ARCoSS

LNCS 6011

Rajiv Gupta (Ed.)

Compiler Construction

19th International Conference, CC 2010
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2010
Paphos, Cyprus, March 2010, Proceedings

European Joint Conferences on Theory And Practice of Software
ETAPS
2010

Springer

Lecture Notes in Computer Science 6011

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

Advanced Research in Computing and Software Science

Subline of Lectures Notes in Computer Science

Rajiv Gupta (Ed.)

Compiler Construction

19th International Conference, CC 2010
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2010
Paphos, Cyprus, March 20-28, 2010
Proceedings

Springer

Volume Editor

Rajiv Gupta
University of California Riverside
Department of Computer Science and Engineering
Riverside, CA 92521, USA
E-mail: gupta@cs.ucr.edu

Library of Congress Control Number: 2010922288

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

ISSN 0302-9743
ISBN-10 3-642-11969-7 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-11969-9 Springer Berlin Heidelberg New York

springer.com

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Foreword

ETAPS 2010 was the 13th 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, FOSSACS, TACAS), 19 satellite workshops (ACCAT, ARSPA-WITS, Bytecode, CMCS, COCV, DCC, DICE, FBTC, FESCA, FOSS-AMA, GaLoP, GT-VMT, LDTA, MBT, PLACES, QAPL, SafeCert, WGT, and WRLA) and seven invited lectures (excluding those that were specific to the satellite events). The five main conferences this year received 497 submissions (including 31 tool demonstration papers), 130 of which were accepted (10 tool demos), giving an overall acceptance rate of 26%, with most of the conferences at around 24%. 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 toward 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 2010 was organised by the University of Cyprus 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)

and with support from the Cyprus Tourism Organisation.

The organising team comprised:

General Chairs:	Tiziana Margaria and Anna Philippou
Local Chair:	George Papadopoulos
Secretariat:	Maria Kittira
Administration:	Petros Stratis
Satellite Events:	Anna Philippou
Website:	Konstantinos Kakousis.

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

Vladimiro Sassone (Southampton, Chair), Parosh Abdulla (Uppsala), Luca de Alfaro (Santa Cruz), Gilles Barthe (IMDEA-Software), Giuseppe Castagna (CNRS Paris), Marsha Chechik (Toronto), Sophia Drossopoulou (Imperial College London), Javier Esparza (TU Munich), 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 (LM Munich), Joost-Pieter Katoen (Aachen), Paul Klint (Amsterdam), Jens Knoop (Vienna), Shriram Krishnamurthi (Brown), Kim Larsen (Aalborg), Rustan Leino (MSR Redmond), Gerald Luettgen (Bamberg), Rupak Majumdar (Los Angeles), Tiziana Margaria (Potsdam), Ugo Montanari (Pisa), Oege de Moor (Oxford), Luke Ong (Oxford), Fernando Orejas (Barcelona) Catuscia Palamidessi (INRIA Paris), George Papadopoulos (Cyprus), David Rosenblum (UCL), Don Sannella (Edinburgh), João Saraiva (Minho), Michael Schwartzbach (Aarhus), Perdita Stevens (Edinburgh), Gabriele Taentzer (Marburg), and Martin Wirsing (LM Munich).

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 2010, George Papadopoulos, for arranging for us to have ETAPS in the most beautiful surroundings of Paphos.

January 2010 Vladimiro Sassone

Preface

The CC 2010 Programme Committee is pleased to present the proceedings of the 19th International Conference on Compiler Construction (CC 2010) which was held during March 25–26 in Paphos, Cyprus, as part of the Joint European Conference on Theory and Practice of Software (ETAPS 2010). As in the last few years, papers were solicited on a wide range of areas including traditional compiler construction, compiler analyses, runtime systems and tools, programming tools, techniques for specific domains, and the design and implementation of novel language constructs. We received submissions from a wide variety of areas and the papers in this volume reflect this variety.

The Programme Committee received 56 submissions. From these, 16 research papers were selected, giving an overall acceptance rate of 28%. The Programme Committee carried out the reviewing and paper selection completely electronically, in two rounds. In the first round at least three Programme Committee members reviewed each paper, and through discussion among the reviewers those papers which were definite "accepts" and those which needed further discussion were identified. Our second round concentrated on the papers needing further discussion, and we added an additional review to help us decide which papers to finally accept.

Many people contributed to the success of this conference. First of all, we would like to thank the authors for all the care they put into their submissions. Our gratitude also goes to the Programme Committee members and external reviewers for their substantive and insightful reviews. Also, thanks go to the developers and supporters of the EasyChair conference management system for providing a reliable, sophisticated and free service.

CC 2010 was made possible by the ETAPS Steering Committee and the local Organizing Committee. Finally, we are grateful to Jim Larus for giving the CC 2010 invited talk.

January 2010 Rajiv Gupta

Conference Organization

Programme Chair

Rajiv Gupta	UC Riverside, USA

Programme Committee

Jack Davidson	Unversity of Virginia, USA
Paul Feautrier	Ecole Normale Supérieure de Lyon, France
Guang Gao	Unversity of Delaware, USA
Antonio Gonzalez	Intel Barcelona Research Center, Spain
Laurie Hendren	McGill University, Canada
Robert Hundt	Google, USA
Suresh Jagannathan	Purdue University, USA
Chandra Krintz	UC Santa Barbara, USA
Julia Lawall	DIKU, Denmark
Madan Musuvathi	Microsoft Research, USA
Michael O'Boyle	University of Edinburgh, USA
Yunheung Paek	Seoul National University, Republic of Korea
Santosh Pande	Georgia Institute of Technology, USA
Christoph von Praun	Georg-Simon-Ohm Hochschule Nürnberg, Germany
Vivek Sarkar	Rice University, USA
Bernhard Scholz	The University of Sydney, Australia
Bjorn De Sutter	Ghent University, Belgium
Andreas Zeller	Saarland University, Germany

External Reviewers

Alex Aleta
Rajkishore Barik
Indu Bhagat
Zoran Budimlic
Bernd Burgstaller
Qiong Cai
Romain Cledat
Josep M. Codina
Jesse Doherty
S. M. Farhad
Enric Gibert
Christian Grothoff
Lang Hames
Surinder Kumar Jain
Surinder Jain
Kyoungwon Kim
Yongjoo Kim
Tushar Kumar
Akash Lal
Nurudeen Lameed
Jongwon Lee
David Li
Pedro Lopez
Marc Lupon

Carlos Madriles
Nagy Mostafa
Sarang Ozarde
Greogory Prokopski
Easwaran Raman
August Schwerdfeger
Tianwei Sheng
Jun Shirako
Jaswanth Sreeram
Neil Vachharajani
Xavier Vera
Eran Yahav
Seungjun Yang
Jonghee Youn
Jisheng Zhao

Table of Contents

Invited Talk

Optimization Techniques

Program Transformations

Program Analysis

Register Allocation

High-Performance Systems

Programming Clouds

James Larus

Microsoft Research
One Microsoft Way
Redmond, WA 98052
larus@microsoft.com

Abstract. Cloud computing provides a platform for new software applications that run across a large collection of physically separate computers and free computation from the computer in front of a user. Distributed computing is not new, but the commodification of its hardware platform—along with ubiquitous networking; powerful mobile devices; and inexpensive, embeddable, networkable computers—heralds a revolution comparable to the PC.

Software development for the cloud offers many new (and some old challenges) that are central to research in programming models, languages, and tools. The language and tools community should embrace this new world as fertile source of new challenges and opportunities to advance the state of the art.

Keywords: cloud computing, programming languages, software tools, optimization, concurrency, parallelism, distributed systems.

1 Introduction

As I write this paper, cloud computing is a hot new trend in computing. By the time you read it, the bloom may be off this rose, and with a sense of disillusionment at yet another overhyped fad, popular enthusiasm may have moved on to the next great idea. Nevertheless, it is worth taking a close look at cloud computing, as it represents a fundamental break in software development that poses enormous challenges for the programming languages and tools.

Cloud computing extends far beyond the utility computing services offered by Amazon's AWS, Microsoft's Azure, or Google's AppEngine. These services provide a foundation for cloud computing by supplying on-demand, internet computing resources on a vast scale and at low cost. Far more significant, however, is the software model this hardware platform enables; one in which software applications are executed across a large collection of physically separate computers and computation is no longer limited to the computer in front of you. Distributed computing is not new, but the commodification of its hardware platform—along with ubiquitous networking; powerful mobile devices; and inexpensive, embeddable, networkable computers—may bring about a revolution comparable to the PC.

Programming the cloud is not easy. The underlying hardware platform of clusters of networked parallel computers is familiar, but not well supported by programming models, languages, or tools. In particular, concurrency, parallelism, distribution, and

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 1–9, 2010.

availability are long-established research areas in which progress and consensus has been slow and painful. As cloud computing becomes prevalent, it is increasingly imperative to refine existing programming solutions and investigate new approaches to constructing robust, reliable software. The languages and tools community has a central role to play in the success of cloud computing.

Below is a brief and partial list of areas that could benefit from further research and development. The discussion is full of broad generalizations, so if I malign or ignore your favorite language or your research, excuse me in advance.

1. **Concurrency.** Cloud computing is an inherently concurrent and asynchronous computation, in which autonomous processes interact by exchanging messages. This architecture gives raise to two forms of concurrency within a process:

- The first, similar to an operating system, provides control flow to respond to inherently unordered events.
- The second, similar to a web server, supports processing of independent streams of requests.

   Neither use of concurrency is well supported by programming models or languages. There is a long-standing debate between proponents of threads and event handling [1-3] as to which model best supports concurrency. Threads are close to a familiar, sequential programming model, but concurrency still necessitates synchronization to avoid unexpected state changes in the midst of an apparently sequential computation. Moreover, the high overhead of a thread and the cost of context switching limits concurrency and constrains system architectures. Event handlers, on the other hand, offer low overhead and feel more closely tied to the underlying events. However, handlers provide little program structure and scale poorly to large systems. They also require developers to explicitly manage program state. Other models, such as state machines or Actors, have not yet emerged in a general-purpose programming language.

2. **Parallelism.** Cloud computing runs on parallel computers, both on the client and server. Parallelism currently is the dominate approach to increasing processor performance without exceeding power dissipation limitations [4]. Future processors are likely to become more heterogeneous, as specialized functional units greatly increase performance or reduce power consumption for specific tasks.

   Parallelism, unfortunately, is a long-standing challenge for computer science. Despite four decades of experience with parallel computers, we have not yet reached consensus on the underlying models and semantics or provided adequate programming languages and tools. For most developers, shared-memory parallel programs are still written in the assembly language of threads and explicit synchronization. Not surprisingly, parallel programming is difficult, slow, and error-prone and will be a major impediment in developing high-performance cloud applications.

   The past few years have seen promising research on new, higher-level parallel programming models, such as transactional memory and deterministic execution [5, 6]. Neither is a panacea, but both abstractions could hide some complexities of parallelism.

3. **Message passing.** The alternative to shared-memory parallel programming is message passing, ubiquitous on the large clusters used in scientific and technical

computing. Because of its intrinsic advantages, message passing will be the primary parallel programming model for cloud computing as well. It scales across very large numbers of machines and is suited for distributed systems with long communications latencies. Equally important, message passing is a better programming model than shared memory as it provides inherent performance and correctness isolation with clearly identified points of interactions. Both aspects contribute to more secure and robust software systems [7].

Message passing can be more difficult to program than shared memory, in large measure because it is not directly supported by many programming languages. Message-passing libraries offer an inadequate interface between the asynchronous world of messages and the synchronous control flow of procedure calls and returns. A few languages, such as Erlang, integrate message into existing language constructions such as pattern matching [8], but full support for messages requires communications contracts, such as Sing# [9], and tighter integration with the type system and memory model.

4. **Distribution.** Distributed systems are a well-studied area with proven solutions for difficult problems such as replication, consistency, and quorum. This field has focused considerable effort on understanding the fundamental problems and in formulating efficient solutions. One challenge is integrating these techniques into a mainstream programming model. Should they reside in libraries, where developers need to invoke operations at appropriate points, or can they be better integrated into a language, so developers can state properties of their code and the run-time system can ensure correct execution?

5. **High availability.** The cloud end of cloud computing provides of services potentially used by millions of clients, and these services must be highly available. Failures of systems used by millions of people are noteworthy events widely reported by the media. And, as these services become integrated into the fabric of everyday life, they become part of the infrastructure that people depend on for their businesses, activities, and safety.

   High availability is not the same as high reliability, the focus of much research on detecting and eliminating software bugs. A reliable system that runs slowly under heavy load may fail to provide a necessary level of service. Conversely, components of a highly available system can fail frequently, but a properly architected system will continue to provide adequate levels of service [10].

   Availability starts at the architecture level of the system, but programming languages have an important role to play in the implementation. Existing language provide little support for systematically handling unexpected and erroneous conditions beyond exceptions, which are notoriously difficult to use properly [11]. Error handling is complex and delicate code that runs when program invariants are violated, but it is often written as an afterthought and rarely thoroughly tested. Better language support, for example lightweight, non-isolated transactions, could help developers handle and recover from errors [12].

6. **Performance.** Performance is primarily a system-level concern in cloud computing. Many performance problems involve shared resources running across large numbers of computers and complex networks. Few techniques exist to analyze a design or

system in advance, to understand bottlenecks or predict performance. As a consequence, current practice is to build, overprovision, measure, tweak, and pray.

One pervasive concern is detecting and understanding performance problems. Amazon's Dynamo system uses service-level agreements (SLA) among system components to quickly identify performance problems [13]. These SLAs are the performance equivalents of pre- and post-conditions. Making performance into a first-class programming abstraction, with full language and tools support, would help with the construction of complex, distributed systems.

7. **Application partitioning.** Current practice is to statically partition functionality between a client and service by defining an interface and writing both endpoints independently. This approach leads to inflexible architectures that forego opportunities to migrate computations to where they could run most efficiently. In particular, battery powered clients such as phones are limited in memory or processing capability. Migrating a running computation from a phone to a server might enable it to complete faster (or at all) or to better utilize limited network bandwidth by moving computation to data rather than the reverse [14].

   Even within a data center, code mobility is valuable. It permits server workloads to be balanced to improve performance or consolidated to reduce power consumption. Currently virtual machines move an entire image, from the operating system up, between computers. Finer-grain support for moving computations could lower the cost of migration and provide mechanisms useful in a wider range of circumstances.

   Statically partitioned systems could benefit from better language support. Microsoft's prototype Volta tool offered a single-source programming model for writing client-server applications [15]. The developer writes a single application, with annotations as to which methods run on the client or server. The Volta compiler partitions the program into two executables, a C# one for running on the server and a Javascript one for the client. Similar programming models could simplify the development of cloud applications by providing developers with a higher-level abstraction of their computation.

8. **Defect detection.** Software defect detection has made considerable progress over the past decade in finding low-level bugs in software. The tools resulting from this effort are valuable to cloud computing, but are far from sufficient. Few tools have looked for bugs in complex systems built from autonomous, asynchronous components. Although this domain appears similar to reactive systems, the complexity of cloud services present considerable challenges in applying techniques from this area.

9. **High-level abstractions.** Google's Map-Reduce and Microsoft Dryad are two higher level programming models that hide much of the complexity of writing a server-side analytic application [16, 17]. A simple programming model hides much of the complexity of data distribution, failure detection and notification, communication, and scheduling. It also opens opportunities for optimizations such as speculative execution. These two abstractions are intended for code that analyzes large amounts of data. There is a pressing need for similarly abstract models for writing distributed client-server applications and web services.

This list of open problems is not exhaustive, but instead is a starting point for research directly applicable to problems facing developers of cloud computing applications.

2 Orleans

Orleans is a project under development in the Cloud Computing Futures (CCF) group in Microsoft Research. Its goal is to achieve significant improvements in productivity of building cloud computing applications. Orleans specifically addresses the challenges of building, deploying, and operating very large cloud applications that encompass thousands of machines in multiple datacenters, constantly evolving software, and large teams to construct, maintain, and administer these properties.
At a coarse level, Orleans consists of three interdependent components:

- Programming model
- Programming language and tools
- Runtime.

Software for a cloud application, both the portion that runs on servers in a data center and the part that runs on clients, will be written in DC#, an extended version of C# that provides explicit support for the Orleans programming model. Orleans tools help a developer build reliable code by providing static and dynamic defect detection and test tools. Application code runs on the Orleans run-time system, which provides robust, tested implementations of the abstractions needed for these systems. These abstractions in turn execute on Azure, Microsoft's data center operating system.

2.1 Design Philosophy

Orleans is frankly a prescriptive system—it strongly encourages the use of software architectures and design patterns that have proven themselves in practice. Because Orleans targets large-scale cloud computing, the key criterion for adopting a principle is that it results in a scalable, resilient, reliable system. Cloud software is scalable if it a system can grow to accommodate a steadily increasing number of clients without requiring major rewrites, even when the increase in volume spans multiple orders of magnitude. The common practice today is to plan on several complete rewrites of a system as an internet property grows in popularity, even though there are multiple examples of scalable internet properties whose design principles are widely known. Today's general-purpose programming languages and tools provide little or no support for these principles, so the burden of scalability is shifted to developers; and consequently most new enterprises choose short-term expediency to get their websites up quickly.

A system is resilient if it can tolerate failures in its components: the computers, communication network, other services on which it relies, and even the data center in which it runs. Toleration requires the system to detect a failure, respond to it in a manner that minimizes the effect of a failure on unrelated components and clients, restore service when possible by using other resources, and resume execution when the failure is corrected.

The distributed systems community has studied techniques for building scalable, resilient software systems for many years. A small number of abstractions have proven their value in building these systems: asynchronous communications and software architecture; data partitioning; data replication; consensus; and consistent, systematic design policies. Orleans will build these ideas into its programming and data model and provide first-class support for them in the DC# language and tools. These abstractions by no means guarantee a well-written program or successful system; it still remains true that it is possible to write a bad program in any language. However, these abstractions have proven their value in many systems and are well studied and understood, and they provide a solid basis for building resilient systems.

2.2 Centrality of Failure

In ordinary software, error-handling code is home to a disproportionate share of defects. This code is difficult to write because invariants and preconditions often are invalid after an error and paths through this code are less well tested because they are uncommon. Distributed systems complicate error handling by introducing new failure modes, such as asynchronous communications and partial failure, which are challenging to reason about and difficult to handle correctly. Much of the difficulty of building a reliable internet property is attributable to asynchrony and failure.

Distributed systems research offer some techniques for masking failures and asynchrony (e.g., Paxos), but they have significant drawbacks and are unsuitable to mask all failures in a responsive service. Paxos and other replication strategies increase the quantity of resources dedicated to a computation task by a significant (3 – 5x) amount. In addition, these techniques increase the time to perform an operation. Because of increased cost and latency, replication strategies must be used sparingly in scalable services.

Other techniques, such as checkpoint and restart, are more successful for non-reactive computations (e.g., large-scale analytic computations implemented with map-reduce or Dryad) in which it is possible to capture input to a portion of a computation and in which a large recovery cost is less than the far-more-expensive alternative of rerunning the entire computation. Another advantage is that it is possible to automate the failure detection and error recovery process.

Programming models also have a significant influence on the correctness and resiliency of code. For example, every client making a remote procedure call (RPC) has to deal with three possibilities: the call succeeds and the client knows it; the call fails and the client knows it; the call times out and the client does not know whether it succeeded or failed. In more sophisticated models that allow simultaneous RPC calls, complexity further increases when calls complete in arbitrary orders. Complicating this reasoning is the syntactic similarity of an RPC call and a conventional call, which encourage a developer to conflate the two, despite their vast difference in cost and semantics. For these reasons, undisciplined use of RPC has proven to be a bad abstraction for building distributed systems.

2.3 Orleans Programming Model

The Orleans programming model is inherently more resilient. An application is composed of loosely coupled components, each of which executes in its own failure container. In Orleans, these components are called grains. A grain consists of a single-threaded computation with its local state. It can fail and be restarted without directly affecting the execution of any other grain—though it may indirectly affect a dependent grain that cannot respond appropriately to its failure. All communications between grains occurs across channels: higher-order (i.e., can send a channel over a channel), strongly typed paths for sending messages between grains. The code within a grain is inherently asynchronous, to deal with the unpredictable arrival of messages across multiple channels or the unpredictable ordering of messages between asynchronous services. This model exposes the reality of a distributed system (communication via messages that arrive at unpredictable times) but constrains it, in single threaded, isolated containers, to simplify reasoning about and analyzing code.

Grains are not distributed objects. The differences between the two models are fundamental. Orleans does not provide a pointer or reference to a grain, nor do grains reside in a global address space. A computation communicates with a grain through a channel, which is a capability, not a reference. A common channel allows two grains to communicate according to the channel's protocol. However, the channel does not uniquely identify either grain since channels can be passed around. Nor does a channel identify the location of a grain, which can migrate between machines while the channel is active.

Moreover, interactions between grains are asynchronous, not RPC. One grain can request another grain perform an operation by sending a message (which could be wrapped in syntactic sugar to look like a method invocation). The receiving grain has the freedom to process this request in any order with respect to its on-going computations and other requests. When the operation completes, the grain can send back its result. In general, the first grain will not block waiting for this value, as it would for a method call, but instead will process other, concurrent operations.

An important property of a grain is that it can migrate between computers. Migration allows Orleans to adaptively execute a system: to reduce communication latency by moving a computation closer to a client or data resource, to increase fault tolerance by moving a computation to a less tightly coupled system, and to balance the load among servers.

Grains encourage an SPMD (single program, multiple data) style of programming. The same computation (code) runs in all grains of a particular type, and each grain's computation executes independently of other grains and the computations are initiated at different times.

However, it is also possible to use grains to implement a dataflow programming model. In this case, a grain is a unit of computation that accepts input and sends results across channels. Dataflow is appropriate for streaming computation and can achieve the scalability of asynchronous data parallelism by replicating dataflow graphs and computations.

What is the appropriate size for a grain? In today's scalable services, it is necessary to partition the data manipulated by the service at a fine granularity, to allow for rebalancing in the face of load and usage skew. For example, code acting on behalf of a Messenger user does not assume it is co-located with another Messenger user, and it must expect the location of a user's data to change when a server is added or removed. Similar properties hold for Hotmail user's address books, subscriptions in Live Mesh's pub-sub service, ongoing meetings in Office Communications Server, rows in Google's BigTable, keys in Amazon's Dynamo, etc. With this fundamental assumption, a system can spread a large and varying collection of data items (e.g., a user's IM presence) across a large number of servers, even across multiple data centers. Though partitioning by user is illustrative, grains can represent many other entities. For example, a user's mailbox may contain grains corresponding to mail messages.

2.4 Orleans Data Model

Data in cloud computing application exists in a richer, more complex environment than in non-distributed applications. This environment has a number of orthogonal dimensions. Unlike the local case, a single model does not meet all needs. Different grains will require different guarantees, and the developer must assume responsibility for selecting the properties that match the importance of data, semantics of operations, and performance constraints on the system. Orleans will implement a variety of different types of gains that support the different models for the data they contain, so an application developer can declare the properties of a particular grain and expect the system to implement its functionality.

Data can be persistent, permitting it to survive a machine crash. Changes to the data are written to durable storage and Orleans can keep independent copies of the data on distinct computers (or data centers), to increase availability in the face of resource failures.

Replicating the data among machines introduces the issue of consistency among the replicas. Strong consistency requires the replicas to change simultaneously, while weaker models tolerate divergence among the copies.

Within a grain, Orleans supports a simple, local concurrency model. Data local to the grain is only modified by code executing in the grain and execution is single-threaded, so from the perspective of this code, the execution model is mostly sequential. However, when code for an operation ends and yields control back to the grain, other operations can execute and modify the grain's local state, so a developer cannot make assumptions across turns in a grain.

Orleans does not impose a single model on the operations exported by a grain. The semantics of concurrent operations has been formalized in numerous ways, and different models (e.g., sequential consistency, serializability, linearizability) offer varying tradeoffs among simplicity, generality, and efficiency. Orleans will need to provide the support that enables a developer to implement these models where appropriate.

3 Conclusion

Until recently, only a handful of people had ever used more than one computer to solve a problem. This is no longer true, as search engines routinely execute a query across a thousand or so computers. Cloud computing is the next step into a world in which computation and data are no longer tightly tied to a specific computer and it is possible to share vast computing resources and data sets to build new forms of computing that go far beyond the familiar desktop or laptop PCs.

Software development for the cloud offers many new (and some old challenges) that are central to research in programming models, languages, and tools. The language and tools community should embrace this new world as fertile source of new challenges and opportunities to advance the state of the art.

