs, the majority of work will be scheduled to the GPU. However, some of the work is always scheduled to the CPUs which leads to significant slow-downs for most of the benchmarks when compared to the maximum performance. Overall, the dynamic scheduler achieves a speedup of 49. Our prediction approach, on the other hand, classifies almost all programs correctly and therefore achieves almost the same performance as the “oracle” scheduler with 113 times of the single-core performance.

In our model, the majority of programs is filtered out by the first-level “GPU-only” predictor. Only very few are passed through to the next level. For those cases, the second-level predictor makes the right decision to schedule the work to the GPU. More detailed information on the accuracy of our model’s predictors will be shown in section 6.2.

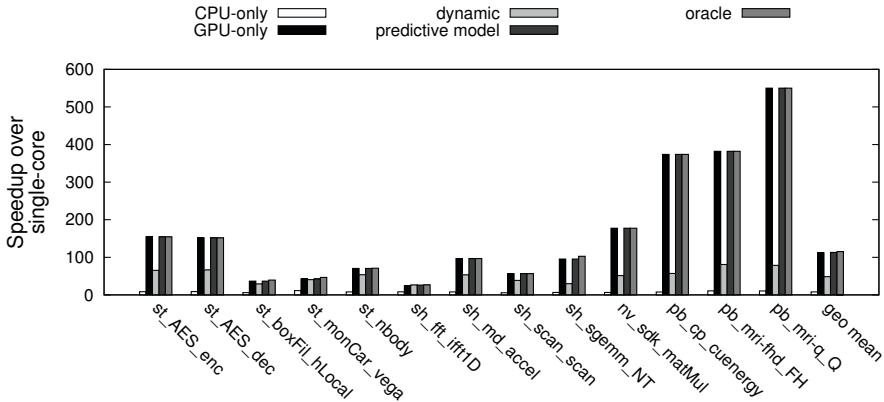


Fig. 4. Performance of those applications where the best speedup is achieved using only the GPU. Our predictive model leads to near-optimal performance compared to the oracle and a speedup of 2.3 over the dynamic scheme.

CPU-friendly benchmarks. The performance of the different partitioning schemes on category 2 programs is shown in figure 5. This time around the static “CPU-only” policy achieves near-optimal performance. The average speedup equates to 6.12, only marginally below the upper bound of 6.36. Unsurprisingly the “GPU-only” method is worst for all programs, most of the time because shipping the data between main memory and GPU memory is not feasible. The average speedup of “GPU-only” is 1.05, i.e. no significant improvement over single-core execution can be observed. Again, the dynamic mapping method only comes second to last. The overhead of having many chunks and sending data to the GPU is simply too big to achieve good performance on these programs and leads to a speedup of only 2.15. Our prediction method comes close to the optimal performance. With an average speedup of 4.81 it is slower than the “CPU-only” policy on the CPU-friendly benchmarks, but significantly better than the dynamic scheme. This again shows our model’s high accuracy for partitioning OpenCL programs. Just like for the GPU-friendly programs, most of the CPU-friendly programs are filtered out in stage 1 of our prediction process. The few other programs are accurately mapped by the second-level predictor. For more detailed information on the predictors’ accuracies see section 6.2.

Remaining benchmarks. Performance results for all the remaining benchmarks are shown in figure 6. To improve readability, we have grouped the programs according to the maximum available speedup, i.e. programs in figure 6a achieve a larger speedup than programs in figure 6b.