References

1. Adya, A., Howell, J., Theimer, M., Bolosky, W.J., Douceur, J.R.: Cooperative Task Management without Manual Stack Management or, Event-driven Programming is Not the Opposite of Threaded Programming. In: Proceedings of the USENIX 2002 Conference, pp. 289–302. Usenix, Monterey (2002)
2. Ousterhout, J.: Why Threads are a Bad Idea (for most purposes). In: Proceedings of the 1996 USENIX Technical Conference. Usenix, San Diego (1996)
3. von Behren, R., Condit, J., Zhou, F., Necula, G.C., Brewer, E.: Capriccio: Scalable Threads for Internet Services. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, pp. 268–281. ACM, Bolton Landing (2003)
4. Larus, J.: Spending Moore's Dividend. Communications of the ACM 52, 62–69 (2009)
5. Larus, J., Kozyrakis, C.: Transactional Memory. Communications of the ACM 51, 80–88 (2008)
6. Bocchino Jr., R.L., Adve, V.S., Adve, S.V., Snir, M.: Parallel Programming Must Be Deterministic by Default. In: First USENIX Workshop on Hot Topics in Parallelism. Usenix, Berkeley (2009)
7. Hunt, G., Larus, J.: Singularity: Rethinking the Software Stack. ACM SIGOPS Operating Systems Review 41, 37–49 (2007)
8. Armstrong, J.: Programming Erlang: Software for a Concurrent World. The Pragmatic Bookshelf, Raleigh (2007)
9. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J.R., Levi, S.: Language Support for Fast and Reliable Message Based Communication in Singularity OS. In: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems, Leuven, Belgium, pp. 177–190 (2006)
10. Barroso, L.A., Hölzle, U.: The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, vol. 6. Morgan & Claypool, San Francisco (2009)
11. Weimer, W., Necula, G.C.: Exceptional Situations and Program Reliability. ACM Transactions on Programming Languages and Systems 30, 1–51 (2008)
12. Lenharth, A., Adve, V.S., King, S.T.: Recovery Domains: An Organizing Principle for Recoverable Operating Systems. In: Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 49–60. ACM, Washington (2009)
13. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's Highly Available Key-value Store. In: Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles, pp. 205–220. ACM, Stevenson (2007)
14. Gray, J.: Distributed Computing Economics. Microsoft Research, p. 6. Redmond, WA (2003)
15. anon.: Volta Technology Preview from Microsoft Live Labs Helps Developers Build Innovative, Multi-Tiered Web Applications with Existing Tools, Technology. Microsoft Press Pass (2007)
16. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM 51, 107–113 (2008)
17. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, pp. 59–72. ACM, Lisbon (2007)

Mining Opportunities for Code Improvement in a Just-In-Time Compiler

Adam Jocksch[1], Marcel Mitran[2], Joran Siu[2],
Nikola Grcevski[2], and José Nelson Amaral[1]

[1] Department of Computing Science
University of Alberta, Edmonton, Canada
{ajocksch,amaral}@cs.ualberta.ca
[2] IBM Toronto Software Laboratory, Toronto, Canada

Abstract. The productivity of a compiler development team depends on its ability not only to the design effective solutions to known code generation problems, but also to uncover potential code improvement opportunities. This paper describes a data mining tool that can be used to identify such opportunities based on a combination of hardware-profiling data and on compiler-generated counters. This data is combined into an Execution Flow Graph (EFG) and then FlowGSP, a new data mining algorithm, finds sequences of attributes associated with subpaths of the EFG. Many examples of important opportunities for code improvement in the IBM® Testarossa compiler are described to illustrate the usefulness of this data mining technique. This mining tool is specially useful for programs whose execution is not dominated by a small set of frequently executed loops. Information about the amount of space and time required to run the mining tool are also provided. In comparison with manual search through the data, the mining tool saved a significant amount of compiler development time and effort.

1 Introduction

Compiler developers continue to face the challenges of accelerated time-to-market and significantly reduced release cycles for both hardware and software. Microarchitectures continue to grow in numbers, complexity, and diversity. In this evolving technological environment, commercial-compiler developing teams must discover and rank the next set of opportunities for code transformations that will provide the highest performance improvement per development cost ratio.

The discovery of opportunities for profitable code transformations in large enterprise applications presents additional challenges. Traditionally, compiler developers have relied on the intuition that the code that is relevant for performance improvement is located in easily identifiable, frequently executed, regions of the code — often called *hot loops*. However, many enterprise applications do not exhibit discernible regions of frequently executed code. Rather, these applications exhibit a *flat profile*: thousands of methods are invoked along an execution path, and no single method accounts for a significant portion of the

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 10–25, 2010.

execution time — even though a typical transaction executes millions of instructions. Thus, focusing development effort on any single method provides negligible overall performance improvement. However, these applications may display code patterns that appear repeatedly throughout the code base. Even though no single instance of such a pattern is executed frequently, the aggregated run time of the pattern may be significant. Applications with flat profiles are becoming increasingly important for commercial compilers that are used to generate code for middleware and enterprise information-technology (IT) infrastructure.

Thus, a challenge when developing a compiler for applications with flat profiles is to discover code patterns whose aggregated execution time is significant so that development efforts can be focused into improving the code generation for such patterns. This paper describes a data mining infrastructure, based on the recently developed FlowGSP algorithm [13], which can be used for automatic analysis of code compiled by the IBM Testarossa Just-in-Time (JIT) Compiler [8]. This infrastructure was used to discover patterns in the code generated for applications running in the IBM® WebSphere® Application Server and for SPECjvm2008 [20] running under Linux® for System Z® [22,19].

WebSphere Application Server is a fully compliant Java™ Enterprise Edition (JEE) application server written in Java code [11]. This paper uses the DayTrader Benchmark in the WebSphere Application Server[7]. This benchmark produces a typical WebSphere Application Server profile reporting the compilation of thousands of methods, with no method representing more than 2% of the total execution time. For instance, cache misses represent 12% of the overall run time in one run of a certain application in application server. But, to account for 75% of the misses requires the aggregation of misses from 750 different methods [8].

SPECjvm2008 exemplifies the growing variety of industry standards that are quickly expanding the scope of benchmarks. The SPECjvm2008 suite comprises more than double the number of benchmarks that were in its predecessor, SPECjvm98 [18]. Some of the benchmarks in the newer suite have flat profiles, making the analysis and identification of opportunities for code improvement more difficult, more tedious and more indeterminate.

The IBM Testarossa JIT compiler ships as part of the IBM Developer Kit for Java which powers thousands of mission-critical applications on everything from embedded devices, to desktops, to high-end servers. The IBM Testarossa JIT is a state-of-the-art commercial compiler that offers a very complete set of traditional OO-based and Java-based optimizations. As a dynamic compiler, Testarossa is also equipped with a sophisticated compilation control system for online feedback-directed re-compilation [21].

The analysis presented in this paper was performed on Linux for System z. System z10™ is the latest and most powerful incarnation of IBM's mainframe family, which continues to provide the foundation for IT centers for many of the world's largest institutions. The System z10 processor has a 4.4 GHz dual core super-scalar pipeline, executes instructions in order, and can be characterized as an address-generation-interlocked pipeline. This processor is a complex instruction set computer with a rich set of register-to-register, register-to-storage, storage-to-storage, and complex branching operations, in addition to hardware co-processors for cryptography, decimal-floating-point, and Lempel-Ziv compression [22]. The System z10 processor also provides an extensive set of performance-monitoring counters that can be used to examine the state of the processor as it executes the program.

The data mining infrastructure was applied to a large set of compiler attributes and hardware counters. The attributes and hardware data are organized in a directed graph representing program flow. Edge frequencies are used to represent the probabilistic flow between basic blocks. The FlowGSP algorithm is general and can mine any flow graph. A vertex in this flow graph may represent any single-entry-single-exit region such as an instruction, a basic block, a byte-code, or a method. Attributes are associated with each vertex, and the algorithm mines for sequences of attributes along a path.

The main contributions of this paper are:

- An introduction of the problem of identifying important code patterns that occur in applications with flat profiles, such as enterprise applications.
- A description of a new data mining framework that can be used to discover important opportunities for code generation improvement in a commercial dynamic compiler environment.
- A demonstration of the effectiveness of the data mining tool through the narrative of several discoveries in the code generated for the System z architecture by the IBM Testarossa compiler.
- Statistics on space and time requirements for the usage of the mining tool in this environment. This information should be relevant for other compiler groups that wish to implement a similar tool, as well as for researchers that wish to improve on our design.

Section 2 explains the need for the mining tool through the description of one of the important discoveries in a very common segment of code. The mining tool is described in Section 3. Several additional improvement opportunities discovered by the tool are described in Section 4. Experimental data describing the time and space requirements for the usage of the tool in the Testarossa environment is presented in Section 5. Section 6 discusses previous work related to the development of similar analysis tools.

2 Motivating Case Study

This section outlines the motivation for the use of data mining to discover patterns that account for significant execution time by describing one such pattern discovered by FlowGSP. The data mined by FlowGSP to discover this pattern includes, instruction type, execution time, cache misses, pipeline interlock, *etc* [13]. This pattern is part of the array-copy code generated by Testarossa for the System z10 platform. FlowGSP identified that, in some benchmarks, more than 5% of the execution time was due to a single instruction called `execute` (EX).

This finding is surprising because the IBM Testarossa JIT compiler uses this instruction in only one scenario – to implement the tail-end of an array copy.[1] More specifically, a variable-length array copy is implemented with a loop that executes an MVC (Move Characters) instruction. The MVC instruction is very efficient at copying up to 256 bytes. The 256-byte copy length is encoded as a literal value of the instruction. Figure 1 shows the code generated for array copying. Any residual of the copy after the repeated execution of MVCs is handled by using the EX instruction. The EX instruction executes a target instruction out of order. Before executing the target, EX replaces an 8-bit literal value specified in the target with an 8-bit field from a register specified in EX. The overloading is done through an OR of the two bit fields. For the residual array-copy code generated by Testarossa, the register specified in EX contains the length of the residual array and the target instruction is a MVC instruction.

```
          Rsrc = Address of source array;
          Rtrgt = Address of target array;
          while(Rlength >= 256)
                MVC  Rsrc, Rtrgt, 256
                 Rsrc = Rsrc + 256;
                 Rtrgt = Rtgt + 256;
                 Rlength = Rlength - 256;
                }
          EX ResLabel, Rlength, mvcLabel;
           ...
ResLabel: MVC Rsrc, Rtrgt, 0
```

Fig. 1. Pseudo-assembly code for array copy

After the data mining tool identified that 5% of the time was spent in EX, we examined the profiling data more carefully to find out that the 5% of time spent in EX is spread over several methods. Therefore, the time spent in the EX instruction would not be apparent from a study of individual methods. Moreover, part of that time is spent in the MVC instruction. Nonetheless, the EX instruction incurs significantly more misses in the data-cache and the translation-look-aside-buffer (TLB) misses than expected. There are two potential reasons for this:

1. The length of many array copies is less than 256 byte long. In this case, data cache misses would occur while fetching the source/target operands of MVC.
2. The EX instruction misses the cache upon fetching the overloaded MVC. This miss occurs because the targeted MVC instruction is located next to other insructions used by the program, and hence resides in the instruction cache. On a z10, the EX instruction needs the targeted MVC in the data

[1] Array copies use 256-byte copy instructions, the tail-end is any final portion of the copy that is smaller than 256 bytes.

cache. Moving the targeted MVC from the instruction cache to the data cache incurs an extra cost that was not apparent to the compiler designers.

This discovery started an important review of the array-copy code generated by the compiler. A suitable strategy must be designed to isolate the targeted MVC from the other data values that are located around it. This strategy must take into consideration the long lines in the architecture.

An important question is why there is the need for a data mining tool to discover such an opportunity. Could simple inspection of the hardware and compiler profiling data reveal this opportunity? Even if a developer were to spot the cache miss caused by the EX instruction, she would have no way to know that the aggregation of occurrences of EX in many infrequently executed methods is amount to significant performance loss that needs to be addressed. Even though profile logs of code generated by this commercial compiler had been inspected by hand for many years, the issue with the use of EX and MVC for array copy had never been regarded as worthy of attention from the team. Once the mining tool reported it, one of the developers remarked: “Now we can see!”.

3 The Mining Tool

The mining tool design is based on a new data mining algorithm called FlowGSP. FlowGSP mines for subpaths in an execution flow graph (EFG). Jocksch formally defines a an EFG as a directed flow graph possibly containing cycles [13]. Each EFG vertex is annotated with a normalized weight and has an associated list of attributes. Each EFG edge is annotated with a normalized execution frequency. A subpath is of interest if either its frequency of execution, called *frequency support*, or vertex weights, called *weight support*, is above a set threshold. A subpath is also of interest if the difference between its frequency and weight support is higher than a *difference support*. FlowGSP reports sequences of attributes whose aggregated support over the entire EFG is higher than the specified supports.

FlowGSP is an extension of the Generalized Sequential Pattern (GSP) algorithm, originally introduced by Agrawal *et al.* [1]. The main difference between FlowGSP and GSP is that GSP was designed to mine for sequences of attributes in a list of totally ordered transactions while FlowGSP enables the mining for sequences of attributes in subpaths of a flow graph, thus allowing a partial order between the transactions (vertices in the EFG). Similar to GSP, FlowGSP allows for windows and gaps. A window allows attributes that occur in distinct vertices that are close in a subpath — within the specified window — to be regarded as occurring in the same vertex. A gap is a maximum number of vertices in the subpath that do not contain attributes in the sequence.

3.1 Preparation of Data for Mining

The overall architecture and flow in a system that uses FlowGSP for mining is shown in Figure 2. Performance-counter data generated by the hardware [12]

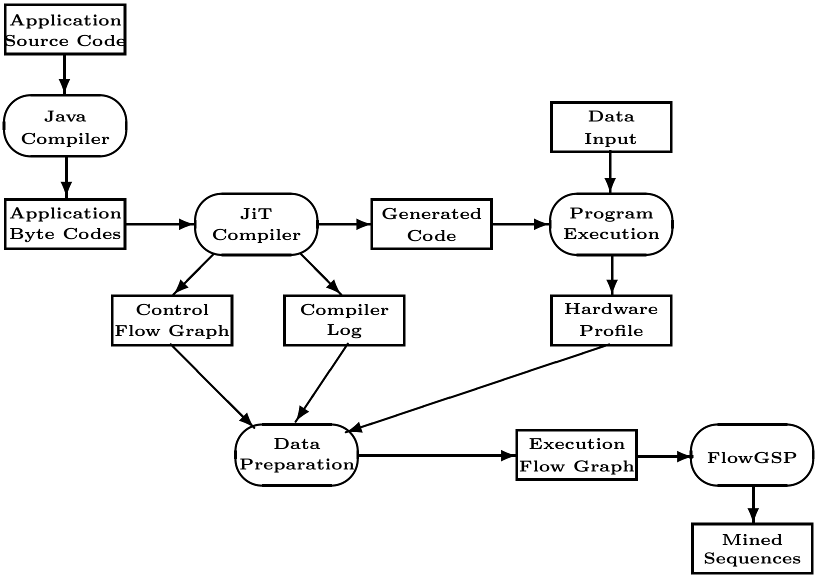


Fig. 2. Overall architecture and flow in system that uses FlowGSP for mining

is added to the control-flow-graph representation of the program created by the compiler to produce the input for the mining tool. The Testarossa compiler comes equipped with a rich set of logging features, including the ability to report all generated machine instructions. The only modification to the compiler was to annotate each instruction with a corresponding basic block so that the log can then be transformed into an EFG. In the implementation of the mining tool, the hardware performance counter information and the control-flow-graph data from the compiler are stored in IBM DB2® Version 9.1 Express Edition for Linux, a relational database. A relational database was chosen because the amount of input data is quite large (some applications running in the WebSphere Application Server contain over 4000 methods). A flat representation of this data could result in a very large input file with very poor random-access performance. Moreover, a relational database allows concurrent access to the data, which enables the use of a parallel implementation of FlowGSP.

For the use of the mining tool reported in this paper, each vertex in the EFG represents an instruction. The weight of each instruction represents the amount of total execution time spent on that instruction. The System Z operating system uses an event-based sampling mechanism: active events and the instruction under execution are recorded when the sample takes place. Instructions that occupy more cycles will be sampled more frequently, and the number of sampling hits or "ticks" is recorded on each instruction. The vertex weights are calculated by counting the number of sampling ticks on each instruction. The edge frequencies in the EFG are a measure of how many times each edge was taken during program execution. In the case of edges that lie between basic blocks, this value can be read directly from the control flow graph in the compiler logs. For intra-basic-block edges, edge weights are assigned the frequency of the basic block in which they reside. Both edge and basic block frequencies in the control flow graph are obtained by the compiler through counters inserted in the JVM interpreter.

Each vertex is assigned attributes based on the corresponding instruction's characteristics or events observed on the instruction in the hardware profile data. Examples of attributes include: opcode, whether an instruction-cache miss was observed, and whether the instruction caused a TLB miss.

In this application FlowGSP is mining for sequences of attributes that occur in subpaths of the EFG, but this search is based on edge frequency collected by the compiler. Precise path execution frequency cannot be derived from edge frequencies [2]. Therefore, the results produced by the mining tool are an approximation. The support reported for a sequence of attributes represents the maximal possible execution of that path that could have occurred based on the edge-frequency information available [13].

FlowGSP is a general flow mining algorithm that can be applied to any flow graph. For instance, each vertex of the EFG could represent any single-entry/single-exit region, including a Java bytecode, a basic block, or an entire method. The vertex weights and edge frequencies would have to be computed accordingly.

3.2 Operation of the Mining Algorithm

When the tool is run, it first recreates the control flow graph from the information taken from the compiler logs. Then, it inserts each instruction from the hardware profile into the correct basic block using the instruction's annotations. The tool constructs and mines only a single method at a time in order to match the level of granularity of the compiler; the Testarossa JIT compiles each individual method in isolation. As a consequence, FlowGSP does not discover patterns that cross method boundaries. However, this restriction is a design decision of the tool, not a limitation of the algorithm.

To mine graphs containing cycles, FlowGSP does not allow a vertex that is the start vertex of a current candidate sequence to start a new sequence. Therefore a vertex within a cycle can only start a sequence the first time that it is visited. FlowGSP can detect frequent subpaths that occur over cycles but avoids looping indefinitely because the lenght of a sequence is bounded by an specified constant. Jocksch provides a detailed description of FlowGSP [13].

FlowGSP is an iterative generate-and-test algorithm. Each iteration creates a set of candidate sequences from the survivors of the previous generation, and then calculates their supports and tests them against the provided thresholds (discussed in Section 3.3). Each iteration discovers longer sequences in the data. Execution terminates when either a specified number of iterations have completed or no new candidate sequences meet the minimum support thresholds.

3.3 Support Thresholds for Mining

FlowGSP accepts a number of parameters that can adjust the type and quantity of sequences that are discovered. FlowGSP takes a maximal support threshold and a differential support threshold. If the support of a sequence does not meet either of these thresholds, then the sequence is excluded from further mining. FlowGSP also accepts a maximum allowable gap size and window size. The maximum gap size determines how much space is allowed between each part of a sequence, and the maximum window size determines how many vertices to consider when searching for one part of a sequence.

Table 1 lists the parameters used in the experimental evaluation for both the SPECjvm2008 benchmarks and the DayTrader 2.0 benchmark in the WebSphere Application Server. The support values for the application server are lower than the corresponding values for the SPECjvm2008 benchmarks because the application server is orders of magnitude larger than any of the SPECjvm2008 benchmarks and has an extremely flat profile. The System z10 instructions are grouped into pairs for execution. Therefore, events that occur on one instruction of a pair can sometimes also appear on the other instruction. A window size of one is used to group paired instructions together so that more accurate patterns can be discovered.

Table 1. FlowGSP parameters used during this study

Parameter	crypto	compiler	sunflow	montecarlo	xml	serial	WebSphere
Maximal support	1%	7%	7%	7%	15%	7%	1%
Diff. support	1%	7%	7%	7%	15%	7%	1%
Gap size	1	0	0	0	0	0	0
Window size	1	0	1	1	1	1	1
Iterations	5	5	5	5	5	5	5

4 Opportunities Discovered

Before the development of the data-mining framework, significant development resources had been invested on the search for performance improvement opportunities in applications running in the WebSphere Application Server. This investment resulted in many observations about potential opportunities for performance improvement. Therefore, a first effort to test the FlowGSP algorithm, and to build confidence in the compiler development team about the efficacy of the framework, was a set of *acid tests* to find out if data mining could discover the opportunities for code improvement that were already known to the team. FlowGSP performed extremely well in these tests: it identified all the patterns that were listed by the developers. Examples of these patterns include:

1. A high correlation between data cache misses, TLB misses, and instruction cache misses. Consultation with hardware experts led to the observation that the page table is loaded through the instruction cache, which explained the

unusual correlation. After FlowGSP confirmed and quantified this correlation, large pages (1 MB instead of 4 KB) were used to reduce the number of TLB misses, resulting in a performance improvement of 3% on applications running in the WebSphere Application Server.

2. A high incidence for instruction-cache misses on entry to JIT code methods. These are cold cache misses for which effective prefetching is a challenge because of dynamic method dispatching. This observation led to additional efforts for inlining and code-cache organization by the compiler team, as well as to discussions on how to mitigate the cache misses in future hardware releases.
3. A high correlation between branch misprediction and instruction cache misses on indirect branches with a higher-than-expected occurrence of these events. A large volume of indirect branches overflows the branch-table buffers. The compiler team implemented code transformations to transform indirect branches into direct branches through versioning. Moreover, the hardware team was engaged to look for solutions to mitigate this issue in future hardware.

The discovery of these issues through manual inspection of performance-monitor data by analysts required orders of magnitude more time and effort than the analysis with the data-mining tool based on FlowGSP. Moreover, the manual approach is not easy to reproduce for a new data set and is less deterministic.

Once the development team was confident about the results produced by the mining tool, they started examining the output of the tool to find new opportunities for code improvement. The time spent in the EX instruction in array copies described in Section 2 is one such opportunity. The team discovered most of the new opportunities when applying the tool to profiling data collected from newer benchmarks, such as the SPECjvm2008. While extensive development effort has been dedicated to discover opportunities in applications running in the WebSphere Application Server over many years, these newer benchmarks have received relatively less attention from the compiler development team. Some of the new discoveries are listed here:

– Stores account for a majority of data cache directory misses [14] in all SPECjvm2008 benchmarks. This is unexpected because the load-to-store ratio in programs is typically on the order of 5:1. Moreover, intuition would indicate that a program writes to locations from which it has read recently. Discussions and analysis are still under way to better understand this ratio. The `serial` benchmark spends three times more time servicing directory lookups for stores than for loads. This benchmark is highly parallel in nature, which, on the surface, would lead developers to dismiss cache contention as a concern. The trends presented by FlowGPS, which would have remained unobserved under manual inspection, have been instrumental in forcing developers to reconsider cache contention as a possible concern.
– Address-generation interlock (AGI) accounts for more than 10% of the execution time in some benchmarks. In the System z architecture, an AGI occurs when the computation of the address required by a memory access

instruction has not completed by the time that the instruction needs it [22]. In some cases, such as in a small pointer-chasing loop, AGIs are difficult to avoid. The mining tool's finding is helping to focus analysis in this benchmark, and the team is planning a review of the instruction scheduling in the compiler to reduce the impact of AGIs on execution time.

- Branch misses account for 9% of execution time in `montecarlo`, a benchmark from the SPECjvm2008 suite. This is unexpected because the execution of this benchmark is dominated by a single method with several hot loops and the benchmark has very good instruction locality. This result led to further analysis that uncovered a limitation in the hardware's instruction fetch unit: the unit stops predicting branches when it cannot detect any previously taken branches within a given window further down the instruction stream. A consequence of this limitation is that when the compiler unrolls a loop, it needs to take into account the size of this window to ensure that the loop backedge is predicted correctly. The compiler team is currently re-examining the loop unrolling strategy to take into account the penalty for branch misses.

Experienced compiler developers will understand the value of the observations above to provide direction to a compiler development team. These observations focus on the z/architecture®, the Testarossa compiler, and are based on mining data from the SPECjvm2008 benchmark suite. A similar approach can be used to most combinations of compiler/architecture/application. Moreover, the mining tool can be used to discover opportunities that might be specific to important applications.

5 Experimental Data on the Usage of the Mining Tool

This section presents statistics on the usage of storage and on the time required to mine several benchmarks. The goal of this section is to provide developers with an idea of the resources needed to deploy such a tool, and to encourage researchers to come up with improvements on our tool design. Information reported here include size of input data, overall running time, number of sequences generated, and the format of the rules output by the tool.

5.1 Profiling and Storage Requirements

This experimental evaluation uses the DayTrader 2.0 benchmark in the WebSphere Application Server 7.0 and programs from the SPECjvm2008 benchmark suite. All programs are run using the IBM Testarossa JIT compiler. The WebSphere Application Server workload is DayTrader 2.0 and the server is run for 5 minutes once a stable throughput has been achieved. This delay is necessary to ensure that the Testarossa JIT has compiled the majority of the methods in the application server to native code. The throughput of the application server increases as methods are compiled to native code. Therefore, stabilization of throughput is an indication that the majority of the code being executed has been natively compiled. A hardware profile of 5 minutes of execution of the WebSphere Application Server results in roughly 37 MB of compressed data. The same run produces a 5.9 GB uncompressed, plain-text compiler log.[2] At the time of this writing, the Testarossa JIT does not have an option to output logs in a compressed format. Compressing the compiler-generated log using gzip reduces its size to around 700 MB.

Table 2. SPECjvm2008 benchmarks studied

Benchmark	# of Methods to Account for 50% of time	# of Methods Compilations	# Unique Methods Invoked
`compiler.compiler`	60	3659	7113
`compiler.sunflow`	55	4009	6946
`crypto.signverify`	2	1219	4654
`scimark.montecarlo`	1	703	4077
`serial`	8	2967	7645
`xml.transform`	25	5374	12430

The SPECjvm2008 benchmarks are profiled for a period of 4 minutes after a 1-minute warm-up time. Only a minute is required until the most of the benchmark code is being executed natively because the SPECjvm2008 benchmarks used in this study are significantly smaller than applications running in the WebSphere Application Server. The 6 SPECjvm2008 benchmarks examined in this study are listed in Table 2. The data in this table provides an indication of how flat the execution profile of each benchmark is by listing the number of methods that need to be examined to account for 50% of the execution time.[3] The table also show the total number of method compilations and the total number of unique methods that are invoked when the benchmark is executed. These benchmarks were chosen because they form a representative sample of the SPECjvm2008 benchmark suite and they produce both flat and non-flat profiles. Running these benchmarks for 5 minutes results in 7 MB of hardware profiling data per benchmark on average, and an average uncompressed compiler log with 1.4 GB of data. The benchmark with largest hardware profile is `compiler.compiler` which produces 12 MB of data. largest compiler log has 3.3 GB of data and is produced by `xml.transform`. The benchmark `scimark.montecarlo` produces the smallest hardware profile (385 KB) and the smallest compiler log (97 MB).

5.2 Time Needed to Mine

The execution time of the tool depends on the size of the log of the program being mined and the parameters passed to the tool. FlowGSP is multi-threaded

[2] The compiler option required to output control flow graph data also outputs a large volume of information that was extraneous to the mining process.

[3] This measurement is an approximation because the number of sampling ticks in the performance monitor that is used to determine the number of methods shown in the table.

in order to exploit the resources available in multi-core architectures. FlowGSP was run with 8 threads on a machine equipped with two AMD 2350 quad-core CPUs and 8 GB of memory. All runs were performed with the parameters outlined in Section 3.

Table 3. Running times of FlowGSP, in seconds

Program	**Execution Time**
Websphere App. Server (DayTrader 2.0)	6399
`compiler.compiler`	815
`compiler.sunflow`	539
`scimark.montecarlo`	2
`xml.transform`	557
`serial`	215
`crypto.signverify`	177

Table 3 lists the running time of FlowGSP on both the DayTrader 2.0 benchmark in the WebSphere Application Server and SPECjvm2008 benchmark profiles with execution time in seconds. The `xml.transform`, `compiler.sunflow`, `serial`, and `scimark.montecarlo` benchmarks terminated when no more candidates with support greater than the minimum threshold remained. `Xml.transform` and `scimark.montecarlo` terminated after three iterations, `compiler.sunflow` and `serial` after four iterations. `Montecarlo` has one small method which occupies almost 100% of total execution time. Therefore the time to mine this benchmark is significantly lower. The times reported in Table 3 indicate that the mining tool based in FlowGSP can be used on a daily basis in the development of a production compiler.

5.3 Sequences Reported by Mining

FlowGSP outputs frequent sequences in the following format:

$$S = \langle s_1, \ldots, s_k \rangle$$

where each $s_i \in s_1, \ldots, s_k$ is a set of attributes:

$$s_i = (\alpha_1, \ldots, \alpha_k)$$

Each sequence is accompanied by four values, which indicate the sequence's weight, frequency, maximal, and differential support. In this use of the data-mining tool the vertices of the EFG are instructions. Examples of attributes include the instruction type, occurrence of cache misses, pipeline interlock, branch missprediction, the type of bytecode the originated the isntruction, *etc.* Results are output to a plain-text file. In the experiments reported here, the DayTrader 2.0 benchmark in WebSphere produced 1286 sequences while the SPECjvm2008 data produced, on average, 64,000 sequences. The SPECjvm2008 benchmarks exhibited a very wide range in terms of the number of sequences generated. The most sequences were discovered in the `scimark.montecarlo` benchmark with roughly 291,000 sequences. On the other hand, the `xml.tranform` benchmark had the smallest number of sequences at around 1,900.

In general, support thresholds for the SPECjvm2008 benchmarks were set generously low because this is an initial exploration of the applications of data mining in the compiler development. These low thresholds ensure that no interesting sequences are overlooked. With experience the support threshold can be increased to allow only the most interesting sequences to be reported. It could be possible in future work to automate this process based on the number of surviving sequences.

We implemented an user interface to display the results of mining. This interface allows sequences to be sorted lexicographically or by any of the support metrics. A maximum and minimum support value can be specified to reduce the number of sequences displayed. The tool can also selectively display sequences based on whether they do or do not contain specific attributes. This filtering is particularly effective at reducing the number of sequences that must be examined by a compiler developer. For instance, the `serial` benchmark contained 16,518 sequences, but only 2,880 involved pipeline stalls due to AGI interlocks. Ranking these resulting sequences by maximal or differential support allows quick identification of the most interesting patterns.

The tool also allows the developer to specify one rule as the baseline against which all other sequences are compared. This feature allows for easy comparison of sequences with respect to the baseline sequence.

6 Related Work

This is potentially the first attempt to use data mining to discover patterns of execution that occur frequently in an application but yet do not necessarily occur inside loops. Work that is related to this approach include performance analysis tools, the use of performance counters in JVMs, and the search for code bloat.

Optiscope is an “optimization microscope” developed to aid compiler developers in understanding low-level differences in the code generated by a compiler executing different code transformations, or between code generated by two different compilers for the same program [15]. Optiscope automatically matches up code in two hardware profiles that originated from the same region of source code. Optiscope focuses on loops. In contrast, FlowGSP focuses on finding interesting patterns within a single hardware profile and aims to discover common patterns that occur throughout the profile.

The design of most existing performance analysis tools, such as the popular Intel VTune for Intel® chipsets [5], focuses on locating small regions of code that are frequently executed to concentrate development efforts on these regions. Chen *et al.* try to capture the most execution time with the least amount of code [4]. Similarly, Schneider *et al.* use hardware performance monitors to “direct the compiler to those parts of the program that deserve its attention” [17]. Contrary to earlier work, the premise of this paper is that in some applications these parts are scattered through the code and not concentrated in smaller regions. Hundt presents HP Caliper, a framework for developing performance analysis tools on the Intel Itanium® platform running HP-UX [10]. Similar to the approach presented here, Caliper integrates sampled hardware performance counters with compiler-generated dynamic instrumentation. Dynamic instrumentation involves changing program instructions on the fly to obtain more accurate program analysis. However, unlike our mining tool, HP Caliper does not attempt to mine the combined data for patterns.

Huck *et al.* present PerfExplorer, a parallel performance analysis tool [9]. PerfExplorer incorporates a number of automated data analysis techniques such as k-means and hierarchical clustering, coefficient of correlation analysis, and comparative analysis. PerfExplorer targets application developers seeking to understand bottlenecks in their code, not compiler developers. Also, PerfExplorer does not search for frequent sequences in the data.

Cuthbertson *et al.* incorporate performance counter information into a production JVM to improve program performance [6]. They use a custom library to retrieve instruction cache miss information on the Intel Itanium platform. This information is used to guide both object allocation and instruction scheduling in order to increase performance. They achieve an average performance increase of 2% on various Java benchmarks. Schneider *et al.* perform similar work using hardware counters on the Intel Itanium platform to guide object co-allocation [17]. However, these approaches can only improve the performance of *existing* code transformations whereas FlowGSP is aimed at discovering opportunities for new code transformations. Also, both approaches only look at a small fraction of all available program data. It is not clear how much increased overhead will result from increasing the amount of data being brought into the compiler.

Buytaert *et al.* use hardware-performance counters to both improve the accuracy and decrease the cost of hot method detection in a production JVM [3]. Their focus is purely on improving the efficiency and accuracy of the JVM and does not provide any insights into new opportunities for code transformations.

Xu *et al.* develop a method for profiling Java programs to identify areas of code bloat [23]. They evaluate the DaCapo benchmark suite, elements of the Java 1.5 standard library, and Eclipse 3.1, and are able to identify a number of specific opportunities to improve performance by decreasing bloat. Similarly, Novark *et al.* develop a tool called Hound to identify memory leaks and sources of bloat in C and C++ programs [16]. Hound was able to achieve a 14% performance increase in one of the studied benchmarks by identifying a single line of code that needed to be changed. While removing code bloat can significantly improve the performance of applications, it only addresses performance from the point of view of the application programmer. Proper use of code transformations by the compiler is equally as important in increasing program performance.

7 Conclusion

In compiler and computer-architecture development, as in Science in general, discovering the question to ask is often as difficult as finding the answer. Recent developments in hardware performance-monitoring tools, and in leaner techniques to insert profiling counters in generated code, have provided developers with an unprecedented amount of data to examine the run-time behavior of a program. The combination of these techniques amounts to a very powerful scope. The mining tool presented in this paper is a mechanism to help focus this powerful scope on patterns that happen frequently enough to warrant the attention of compiler or hardware developers. This paper describes the methodology and the tool used for this mining task. It also presents several examples of discoveries that were done using the tool. Then, it presents statistics on the amount of space and time that is required to use the tool to mine the data produced by enterprise software in a high-end hardware platform with a mature compiler infrastructure. This data indicates that this methodology can be used routinely for the development of production compilers.

Acknowledgments

We are very thankful to Jane Bartik and John Rankin from the IBM Poughkeepsie campus for sharing their invaluable insight into the z/Architecture. This work was supported by an IBM Centre for Advanced Studies fellowship and by grants from the Natural Science and Engineering Research Council (NSERC) of Canada through its Collaborative Research and Development program.

Trademarks

References

1. Agrawal, R., Srikant, R.: Mining sequential patterns. In: International Conference on Data Engineering (ICDE), March 1995, pp. 3–14 (1995)
2. Ball, T., Mataga, P., Sagiv, M.: Edge profiling versus path profiling: the showdown. In: Symposium on Principles of Programming Languages (POPL), San Diego, CA, USA, pp. 134–148 (1998)
3. Buytaert, D., Georges, A., Hind, M., Arnold, M., Eeckhout, L., De Bosschere, K.: Using HPM-sampling to drive dynamic compilation. In: Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Montreal, Quebec, Canada, pp. 553–568 (2007)
4. Chen, H., Hsu, W.-C., Lu, J., Yew, P.-C., Chen, D.-Y.: Dynamic trace selection using performance monitoring hardware sampling. In: Code Generation and Optimization (CGO), San Francisco, CA, USA, pp. 79–90 (2003)
5. Intel Corporation. Intel v-Tune performance analyzer, `http://software.intel.com/en-us/articles/intel-vtune-performance-analyzer-white-papers/`

6. Cuthbertson, J., Viswanathan, S., Bobrovsky, K., Astapchuk, A., Kaczmarek, E., Srinivasan, U.: A practical approach to hardware performance monitoring based dynamic optimizations in a production JVM. In: Code Generation and Optimization (CGO), Seattle, WA, USA, pp. 190–199 (2009)
7. Geronimo, A.: Apache daytrader benchmark sample (October 2009), `http://cwiki.apache.org/GMOxDOC20/daytrader.html`
8. Grcevski, N., Kielstra, A., Stoodley, K., Stoodley, M., Sundaresan, V.: Java just-in-time compiler and virtual machine improvements for server and middleware applications. In: Conference on Virtual Machine Research and Technology Symposium (VM), San Jose, CA, USA, pp. 12–12 (2004)
9. Huck, K.A., Malony, A.D.: PerfExplorer: A performance data mining framework for large-scale parallel computing. In: ACM/IEEE Conference on Supercomputing (SC), Seattle, WA, USA, p. 41 (2005)
10. Hundt, R.: HP Caliper: A framework for performance analysis tools. IEEE Concurrency 8(4), 64–71 (2000)
11. IBM Corporation. WebSphere Application Server (October 2009), `http://www-01.ibm.com/software/websphere/`
12. Jackson, K.M., Wisniewski, M.A., Schmidt, D., Hild, U., Heisig, S., Yeh, P.C., Gellerich, W.: Ibm system z10 performance improvements with software and hardware synergy. IBM J. of Res. and Development 53(1), Paper 16:1–8 (2009)
13. Jocksch, A.: Data mining flow graphs in a dynamic compiler. Master's thesis, University of Alberta, Edmonton, AB, Canada (October 2009)
14. Mak, P., Walters, C.R., Strait, G.E.: IBM system z10 processor cache subsystem microarchitecture. IBM J. of Res. and Development 53(1), Paper 2:1–12 (2009)
15. Moseley, T., Grunwald, D., Peri, R.V.: Optiscope: Performance accountability for optimizing compilers. In: Code Generation and Optimization (CGO), Seattle, WA, USA (2009)
16. Novark, G., Berger, E.D., Zorn, B.G.: Efficiently and precisely locating memory leaks and bloat. In: Conference on Programming Language Design and Implementation (PLDI), Dublin, Ireland, pp. 397–407 (2009)
17. Schneider, F.T., Payer, M., Gross, T.R.: Online optimizations driven by hardware performance monitoring. In: Conference on Programming Language Design and Implementation (PLDI), pp. 373–382 (2007)
18. Shiv, K., Chow, K., Wang, Y., Petrochenko, D.: SPECjvm2008 performance characterization. In: SPEC Workshop on Computer Performance Evaluation and Benchmarking, Austin, TX, USA, pp. 17–35 (2009)
19. Shum, C.-L.K., Busaba, F., Dao-Trong, S., Gerwig, G., Jacobi, C., Koehler, T., Pfeffer, E., Prasky, B.R., Rell, J.G., Tsai, A.: Design and microarchitecture of the IBM system z10 microprocessor. IBM J. of Res. and Development 53(1), Paper 1:1–12 (2009)
20. Standard Performance Evaluation Corporation. SPEC: The standard performance evaluation corporation, `http://www.spec.org/`
21. Sundaresan, V., Maier, D., Ramarao, P., Stoodley, M.: Experiences with multithreading and dynamic class loading in a java just-in-time compiler. In: Code Generation and Optimization (CGO), New York, NY, USA, pp. 87–97 (2006)
22. Webb, C.F.: IBM z10: The next generation mainframe microprocessor. IEEE Micro 28(2), 19–29 (2008)
23. Xu, G., Arnold, M., Mitchell, N., Rountev, A., Sevitsky, G.: Go with the flow: profiling copies to find runtime bloat. In: Conference on Programming Language Design and Implementation (PLDI), Dublin, Ireland, pp. 419–430 (2009)

Unrestricted Code Motion: A Program Representation and Transformation Algorithms Based on Future Values[⋆]

Shuhan Ding and Soner Önder

Department of Computer Science
Michigan Technological University
shding@mtu.edu, soner@mtu.edu

Abstract. We introduce the concept of *future values*. Using future values it is possible to represent programs in a new control-flow form such that on any control flow path the data-flow aspect of the computation is either traditional (i.e., definition of a value precedes its consumers), or reversed (i.e., consumers of a value precede its definition). The representation hence allows unrestricted code motion since ordering of instructions are not prohibited by the data dependencies. We present a new program representation called *Recursive Future Predicated Form* (RFPF) which implements the concept. RFPF subsumes general *if-conversion* and permits unrestricted code motion to the extent that the whole procedure can be reduced to a single block. We develop algorithms which enable instruction movement in acyclic as well as cyclic regions and give examples of various optimizations in RFPF form.

1 Introduction

Code motion is an essential tool for many compiler optimizations. By reordering instructions, a compiler can eliminate redundant computations [4,9,10], schedule instructions for faster execution [17], or enable early initiation of long latency operations, such as possible cache misses. In these optimizations, the range of code motion is limited by data and control dependencies [4,5]. Therefore, code-optimization algorithms which rely on code-motion have to make sure that control and data dependencies are not violated.

Ability to move code in a control-flow setting in an unrestricted manner would have several significant benefits. Obviously, having the necessary means to move instructions in an unrestricted manner while maintaining correct program semantics could enable the development of simpler algorithms for program optimization. More importantly however, when we permit code motion beyond the obvious limits, code-motion itself can become a very important tool for program analysis.

[⋆] This work is supported in part by a NSF CAREER award (CCR-0347592) to Soner Önder.

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 26–45, 2010.

In this paper, we first present the concept of *future values*. Future values allow a consumer instruction to be placed before the producer of its source operands. Using the concept, we develop a program representation which is referred to as *Recursive Future Predicated Form* (RFPF). RFPF is a new control-flow form such that on any control flow path the data-flow aspect of the computation is either traditional (i.e., definition of a value precedes its consumers), or reversed (i.e., consumers of a value precede its definition). When an instruction is to be hoisted above an instruction that defines its source operands, the representation updates the data-flow aspect to become reversed (i.e., *future*, meaning that the instruction will encounter the definition of its source operands in a future sequence of control-flow). If, on the other hand, the same instruction is propagated down, the representation will update the data-flow aspect to become traditional again. The representation hence allows unrestricted code motion since ordering of instructions is not prohibited by the data dependencies. Of course, for correct computation, the values still need to be produced before they can be consumed. However, with the aid of a future values based program representation, the actual time that this happens will appropriately be delayed.

RFPF is a representation built on the principle of *single-assignment* [3,7] and it subsumes general *if-conversion* [2]. In this respect, RFPF properly extends the SSA representation and covers the domain of legal transformations resulting from instruction movements. Possible transformations range from the starting SSA form where all data-flow is traditional, to a final reduction where the *entire procedure becomes a single block* through upward code motion, possibly with mixed (i.e., traditional and future) data-flow. We refer to a procedure which is reduced to a single block through code motion to be in *complete RFPF*. *Complete RFPF* expresses the program semantics without using control-flow edges except sequencing. During the upward motion of instructions, valuable information is collected and as it is shown later in the paper, this information can be used to perform several sophisticated optimizations such as *Partial Redundancy Elimination* (PRE). Such optimizations typically require program analysis followed by code motion and/or code restructuring [12,9,4].

Our contributions in this paper are as follows: (1) We introduce the novel concept of *future values* which permits a consumer instruction to be encountered before the producer of its source operand(s) in a control-flow setting; (2) Using future values, we introduce the concept of *future predicates* which permits instruction hoisting above the controlling instructions by specifying *future control flow*; (3) We introduce the concept of *instruction-level recursion*. This concept allows the loops to be represented as straight-line code and analyzed with ease. Combination of future predicates and instruction-level recursion enables predication of backward branches; (4) Using the concepts of future values, future predicates and instruction-level recursion, we develop a unified representation (RFPF) which is control-flow based, yet instructions can freely be reordered in this representation by simply comparing the instruction's predicate, source and destination variables to the neighboring instruction; (5) We illustrate that unrestricted code motion itself can be used to analyze programs for optimization opportunities.

We present a PRE example in which redundancy cannot be eliminated using code motion alone and restructuring is necessary, yet both the discovery and the optimization of the opportunity can be performed with ease; (6) We present algorithms to convert conventional programs into the RFPF. These algorithms are low in complexity and with the exception of identification of loop headers and the nesting of the loops in the program, they do not need additional external information to be represented. Instead, these algorithms operate by propagating instructions and predicates and use only the local information available at the vicinity of moved instructions; (7) We illustrate that for any graph with mixed-mode data-flow, there is a path through instruction reordering and control flow node generation to convert the future data-flow in the representation back to a traditional SSA graph, or generate code directly from the representation.

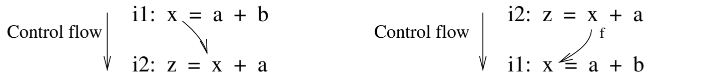
In the remainder of the paper, in Section 2, we first present the concept of future values. Section 3 through Section 6 illustrate a process through which instructions can be hoisted to convert a program into RFPF while collecting the data and control dependencies necessary to perform optimizations. For this purpose, we first illustrate how the concept can be used for instruction movement in an acyclic region in Section 3. This set of algorithms can be utilized by existing optimization algorithms that need code motion by incorporating the concept of future-values into them. Code motion in cyclic regions requires conversion of loops into instruction-level recursion. We introduce the concept of instruction-level recursion in Section 4. This section presents the idea of recursive predicates and illustrates how backward branches can be predicated. Next, in Section 5 we give an algorithm for computing recursive predicates. Combination of code motion in acyclic and cyclic regions enables the development of an algorithm that generates procedures in *complete RFPF* from a given SSA program using a series of topological traversals of the graph and instruction hoisting. Since reordering of instructions has to deal with explicit dependencies, memory dependencies pose specific challenges. We discuss the handling of code motion involving memory dependencies in Section 6. Section 7 gives examples of optimizations using the RFPF form. We discuss the conversion back into CFG in Section 8. Finally we describe the related work in Section 9 and summarize the paper in Section 10.

2 The Concept of Future Values

Any instruction ordering must respect the true data dependencies as well as the control dependencies. As a result, an instruction cannot normally be hoisted beyond an instruction which defines the hoisted instruction's source operand(s). When such a hoisting is permitted, a *future dependency* results:

Definition 1. *When instructions I and J are* true dependent *on each other and the instruction order is reversed, the true dependency becomes a* future dependency *and is marked on the source operand with the subscript* f.

Consider the statements shown in Figure 1(a). In this example, the control first encounters instruction `i1` which computes the value `x`, and then encounters the



(a) True dependence (b) Future (reversed) dependence

i_1: if($a < b$)
i_2: x = x +1

(c) Traditional control-flow

i_1: P=($a < b$)
i_2: $[P]x = x + 1$

(d) If-conversion

i_2: $[P_f]x = x + 1$
i_1: P=($a < b$)

(e) Future control-dependence

Fig. 1. The concept of Future data and control dependences

instruction `i2` which consumes the value. In Figure 1(b), the instruction `i2` has been hoisted above `i1`, and its source operand `x` has been marked to be a *future value* using the subscript `f`. If the machine buffers any instructions whose operands are future values alongside with any operand values which are not future until the producer instruction is encountered, the instructions can be executed with proper data flow between them even though the order at which the control has discovered them is reversed. Similarly, we can represent control dependencies in future form as well. Consider Figure 1(c). In this example, i_2 is control dependent on i_1. In Figure 1(d) predicate P is used to guard i_2, which represents the same control dependence. When the order of i_1 and i_2 is reversed(Figure 1(e)), predicate P becomes a future value and thus the original control dependence becomes future control dependence.

The combination of future data and control dependencies and single-assignment semantics permit unrestricted code motion. In the rest of the paper, single-assignment semantics is assumed and all the transformations maintain the single-assignment semantics. We first discuss code motion using future values in acyclic regions involving control dependencies.

3 Code Motion in Acyclic Code

For an acyclic control-flow graph $G =< s, N, E >$ such that, s is the start node, N is the set of nodes and E is the set of edges, instruction hoisting involves one of three possible cases. These are: (1) movement that does not involve control dependencies (i.e., straight-line code), (2) *splitting* (i.e., parallel move to predecessor basic blocks), and (3) *merging* (i.e., parallel move to a predecessor block that dominates the source blocks). Note that movement of a ϕ-node is a special case and normally would destroy the single-assignment property. We examine each of these cases below:

Case 1 (Basic block code motion). Consider instructions I and J. Instruction J follows instruction I in program order. If I and J are true dependent, hoisting J above I converts the true dependency to a future dependency. Alternatively, if the instructions are future dependent on each other, hoisting J above I converts the future dependency to a true dependency (Figure 1(a) and (b)).

When code motion involves control dependencies, the instruction propagation is carried out using instruction predication, instruction cloning and instruction merging. An instruction is cloned when the instruction is moved from a control independent block to a control dependent block. Cloned copies then propagate along the code motion direction into different control dependent blocks. When cloned copies of instructions arrive at the same basic block they can be merged.

Case 2 (Splitting code motion). Consider instruction I that is to be hoisted above the block that contains the instruction. For each incoming edge e_i a new block is inserted, a copy of the instruction is placed in these blocks and a ϕ-node is left in the position of the moved instruction (Figure 2).

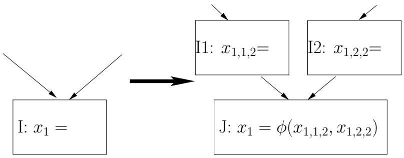


Fig. 2. Splitting code motion

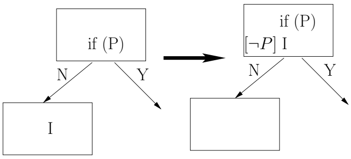


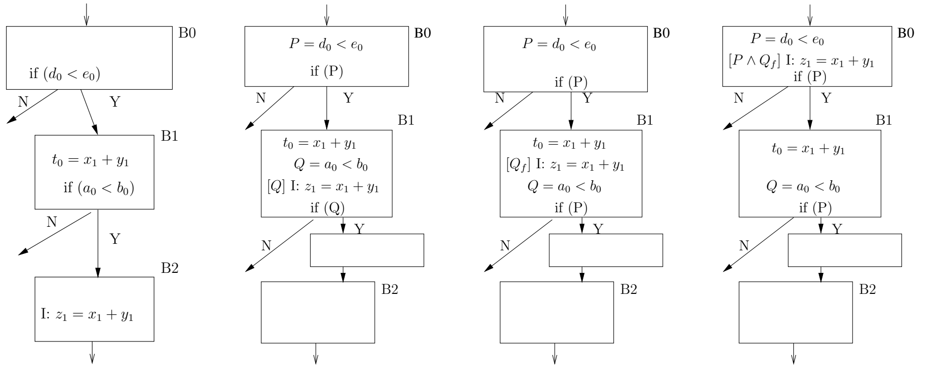
Fig. 3. Merging code motion

Note that in Figure 2, when generated copies I1 and I2 are merged back into a single instruction, the inserted ϕ-node can safely be deleted and the new instruction can be renamed back to x_1. The two new names created during the process, namely, $x_{1,1,2}$ and $x_{1,2,2}$ are eliminated as part of the merging process. In order to facilitate easy merging of clones, we adopt the naming convention $v_{i,j,k}$ where v_i is an SSA name, j is the copy version number and k is the total number of copies. Generated copies can be merged when they arrive at the immediate dominator of the origin block, and in case of reduction to a single block, all copies can be merged. We discuss these aspects of merging later in Section 3.3.