Both non-adaptive policies, “CPU-only” and “GPU-only”, do not do very well on most of these benchmarks, because the optimal performance is achieved when the work is distributed across both devices. On average, the “GPU-only” mapping achieves higher speedups (6.26 compared to 4.59), because the “CPU-only” policy misses out on potentially high speedups as shown in figure 6a. The

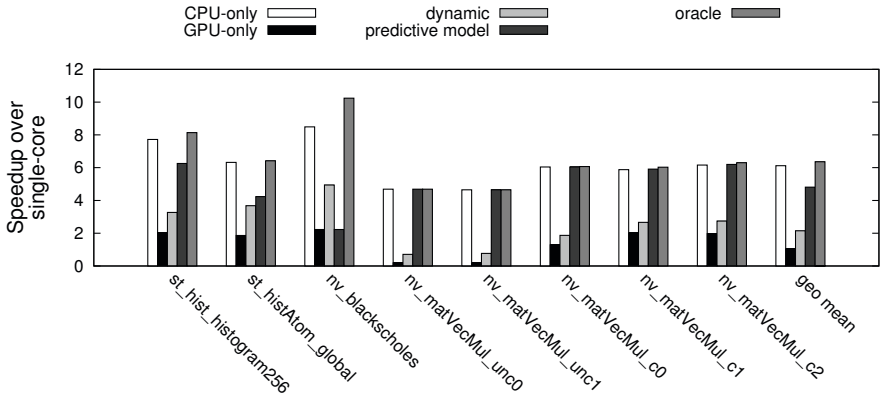


Fig. 5. Performance of those applications where using only the CPU leads to almost optimal speedup. Our predictive model achieves a speedup of 2.2 over the dynamic scheme.

dynamic scheme does reasonably well and achieves near-optimal performance for some benchmarks and an average speedup of 8.00. However, all schemes are outperformed by our prediction approach which achieves a speedup of 9.31 on average. Hence, even though the dynamic scheme shows its potential on these kind of programs, it is still outperformed by our prediction approach due to reduced scheduling overhead and more accurate work distribution.

Summary. As was shown in this section, a fixed partitioning that is agnostic to program characteristics is unsuitable for heterogeneous environments. The “CPU-only” and “GPU-only” methods only apply to a limited set of benchmarks and cannot adapt to different programs. Different programs need different mappings, highlighting the need for adaptive techniques.

For the majority of programs, a partitioning of the work between CPUs and GPU is beneficial. While the dynamic partitioning method described in section 5.2 is designed to handle these cases, it is often unable to achieve good performance due to scheduling overhead and the inability to account for cases where a mixed execution is harmful. Our approach, in contrast, explicitly handles all cases and minimises overhead by making a static decision based on program code features.

Figure 7 shows the geometric mean over all benchmarks for the techniques presented in this paper. The “CPU-only” scheme is by far the worst technique because it does not adapt to programs and misses out on large speedups with GPU-friendly programs. Although the “GPU-only” mapping does not adapt to programs either, it achieves a better overall speedup because it correctly maps the programs that show high performance improvements over single-core execution. With an average speedup of 9.21 it is even marginally better than the dynamic method which achieves only 9.13 times the performance of single-core execution. This again is because the potential of GPU-friendly programs

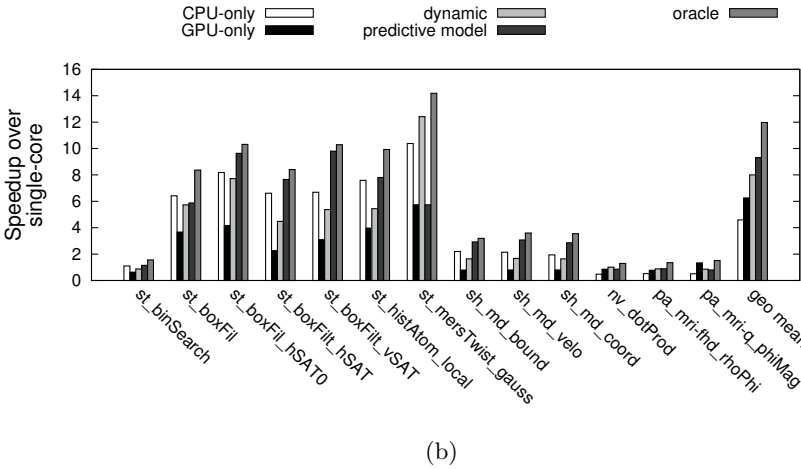
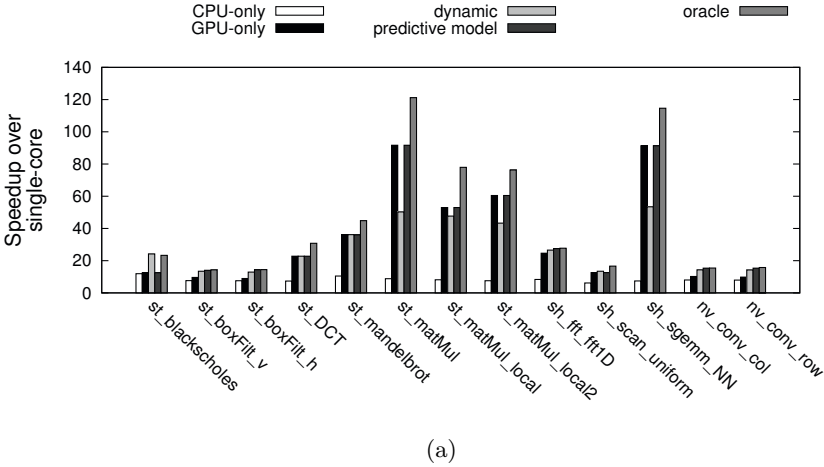


Fig. 6. Performance of those applications where a mix of both CPU- and GPU-execution leads to the best performance. Our predictive model achieves a speedup of 1.2 over the dynamic scheme and clearly outperforms both the CPU-only and GPU-only mapping.

is not realised. Our approach presented in this paper, on the other hand, leads to significantly higher speedups. With an average speedup of 14.30, we achieve more than 80% of the upper bound which equates to a speedup of 1.6 over the dynamic scheduler.

6.2 Prediction Accuracy

Let us take a closer look at the individual predictors in our model. Table 3 shows the number of applications that achieve the best performance with GPU-only

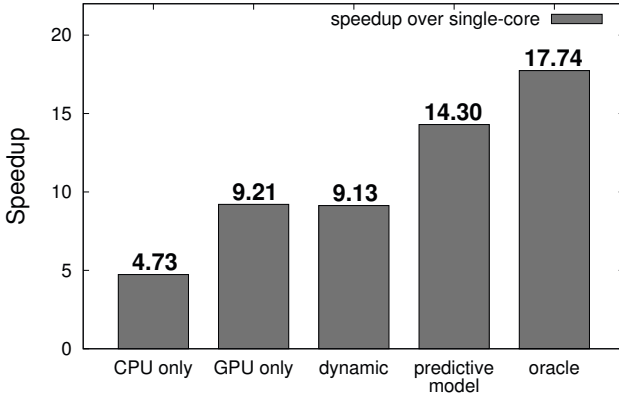


Fig. 7. The average speedup (geometric mean) over all benchmarks. Our predictive modelling approach clearly outperforms all other competing techniques.

and CPU-only execution respectively. The job of our two first-level predictors is to filter out these applications and to pass on the rest. Therefore, the table shows the numbers broken down according to the predictions in the first level of our model.

Out of the 220 program-input pairs we examined, 61 should be entirely mapped to the GPU. 52 of those programs are identified by our first-level predictor and thus mapped to the GPU straightaway. The remaining 9 inputs are misclassified. However, they are passed on to the second-level predictor, which correctly maps them entirely to the GPU. 10 inputs are incorrectly classified as GPU-only and therefore mapped to the GPU although it is not optimal. However, in half of the cases this still leads to more than 90% of the optimal performance. Overall, the GPU-only classifier has an accuracy of 91%.

19 of the 23 program-input pairs that should be entirely mapped to CPUs are correctly classified by our second predictor in the first level of our model. For the 10 misclassified inputs we still achieve an average performance of 78% of the optimum. Overall, the CPU-only classifier achieves an accuracy of 95%.