Case 3 (Merging code motion). Consider instruction I that is to be hoisted into a block where the source block is control dependent on the destination block. The instruction I is converted to a predicated instruction labeled with the controlling predicate of the edge (Figure 3).

3.1 Future Predicated Form

When a predicated instruction is hoisted above the instruction which defines its predicate, the predicate guarding the instruction becomes *future* as the predicate is also a value and the data dependence must be updated properly. Figure 4 shows a control dependent case. Instruction I is control dependent on condition $a_0 < b_0$. When the instruction I is moved from $B2$ to $B1$, it becomes predicated and is guarded by Q (Figure 4(b)). In the next step, the instruction is hoisted above the definition of Q and its predicate Q becomes future (i.e., Q_f) (Figure 4(c)).



(a) before code motion

(b) after code motion

(c) future predicate

(d) nested predicate

Fig. 4. Code motion across control dependent regions

When a predicated instruction is hoisted further, it may cross additional control dependent regions and will acquire additional predicates. Consider Figure 4(c). Since the target instruction is already guarded by the predicate Q_f, when it moves across the branch defined by P, it becomes guarded by a nested predicate (Figure 4(d)). In terms of control flow, it means that predicate P must appear, and it will appear before Q. Similarly, if P is true, then Q must also appear since if the flow takes the true path of P the predicate Q will eventually be encountered. In other words, the conjunction operator has the short-circuit property and it is evaluated from left to right. Semantically, a nested predicate which involves future predicates is quite interesting as it defines *possible* control flow.

3.2 Elimination of ϕ-Nodes

RFPF transformations aim to generate a single block representing a given procedure. The algorithms developed for this purpose hoist instructions until all the blocks, except the start node are empty. Proper maintenance of the program semantics during this process requires the graph to be in single-assignment form. On the other hand, movement of ϕ-nodes as regular instructions is not possible and the elimination of ϕ-nodes result in the destruction of the single-assignment property. For example, elimination of the ϕ-node $x_3 = \phi(x_1, x_2)$ involves insertion of copy operations $x_3 = x_1$ and $x_3 = x_2$ across each incoming edge in that order. Such elimination creates two definitions of x_3 and the resulting graph is no longer in single-assignment form. Our solution is to delay the elimination of ϕ-nodes until the two definitions can be merged, at which time a *gating function* [13] can be used if necessary:

Definition 2. *We define the gating function* $\psi_p(a1, a2)$ *as an executable function which returns the input* $a1$ *if the predicate* p *is true and* $a2$ *otherwise.*

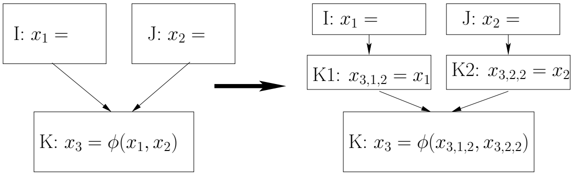


Fig. 5. ϕ-node elimination

Note that during merging, cloned copies already bring in the necessary information for computing the controlling predicate for the gating function. The merging process is enabled by transforming the ϕ-node in a manner similar to the *splitting* case described above:

Case 4 (ϕ-node elimination). Consider the elimination of the ϕ-node x_3=$\phi(x_1, x_2)$ (Figure 5). ϕ-node elimination can be carried out by placing copy operations $x_{3,1,2} = x_1$ and $x_{3,2,2} = x_2$ across each incoming edge in that order and updating the ϕ-node with the new definitions to become $x_3 = \phi(x_{3,1,2}, x_{3,2,2})$.

Merging of the instructions $x_{3,1,2} = x_1$ and $x_{3,2,2} = x_2$ requires the insertion of a *gating function* since the right-hand sides are different. Once the instructions are merged, the ϕ-node can be eliminated. It is important to observe that until the merging takes place and the deletion of the ϕ-node, instructions which use the ϕ-node destination x_3 can be freely hoisted by converting their dependencies to *future dependencies*.

3.3 Merging of Instructions

In general, upward instruction movement will expose all paths resulting in many copies of the same instruction guarded by different predicates. This is a desired property for optimizations that examine alternative paths such as PRE and related optimizations since partial redundancy needs to be exposed before it can be optimized. We illustrate an example of PRE optimization in Section 7. On the other hand, the code explosion that results from the movement must be controlled. RFPF representation allows copies of instructions with different predicates to be merged. Merging can be carried out between copies of instructions which result from a splitting move, as well as those created by ϕ-node elimination. As previously indicated, merging of two instructions with the same derivative destination (i.e., such as those which result from ϕ-node elimination) requires the introduction of the *gating function* ψ into the representation, whereas merging of the two copies of the same instruction can be conducted without the use of a gating function. When the merged instructions are the only copies, the resulting instruction can be renamed back to the ϕ destination. Otherwise, a new name is created for the resulting instruction, which will be merged with other copies later during the instruction propagation.

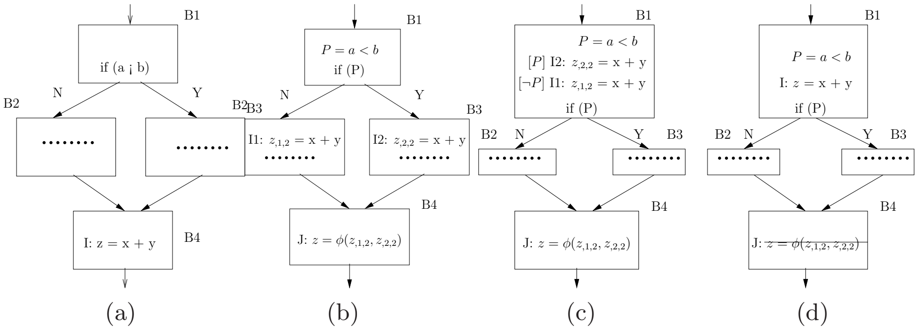
Definition 3. *Two instructions $\gamma : x_{i,m,k} \leftarrow e1$ and $\delta : x_{i,n,k} \leftarrow e2$, where γ and δ are predicate expressions, represent the single instruction $\gamma \vee \delta : x_{i,(m,n),k} \leftarrow e1$ if e1 and e2 are identical.*

Definition 4. *Two instructions $\gamma : x_{i,m,k} \leftarrow e1$ and $\delta : x_{i,n,k} \leftarrow e2$, where γ and δ are predicate expressions represent the single instruction $\gamma \vee \delta : x_{i,(m,n),k} \leftarrow \psi_P(e1, e2)$ if e1 and e2 are not identical. The predicate expression P is the first predicate expression in γ and δ such that P controls γ and $\neg P$ controls δ.*

Definition 5. *Instruction $\gamma : x_{i,(p,\ldots,q),k} \leftarrow e$ can be renamed back to $\gamma : x_i \leftarrow e$ if $(p, \ldots, q)$ contains a total of k version numbers.*

Theorem 1. *Copy instructions generated from a given instruction I during upward propagation are merged at the immediate dominator of the source node of I, since all generated copies will eventually arrive at the immediate dominator of the source block.*

Proof. Let node A be the immediate dominator of the source node I has originated from in the forward CFG. Assume there's one copy instruction I' which does not pass through A during the whole propagation. For this to happen, there must be a path p, which from the start node reaches I' and then reaches the source node of I. The fact that p does not pass through node A conflicts the assumption that A is the immediate dominator node of I.



(a) (b) (c) (d)

Fig. 6. Instruction propagation

Let us now see through an example how the instruction merging effectively eliminates unnecessary code duplication. Consider the CFG fragment shown in Figure 6(a). Suppose that instruction I needs to be moved to block $B1$. Further note that instruction I is control independent of the block B1. We first insert the branch condition $P = a < b$ in block $B1$. Moving of I is accomplished by applying the *splitting* transformation, followed by progression of $I1$ and $I2$ into blocks B2 and B3 respectively and the deletion of temporary nodes inserted during the movement (Figure 6(b)). Next, the instructions $I1$ and $I2$ are propagated using a *merge* move which predicates them with $\neg P$ and P respectively and places them in block $B1$ (Figure 6(c)). At this point, using Definition 3, the two instructions can be reduced to a single instruction I without a predicate (Figure 6(d)) and the ϕ-node can be deleted. Note that the merging of the instructions and the deletion of ϕ node must be carried out at the same step to maintain single-assignment property.

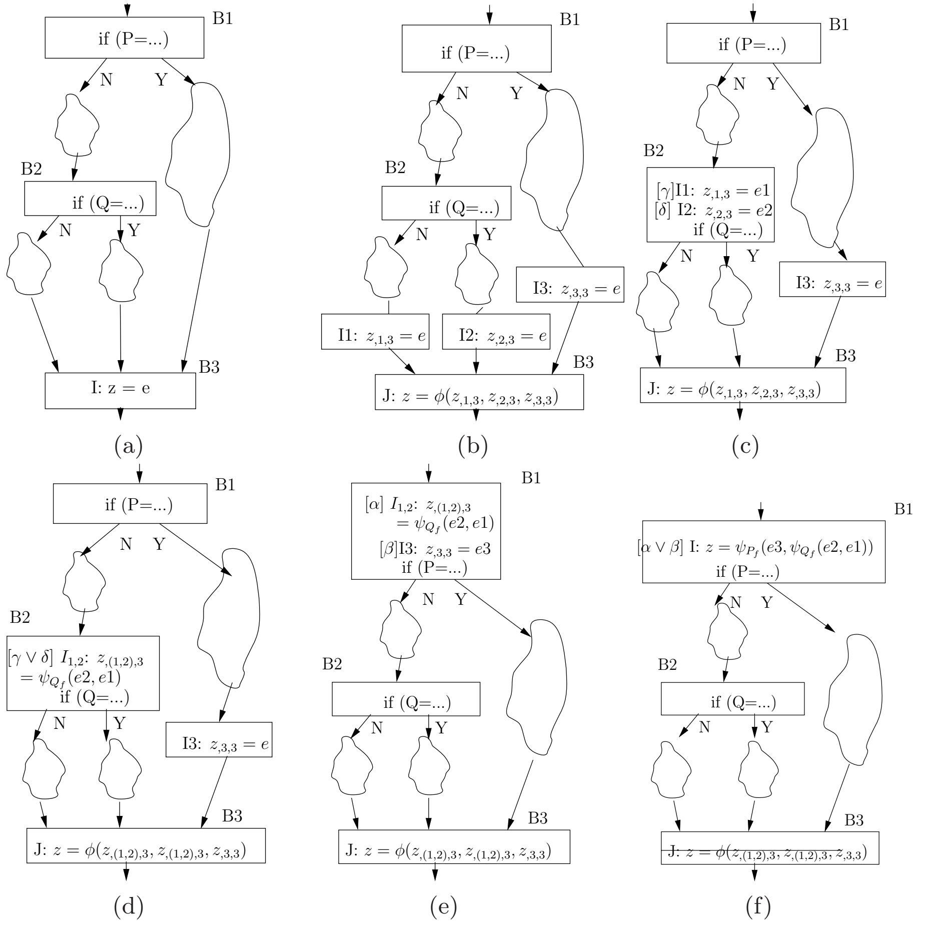

B1
if (P=...)
N
Y
B2
if (Q=...)
B3
I: z = e
(a)
I1: $z_{,1,3} = e$
I2: $z_{,2,3} = e$
I3: $z_{,3,3} = e$
J: $z = \phi(z_{,1,3}, z_{,2,3}, z_{,3,3})$
(b)
$[\gamma]$I1: $z_{,1,3} = e1$
$[\delta]$ I2: $z_{,2,3} = e2$
(c)
$[\gamma \vee \delta]$ $I_{1,2}$: $z_{,(1,2),3} = \psi_{Q_f}(e2, e1)$
J: $z = \phi(z_{,(1,2),3}, z_{,(1,2),3}, z_{,3,3})$
(d)
$[\alpha]$ $I_{1,2}$: $z_{,(1,2),3} = \psi_{Q_f}(e2, e1)$
$[\beta]$I3: $z_{,3,3} = e3$
(e)
$[\alpha \vee \beta]$ I: $z = \psi_{P_f}(e3, \psi_{Q_f}(e2, e1))$
~~J: $z = \phi(z_{,(1,2),3}, z_{,(1,2),3}, z_{,3,3})$~~
(f)

Fig. 7. Instruction merging

A detailed example which shows how the adopted naming convention facilitates instruction merging is illustrated in Figure 7(a). In this example regions represented as *clouds* are arbitrary control and dataflow regions an instruction has to pass through and cloud regions have no incoming or outgoing edges except for the explicitly indicated ones. Instruction I, which computes e is moved across block $B3$ by applying a *splitting* transformation (Figure 7(b)). Next, two of the total three copy instructions, namely, $I1$ and $I2$ converge in block $B2$ and during propagation may acquire different expressions, namely, $e1$ and $e2$. These two instructions are merged into $I_{1,2}$ using Definition 4 (Figure 7(c)(d)).

Note that future predicate Q_f is used in the gating function for choosing between $e1$ and $e2$. At this point, checking the name of destination $z_{,(1,2),3}$, indicates that there are unmerged copies. Further instruction propagation results in the merging of $I_{1,2}$ and $I3$ in block $B1$. Applying Definition 4 and 5, all the copy instructions are reduced into a single instruction I, which is represented through a nested gating function. At this point the ϕ-node can be deleted. The final result is shown in Figure 7(f).

4 Instruction-Level Recursion

In a reducible control-flow graph, a loop region is a strongly connected region where the loop header forms the upward propagation boundary. Therefore moving instructions across the loop header requires a new approach. This approach is to convert every instruction within the loop region to an equivalent instruction that can iterate in parallel with the loop execution independently. We define an instruction that *schedules* its next iteration, a *recursive instruction.*

Conceptually, a recursive instruction appears as a function call that is spawned at the point the control visits the instruction. The instruction executes within this envelope and checks a predicate to see if it should execute in the next iteration. If the predicate is true, a recursive call is performed. Otherwise the function returns the last value it had computed. In this way, as long as the predicate which controls the loop iteration is known, any loop instruction can iterate itself and hence it can be separated from the loop structure (or pushed out of the loop region). In other words, an instruction that is hoisted above the loop header becomes a recursive instruction controlled by a special predicate called the *Recursive Predicate*:

Definition 6. *Recursive Predicate: In a loop L that has a single loop header H and a single backedge e, the predicate expression which allows control flow to reach e from H without going through e is Recursive Predicate for L.*

For loops with multiple edges we can use the disjunction of the recursive predicates computed for each edge. This follows from the observation that we can insert an empty block such that all the backedges are connected to this block and removed from the loop header and a single exit from this block becomes the single backedge for the graph. Since the controlling predicate of the newly inserted block's outgoing edge is the disjunction of the controlling predicates of all the incoming edges, such graphs can be reduced into a single backedge case described above.

Since the instruction returns only its last value, we can establish proper data dependencies with instructions outside the loop region. Note that, a recursive instruction should also include a predicate to implement the control flow within the loop body:

Definition 7. Recursive Predicated Instruction: $x_i = (R)[P]\{I : x_{i_j} = ...\}$*, where I is the instruction,* x_i *is the SSA name of the instruction's destination, j is the loop nest level, P is the predicate guarding I obtained through acyclic instruction propagation into the loop header and R is the recursive predicate the instruction iterates on.*

Note that the recursive instruction renames the destination of the original instruction by appending the loop nest level, and the function returns the original name. In Section 5.2 we revisit this renaming. From an executable semantics perspective, a recursive predicate must need to know the number of readers it is being waited by and should generate a new value after all the readers have read it.

5 Code Motion in Cyclic Code and Recursive Future Predicated Form

We follow a hierarchical approach to perform code motion in cyclic code. For this purpose, starting with the inner-most loops, we convert the loops into groups of recursive instructions, propagate them to the loop header of the immediately enclosing loop and apply the procedure repeatedly until all cyclic code is converted into recursive instruction form, eventually leading to a single block for the procedure.

5.1 ϕ-Nodes in Loop Header

Although any code movement within a given loop can be carried out using the acyclic code motion techniques, the ϕ-nodes in the loop header cannot be eliminated using the techniques developed for acyclic regions. Instead, we adopt the executable function from [13]:

Definition 8. *We define the gating function* $\mu(a^{init}, a^{iter})$ *as an executable function.* a^{init} *represents external definitions that can reach the loop header prior to the first iteration.* a^{iter} *represents internal definitions that can reach loop header from within the loop following an iteration.* a^{init} *is returned when control reaches loop header from outside of the loop.* a^{iter} *is returned in all subsequent iterations.*

5.2 Conversion of Loops into Instruction-Level Recursion

The conversion is achieved by following the following steps:

1. Identify a single-entry, multi-exit region where the entire region is dominated by an inner-most loop header.
2. Propagate all instructions except branches to the loop header using acyclic code motion discussed before.
3. Calculate the controlling predicates for the exit edges and calculate the *Recursive Predicate* using *Algorithm 1* shown in Figure 8.

```
Algorithm 1
for each back-edge and exit edge $e$ do

begin
  let $b$ be the block which $e$ originates from
  let $I$ be any instruction originally in block $b$
  let $\alpha(I)$ be the predicate expression guarding $I$
  if block $b$ has a branch on predicate P then
    if $e$ is on the true path of the branch then
      $p(e) \leftarrow \alpha(I) \wedge P$ if e is a back-edge
      $q(e) \leftarrow \alpha(I) \wedge P$ if e is an exit edge
    else
      $p(e) \leftarrow \alpha(I) \wedge \neg P$ if e is a back-edge
      $q(e) \leftarrow \alpha(I) \wedge \neg P$ if e is an exit edge
    end
  else /* e is fall through backedge */
    $p(e) \leftarrow \alpha(I)$ if e is a back-edge
  end
  if $e$ is a back edge then
    $RP \leftarrow p(e)$
end
```

Fig. 8. Algorithm 1: Compute RecursivePredicate and ExitPredicate

4. Pick an unused SSA name for the *RecursivePredicate*.
5. Convert ϕ-nodes to gating function μ.
6. Insert $(RP)[T]RP = ...$ at the very beginning of loop header where RP is the SSA name picked in the previous step and it is assigned to the computed *RecursivePredicate* by converting all the predicate variables in the computed predicate to future form.
7. Convert every instruction in the header to recursive form using RP and delete the back edges and branches. The conversion involves renaming all instructions which are in the loop body such that each SSA name that is defined in the block is appended the loop nest level, starting with zero at the inner-loop and incrementing. This renaming will update any uses which are loop carried to the new name while keeping names which are defined outside the loop unchanged.

Once the above process is completed, an inner-most loop has been converted to a sequential code. We apply the above process until the entire procedure is converted into a single block.

Theorem 2. *The predicate expression controlling the backedge e can be computed correctly using Algorithm 1.*

Proof. Figure 9 that contains an arbitrary innermost loop is used to demonstrate the proof. $B1$ is the loop header, $e1$ is a backedge originating from block $B3$ which contains instruction J. Assume a trivial instruction K is inserted in $e1$ as shown in Figure 9(b). The predicate expression controlling K, namely γ is the same as the one controlling $e1$. γ is computed by propagating instruction K to $B1$. For that purpose, K is first moved into block $B3$. K becomes $\beta : K$ in $B3$ where three cases may happen:

case1: $\beta = P$ if $e1$ is the taken edge of $B3$,
case2: $\beta = \neg P$ if $e1$ is the fall through edge of $B3$,
case3: $\beta = \mathit{true}$ which means K is not guarded by any predicate if $B3$ is ended with an unconditional jump.

Propagate $\beta : K$ and instruction J to $B1$. Since $\beta : K$ and J propagate from the same block, the predicates guarding these two instructions are the same when they reach $B1$. Assume J becomes $\alpha : J$ in $B1$, then K becomes $\alpha : \{\beta : K\}$. Combining nested predication yields $\gamma = \alpha \wedge \beta$.

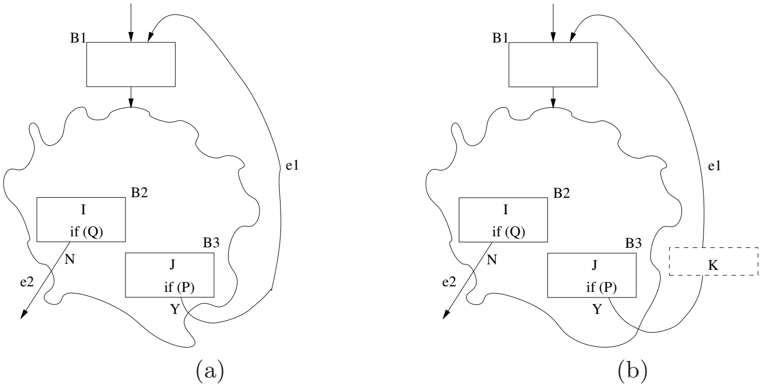
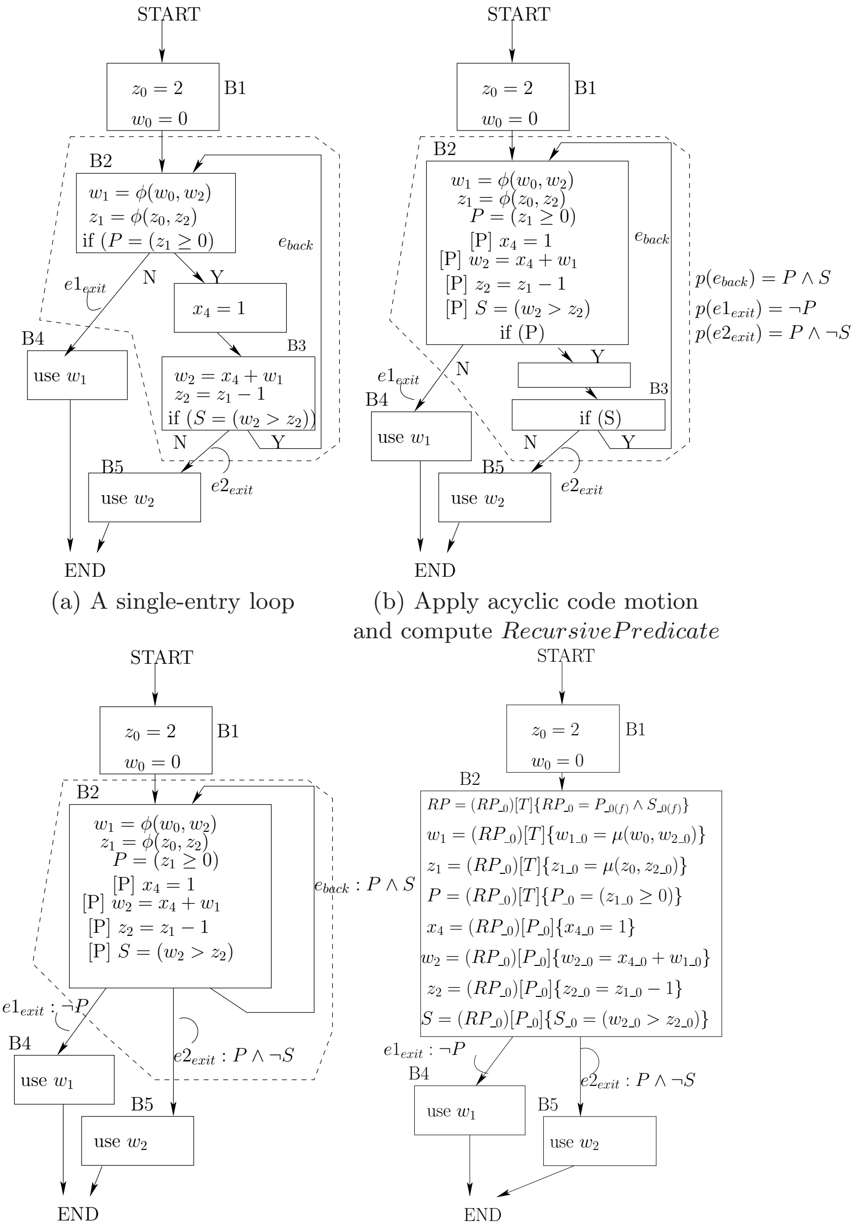


Fig. 9. Theorem 2

Note that although K may be split into multiple copies during the propagation, the last copy instruction is merged and hence the resulting instruction is renamed back to K in the loop header $B1$ if it is not merged before reaching $B1$.