As expected, the second-level predictor has a lower accuracy than its counterparts in level 1. This is because its solving a much harder problem: instead of making a binary decision, one out of 11 classes needs to be predicted. However,

Table 3. The accuracy of the binary classifiers in the first level our predictor. The “GPU-only” model achieves an accuracy of 91% while the “CPU-only” model even achieves a 95% accuracy.

	GPU	¬ GPU		CPU	¬ CPU
GPU predicted	52	10		19	6
¬ GPU predicted	9	149		4	191

being off by just one or two classes often still leads to good performance, on average a performance of 80% is still achieved. Our level 2 predictor is within these bounds for 65% of all programs.

Looking at the model as a whole, we achieve an accuracy of 52%, i.e. in 52% of all program-input pairs we *exactly* predict the optimal mapping. This leads to an average 85% of the optimal performance, compared to only 58% of the dynamic partitioning method.

7 Related Work

The mapping or scheduling of tasks to heterogeneous systems is an extensively studied subject. In early publications, heterogeneous systems were often based on single-core CPUs running at different speeds. In 1977, for example, Ibarra et al. [13] described some of the first strategies for scheduling independent tasks to heterogeneous architectures. Given a function for each architecture that describes the run-time of a task on this processing unit they propose different heuristics to minimize the overall finishing time. In addition to those and other static techniques [6], several dynamic methods have been proposed [21,29,17].

On heterogeneous *multi-core* architecture consisting of multi-core CPUs and GPUs task mapping becomes more complex, because the devices are based on entirely different architectures. The tasks considered are also often data-parallel which means they can be split to use multiple devices concurrently.

In the Harmony [10] framework programs are represented as a sequence of kernels. Whenever a kernel is available for execution it is dynamically scheduled to a device based on the suitability of the device for this kernel. The suitability is computed with a multivariate regression model that is built at run-time based on previous runs of the kernels. Harmony does not consider partitioning work and thus only schedules entire tasks.

Qilin [20] also relies on a regression model to predict kernel run-times. In contrast to Harmony [10] which builds the model on-line, Qilin uses off-line profiling to create a regression model at compile time. Each kernel version is executed with different inputs and a linear model is fit to the observed run-times. Rather than scheduling individual kernels, a single data-parallel task is split into sub-tasks which are each executed on one device. Qilin requires extensive off-line profiling which may be prohibitive in some situations. Our approach, on the other hand, does not require any profiling and is entirely based on static code features.

Merge [19] is a framework for developing map-reduce applications, i.e. programs comprised of data-parallel maps and reductions, on heterogeneous platforms. At run-time user-provided constraints are used to dynamically map sub-tasks to devices, favoring a more specific implementation over a more general implementation. In contrast to our scheme, they rely on the user to provide information on the suitability of a kernel for the devices.

Ravi et al. [24] also propose a dynamic scheduling technique for map-reduce programs. Tasks are split into chunks that are then distributed across the devices similar to a master-slave model: When a device finishes processing a chunk of work, it requests a new one from a list of remaining work items. While the general idea of this model is straightforward, choosing the right chunk size has a non-negligible effect on performance as Ravi et al. show. However, they leave it for future work to predict the optimal chunk size. While purely dynamic approaches neither require user intervention nor profiling, they do not take any kernel characteristics into account which often leads to poor performance as shown in this paper.

Jiménez et al. [14] consider scheduling in multi-programmed heterogeneous environments. At run-time each program will be executed on all devices and the performance is collected. This information is then used to decide which processor a program will be executed on. A similar idea is proposed by Gregg et al. [12]. Based on the contention of devices and historical performance data they dynamically schedule programs to devices. Both methods do not consider partitioning tasks.

StarPU [4] is a framework for programming on heterogeneous systems. The user provides multiple versions of each task and the run-time schedules them to the devices. Various scheduling techniques are presented, including greedy scheduling and performance-based scheduling. The performance estimations are either provided by the user or based on history information collected by the run-time [3].

Apart from Gregg et al. [12], none of the above approaches use OpenCL. While some use code generation from domain-specific high-level language representations, most of them rely on the user to provide separate kernel version for CPUs and GPUs. We circumvent this problem by using OpenCL which only requires a single kernel code version for both CPUs and GPUs.

8 Conclusion

This paper has developed a new approach for partitioning and mapping OpenCL programs on heterogeneous CPU-GPU systems. Given a data-parallel task, our technique predicts the optimal partitioning based on the task's code features. Our model relies on machine-learning techniques, which makes it easily portable across architectures and OpenCL implementations. This is a desirable property as both hardware and software implementations are going to evolve.

When evaluated over 47 benchmark kernels, each with multiple input sizes, we achieve an average speedup of 14.3 over single-core execution. Compared to a state-of-the-art dynamic partitioning approach this equates to a performance boost of 1.57 times. Our approach also clearly outperforms the default strategies of using only the multi-core CPUs or only the GPU, which lead to a speedup of 4.73 and 9.21 over single-core execution, respectively.

Future work will investigate the use of our partitioning and mapping technique for multi-kernel OpenCL programs. Furthermore, guided dynamic schemes will

be explored that use kernel-specific information to improve the scheduling. This can be particularly useful in situations where a static, machine-learning based model has a low confidence of making the optimal decision, e.g. due to lack of training data.

References

1. Clang: a C language family frontend for LLVM (2010), <http://clang.llvm.org/>
2. AMD/ATI. ATI Stream SDK (2009), <http://www.amd.com/stream/>
3. Augonnet, C., Thibault, S., Namyst, R.: Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. In: Lin, H.-X., Alexander, M., Forsell, M., Knüpfer, A., Prodan, R., Sousa, L., Streit, A. (eds.) Euro-Par 2009. LNCS, vol. 6043, pp. 56–65. Springer, Heidelberg (2010)
4. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: STARPU: A unified platform for task scheduling on heterogeneous multicore architectures. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 863–874. Springer, Heidelberg (2009)
5. Bishop, C.M.: Pattern Recognition and Machine Learning (Information Science and Statistics). Springer-Verlag New York, Inc., Secaucus (2006)
6. Braun, T.D., Siegel, H.J., Beck, N., Bölöni, L., Maheswaran, M., Reuther, A.I., Robertson, J.P., Theys, M.D., Yao, B., Hensgen, D.A., Freund, R.F.: A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. In: Heterogeneous Computing Workshop (1999)
7. Buck, I., Foley, T., Horn, D.R., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: stream computing on graphics hardware. ACM Trans. Graph. 23(3) (2004)
8. Chang, C.-C., Lin, C.-J.: LIBSVM: a library for support vector machines (2001), Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>
9. Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S.: The scalable heterogeneous computing (SHOC) benchmark suite. In: GPGPU (2010)
10. Diamos, G.F., Yalamanchili, S.: Harmony: an execution model and runtime for heterogeneous many core systems. In: HPDC (2008)
11. Dolbeau, R., Bihan, S., Bodin, F.: HMPP: A hybrid multi-core parallel programming environment. In: Workshop on General Purpose Processing Using GPUs (2007)
12. Gregg, C., Brantley, J., Hazelwood, K.: Contention-aware scheduling of parallel code for heterogeneous systems. Technical report, Department of Computer Science, University of Virginia (2010)
13. Ibarra, O.H., Kim, C.E.: Heuristic algorithms for scheduling independent tasks on nonidentical processors. J. ACM 24(2) (1977)
14. Jiménez, V.J., Vilanova, L., Gelado, I., Gil, M., Fursin, G., Navarro, N.: Predictive runtime code scheduling for heterogeneous architectures. In: Sez nec, A., Emer, J., O'Boyle, M., Martonosi, M., Ungerer, T. (eds.) HiPEAC 2009. LNCS, vol. 5409, pp. 19–33. Springer, Heidelberg (2009)
15. Khokhar, A.A., Prasanna, V.K., Shaaban, M.E., Wang, C.-L.: Heterogeneous computing: Challenges and opportunities. IEEE Computer 26(6) (1993)
16. Khronos. OpenCL: The open standard for parallel programming of heterogeneous systems (October 2010), <http://www.khronos.org/opencv/>