Figure 10(a) is an example that shows the steps of transforming cyclic code. The region cut out is a loop region with a single loop header $B2$. Following the algorithm, we first propagate every instruction inside the loop into the loop header(Figure 10(b)). During the instruction propagation, the necessary predicate information to compute the *RecursivePredicate* and controlling predicates for the exit edges are collected naturally, shown on the right side of Figure 10(b). Next, everything in the loop region except the loop header and the back edge is deleted.(Figure 10(c)). The result of the conversion is shown in Figure 10(d).



(a) A single-entry loop

(b) Apply acyclic code motion and compute *RecursivePredicate*

(c) Eliminate loop except loop header

(d) Convert to recursive form

Fig. 10. Program 1: Conversion of a cyclic program into RFPF

6 Code Motion Involving Memory Dependencies and Function Calls

Memory dependencies pose significant challenges in code motion. There are many cases a compile time analysis of memory references does not yield precise answers. Our solution is to assume dependence and enforce the original memory ordering in the program through predication. Since a series of consecutive load operations without intervening stores have no dependence on each other, RFPF allows these loads to be executed in any order once the dependence of the first load in the series is satisfied. We define the memory operations as: `MEM, @P` where MEM represents a Load/Store operation and P is a predicate whose value is set to 1 when the memory operation MEM gets executed. Any memory operation that has a dependence with MEM will be guarded by P as a predicated operation. In this way, the dependence among memory operations are converted into data dependencies explicitly. Once the memory operations are converted in this manner, they can be moved like any other instruction. Because of the predication, if a memory operation is hoisted above another which defines its controlling predicate, the controlling predicate becomes a future value (Figure 11).

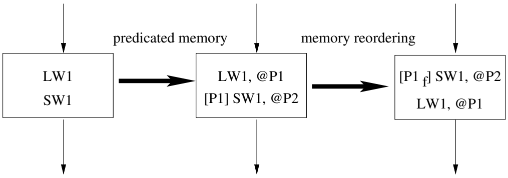


Fig. 11. Predicated memory and reordered memory

Our algorithm to rewrite memory operations is based on Cytron et al's SSA construction algorithm [7]. Since all the load/store operations can be treated as assignments to the same variable, Cytron et al's algorithm can be modified to accomplish the rewriting. Due to lack of space, we are unable to include the algorithm.

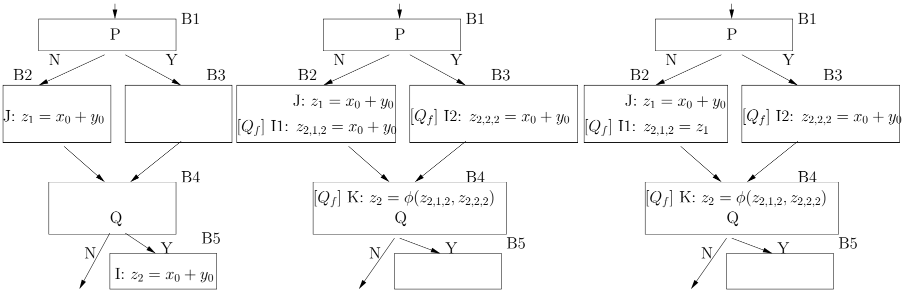
We employ a similar algorithm for handling function calls. Because of their side effects such as input/output, function calls may not be reordered without a proper analysis of the functions referenced. Therefore, we introduce a single predicate for each call instruction which is set when the call is executed. A single ϕ node is needed at merge steps to enforce the function call order on any path.

7 Optimizations Using RFPF

Many optimizations can be carried out on the *complete RFPF* and as well as during the transformation process. One of the advantages of RFPF is its ability to perform traditional optimizations while keeping the graph in single-assignment form with minimal book keeping. We show two examples of optimizations, one which can be employed during the transformation and another after the graph is converted into full RPFP.

Case study 1. *PRE during the transformation:*

Consider Figure 12(a). There's a redundant computation of x_0+y_0 along the path (B2 B4 B5). Most PRE algorithms cannot capture this redundancy because node B4 destroys the available information for $x_0 + y_0$. On the other hand, instruction propagation and RFPF cover the case. Observe that during the instruction propagation, one of the clones, namely, $(I1)$ reaches node B2(Figure 12(b)). By applying *Value numbering* [1] in the basic block, $x_0 + y_0$ in I1 is subsumed by z_1(Figure 12(c)).


B1
P
N
Y
B2
B3
J: $z_1 = x_0 + y_0$
B4
Q
N
Y
B5
I: $z_2 = x_0 + y_0$
B1
P
N
Y
B2
B3
J: $z_1 = x_0 + y_0$
$[Q_f]$ I1: $z_{2,1,2} = x_0 + y_0$
$[Q_f]$ I2: $z_{2,2,2} = x_0 + y_0$
B4
$[Q_f]$ K: $z_2 = \phi(z_{2,1,2}, z_{2,2,2})$
Q
N
Y
B5
B1
P
N
Y
B2
B3
J: $z_1 = x_0 + y_0$
$[Q_f]$ I1: $z_{2,1,2} = z_1$
$[Q_f]$ I2: $z_{2,2,2} = x_0 + y_0$
B4
$[Q_f]$ K: $z_2 = \phi(z_{2,1,2}, z_{2,2,2})$
Q
N
Y
B5

(a) A PRE example (b) Code motion (c) *Value numbering*

Fig. 12. Partial redundancy elimination during the code motion

By further propagating and merging, instruction I1 and I2 are merged in B1 with the addition of the gating function ψ (Figure 13(a)) yielding the complete RFPF:

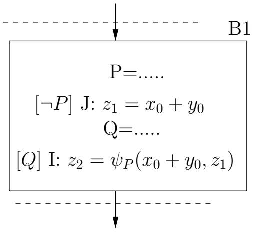
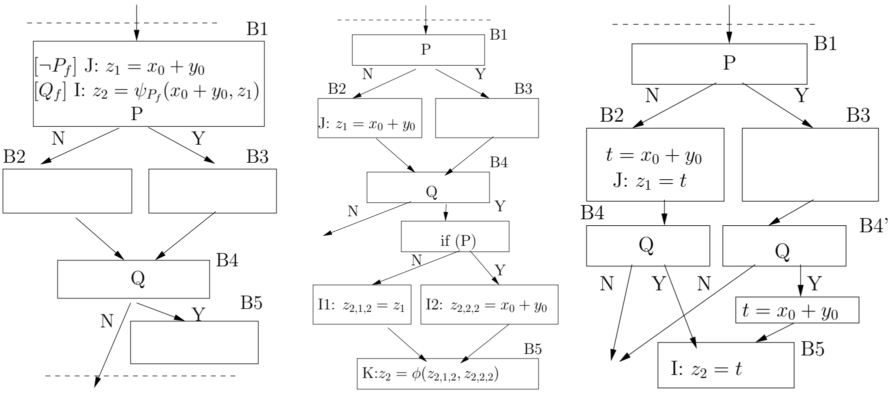

B1
P=.....
$[\neg P]$ J: $z_1 = x_0 + y_0$
Q=.....
$[Q]$ I: $z_2 = \psi_P(x_0 + y_0, z_1)$

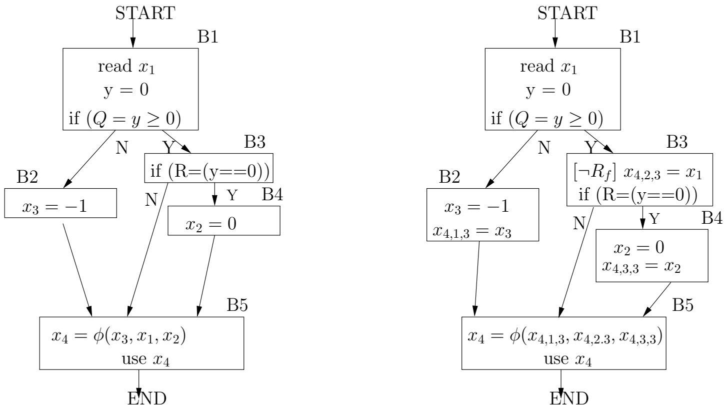
Figure 13(b) gives the result of transforming RFPF back into SSA using the algorithm in Section 8. This graph is functionally equivalent to Figure 13(c), which shows the result by using the PRE algorithm of Bodik et al. [4]. This algorithm separates the expression available path from the unavailable path by node cloning which eliminates all redundancies. As it can be seen, RFPF can perform PRE and keep the resulting representation in the SSA form.

The dependency elimination in our example is not a coincidence. By splitting instructions into copies, we naturally split the dataflow information available



(a) Instruction merging (b) RFPF to CFG (c) Bodik et al., [4]

Fig. 13. Merging and Converting Back to CFG



(a) A CP Example (b) Transform to RFPF

read x_1
$y = 0$
$Q = (y \geq 0)$
$Q : R = (y == 0)$
$Q \wedge R : x_2 = 0$
$\neg Q : x_3 = -1$
$x_4 = \psi_Q(\psi_R(x_2, x_1), x_3)$
use x_4

(c) Complete RFPF

read x_1
$y = 0$
$Q = true$
$true : R = true$
$true : x_2 = 0$
$false : x_3 = -1$
$x_4 = \psi_{true}(\psi_{true}(0, x_1), x_3)$
use $0(x_4)$

(d) Apply CCP

Fig. 14. Constant propagation on RFPF

path from unavailable path. From the perspective of the total number of the computations, RPFP yields essentially the same result. The optimality of RPFP and code motion based PRE in RFPF is yet to be studied, but its ability to catch difficult PRE cases is quite promising.

Case study 2. *Constant propagation in* complete RFPF*:*

We use another example(Figure 14(a)) to show how to do constant propagation(CP) in *complete RFPF*. As in the PRE example, constant propagation chances are caught in node B2 and B4(Figure 14(b)). Figure 14(c) and (d) shows complete RFPF of the program and the result after optimization. We use the conditional constant propagation(CCP) approach described in [18]. Note that x_4 becomes a constant in our representation because gating function ψ can be evaluated given the constant information of the predicate and the variable values.

The choice of applying various optimizations during or after the transformation has to be decided based on foreseen benefits. This is an open research problem and it's a part of our future work.

8 Algorithms for Converting RFPF Back to CFG

The inverse transformation algorithms are necessary because the existing algorithms can be applied on CFG for further optimizations and to produce machine code. In different stages of compilation, the conversion algorithms have different goals. Before scheduling, the goal is to minimize number of nodes in the resulting CFG. After scheduling, the goal is to maximize the issue rate on the resulting CFG. At the register allocation stage, the goal is to minimize live range of variables in the resulting CFG. So we must take into account different optimality criteria for different conversion stages. The basic algorithm to transform RFPF back to CFG consists of three steps:

1. Reorder RFPF in a way that no future values occur by pushing-down or moving-up instructions, which forms an initial instruction list.
2. Group instructions with identical predicates together. Such grouping reduces the multiple node insertions for a branch condition and forms the loop structures.
3. Iterate through the instruction list and insert instructions one by one into corresponding basic blocks.

The optimality of the resulting graph is dependent on how the predicate expressions are analyzed and combined. A complete inverse transformation framework is part of our future work.

9 Related Work

Intermediate program representation design has always been a very important topic for optimizing compiler research since the choice of program representation affects significantly the design and complexity of optimization algorithms. Some of the most relevant to this work are the control flow graph [1], def-use chains [1], program dependence graph [8], static single assignment(SSA) [7,3], and the program dependence web [13]. We directly use these prior art in this paper. The SSA form as well as the gating functions that the program dependence web proposes are significant for correct translation of programs into RFPF. The dependence flow graph [14] contributed to our thinking in designing the representation.

Allen et al. proposed the idea of isomorphic control transformation(ICT) [2] which converts the control dependencies into data dependencies. This idea forms the basis of hyperblock formation in many techniques, including ours as well as others [11]. Warter et al. [17] proposes a technique which uses ICT and apply local scheduling techniques on the hyperblock and then transforms the scheduled code back to CFG representation. RFPF follows a similar, but a more comprehensive path.

Partial redundancy elimination(PRE) proposed by Morel and Renvoise [12] is a powerful optimization technique which is usually carried out using code motion [9,10]. As it is well known, code motion alone cannot completely eliminate partial redundancies. Click proposed an approach using global value numbering supported by code motion is proposed to eliminate redundancies [6]. This approach may insert extra computations along some path. Bodik et al. [4], give an algorithm based on the integration of code motion and CFG restructuring which achieves the complete removal of partial redundancies. Chow et al. [5] proposes a similar PRE algorithm for SSA yielding similar optimality to lazy code motion. The algorithm maintains its output in the same SSA form. VanDrunen and Hosking [16] present a structurally similar PRE for SSA covering more cases. Control flow obfuscate data-flow information needed by many optimization algorithms. Thakur and Govindarajan [15] proposes a framework to find out the merge region in a CFG which prevents the data-flow analysis, and restructure the CFG to make data-flow analysis more accurate. Our technique of instruction propagation and merging exposes similar opportunities.

10 Conclusion

We have presented a new approach to program representation and optimization. The most significant difference of our approach is to move instructions to collect the necessary data and control flow information, and in the process yield a representation in which compiler optimizations can be carried out. Our future work involves transformation and adaptation of state-of-the-art optimization algorithms into the new framework.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston (1986)
2. Allen, J.R., Kennedy, K., Porterfield, C., Warren, J.: Conversion of control dependence to data dependence. In: POPL 1983: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 177–189. ACM, New York (1983)

3. Bilardi, G., Pingali, K.: Algorithms for computing the static single assignment form. J. ACM 50(3), 375–425 (2003)
4. Bodík, R., Gupta, R., Soffa, M.L.: Complete removal of redundant expressions. In: PLDI 1998: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, pp. 1–14. ACM, New York (1998)
5. Chow, F., Chan, S., Kennedy, R., Liu, S.M., Lo, R., Tu, P.: A new algorithm for partial redundancy elimination based on ssa form. In: PLDI 1997: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation, pp. 273–286. ACM, New York (1997)
6. Click, C.: Global code motion/global value numbering. SIGPLAN Not. 30(6), 246–257 (1995)
7. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13(4), 451–490 (1991)
8. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9(3), 319–349 (1987)
9. Knoop, J., Rüthing, O., Steffen, B.: Lazy code motion. In: PLDI 1992: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation, pp. 224–234. ACM, New York (1992)
10. Knoop, J., Rüthing, O., Steffen, B.: Optimal code motion: theory and practice. ACM Trans. Program. Lang. Syst. 16(4), 1117–1155 (1994)
11. Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E., Bringmann, R.A.: Effective compiler support for predicated execution using the hyperblock. In: MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture, pp. 45–54. IEEE Computer Society Press, Los Alamitos (1992)
12. Morel, E., Renvoise, C.: Global optimization by suppression of partial redundancies. Commun. ACM 22(2), 96–103 (1979)
13. Ottenstein, K.J., Ballance, R.A., MacCabe, A.B.: The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. SIGPLAN Not. 25(6), 257–271 (1990)
14. Pingali, K., Beck, M., Johnson, R.C., Moudgill, M., Stodghill, P.: Dependence flow graphs: An algebraic approach to program dependencies. Tech. rep., Cornell University, Ithaca, NY, USA (1990)
15. Thakur, A., Govindarajan, R.: Comprehensive path-sensitive data-flow analysis. In: CGO 2008: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization, pp. 55–63. ACM, New York (2008)
16. VanDrunen, T., Hosking, A.L.: Anticipation-based partial redundancy elimination for static single assignment form. Softw. Pract. Exper. 34(15), 1413–1439 (2004)
17. Warter, N.J., Mahlke, S.A., Hwu, W.M.W., Rau, B.R.: Reverse if-conversion. In: PLDI 1993: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation, pp. 290–299. ACM, New York (1993)
18. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. 13(2), 181–210 (1991)

Optimizing MATLAB through Just-In-Time Specialization*

Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge

School of Computer Science, McGill University, Montreal, QC, Canada
{mcheva,hendren,clump}@cs.mcgill.ca

Abstract. Scientists are increasingly using dynamic programming languages like MATLAB for prototyping and implementation. Effectively compiling MATLAB raises many challenges due to the dynamic and complex nature of MATLAB types. This paper presents a new JIT-based approach which specializes and optimizes functions on-the-fly based on the current types of function arguments.

A key component of our approach is a new type inference algorithm which uses the run-time argument types to infer further type and shape information, which in turn provides new optimization opportunities. These techniques are implemented in McVM, our open implementation of a MATLAB virtual machine. As this is the first paper reporting on McVM, a brief introduction to McVM is also given.

We have experimented with our implementation and compared it to several other MATLAB implementations, including the Mathworks proprietary system, McVM without specialization, the Octave open-source interpreter and the McFor static compiler. The results are quite encouraging and indicate that specialization is an effective optimization—McVM with specialization outperforms Octave by a large margin and also sometimes outperforms the Mathworks implementation.

1 Introduction

Scientists are increasingly using dynamic languages to prototype and implement their applications. MATLAB is particularly appealing because it has an interactive development environment, a rich set of libraries, and highly expressive semantics due to its dynamic nature. However, even though the dynamic nature of MATLAB may be convenient for scientists, it provides many challenges for effective and efficient compilation and execution. Furthermore, scientists would like to have reasonable performance as many scientific applications are computation-heavy and execute for a long time. Ideally this performance should be achieved without requiring a rewrite of MATLAB code to a more static language such as Fortran.

For good performance, we require an optimizing compiler that works directly on MATLAB programs. However, MATLAB poses several challenges. Firstly, MATLAB programs are normally developed incrementally, using an interactive development loop and mixing MATLAB scripts (a sequence of commands like those

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

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 46–65, 2010.

typed into the interactive loop prompt) with functions that are defined in separate source files. This means that code is dynamically-loaded and not all code is known ahead-of-time. Secondly, MATLAB's type system is both dynamic and intricate. The types of variables are not declared, but rather change as the computation proceeds. For example, it is not even straightforward to determine which values are scalars and which are arrays since a scalar assignment, such as `x = 1`, is assumed to define `x` as an 1×1 array. Furthermore, the size of an array dynamically increases as new values are written outside the current array bounds, and the effective base type of an array can change when an element of a more general type is written into it.

All of these challenges suggest that MATLAB is best optimized on-the-fly using a JIT compiler within a MATLAB Virtual Machine. We have developed a new open MATLAB VM called McVM which includes a JIT compiler built upon LLVM [1] which we briefly introduce in this paper. The main feature of the McVM JIT is a new on-the-fly specialization algorithm which specializes functions based on the run-time types of their arguments. This relies on a type and shape inference analysis which is specifically tailored to abstract the key features of the types in the function body. This type and shape analysis must be simple enough to work in the JIT context, but at the same time it must abstract the key features needed for optimization. Our approach is to combine 8 different simple abstractions, consisting of a variable's overall type, whether or not it is a scalar or a 2D matrix, its shape, and so on. The results of this type and shape inference analysis are then used to compile a specialized and optimized version of the function.

In order to determine the effectiveness of this argument-type-based specialization approach, we have implemented it and compared it against both McVM without specialization and three other existing MATLAB implementations: the Mathworks proprietary implementation, Octave[1] which is an open-source MATLAB interpreter, and McFor which is our group's static MATLAB-to-Fortran compiler. Initial results are quite encouraging and show that specialization works, provides good performance and that a reasonable number of specialized versions of functions are created.

The main contributions of this paper are:

McVM: an introduction of McVM giving our design criteria and an overview of the architecture of the system (Section 3);

Specialization: an introduction our approach for specializing functions on-the-fly based on the run-time types of function arguments (Section 4);

Type and Shape Inference: a new type and shape inference algorithm which approximates type and shape information based on argument types (Section 5); and

Experimental Validation: an experimental validation showing the overall effectiveness of McVM and the the effectiveness of specialization and type inference, in particular (Section 6).

[1] `www.gnu.org/software/octave/`

In the remainder of this paper we first describe the challenges of compiling MATLAB in Section 2, then we address each of our main contributions in Sections 3 through 6. We then discuss related work in Section 7 and give conclusions and future work in Section 8.

2 Optimization Challenges

MATLAB presents many challenges to an optimizing compiler. Traditional static optimization techniques do not work because of the highly dynamic nature and the complex semantics of the language. Dynamic loading of functions and scripts prevents us from assuming the entire program is known ahead of time, for example. One of the main challenges, however, is dealing with types, since the language is dynamically typed and follows intricate type rules.

Listing 1 shows an example of a simple program that illustrates some of the intricacies of the MATLAB type system. In this example, the `caller` function calls the `sumvals` function twice, with different argument types each time. The `sumvals` function is designed to sum numbers within a range of values. However, as this example illustrates, in MATLAB, it can be applied to both scalar types and arrays of values. Specifically, the variable `a` will be assigned the scalar integer value `5 * 10e11`, while `b` will be assigned the 1×2 floating-point array `1.0e12 * [0.8533 1.7067]`. These two values are then concatenated into `c`, a 1×3 array.

```
function s = sumvals(start, step, stop)
  i = start;
  s = i;

  while i < stop
    i = i + step;
    s = s + i;
  end
end

function caller()
  a = sumvals(1, 1, 10^6);
  b = sumvals([1 2], [1.5 3], [20^5, 20^5]);
  c = [a b];

  disp(c);
end
```

Listing 1. Implicit typing in MATLAB programs

Since the `sumvals` function can apply to either scalars or arrays, and the values operated on could be either integer, real or complex, compiling this program into efficient machine code can be challenging: type information is not explicit, and can change dynamically. A naive compiler could always store the variables inside the `sumvals` function as the widest available type (i.e.: complex matrices) or even generate code based on the idea that the type of all variables in the function are unknown, which is clearly very inefficient.

To generate efficient code, type inference is needed to extract implicit type information in the source program. In the case where `sumvals` is called with only scalar integer inputs, it is possible to logically infer that all of the intermediate variables will also be scalar integers, and generate efficient code for this case. As for the case where `sumvals` is called with arrays as input, it should be possible to at least infer that complex values will never occur in the computation. This example motivates our approach of specialization based on the run-time types of arguments. Our approach will compile two different versions of the function based on the call signatures. This ensures that efficient code can be generated for each case. More details of our specialization technique are given in Section 4 and our type inference analysis is described in Section 5.

3 Design Overview

Our approach to optimization requires the ability to both interpret and compile multiple versions of code. The McVM virtual machine thus implements a mixed mode design, consisting of both interpreter and JIT components. The design is modular, making use of external front-end and low-level back-end components to simplify implementation complexity; Figure 1 shows the overall structure, which we now describe in more detail.

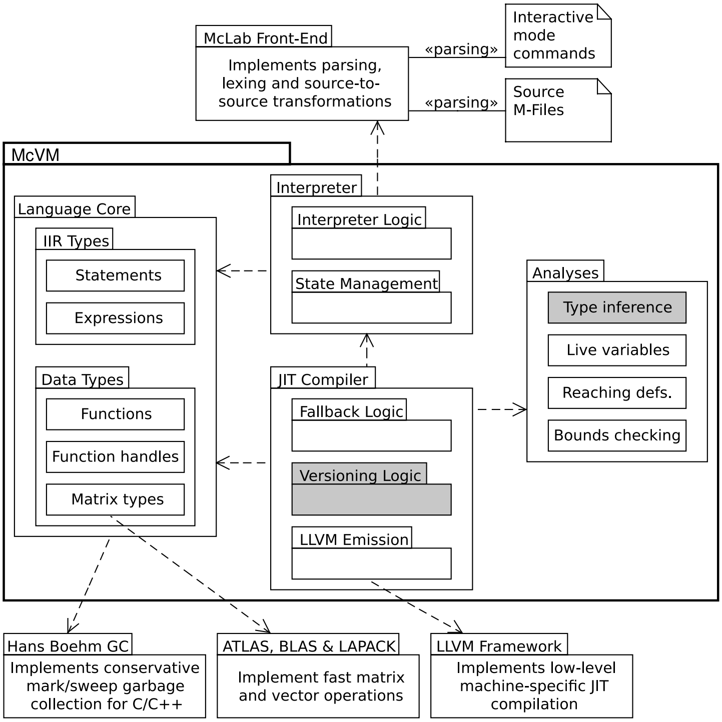


Fig. 1. Structure of the McVM Virtual Machine

The *McLab* front-end is used to parse interactive-mode commands and M-file source code, producing a common Abstract Syntax Tree (AST) representation for both interpretation and compilation. The functionality of the interpreter is divided into interpretation logic and state management (housekeeping), while the JIT compiler manages the function specialization/versioning system, and generates low-level intermediate code for the statements it can compile. Our design allows for incremental and flexible development, with the JIT relying on the interpreter as fallback to evaluate code for which there is not yet compiler support.

At the core, McVM's implementation of matrix types depends directly on a set of mathematical libraries (ATLAS, BLAS and LAPACK) to perform fast matrix and vector operations. We use the Boehm garbage collector library for garbage collection [2], and our JIT compiler uses the LLVM framework to implement low-level JIT compilation and generate machine code [1]. The JIT compiler also implements several analyses to gain additional information about source programs being compiled, including basic analyses and optimizations such as live variables, reaching definitions, bounds check alimination, as well as type inference.