17. Kim, J.-K., Shivle, S., Siegel, H.J., Maciejewski, A.A., Braun, T.D., Schneider, M., Tideman, S., Chitta, R., Dilmaghani, R.B., Joshi, R., Kaul, A., Sharma, A., Sripada, S., Vangari, P., Yellampalli, S.S.: Dynamic mapping in a heterogeneous environment with tasks having priorities and multiple deadlines. In: IPDPS (2003)
18. Kumar, R., Tullsen, D.M., Jouppi, N.P., Ranganathan, P.: Heterogeneous chip multiprocessors. *IEEE Computer* 38(11) (2005)
19. Linderman, M.D., Collins, J.D., Wang, H., Meng, T.H.Y.: Merge: a programming model for heterogeneous multi-core systems. In: ASPLOS (2008)
20. Luk, C.-k., Hong, S., Kim, H.: Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: MICRO (2009)
21. Maheswaran, M., Siegel, H.J.: A dynamic matching and scheduling algorithm for heterogeneous computing systems. In: Heterogeneous Computing Workshop (1998)
22. NVIDIA Corp. NVIDIA CUDA (2010), <http://developer.nvidia.com/object/cuda.html>
23. University of Illinois at Urbana-Champaign. Parboil benchmark suite (2010), <http://impact.crhc.illinois.edu/parboil.php>
24. Ravi, V.T., Ma, W., Chiu, D., Agrawal, G.: Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In: ICS (2010)
25. Rifkin, R.M., Klautau, A.: In defense of one-vs-all classification. *Journal of Machine Learning Research* (2004)
26. Ryoo, S., Rodrigues, C.I., Bagsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.-m.W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: PPOPP (2008)
27. Venkatasubramanian, S., Vuduc, R.W.: Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In: ICS (2009)
28. Wolfe, M.: Implementing the PGI accelerator model. In: GPGPU (2010)
29. Yarmolenko, V., Duato, J., Panda, D.K., Sadayappan, P.: Characterization and enhancement of dynamic mapping heuristics for heterogeneous systems. In: ICCP Workshops (2000)

Author Index

- Agrawal, Gagan 266
Axelsen, Holger Bock 144
- Bersch, Thomas 42
Bigonha, Mariza A.S. 2
Brunthaler, Stefan 164
Buchwald, Sebastian 42
Budimlić, Zoran 246
- Chen, Yuting 62
- d'Amorim, Marcelo 124
Demsky, Brian 198
- Eom, Yong hun 198
- Franchetti, Franz 225
- Grewe, Dominik 286
Guillon, Christophe 2
- Hendren, Laurie 22
Henretty, Tom 225
- Jenista, James C. 198
Joyner, Mackale 246
- Krishnamoorthy, Sriram 266
- Lameed, Nurudeen 22
Lhoták, Ondřej 82, 179
- Ma, Wenjing 266
- Naeem, Nomair A. 82
- O'Boyle, Michael F.P. 286
Odersky, Martin 1
- Pearce, David J. 104
Pouchet, Louis-Noël 225
- Quintão Pereira, Fernando Magno 2,
124
- Ramanujam, J. 225
Rimsa, Andrei 124
Rodriguez, Jonathan 179
- Sadayappan, P. 225
Sarkar, Vivek 246
Sol, Rodrigo 2
Stock, Kevin 225
Sun, Qiang 62
- Zhao, Jianjun 62
Zwinkau, Andreas 42