The McVM interpreter performs a straightforward, pre-order traversal of the internal AST in order to execute the input code. This interpretation approach is naive, but provides a correct, if low-performance execution that can serve both as a reference and act as a fallback when JIT compilation cannot be performed. The interpreter also serves housekeeping roles, providing essential run-time services. These include taking care of loading MATLAB files on-demand, executing interactive-mode commands, hosting library function bindings, maintaining bindings to global variables, and so forth.

The JIT compiler improves performance by translating high-level source code into a more efficient low-level form. A fundamental design goal in our VM was to aim for a simple and easily extensible design—similar to the `phc` compiler [3], our JIT compiler is built as an extension of the interpreter. The compiler can thus fall back to interpreting sections of code it cannot compile, mixing sections of both compiled and interpreted code in the execution of a given function. This allows for incremental JIT development, and also for language modifications to be more easily incorporated—new data types or statements can be added by modifying only the interpreter, relying on the fallback mechanism for any new features. The JIT compiler can later be modified, if necessary, to gain performance benefits from any additional optimization opportunities.

The JIT compiler performs actual code generation in conjunction with LLVM. During run-time, the input AST is first translated by our JIT compiler into a low-level, RISC-like Static Single Assignment (SSA) representation. From this, LLVM generates machine-specific executable code; LLVM also performs basic optimization passes on the code, such as constant propagation, dead code elimination and redundant operation elimination. As such, it greatly simplifies the construction of a JIT compiler by completely hiding much of the platform-specific details and providing low-level optimizations.

Our fallback mechanism requires a high-level strategy to coordinate the transition from compiled code to interpretation and vice-versa. In particular, at each step of the compilation process the JIT must track how and where each live variable is stored in order to appropriately transfer execution context. When interpreter fallback code is generated, instructions are issued to flush any register variables into memory for interpreter consumption. Upon returning to compiled execution, variables are copied back into their original registers. While spilling variables in this way is expensive, it has the advantage that the interpreter fallback mechanism does not impose extra penalties on compiled code in the case of functions which do not need to use it.

The McVM JIT compiler is able to compile and make use of specialized versions of functions based on call signatures. This corresponds to the two shaded boxes in Figure 1 labeled "Versioning Logic" and "Type Inference". In the next two sections we examine these two important components in more detail.

4 Just-In-Time Specialization

Exposing and using type information is central to most existing approaches to Matlab optimization [4,5]. McVM uses run-time type information to create multiple specialized versions of Matlab functions. This allows for optimized function dispatch and improved code generation for many common operations, greatly reducing overhead costs necessary in a more generic design. Below we describe our precise versioning strategy, followed by core optimizations so enabled.

4.1 Function Versioning

Specialization requires creating type-specific versions of function bodies. This process is performed at run-time, by "trapping" commands issued through the interpreter (including calls made in the read-eval-print loop of the interactive mode). If the command is a call to a function (and not a script), the interpreter will try and pass control to the JIT compiler. When this happens, the JIT compiler builds an argument type string from the input arguments to the function, and attempts to locate a previously compiled version of the function with a matching argument type string. If none exists a new version will first be compiled, appropriately specialized to the given argument types. This removes significant dispatch overhead, allowing, for instance, scalar variables to be stored on the stack instead of as objects allocated on the heap. While compiling specialized function versions, the JIT compiler also considers functions called by the function being compiled, compiling them as direct calls to specialized versions as well. Thus entire executions can be specialized in a "deep" fashion.

As an example of how our function versioning works, consider the `sumvals` function shown earlier in Listing 1. This function is meant to sum numerical values in the range from start to stop, inclusively. In the absence of type information and specialization a compiler must make conservative assumptions, assuming iteration is potentially performed over arrays. Expensive heap storage is thus required, as well as function calls to generically perform every operation (addition, comparison, etc.).

```
function s <scalar int> = sumvals(start <scalar int>, step <scalar int>,
     stop <scalar int>)

    i <scalar int> = start;
    s <scalar int> = i;

    while i < stop
        i <scalar int> = i + step;
        s <scalar int> = s + i;
    end

end
```

Listing 2. The type-annotated sumvals function

At an actual invocation of the function, however, such as in Listing 1: `a = sumvals (1, 1, 10^6);`, argument types are known to be scalar integers. This information is flowed through the function by our type inference, producing a type-annotated version as shown in Listing 2. From this, efficient code can be generated: all variables are easily stored on the stack, and there is no need to make expensive dispatches, because there are efficient machine instructions to add and compare scalar integer values.

The obvious downside is that this scheme has the potential to generate many specialized versions of a function, with each requiring additional compilation time, and potentially impacting the performance of the instruction cache, should multiple versions be executed. We will see that this is not the case in practice (see Section 6). From our observations, Matlab programs tend to have few long functions and fewer call sites than code written in other programming languages.

4.2 Additional Optimizations

Type-based specialization greatly simplifies basic arithmetic operations, allowing many uses of scalars to be implemented in just a few machine instructions. The type information, however, also facilitates the optimization of a number of other common operations, in particular certain array access operations, and use of library function calls. These optimizations improve performance by both taking advantage of type information, and eliminating cases where interpreter fallback is otherwise required.

Matlab possesses a sophisticated array indexing scheme that allows programmers to read or write to n-dimensional slices (sub-arrays) based on ranges of indices, specified independently for each dimension. This behaviour is implemented through the interpreter, using the fallback mechanism to evaluate complex array reads and writes. When types are known, however, such as in `x = a(i);` where `i` is a scalar, optimized code can be generated to read or write the value directly. Type information includes array dimensions as well, eliminating the need for many dynamic array bounds checks.

Library functions are implemented in our virtual machine as native C++ functions which take as input (and return as output) dynamically allocated arrays of pointers to data objects. This strategy is conservatively correct in the presence of unknown types, but can be inefficient because each call to these functions requires array allocation. Even for variables known to be scalar, the use of a generic library routine requires boxing and unboxing arguments and return values respectively, reducing the benefit from other optimizations.

To address these issues, we have devised a further simple specialization scheme for some library functions. Multiple, type-specific versions of library functions are first registered ahead-of-time in McVM. When a library function call is encountered, the JIT compiler will attempt to locate an appropriately specialized version, matching function argument and return types. An obvious example where this is beneficial is in the case of functions like `abs` or `sin`, where scalar data allows the direct use of the native C++ versions of these library functions.

5 Type and Shape Inference System

The McVM JIT compiler uses data provided by our type inference analysis to implement the just-in-time function specialization scheme described in Section 4. The more information the analysis provides about the concrete types and shapes of program variables, the more interpretive dispatching and storage overhead can be eliminated, and the faster the resulting compiled code will be, as demonstrated in Section 6.

Our type inference analysis works on a per-function basis, with the assumption that the whole program is not necessarily known at run-time, and new functions could be loaded at any time. The analysis assumes that the set of possible types for each input argument of a given function are known, and infers the set of possible types for every variable at every point (before and after every statement) in the function, given those possible input argument types.

The analysis is an abstract interpretation style analysis, which implements a compositional forward analysis directly on the structured AST representation. The analysis computes an abstraction of the actual types and shapes of variables at each program point. The actual abstraction is a carefully designed combination of simple abstractions, where each element of the abstraction captures a key aspect of the variable's type or shape. For example the *isScalar* flag indicates when a variable is definitely a scalar variable. If this flag is true, then the JIT compiler can allocate it to a register, which is much more efficient than storing it as a matrix. Another key point of our analysis is that it is flow-sensitive, and we thus have type and shape information for each program point.

5.1 Abstract Domain

In the real domain of Matlab programs, variables at different program points are bound to actual values (data objects). In our abstract domain, variables instead map to sets of possible abstract types. These sets contain zero or more *type abstractions* summarizing all possible types and shapes the specific variable can have. Each type abstraction is actually an 8-tuple: $\langle overallType, is2D, isScalar, isInteger, sizeKnown, size, handle, cellTypes \rangle$.

If an abstract type set contains multiple type abstractions, it means that the variable whose potential types are represented by the set at that program point could be one of the several types represented by each type abstraction in the set. The empty set is the $\perp$ element of the type lattice, representing situations where no information has been computed yet. The set of all type objects is the $\top$ element of the lattice, representing the situation where the type of a variable cannot be determined.

The core of the abstraction is the first item of the 8-tuple, the *overallType*, which represents a specific MATLAB language type, such as character array, floating-point matrix, complex number matrix, etc. Figure 2 represents the hierarchical type lattice of McVM *overallType* values.

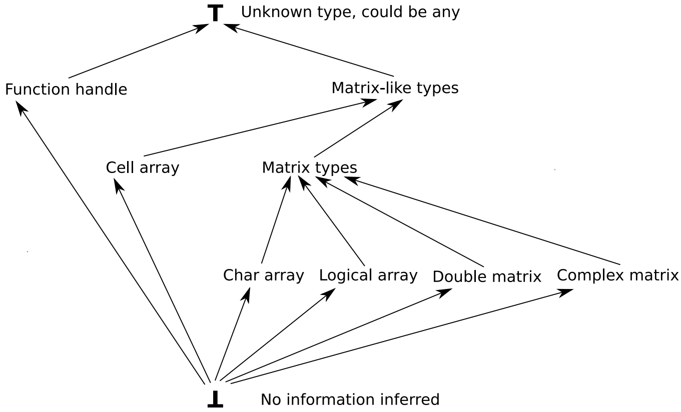


Fig. 2. Hierarchical lattice of McVM types

The remaining elements of each 8-tuple provide abstractions of different features of the type. Table 1 describes the fields stored in type objects. These fields cannot hold arbitrary values. For example, if the *isScalar* flag is set to True, then the *sizeKnown* flag must also be True. However, the *is2D* flag does not necessarily indicate that the matrix size is known.

For each statement in a program, our analysis produces a mapping of symbols to sets of type abstractions representing the type that each variable in the current function may hold before the statement is executed. Formally, if O is the set of all possible type abstractions and S is the set of all symbols, then our analysis operates in the domain of subsets of M, where M is the set of all pairs of symbols and subsets of O (mappings of symbols to type sets):

$$M = \{\, (s,t) |\, s \in S, t \in P\,(O)\}$$

5.2 Merge Operator

A merge operator is required to implement inference rules for control flow statements. This is because when multiple control paths join at a given point in a

Table 1. Description of type object fields

Field	Meaning/Description	Default
overallType	An element of the set of possible McVM data types.	Undefined
is2D	Flag whose value applies to matrix types only. A True value indicates that the matrix has at most two dimensions. False means it is not known how many dimensions the matrix has.	False (unknown)
isScalar	Flag whose value applies to matrix types only. A True value indicates that the matrix is a scalar. False means the matrix may not be scalar.	False (unknown)
isInteger	Flag whose value applies to matrix types only. A True value indicates that the matrix contains only integer values. False means the matrix may contain non-integer values.	False (unknown)
sizeKnown	Flag whose value applies to matrix types only. A True value indicates the size of the matrix is known. False means the size is not known.	False (unknown)
size	Applies to matrix types only. A vector of integers storing the dimensions of the matrix. This is only defined if the sizeKnown flag is set to True.	Undefined
handle	Applies to function handles types only. Stores a pointer to the function object the handle points to. This value can be null if the specific function is not known at inference time.	null (unknown)
cellTypes	Applies to cell array types only. Set of type objects representing the possible types the cell array stores.	⊥ (undefined)

program, our analysis needs to merge the mappings of symbols to type sets for each of these control flow paths into one single mapping. In our analysis, the merging of two type mappings is accomplished by performing, for each symbol, the joining of the type sets for each type mapping:

$$merge(M_1, M_2) = \{\, (s, t) |\, (s, t_1) \in M_1, (s, t_2) \in M_2, t = join(t_1, t_2)\}$$

The joining of type sets is accomplished by using set union as a merge operator and then applying a filter operator to the result:

$$join(t_1, t_2) = filter\,(t_1 \cup t_2)$$

The filter operator takes a type set as input and returns a new type set in which all type objects having the same *overallType* value have been merged into one. It does so in a pessimistic way, that is, if one of the type objects to be merged has an unknown value for one of its flags, the merged type object will have the unknown value for this flag. For example, if we are filtering a type set containing multiple `double` matrix type objects, the resulting type object will have the *integer* flag set to true only if all input type objects did.

5.3 Inference Rules

Our type inference analysis follows inference rules to determine the mapping of possible variable types after a given statement based on the possible types before that same statement. Each kind of statement has an associated type inference rule that takes the mapping of possible input types as input and returns the mapping of possible output types as output. Expression statements, such as `disp(3);` use the identity type mapping, that is, the output types they produce are the same as the input types.

The statements that are at the core of our type inference analysis are assignment statements. They are the only kind of statement that can define a variable, and thus, change its type. In the case of an assignment statement of the form `v = op(a, b);`, where `op` is an element of the set `R` of all possible binary operators, we have that the type of `v` is redefined as the set of possible output types of the operator being applied to the possible types of `a` and `b`, according to its own type rule:

$$typeRule_{v=op(a,b)}\left(M_{in}\right) = \left\{\left(s,t\right) \in M_{in} \middle| \, s \neq v\right\} \cup typeRule_{op(v,a,b)}\left(M_{in}\right)$$

$$typeRule_{op(v,a,b)}\left(M_{in}\right) = \left\{\left(v,t\right) \middle| \, t = outtype_{op}\left(\left\{(a,t) \in M_{in}\right\}, \left\{(b,t) \in M_{in}\right\}\right)\right\}$$

As an example, we can look at the assignment `c = [a b];` in Listing 1. This represents the horizontal concatenation of arrays `a` and `b`. In this case, `a` holds the value `5 * 10e11`, which is a scalar integer value, and `b` holds the value `1.0e12 * [0.8533 1.7067]`, a 1 × 2 floating-point array. Thus, the type abstractions for `a` and `b` are:

$$\begin{aligned} type(a) = \{\langle & overallType = double, is2D = T, isScalar = T, isInteger = T, \\ & sizeKnown = T, size = (1,1), handle = null, cellTypes = \bot\rangle\}\end{aligned}$$

$$\begin{aligned} type(b) = \{\langle & overallType = double, is2D = T, isScalar = F, isInteger = F, \\ & sizeKnown = T, size = (1,2), handle = null, cellTypes = \bot\rangle\}\end{aligned}$$

The type rule associated with the horizontal concatenation operation allows us to infer that `c` will be a 1 × 3 floating-point array, that is:

$$\begin{aligned} outtype_{hcat}(type(a), type(b)) = \{\langle & overallType{=}double, is2D = T, isScalar = F, \\ & isInteger = F, sizeKnown = T, size = (1,3), \\ & handle = null, cellTypes = \bot\rangle\}\end{aligned}$$

In the case of `if` statements, the type inference process is handled differently. The “true” and “false” branches of the statement are both treated as compound statements, as if all statements on either branch were one statement. The output type mappings are determined separately for both branches and then merged together into one mapping of the possible types at the output of the `if` statement itself:

$$typeRule_{if}\left(M_{in}\right) = merge\left(typeRule_{trueStmt}\left(M_{in}\right), typeRule_{falseStmt}\left(M_{in}\right)\right)$$

Handling of loop statements is slightly more complex. Because types at the input of the loop depend on types at the output, a fixed point must be iteratively computed. Before we apply our type inference analysis, all loop statements are converted to `while` loops. As is the case for `if` statements, statements in the loop body are treated as one single compound statement. Special care is taken to properly deal with both `break` and `continue` statements.

5.4 Inference Process

In terms of abstract interpretation, we wish to compute, for a given function, the least fixed point of the mapping of program statements and variables to sets of possible types before that given program point. The type inference process for a function begins with the type sets for the input parameters of the function being given. Because of the Matlab semantics, the possible types of all other variables are initialized to $\top$. This is because undeclared variables could be globals, and thus, could potentially hold any type.

The body of the function is then analyzed. The function body itself is a compound statement. When inferring the types in a compound statements, the statements it contains are traversed in order, and the inferred output type of each statement is stored in a global mapping (e.g.: hash map) of the types at the output of each statement.

6 Evaluation

In order to assess the performance of our virtual machine we compare the actual performance of McVM to that obtained by several related systems: Mathworks Matlab, GNU Octave (the GNU Matlab environment) and McFor (a Matlab to Fortran translator built by Jun Li, a member of the McLab team). The Octave and Matlab performance numbers are intended to give us some idea of how well our current solution performs against competing implementations. The McFor numbers are provided as a rough “upper bound” on performance—Fortran compilers are known to perform very well on numerical computations, giving an indication of potential compiler performance for non-interactive code.

We have performed our tests on a total of 20 benchmark programs. These benchmarks are gathered from previous work on optimizing Matlab[2], in the FALCON [6] and OTTER projects, Mathworks' CentralFile Exchange, Chalmers University, and from individual course work and student projects at McGill. Several of these are currently unsupported by the McFor Fortran translator as it lacks support for cell arrays, closures and function handles at this time. The left part of Table 2 provides characteristic numbers for each of the benchmarks supported by McVM. Number of functions and statements (3-address form) relate to the overall (static) input load on our system, while number of call sites directly affects specialization. Maximum loop nesting depth affects the theoretical efficiency of our dataflow analysis.

Not all benchmarks benefit equally from our optimizations of course, and in the following sections we show further profiling numbers intended to explain where specific performance bottlenecks occur. Section 6.2 describes the behaviour of the type inference system, while Section 6.3 gives data on the specialization system, including compiler overhead. All of our benchmarking metrics were gathered on a system equipped with an Intel Core 2 Quad Q6600 processor (quad core, 2.4GHz) and 4GB of dual channel DDR2 RAM, running Ubuntu 9.10

[2] http://www.ece.northwestern.edu/cpdc/pjoisha/MAT2C/

(linux kernel 2.6.31, 32-bit). We have gathered our Matlab performance numbers using Matlab R2009a, and our GNU Octave numbers on Octave version 3.0.5. The Fortran code produced by McFor was compiled using the GNU Fortran compiler version 4.4.1. Because of significant variance when timing benchmarks, attributable to i-cache effects and the garbage collector, all benchmark timing measurements are based on an average over 10 runs.

6.1 Baseline Performance

The rightmost columns of Table 2 show a comparison of benchmark running times under our four execution environments, as well as a version of McVM with the JIT and specialization disabled, giving absolute time as well as times normalized to the McVM JIT (values greater than 1 are running slower than McVM with JIT). As we can see, McVM with JIT performs better than Matlab in 8 out of 20 benchmarks, sometimes by a fair margin. In the cases where it does worse than Matlab, the running times can be relatively close (as with `nnet`), or, as exemplified by the `crni` benchmark, sometimes dramatically less; we discuss reasons for this poor performance in Section 6.2.

GNU Octave, possessing no JIT compiler, does rather poorly in general. It trails far behind Matlab and outperforms McVM with JIT on only a single benchmark. Interestingly, McVM in interpreted mode, although it performs much worse than the JIT on several benchmarks, actually performs better on some (this will also be discussed further). The McFor running times are generally well ahead of Matlab and McVM, with the exception of the `clos` benchmark. This suggests that Matlab and McVM both are still far from the "optimal" performance level.

6.2 Type Inference Efficiency

Our ability to optimize strongly depends on the behaviour of our type inference system. The leftmost part of Table 3 thus shows relevant run-time profiling information, dynamically weighted by the relative execution counts of the associated statements. The first data column gives the percentage of type sets that are at top, providing no type information, while column 3 shows the percentage of type sets which contain only one type, and so give exact type data. The third column shows the percentage of times where variables holding scalar values were known ahead of time to be scalar, and the fourth column is the percentage of times where the size of matrix variables was known by the type inference system.

In general the more type information our system has the better it will be able to optimize code generation. Knowledge of which variables are scalars is even more critical, however, as it lets the JIT compiler know which variables can be stored on the stack. As we can see, this matches our results: benchmarks with speedups of over 99% all have 100% of scalar variables known. The behaviour of the `crni` benchmark can also be explained by this data. As can be seen in Table 3, scalars are known in only 68.7% of cases, one of the lowest such ratios. An examination of the code reveals this benchmark uses matrix "creation on

Table 2. Benchmark characteristics and comparison of running times. Columns 6–10 give absolute running times, while columns 11–14 are performance normalized to McVM JIT. The geometric mean was used for relative values (columns 11–14).

	Static Measurements				Performance (s)					Relative to McVM JIT			
Benchmark	Functions	Stmts	Loop Nesting	Call Sites	McVM JIT	MATLAB	McVM no JIT	Octave	McFor	MATLAB	McVM no JIT	Octave	McFor
adpt	2	196	2	6	13.4	2.66	12.6	45.9	0.72	0.20	0.94	3.42	0.05
beul	10	511	1	38	3.07	3.09	1.56	7.62	N/A	1.01	0.51	2.49	N/A
capr	5	214	2	10	3.51	8.10	1674	5256	1.26	2.31	478	1499	0.36
clos	2	58	2	3	6.84	0.75	13.6	17.5	7.87	0.11	1.99	2.56	1.15
crni	3	142	2	7	1321	6.95	1788	5591	3.56	0.01	1.35	4.23	0.00
dich	2	144	3	7	2.80	4.71	1149	4254	1.88	1.68	410	1517	0.67
diff	2	253	3	6	30.0	5.26	41.9	120	0.65	0.17	1.39	3.98	0.02
edit	2	130	2	6	54.9	11.0	81.4	394	0.13	0.20	1.48	7.17	0.00
fdtd	2	157	1	3	20.1	3.32	8.56	172	0.29	0.17	0.43	8.55	0.01
fft	2	159	3	8	12.8	16.2	2470	8794	9.13	1.27	193	689	0.72
fiff	2	120	2	4	5.37	6.97	1528	4808	0.99	1.30	285	895	0.18
mbrt	3	78	2	11	34.6	4.53	98.6	295	0.96	0.13	2.84	8.51	0.03
nb1d	3	194	2	11	4.10	9.85	4.24	43.9	0.74	2.40	1.03	10.7	0.18
nb3d	3	164	2	12	3.88	1.54	2.51	40.8	0.89	0.40	0.65	10.5	0.23
nfrc	5	151	2	11	15.7	4.94	26.0	80.3	N/A	0.32	1.66	5.13	N/A
nnet	4	186	3	16	6.95	6.35	7.32	26.5	N/A	0.91	1.05	3.81	N/A
play	6	364	2	29	3.37	8.68	4.24	29.0	N/A	2.57	1.26	8.60	N/A
schr	8	203	1	32	2.48	2.07	3.03	2.31	N/A	0.84	1.22	0.93	N/A
sdku	9	363	2	49	1.23	9.74	16.0	112	N/A	7.93	13.1	90.9	N/A
svd	11	308	3	42	8.24	2.38	7.02	10.9	N/A	0.29	0.85	1.33	N/A
mean	4.3	205	2.1	15.6	77.7	5.96	447	1505	2.24	0.49	3.91	15.4	0.08

assignment" to initialize its input data, resulting in several unknown types being propagated through the entire program. We examine ways to fix this weakness of our type inference system as part of future work.

While our JIT compiler is able to speed up most benchmarks, sometimes by very significant margins, some still show slowdowns over interpreted performance. These do not necessarily have poor type information. The `nb3d` benchmark, for example, has 100% scalar variables known and 96.9% singleton type sets. Most of these benchmarks makes heavy use of complex slice read operations operating on entire columns or rows of a matrix at a time, and these are currently implemented through our (expensive) interpreter fallback mechanism.

6.3 JIT Specialization

The benefit of JIT specialization depends on how well it improves the code as well as any introduced overhead. The rightmost three columns of Table 3 show

Table 3. Profiled performance. All values are percentages.

Benchmark	Top sets	Singleton sets	Scalars known	Size known	JIT speedup	Matrices created	Slice reads	Env. lookups
adpt	4.18	95.8	100	90.0	-6.82	24.8	16.8	39.2
beul	55.2	44.8	71.3	29.5	-96.3	85.5	49.8	114
capr	0.01	100	100	82.8	99.8	0.00	0.00	0.00
clos	0.00	100	100	99.9	49.7	0.00	100	0.00
crni	19.1	71.4	68.7	54.8	26.1	66.7	69.2	55.2
dich	2.09	97.9	100	85.1	99.8	0.00	0.00	0.00
diff	14.3	82.1	66.7	66.7	28.2	68.3	100	2.45
edit	5.14	94.9	96.8	81.5	32.5	65.0	40.0	81.6
fdtd	0.01	100	100	49.8	-135	88.1	90.0	90.5
fft	0.00	100	100	80.3	99.5	0.00	0.00	0.00
fiff	0.01	100	100	86.1	99.6	0.01	0.00	0.00
mbrt	9.09	90.9	100	100	64.9	33.3	100	0.00
nb1d	5.84	94.2	88.1	34.5	3.33	75.6	0.00	14.9
nb3d	3.13	96.9	100	16.5	-54.6	94.0	98.3	76.2
nfrc	16.4	82.7	100	98.9	39.8	42.5	100	19.8
nnet	52.6	47.4	98.7	55.1	5.08	86.9	100	82.8
play	23.3	66.6	77.5	52.1	20.6	72.5	100	45.9
schr	31.8	55.3	99.5	41.7	18.3	65.5	54.0	84.6
sdku	14.8	85.2	83.8	49.7	92.3	7.55	5.69	4.65
svd	16.4	73.8	94.2	59.7	-17.4	84.7	100	60.2
mean	13.7	84.0	92.3	65.7	23.5	48.0	56.2	38.6

the effect of JIT compilation on three profile measures, the number of matrices created, the number of slice reads, and the number of environment lookups, in each case presented as a percentage of the original, interpreted quantity. These are all expensive operations, and so large reductions should map to large improvements from JIT compilation. The `fft` benchmark, for instance, has 100% of its 789 million interpreter slice reads eliminated, and runs over 190 times faster with the JIT compiler enabled.

For a better understanding of the cost/benefit of different components of our system, we also evaluate the performance of McVM with specific JIT optimizations disabled. Relative to the McVM JIT compiler with all optimizations enabled, the five leftmost columns in Table 4 show the ratio of run-times of McVM with optimizations to arithmetic operations, array operations, function calls, specialized library functions, and the entire JIT selectively disabled (a number greater than one signifies a slowdown). Clearly, arithmetic operation and array access optimizations have a tremendous impact as they speed up several benchmarks by two orders of magnitude. In certain cases, such as `dich`, optimizing library functions also has a large impact.

Table 4. Relative JIT performance with specific optimizations disabled (columns 2–6), and overhead of the optimization system (columns 7–10). The geometric mean was used for relative values (columns 2–6).

Benchmark	Arith.	Array	Direct calls	Library	JIT	# functions	# versions	Compile (s)	Analysis (s)
adpt	1.43	1.12	0.97	1.07	0.94	2	2	0.86	0.79
beul	1.03	1.00	1.00	1.00	0.51	9	16	1.20	0.90
capr	590	428	1.73	1.05	478	5	5	0.50	0.43
clos	3.40	1.01	1.00	1.00	1.99	2	2	0.14	0.12
crni	1.63	1.27	0.75	0.99	1.35	3	3	0.32	0.26
dich	459	282	1.00	29.7	410	2	2	0.38	0.32
diff	2.20	1.03	1.01	0.96	1.39	2	2	1.19	1.10
edit	1.90	1.46	0.61	0.98	1.48	2	2	0.22	0.17
fdtd	1.25	1.10	1.01	0.87	0.43	2	2	0.48	0.38
fft	144	143	1.02	1.01	193	2	2	0.58	0.54
fiff	280	204	1.01	1.05	285	2	2	0.24	0.20
mbrt	3.57	1.05	1.05	0.99	2.84	3	3	0.14	0.11
nb1d	0.90	1.22	1.06	0.97	1.03	3	3	0.51	0.42
nb3d	0.66	1.07	1.08	0.97	0.65	3	3	0.57	0.46
nfrc	1.33	1.04	1.77	0.98	1.66	5	5	0.22	0.15
nnet	1.20	1.01	1.02	0.98	1.05	4	4	0.36	0.29
play	1.21	1.03	1.11	0.98	1.26	6	10	0.58	0.42
schr	1.47	1.00	1.02	1.00	1.22	8	9	0.55	0.45
sdku	1.42	1.67	1.13	0.97	13.1	9	11	1.08	0.85
svd	3.92	0.98	1.05	0.98	0.85	11	15	0.79	0.61
mean	4.56	3.28	1.04	1.17	3.91	4.2	5.2	0.55	0.45

The direct call mechanism has much less impressive benefits. It improves benchmarks that perform many function calls, but can also yield lower performance in cases where the types of input parameters to a function are unknown. A version of the function then gets compiled with insufficient type information, whereas the interpreter can extract exact type information on-the-fly when a call is performed with direct calls disabled.

Given our specialization strategy, compilation overhead is a concern—if types are highly variable, many function versions will be compiled, adding CPU and memory overhead. We thus measured the number of functions compiled, as well as the total number of specialized versions for each of our benchmarks. Columns 7 and 8 in Table 4 show that excessive specialization is not a problem in practice. In most cases functions are always called with the same argument types, and there are never more than twice as many versions as compiled functions.

The last two columns of Table 4 give the absolute compile-time overhead and its analysis-time constituent. As we can see, most of the compilation time is spent performing analyses on the functions to be compiled, as opposed to code generation. The slowest compilation time is associated with the `diff` benchmark. We attribute this to the large quantity of code contained in a triple nested loop in this benchmark, for which our analyses take longer to compute a fixed point. In most cases these costs are not excessive and are easily overcome by the performance improvement, especially for longer running benchmarks.

7 Related Work

Our approach to optimizing MATLAB has concentrated on dynamic features of the language that interfere with more traditional optimization. This brings together more traditional work on compiling scientific, array-intensive languages and techniques for optimizing dynamic languages, and specifically dynamic specialization and type inference.

Previous compiler approaches to MATLAB have mainly focused on numerical performance, primarily in the context of static language subsets or contexts. As well as more traditional loop and array optimizations, code restructuring can be performed to ensure programs take good advantage of optimized instrinics [7]. Good performance can also be achieved by translating MATLAB code to other static languages, such as C [8] or Fortran 90 [6,9], where further aggressive optimization or parallelization can be performed. A major source of complexity for almost all MATLAB optimizations, as in our case, is analyzing and understanding array properties, such as shape and size [10]. Elphick et al. identify similar typing and dynamic language concerns in their partial evaluation approach to optimizing MATLAB programs [5]. They develop *MPE*, an online system to partially evaluate MATLAB source functions into more efficient MATLAB code. Their design is intra-procedural and does not handle polyvariant types, but as such may provide an additional and orthogonal benefit to our approach.

Full VM approaches have also been applied, including JIT-based solutions. *MaJIC* combines JIT-compilation with an offline code cache maintained through speculative compilation of MATLAB code into C/Fortran [4]. They derive the most benefit from optimizations such as array bounds check removals and register allocation. The *Match* VM project translates MATLAB programs to a lower-level intermediate form which is then analyzed for dependencies and used to automatically parallelize computation [11]. The result is invisible to the user, and by relying on run-time estimates for scheduling avoids static array analysis requirements.

Program Specialization. We use program specialization [12] in order to optimize effectively in the presence of imprecise type information. More specifically, we apply procedure cloning [13] to create specialized copies of function bodies in which we can make stronger typing assumptions. Such specialization techniques have previously been used offline to translate MATLAB code into optimized C or Fortran code [14]. Our design extends on run-time specialization techniques used by languages such as SELF [15] and is similar to the approach used to optimize the JIT compilation of generics for the C# language [16]. More general specialization designs have also been applied [17]. In practice this can yield very significant performance gains—Schultz and Consel report speedups of up to 300% for their specializing *JSpec* Java compiler [18].

Run-time specialization accommodates MATLAB's dynamic nature, and is a technique that has been applied in many other dynamic optimization contexts. The *Psyco* python virtual machine, for instance, implements specialization "by need" [19], a process similar to the online partial evaluation approaches applied to MATLAB [5] and Maple [20]. This specialization technique involves interleaving program specialization and execution; the specializer can request facts such as the type of variables while a procedure is executing, and depending on the result, potentially modify the compiled code to be more efficient. The design goal was to eliminate much of the interpretative overhead through the use of JIT compilation, without sacrificing the dynamic features of the language. Approaches such as *Psyco* differ from our system by working on fine-grain code fragments rather than functions, trading simpler code-generation and analysis requirements for smaller specialized sequences.

Similar to the Psyco effort, the *TraceMonkey* VM for the JavaScript language has focused on just-in-time specialization based on type information in order to increase performance [21]. The design is based on a bytecode interpreter that can identify frequently executed bytecode sequences (traces) going through loops and compile them to efficient native code based on collected type information. A crucial assumption of their system is that programs will spend most of their time in loops, and that the types of variables will remain mostly stable throughout the execution of loops. They have achieved speedups of up to 25 times on some benchmarks. However, their current VM does poorly on benchmarks making extensive use of recursion.

Type Inference. Our specialization approach is facilitated by a type inference analysis [22], where we use a straightforward, if non-trivial dataflow analysis to determine type information. The problem, of course, has been examined in many contexts, and poses an efficiency and accuracy trade-off even in the case of statically typed languages, such as C++ [23] or Java [24]. In these cases relatively cheap *flow-insensitive* approaches to type analysis have been shown effective. In a more general and flow-sensitive sense the type inference problem can also be seen as a bidirectional dataflow analysis, propagating type information both along and against the direction of control flow [25]. In most such analyses types are considered static, although dynamic types may be reduced to static types through the use of a Static Single Assignment (SSA) representation.

Type inference on dynamic languages brings additional complexity. Constructs like `eval`, MATLAB's `cd`, as well as dynamic loading and reflection features, make it difficult or impossible to know the entire call graph of a program ahead of time. Despite this, there have been efforts to statically perform type inference on dynamic languages such as MATLAB [26] and Ruby [27]. These approaches show potential to detect type errors ahead of time, but they do not address the aforementioned problems. Our approach, on the other hand, can operate on programs whose call graphs are not fully known ahead of time.

8 Conclusions and Future Work

Our experience with McVM demonstrates that online specialization is an effective and viable technique for optimizing Matlab programs. Although other specialization and partial evaluation approaches have been applied to Matlab [4,5] and similar dynamic language contexts [19,21], we provide an efficient and full JIT solution. Our approach focuses on optimizing code generation, uses a coarse-grained strategy that minimizes specialization overhead, and is specifically designed to accommodate complex dynamic language properties. Combined with an effective type and shape inference strategy, McVM is able to achieve performance up to three orders of magnitude faster than competing Matlab implementations such as GNU Octave, and in several cases faster than the commercial product.

Further improvements to performance are possible in a number of ways. The need to be conservative in our type inference analysis means that unknown types dominate in merges. The result is that once "unknown" types are introduced, they often propagate and undermine the type inference efforts. Our code generation strategy is then left with very little information to operate on. In many cases, however, even if the type of a variable cannot be determined with 100% certainty, it may be possible to mitigate the impact of unknown types by predicting the most likely outcome.

A speculative design enables heuristic judgements. It is likely, for example, that if a variable is repeatedly added to integer matrices, that it is also an integer matrix. Our code generation system could use these "best guesses" to generate an optimized code path. The types of variables can then be tested during execution and either an optimized path or default code chosen as appropriate. Speculative approaches have been successful based on external compilation [4], and a JIT-based solution has potential to yield further significant speed gains.

References

1. Lattner, C.: LLVM: An infrastructure for multi-stage optimization. Master's thesis, Comp. Sci. Dept., U. of Illinois at Urbana-Champaign (December 2002)
2. Boehm, H., Spertus, M.: Transparent programmer-directed garbage collection for C++ (2007)
3. Biggar, P., de Vries, E., Gregg, D.: A practical solution for scripting language compilers. In: SAC 2009, pp. 1916–1923. ACM, New York (2009)
4. Almási, G., Padua, D.: MaJIC: compiling MATLAB for speed and responsiveness. In: PLDI 2002, pp. 294–303. ACM, New York (2002)
5. Elphick, D., Leuschel, M., Cox, S.: Partial evaluation of MATLAB. In: Pfenning, F., Smaragdakis, Y. (eds.) GPCE 2003. LNCS, vol. 2830, pp. 344–363. Springer, Heidelberg (2003)
6. Derose, L., Rose, L.D., Gallivan, K., Gallivan, K., Gallopoulos, E., Gallopoulos, E., Marsolf, B., Marsolf, B., Padua, D., Padua, D.: 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)

7. Birkbeck, N., Levesque, J., Amaral, J.N.: A dimension abstraction approach to vectorization in MATLAB. In: CGO 2007, pp. 115–130. IEEE Computer Society, Los Alamitos (2007)
8. Joisha, P.G., Banerjee, P.: A translator system for the MATLAB language: Research articles. Softw. Pract. Exper. 37(5), 535–578 (2007)
9. Rose, L.D., Padua, D.: A MATLAB to Fortran 90 translator and its effectiveness. In: ICS 1996, pp. 309–316. ACM, New York (1996)
10. Joisha, P.G., Banerjee, P.: An algebraic array shape inference system for MATLAB®. ACM Trans. Program. Lang. Syst. 28(5), 848–907 (2006)
11. Haldar, M., Nayak, A., Kanhere, A., Joisha, P., Shenoy, N., Choudhary, A., Banerjee, P.: Match virtual machine: An adaptive runtime system to execute MATLAB in parallel. In: ICPP 2000, pp. 145–152 (2000)
12. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Prentice-Hall, Inc., Englewood Cliffs (1993)
13. Cooper, K.D., Hall, M.W., Kennedy, K.: Procedure cloning. Computer Languages, 96–105 (1992)
14. Chauhan, A., McCosh, C., Kennedy, K., Hanson, R.: Automatic type-driven library generation for telescoping languages. In: SC 2003, vol. 1, pp. 58113–695 (1917)
15. Chambers, C., Ungar, D.: Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. SIGPLAN Not. 24(7), 146–160 (1989)
16. Kennedy, A., Syme, D.: Design and implementation of generics for the .NET Common Language Runtime. In: PLDI 2001, pp. 1–12. ACM, New York (2001)
17. Shankar, A., Sastry, S.S., Bodík, R., Smith, J.E.: Runtime specialization with optimistic heap analysis. SIGPLAN Not. 40(10), 327–343 (2005)
18. Schultz, U., Consel, C.: Automatic program specialization for Java. ACM Trans. Program. Lang. Syst. 25(4), 452–499 (2003)
19. Rigo, A.: Representation-based just-in-time specialization and the Psyco prototype for Python. In: PEPM 2004, pp. 15–26. ACM, New York (2004)
20. Carette, J., Kucera, M.: Partial evaluation of Maple. In: PEPM 2007, pp. 41–50. ACM, New York (2007)
21. Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M.R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E.W., Reitmaier, R., Bebenita, M., Chang, M., Franz, M.: Trace-based just-in-time type specialization for dynamic languages. In: PLDI 2009, pp. 465–478. ACM, New York (2009)
22. Duggan, D., Bent, F.: Explaining type inference. Science of Computer Programming, 37–83 (1996)
23. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: OOPSLA 1996, pp. 324–341. ACM, New York (1996)
24. Tip, F., Palsberg, J.: Scalable propagation-based call graph construction algorithms. In: OOPSLA 2000, pp. 281–293. ACM, New York (2000)
25. Singer, J.: Sparse bidirectional data flow analysis as a basis for type inference. In: Web proceedings of the Applied Semantics Workshop (2004)
26. Joisha, P.G., Banerjee, P.: Correctly detecting intrinsic type errors in typeless languages such as MATLAB. In: APL 2001, pp. 7–21. ACM, New York (2001)
27. Furr, M., An, J.h.D., Foster, J.S., Hicks, M.: Static type inference for Ruby. In: SAC 2009, pp. 1859–1866. ACM, New York (2009)

RATA: Rapid Atomic Type Analysis by Abstract Interpretation – Application to JavaScript Optimization

Francesco Logozzo and Herman Venter

Microsoft Research, Redmond, WA (USA)
{logozzo,hermanv}@microsoft.com

Abstract. We introduce RATA, a static analysis based on abstract interpretation for the rapid inference of atomic types in JavaScript programs. RATA enables aggressive type specialization optimizations in dynamic languages. RATA is a combination of an interval analysis (to determine the range of variables), a kind analysis (to determine if a variable may assume fractional values, or NaN), and a variation analysis (to relate the values of variables). The combination of those three analyses allows our compiler to specialize Float64 variables (the only numerical type in JavaScript) to Int32 variables, providing large performance improvements (up to 7.7×) in some of our benchmarks.

1 Introduction

JavaScript is probably the most widespread programming platform in the world. JavaScript is an object-oriented, dynamically typed language with closures and higher-order functions. JavaScript runtimes can be found in every WEB browser (*e.g.*, Internet Explorer,Firefox, Safari and so on) and in popular software such as Adobe Acrobat and Adobe Flash. Large and complex WEB applications such as Microsoft Office WEB Apps or Google Mail, rely on JavaScript to run inside every browser on the planet.

A fast JavaScript implementation is crucial to provide a good user experience for rich WEB applications and hence enabling their success. Because of its dynamic nature, a JavaScript program cannot statically be compiled to efficient machine code. A fully interpreted solution for JavaScript runtime is generally acknowledged to be too slow for the new generation of web applications. Modern implementations rely on Just-in-time (JIT) techniques: When a function f is invoked at runtime, f is compiled to a function f′ in machine code, and it is then executed. The performance gain of executing f′ pays off the extra time spent in the compilation of f. The quality of the code that the JIT generates for f′ depends on the amount of dynamic and static information that is available to it at the moment of the invocation of f. For instance, if the JIT knows that a certain variable is of an atomic type then it generates specialized machine instructions (*e.g.*, incr for an Int32) instead of relying on expensive boxing/unboxing operations.

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 66–83, 2010.

Motivating Example. Let us consider the `nestedLoops` function in Fig. 1. Without any knowledge of the concrete types of `i` and `j`, the JIT should generate a value wrapper containing: (i) a tag with the dynamic type of the value, and (ii) the value. Value wrappers are disastrous for performance. For instance, the execution of `nestedLoops` takes 310ms on our laptop. [1] In fact, the dynamic execution of the statement `i++` involves: (i) an "unbox" operation to fetch the old value of `i` and check that it is a numerical type; (ii) incrementing `i`; (iii) a "box" operation to update the wrapper with the new value. The JIT can specialize the function if it knows that `i` and `j` are numerical values. In `JavaScript`, the only numerical type is a 64 bits floating point (`Float64`) which follows the IEEE754 standard [16,19]. In our case, a simple type inference can determine that `i` and `j` are `Float64`: they are initialized to zero and only incremented by one. The execution time then goes down to 180ms.

The JIT may do a better job if it knows that `i` and `j` are `Int32`: floating point comparisons are quite inefficient and they usually requires twice or more instructions to perform than integer comparisons on a x86 architecture. A simple type inference does not help, as it cannot infer that `i` and `j` are bounded by 10000. In fact, it is safe to specialize a numerical variable `x` with type `Int32` when one can prove that for all possible executions:

(i) `x` never assumes values outside of the range $[-2^{31}, 2^{31} - 1]$; *and*
(ii) `x` is never assigned a fractional value (*e.g.*, 0.5).

Contribution. We introduce RATA, Rapid Atomic Type Analysis, a new static analysis based on abstract interpretation, to *quickly* and *precisely* infer the numerical types of variables. RATA is based on a combination of an interval analysis (to determine the range of variables), a kind analysis (to determine if a variable may assume fractional values, or `NaN`) and a variation analysis (to relate the values of variables). In our example, the first analysis discovers that $\mathtt{i} \in [0, 10000], \mathtt{j} \in [0, 10000]$ and the second that $\mathtt{i}, \mathtt{j} \in \mathbb{Z}$. Using this information, the JIT can further specialize the code so that `i` and `j` are allocated in integer registers, and as a matter of fact the execution time (inclusive of the analysis time) drops to 31ms!

The function `bitsinbyte` in Fig. 1 (extracted from the SunSpider benchmarks [31]) illustrates the need for the variation analysis. The interval analysis determines that $\mathtt{m} \in [1, 256], \mathtt{c} \in [0, +\infty]$. The kind analysis determines that $\mathtt{m}, \mathtt{c} \in \mathbb{Z}$. If we infer that $\mathtt{c} \leq \mathtt{m}$ then we can conclude that `c` is an `Int32`. In general, we can solve this problem using a relational, or weakly relational abstract domain, such as Polyhedra [12], Subpolyhedra [23], Octagons [26], or Pentagons [24]. However, all those abstract domains have a cost which is quadratic (Pentagons), cubic (Octagons), polynomial (Subpolyhedra) or exponential (Polyhedra) and hence we rejected their use, as non-linear costs are simply not tolerable at runtime. Our variation analysis infers that: (i) `m` and `c` differ by one

[1] The data we report is based on the experience with our own implementation of a `JavaScript` interpreter for `.Net`. More details will be given in Sect. 6.

```
function nestedLoops()
{
  var i, j;
  for(i = 0; i < 10000; i++)
    for(j = 0; j < i; j++) {
      // do nothing...
    }
}
```

```
function bitsinbyte(b) {
  var m = 1, c = 0;
  while(m<0x100) {
    if(b & m) c++;
    m <<= 1;
  }
  return c;
}
```

Fig. 1. Two small `JavaScript` functions showing the impact of type specialization on performance. With no type information the execution of `nestedLoops` takes 310ms, when `i` and `j` are treated as `Float64` it takes 180ms and when they are treated as `Int32` it only takes 31ms. To infer `i`, `j` to be `Int32`, one needs a more powerful analysis than a simple type inference. In `bitsinbyte` one needs to discover that `c` is bounded by `m` in order to determine thas is an `Int32`.

at the entry of the loop; (ii) `c` is incremented by 0 or 1 at each loop iteration; and (iii) `m`, the guard variable of the loop, monotonically increases at each loop iteration (even if non-linearly). As a consequence, $\mathtt{m} \leq 256$ implies that $\mathtt{c} \leq 256$, which combined with the interval and kind information allows the analysis to conclude that `c` is a `Int32`.

The precision of RATA is in between Intervals [10] and Octagons. It is more precise than Intervals, as it can express kind information and relative variable growth. It is less precise than Octagons, for whereas the Octagon abastract domain can exactly represent relations such as $\mathtt{c} \leq \mathtt{m} \wedge \mathtt{m} \leq 256$ our analysis considers the weaker property $\exists \mathtt{x} \in \mathtt{Vars}.\mathtt{c} \neq \mathtt{x} \wedge \mathtt{c} \leq \mathtt{x} \wedge \mathtt{x} \leq 256$. It is worth remarking that RATA is designed to be very fast, to be invoked by the JIT at runtime, and to be used for program optimization.

2 The JavaScript$^{=}$ Language

We illustrate our analysis using a small untyped imperative language, `JavaScript`$^{=}$, defined in Fig. 2, which models the subset of `JavaScript` we consider in our analysis. A program is a sequence of function declarations and a statement (the global statement). For simplicity we assume functions to have only one parameter. Local (global) variables are declared with the `var` (`global`) keyword. The `JavaScript` language does not provide immediate syntax to differentiate globals from locals, which can be easily determined by the parser. The difference is relevant for the soundness of our analysis, so we make the distinction explicit in the syntax of `JavaScript`$^{=}$. Variable assignment, function invocation, statement concatenation, loop, conditional are as usual. The statement $\mathtt{Ignored}_{\mathtt{C}}$ abstracts the language statements which do not affect locals such as object creation, closures and so on. The statement $\mathtt{Havoc}_{\mathtt{C}}$ models any statement that we do not consider in the analysis, and that may have some effect of locals, *e.g.* `throw` and `eval`. To ease the presentation we admit only strict inequalities and equalities for guards. Expressions can be constants or variables, and they are

$$\begin{array}{rl}
\texttt{Prog} & ::= \texttt{F C}\\
\texttt{F} & ::= \texttt{function f(x) \{C\} F} \mid \epsilon\\
\texttt{C} & ::= \texttt{var x;} \mid \texttt{global x;} \mid \texttt{x = e;} \mid \texttt{x = f(e);} \mid \texttt{C C} \mid \texttt{while(b) \{C\};}\\
 & \quad\quad \mid \texttt{if(b) \{C \}else \{C \};} \mid \texttt{Havoc}_{\texttt{C}}\texttt{;} \mid \texttt{Ignored}_{\texttt{C}}\\
\texttt{b} & ::= \texttt{e} < \texttt{e} \mid \texttt{e} \leq \texttt{e} \mid \texttt{e} == \texttt{e}\\
\texttt{e} & ::= k \mid \texttt{x} \mid \texttt{e} + \texttt{e} \mid \texttt{e op}_{num}\texttt{e} \mid \texttt{e op}_{int}\texttt{e} \mid \texttt{Ignored}_{\texttt{e}}\\
k & ::= \texttt{NumericalConstant} \mid \texttt{StringConstant} \mid \texttt{Ignored}_{\texttt{k}}\\
\texttt{op}_{int} & ::= << \mid >> \mid \& \mid \hat{} \qquad \texttt{op}_{num} ::= / \mid * \mid \% \mid -
\end{array}$$

Fig. 2. The syntax of the JavaScript$^{=}$ language

combined with binary operators. We distinguish three kinds of binary operators: (i) sum, +, which can be either the usual IEEE754 addition when its operands are numerical values or string concatenation otherwise; (ii) numerical operations which return a numerical value (or NaN if the operation is undefined, *e.g.* 0/0); (iii) int operations, which always return a Int32 value. The expression $\texttt{Ignored}_{\texttt{e}}$ abstracts the expressions that we do not consider here such as Boolean operators and casting. A constant can either be an IEEE754 64-bits numerical constant, a string literal or some constant we do not deal with (*e.g.*, Boolean constants).

It is worth mentioning that even if in the definition of JavaScript$^{=}$ we ignore some language constructs, our implementation takes care of them *e.g.* by syntax rewriting (“x+ = 2” → “x = x + 2”).

3 Background

3.1 IEEE754 Standard

The IEEE754 standard defines, among other things, the arithmetic format for floating point computations. When using 64-bits (Float64), the standard format allows numbers as large as $\pm 1.7976931348623157 \cdot 10^{308}$ and as small as $\pm 5 \cdot 10^{-324}$ to be represented. All the integers between -2^{53} and 2^{53} are exactly represented. Outside of this interval, one may lose precision in the trailing digits. Unlike machine integers: (i) Float64 numbers do not overflow, and (ii) two special values represent infinities: $\pm\infty$. For instance, $1/0 = +\infty = +\infty + 10$. The Float64 format also specifies a special value NaN (Not-a-Number) as the result of invalid operations, *e.g.*, ∞/∞. A peculiar property of NaN is that $\texttt{NaN} \neq \texttt{NaN}$.

One can specialize a Float64 variable x to a Int32 without changing the semantics of the program if one can prove that x will never assume: (i) a fractional value, a NaN or an infinity; and (ii) a value outside of the range $[-2^{31}, 2^{31} - 1]$. The goal of RATA is to enable such specialization.

3.2 Abstract Interpretation

Abstract interpretation [10,11] is a general theory of semantic approximations. Its more interesting application is to define and prove soundness of program analyses. From the abstract interpretation perspective, a static analysis is a program semantics that is coarse enough to be computable and precise enough to capture the properties of interests. The concrete semantics of a program is defined over a complete lattice $\langle \mathsf{C}, \sqsubseteq \rangle$. The abstract semantics is defined as a fixpoint over a complete lattice $\langle \mathsf{A}, \bar{\sqsubseteq} \rangle$, which is related to C by a Galois connection, *i.e.*, a pair of monotonic functions $\langle \alpha, \gamma \rangle$ such that $\forall \mathsf{c} \in \mathsf{C}.\ \mathsf{c} \sqsubseteq \gamma \circ \alpha(\mathsf{c})$ and $\forall \bar{\mathsf{a}} \in \mathsf{A}.\ \alpha \circ \gamma(\bar{\mathsf{a}}) \bar{\sqsubseteq} \bar{\mathsf{a}}$. We write $\langle \mathsf{C}, \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathsf{A}, \bar{\sqsubseteq} \rangle$ to denote that. An abstract transfer function $\bar{\tau}$ is a sound approximation of a concrete τ if $\forall \bar{\mathsf{a}} \in \mathsf{A}.\ \alpha \circ \tau \circ \gamma(\bar{\mathsf{a}}) \bar{\sqsubseteq} \bar{\tau}(\bar{\mathsf{a}})$. In general, the abstract domain A may contain strictly increasing infinite (or very long) chain. Hence, to ensure the convergence of fixpoint iterations one should use a widening operator, which extrapolates the limit of the sequence. Precision lost by the widening can be recovered using a narrowing operator.

4 Numerical Abstract Domains

The Rapid Atomic Type Analysis (RATA) is meant to be used in an online context, as an oracle for the JIT that can use the inferred types to generate more specialized code. RATA is a combination of three different static analyses. An interval analysis to determine the range of the variables. A kind analysis to infer if a variable can assume a fractional or a `NaN` value. A variation analysis to infer loose relationships about program variables, and hence refine the ranges and the kinds. The analysis should be very fast, to avoid causing untoward pauses in normal program execution. We rejected the use of expressive yet expensive numerical abstract domains. For instance, Octagons have a cubic complexity (in the number of program variables), Polyhedra are exponential, and Subpolyhedra lay in between.

4.1 Extended Intervals

The interval abstract domain was introduced by Cousot & Cousot in [10] as example of the application of Abstract Interpretation to program optimization (specifically array bounds check removal). Inspired by this idea, we use it for type specialization. Our extended intervals are a little bit different from the originals, in that we also consider intervals potentially containing `NaN`, intervals abstracting non-numerical values, intervals abstracting floats and intervals bounded *only* by `Int32` values. An interval can either be the empty interval, the interval containing only `NaN`, a `Int32`-bounded interval, an open interval or the unknown interval ($\top_i$):

$$\begin{aligned}\mathsf{Intv} = {} & \bot_i \mid \mathsf{NaN} \mid \mathsf{Normal}(a,b) \mid \mathsf{OpenLeft}(b) \mid \mathsf{OpenRight}(a) \mid \top_i \\ & a, b \in \mathtt{Int32}\end{aligned}$$

More formally, the meaning of an interval is given by the concretization function $\gamma_i \in [\mathsf{Intv} \to \mathcal{P}(\mathsf{Val})]$. The set Val is the set of concrete $\mathtt{JavaScript}^{=}$ values. We are interested only in numerical values, so we let $\mathsf{Val} = \mathsf{Ignored}_{\mathsf{Val}} \cup \mathbb{R} \cup \{\pm\infty, \mathtt{NaN}\}$. For simplicity, we let $\mathbb{R}^* = \mathbb{R} \cup \{\pm\infty, \mathtt{NaN}\}$, and we extend the usual axioms over reals so that $\forall r \in \mathbb{R}. -\infty < r < +\infty$ and $\forall r \in \mathbb{R}^*.\ r \neq \mathtt{NaN}$. The concretization function and the induced order $\sqsubseteq_i$ are in Fig. 3.

$$
\begin{aligned}
\gamma_i(\bot_i) &= \emptyset\\
\gamma_i(\mathsf{NaN}) &= \{\mathtt{NaN}\}\\
\gamma_i(\mathsf{Normal}(a,b)) &= \{r \mid r \in \mathbb{R}, a \le r \le b\} \cup \{\mathtt{NaN}\}\\
\gamma_i(\mathsf{OpenLeft}(b)) &= \{r \mid r \in \mathbb{R}, r \le b\} \cup \{\mathtt{NaN}\}\\
\gamma_i(\mathsf{OpenRight}(a)) &= \{r \mid r \in \mathbb{R}, a \le r\} \cup \{\mathtt{NaN}\}\\
\gamma_i(\top_i) &= \mathsf{Val}
\end{aligned}
$$

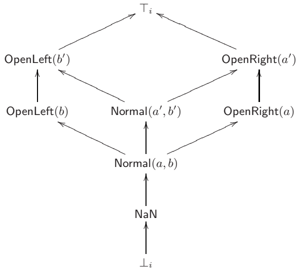


Fig. 3. The concretization γ_i and the order $\sqsubseteq_i$ on the extended intervals. We assume that $a' \le a \le b \le b'$.

Example 1. $\gamma_i(\mathsf{OpenRight}(10)) = \{10 \ldots 11 \cdots + \infty\} \cup \{\mathtt{NaN}\}$.

The abstraction function $\alpha_i \in [\mathcal{P}(\mathsf{Val}) \to \mathsf{Intv}]$ is defined as

$$
\alpha_i(R) = \bigsqcup_i \{\dot{\alpha}_i(r) \mid r \in R\} \text{ where}
$$

$$
\dot{\alpha}_i(r) = \begin{cases}
\mathsf{NaN} & r \text{ is } \mathtt{NaN}\\
\mathsf{OpenRight}(2^{31}-1) & 2^{31}-1 < r \le +\infty\\
\mathsf{OpenLeft}(-2^{31}) & -\infty \le r < -2^{31}\\
\mathsf{Normal}(\mathsf{floor}(r), \mathsf{ceiling}(r)) & -2^{31} \le r \le 2^{31}-1\\
\top_i & \text{otherwise}
\end{cases}
$$

($\mathsf{floor}(r) = \max\{x \in \mathbb{Z} \mid x \le r\}$ and $\mathsf{ceiling}(r) = \min\{x \in \mathbb{Z} \mid r \le x\}$).

Example 2. $\alpha_i(\{10.3, +\infty, \mathtt{NaN}\}) = \mathsf{OpenRight}(10)$, $\alpha_i(\{3.14\}) = \mathsf{Normal}(3,4)$.

Theorem 1. $\langle \mathcal{P}(\mathsf{Val}), \subseteq \rangle \underset{\alpha_i}{\overset{\gamma_i}{\leftrightarrows}} \langle \mathsf{Intv}, \sqsubseteq_i \rangle$.

It is worth noting that for Th. 1 to hold we need to map ±∞ to the smallest abstract element containing ±∞.

The abstract domain Intv is precise enough to capture that the value of a variable is always within the `Int32` range, but it cannot capture the fact that a variable never assumes fractional values, crucial for soundness : *e.g.*, 1/2 is 0.5 with `Float64` semantics and 0 with `Int32` semantics.

Example 3. For the function `bitsinbyte`, the analysis with Intv infers that `m` : $\mathsf{Normal}(0, 512)$, `c` : $\mathsf{OpenRight}(0)$, `b` : $\top_i$.

4.2 Kinds

The elements of the Kind abstract domain are either the empty kind, a 32-bits integer, a 64-bit floating point number or an unknown kind of value:

$$
\mathsf{Kind} = \bot_k \mid \mathtt{Int32} \mid \mathtt{Float64} \mid \top_k .
$$

$$\begin{aligned}
\gamma_k(\bot_k) &= \emptyset \\
\gamma_i(\texttt{Int32}) &= \{r \mid r \in \mathbb{Z}, -2^{31} \leq r \leq 2^{31} - 1\} \\
\gamma_i(\texttt{Float64}) &= \{r \mid r \in \mathbb{R}^*, r \text{ is a 64 bits IEEE 754 number}\} \\
\gamma_i(\top_k) &= \mathsf{Val}
\end{aligned}$$

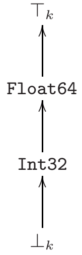


Fig. 4. The concretization γ_k and the order $\sqsubseteq_k$ of the Kinds

The meaning function $\gamma_k \in [\mathsf{Kind} \to \mathcal{P}(\mathsf{Val})]$ and the induced order $\sqsubseteq_k$ are in Fig. 4. The abstraction function $\alpha_k \in [\mathcal{P}(\mathsf{Val}) \to \mathsf{Kind}]$ is

$$\alpha_k(R) = \bigsqcup_k\{\dot{\alpha_k}(r) \mid r \in R\} \;\text{ where }\; \dot{\alpha_k}(r) = \begin{cases} \texttt{Int32} & r \text{ is a } \texttt{Int32} \\ \texttt{Float64} & r \text{ is a } \texttt{Float64} \\ \top_k & r \text{ otherwise} \end{cases}$$

Example 4. $\alpha_k(\{10.3, +\infty, \texttt{NaN}\}) = \alpha_k(\{3.14\}) = \texttt{Float64}$.

Theorem 2. $\langle \mathcal{P}(\mathsf{Val}), \subseteq\rangle \underset{\alpha_k}{\overset{\gamma_k}{\leftrightarrows}} \langle \mathsf{Kind}, \sqsubseteq_k\rangle$.

The abstract domain of Kind in isolation is of almost no use (maybe except for trivial, loop free programs). In the `nestedloops` example, knowing that `i` is initialized to a `Int32`, it is compared to a `Int32`, and only incremented by one it is not enough to deduce that `i` is an `Int32`. In fact if the loop guard were instead $\texttt{i}\leq 2^{31}$, then after the last iteration of the loop $\texttt{i} = 2^{31} + 1$ which is a fine `Float64` value, but not an `Int32`.

4.3 K-Intervals

The combination of extended intervals and kinds allow the derivation of very powerful yet rapid analyses. We call the reduced product of Kind and Intv a k-interval. The elements of the abstract domain are pairs in $\mathsf{Intv} \times \mathsf{Kind}$, and the concretization $\gamma_{ki} \in [\mathsf{Intv} \times \mathsf{Kind} \to \mathcal{P}(\mathsf{Val})]$ is $\gamma_{ki}(\langle i, k\rangle) = \gamma_i(i) \cap \gamma_k(k)$. The abstraction $\alpha_{ki} \in [\mathcal{P}(\mathsf{Val}) \to \mathsf{Intv} \times \mathsf{Kind}]$ is the simple pairwise abstraction: $\alpha_{ki}(R) = \langle \alpha_i(R), \alpha_k(R)\rangle$. The order $\sqsubseteq_{ki}$ is the pairwise extension of the order on the basic domains. We write $\texttt{x} : t$ to denote that the variable `x` has a k-interval t.

Theorem 3. $\langle \mathcal{P}(\mathsf{Val}), \subseteq\rangle \underset{\alpha_{ki}}{\overset{\gamma_{ki}}{\leftrightarrows}} \langle \mathsf{Intv} \times \mathsf{Kind}, \sqsubseteq_{ki}\rangle$.

K-Intervals are more expressive than the single domains and can represent addition information, crucial to type specialization:

```
function loop() {
  var x;
  x = 0;
  while(x < 10000) {
    x = x + 1;
  }
}
```

```
function loopToN(n) {
  var x;
  x = 0;
  while(x < n) {
    x = x + 1;
  }
}

loopToN(99999);
loopToN(1234);
```

Fig. 5. In order to infer that `x : Int32`, in the first example RATA uses a widening with a threshold, and in the second a narrowing as re-execution. The function `loopToN` is analyzed at the first invocation, it is specialized for `Int32`, and the specialization is re-used at the second invocation.

Example 5. The k-interval $t = \langle \mathsf{OpenRight}(10), \mathtt{Int32} \rangle$ represents the set of `Int32` larger than or equal to 10:

$$\begin{aligned}\gamma_{ki}(t) = \{10 \dots 11 \dots + \infty, \mathtt{NaN}\} \cap \{r \mid r \in \mathbb{Z}, -2^{31} \leq r \leq 2^{31} - 1\} \\ = \{10, 11 \dots 2^{31} - 1\}.\end{aligned}$$

It is worth noting that $t' = \langle \mathsf{Normal}(10, 2^{31} - 1), \mathtt{Int32} \rangle$ is such that $\gamma_{ki}(t') = \gamma_{ki}(t)$, so that t and t' are two abstract elements with the same concretization. To keep a low analysis overhead, we do *not* impose a canonical form for abstract elements.

Roughly, if the analysis determines that $\mathtt{x} : t$ then there exists a variable `y`, which is known to be an `Int32`, such that $\mathtt{x} \leq \mathtt{y}$. This information is weaker than that one gets for instance with Octagons, which automatically discovers the particular `y` and $v \geq 0$ such that $\mathtt{x} \leq \mathtt{y} - v$.

5 Rapid Atomic Type Analysis

The Rapid Atomic Type Analysis is defined by structural induction on the program syntax. It has two main phases: (i) numerical invariant inference with $\mathsf{Intv} \times \mathsf{Kind}$; and (ii) type refinement via variation analysis.

5.1 Numerical Analysis

The numerical invariant analysis $\mathbb{N}[\![\cdot]\!]$ infers, for each program point an abstract state $\sigma \in \Sigma = [\mathtt{Vars} \rightarrow \mathsf{Intv} \times \mathsf{Kind}]$, that is a map from variables to k-intervals.

Invocation of the Analysis. When the JIT encounters a function call $\mathtt{f}(v)$, where v is a value of dynamic type t, it first searches the cache to see if it has already specialized `f` for the type t. If this is not the case, it invokes RATA to infer the atomic numerical types for `f`'s locals, to be used for type specialization.

$$
\begin{aligned}
\mathsf{eval}(k,\sigma) &= \begin{cases} \langle \mathsf{Normal}(k,k), \mathtt{Int32}\rangle & k \text{ is } \mathtt{Int32} \\ \langle \top_i, \mathtt{Float64}\rangle & k \text{ is } \mathtt{Float64} \\ \langle \top_i, \top_k\rangle & \text{otherwise}\end{cases}\\
\mathsf{eval}(\mathtt{x},\sigma) &= \sigma(\mathtt{x})\\
\mathsf{eval}(\mathsf{e}_1 + \mathsf{e}_2,\sigma) &= \mathsf{let}\ v_1 = \mathsf{eval}(\mathsf{e}_1,\sigma), v_2 = \mathsf{eval}(\mathsf{e}_2,\sigma)\ \mathsf{in}\\
&\ \mathsf{if}\ v_1 == \langle\top_i,\top_k\rangle \vee v_2 == \langle\top_i,\top_k\rangle \mathsf{then}\ \langle\top_i,\top_k\rangle\\
&\ \mathsf{else}\ (v_1 \bar{+} v_2) \sqcap_{ki} \langle\top_i,\mathtt{Float64}\rangle\\
\mathsf{eval}(\mathsf{e}_1\ \mathsf{op}_{num}\ \mathsf{e}_2,\sigma) &= (\mathsf{eval}(\mathsf{e}_1,\sigma)\bar{\mathsf{op}}_{num}\ \mathsf{eval}(\mathsf{e}_2,\sigma)) \sqcap_{ki} \langle\top_i,\mathtt{Float64}\rangle\\
\mathsf{eval}(\mathsf{e}_1\ \mathsf{op}_{int}\ \mathsf{e}_2,\sigma) &= (\mathsf{eval}(\mathsf{e}_1,\sigma)\bar{\mathsf{op}}_{int}\ \mathsf{eval}(\mathsf{e}_2,\sigma)) \sqcap_{ki} \langle\top_i,\mathtt{Int32}\rangle\\
\mathsf{eval}(\mathtt{f}(\mathsf{e})) &= \top_i
\end{aligned}
$$

Fig. 6. The abstract evaluation of expressions. The abstract operators $\bar{+}$, $\bar{\mathsf{op}}_{num}$, $\bar{\mathsf{op}}_{int}$ are the abstract counterparts of concrete the concrete operators.

Initial State. At the entry point of `f`, the global values are set to $\langle\top_i,\top_k\rangle$ (any value), the local values are set to $\langle\bot_i,\bot_k\rangle$ (uninitialized), and the actual value v of the parameter `x` is generalized to $\langle\top_i,\alpha_k(\{v\})\rangle$. We generalize the actual value of the parameter so that the result of the analyses can be re-used.

Example 6. The initial abstract state for the analysis of `loopToN`(99999) in Fig. 5 is $\sigma_0 = [\mathtt{n} \mapsto \langle\top_i,\mathtt{Int32}\rangle, \mathtt{x} \mapsto \langle\bot_i,\bot_k\rangle]$. The specialization of `loopToN` can be cached and reused for `loopToN`(1234) as σ_0 is an over-approximation of $[\mathtt{n} \mapsto \langle\mathsf{Normal}(1234,1234),\mathtt{Int32}\rangle, \mathtt{x} \mapsto \langle\bot_i,\bot_k\rangle]$.

Variables. RATA is a modular analysis, run on a per-method basis. In the general case, at the moment of the invocation of RATA, we have not seen all the assignments to globals, so that the only sound assumption for globals is the open-world assumption, *i.e.*, they can assume any value.

Assignment. An assignment $\mathtt{x} = \mathsf{e}$ in a pre-state σ_0, updates the entry for `x` with $\mathsf{eval}(\mathsf{e},\sigma_0)$ if `x` is a local variable (or a parameter) or it does nothing otherwise. The evaluation function $\mathsf{eval} \in [\mathsf{e} \times \Sigma \rightarrow \mathsf{Intv} \times \mathsf{Kind}]$ is in Fig. 6. The k-interval for a constant is assigned according to its type. The "+" operator is polymorphic in `JavaScript`: it can either be string concatenation or numerical addition. As a consequence, if no information on the operands is available, nothing can be inferred on the result. Otherwise, we know that it is at least a `Float64`. The result of a op_{num} (op_{int}) is at least a `Float64` (an `Int32`). The return value of a function call is ignored: to statically determine which function is invoked requires a quite complex global program analysis, out of the scope of this paper.

Test. A precise handling of tests enables the refinement of the abstract states, and hence a more precise analysis. A *too* precise analysis of tests (*e.g.*, using forward/backwards iterations [9]) may cause slowdowns unacceptable for an online analysis. In our implementation we only consider comparisons between a variable and an expression, or between two variables. For equalities, we have that:

$$
\begin{aligned}
&\mathbb{N}[\![\mathtt{x} == \mathsf{e}]\!](\sigma_0) = \sigma_0[\mathtt{x} \mapsto \sigma_0(\mathtt{x}) \sqcap_{ki} \mathsf{eval}(\mathsf{e},\sigma_0)], \text{ and}\\
&\mathbb{N}[\![\mathtt{x} == \mathsf{y}]\!](\sigma_0) = \sigma_0[\mathtt{x},\mathsf{y} \mapsto \sigma_0(\mathtt{x}) \sqcap_{ki} \sigma_0(\mathsf{y})].
\end{aligned}
$$

For an inequality $\mathtt{x} < \mathtt{e}$, the upper bound of $\mathtt{x}$ is refined by the upper bound of $\mathsf{eval}(\mathtt{e}, \sigma_0)$ ($\mathsf{upp}(\langle i, t\rangle)$ is the open k-interval bounded by the upper bound of i, it can be $+\infty$):

$$\mathbb{N}[\![\mathtt{x} < \mathtt{e}]\!](\sigma_0) = \sigma_0[\mathtt{x} \mapsto \sigma_0(\mathtt{x}) \sqcap_{ki} \mathsf{upp}(\mathsf{eval}(\mathtt{e}, \sigma_0))].$$

Similarly for an inequality $\mathtt{x} < \mathtt{y}$, the upper bound for $\mathtt{x}$ can be refined by the upper bound of $\mathtt{y}$, and the lower bound of $\mathtt{y}$ can be refined by the lower bound of $\mathtt{x}$:

$$\mathbb{N}[\![\mathtt{x} < \mathtt{y}]\!](\sigma_0) = \sigma_0[\mathtt{x} \mapsto \sigma_0(\mathtt{x}) \sqcap_{ki} \mathsf{upp}(\mathsf{eval}(\mathtt{y}, \sigma_0)), \mathtt{y} \mapsto \sigma_0(\mathtt{y}) \sqcap_{ki} \mathsf{low}(\mathsf{eval}(\mathtt{x}, \sigma_0))].$$

Example 7. Let us assume that $\sigma_0 = [\mathtt{x} \mapsto \langle \mathsf{OpenRight}(10), \top_i\rangle]$. Then:

$$\mathbb{N}[\![\mathtt{x} < \mathtt{1000}]\!](\sigma_0) = [\mathtt{x} \mapsto \langle \mathsf{Normal}(10, 1000), \top_i\rangle].$$

Note that it would be unsound to assume that $\mathtt{x}$ is an `Int32` or that $\mathtt{x} : \mathsf{Normal}(10, 999)$.

Sequence. The analysis of a sequence of statements is the composition of the analyses: $\mathbb{N}[\![\mathtt{C}_1\ \mathtt{C}_2]\!](\sigma_0) = \mathbb{N}[\![\mathtt{C}_2]\!](\mathbb{N}[\![\mathtt{C}_1]\!](\sigma_0))$.

Conditional. For a conditional the analysis first refines the pre-state with the guards, and then joins the results (the function `Not` negates the Boolean expression `b`):

$$\mathbb{N}[\![\mathtt{if(b)}\ \{\mathtt{C}_1\}\mathtt{else}\ \{\mathtt{C}_2\};]\!](\sigma_0) = \mathbb{N}[\![\mathtt{C}_1]\!](\mathbb{N}[\![\mathtt{b}]\!](\sigma_0)) \sqcup_{ki} \mathbb{N}[\![\mathtt{C}_2]\!](\mathbb{N}[\![\mathtt{Not(b)}]\!](\sigma_0)).$$

Loop. A loop invariant for `while(b) {C};` is a fixpoint of the functional $F \in [\Sigma \to \Sigma]$:

$$F(X) = \sigma_0 \dot{\sqcup}_{ki} \mathbb{N}[\![\mathtt{C}]\!](\mathbb{N}[\![\mathtt{b}]\!](X)),$$

where σ_0 is the abstract state at the entry point of the loop and $\dot{\sqcup}_{ki}$ is the pointwise extension of $\sqcup_{ki}$. An invariant can be computed with the usual fixpoint iteration techniques. The abstract domain $\mathsf{Intv} \times \mathsf{Kind}$ does not contain infinite ascending chains, but it contains very very long chains (up to $2^{32} + 3$ elements). We need a widening operator to speed up the convergence of the iterations to a post-fixpoint. A widening with thresholds [6,22], and the re-execution from a post-fixpoint (a form of narrowing [22]) guarantee a good precision yet providing good performance. We illustrate those two techniques with examples.

Example 8. The iterations with the classical widening for the `loop` function of Fig. 5 produce the following sequence of abstract values for `x`:

$$\langle \mathsf{Normal}(0, 0), \mathtt{Int32}\rangle \sqsubseteq_{ki} \langle \mathsf{Normal}(0, 1), \mathtt{Int32}\rangle \sqsubseteq_{ki} \langle \mathsf{OpenRight}(0), \mathtt{Float64}\rangle,$$

as the upper bound for `x` is extrapolated to $+\infty$. The threshold (or staged) widening tries to extrapolate the upper bound to constants appearing in guards, producing the sequence:

$$\langle \mathsf{Normal}(0, 0), \mathtt{Int32}\rangle \sqsubseteq_{ki} \langle \mathsf{Normal}(0, 1), \mathtt{Int32}\rangle \sqsubseteq_{ki} \langle \mathsf{Normal}(0, 10000), \mathtt{Int32}\rangle.$$

In general, during the analysis we collect all the constants that appear in the tests, and we use them as steps for widening with a threshold.

Example 9. The type of x in function loopToN of Fig. 5 depends on the input parameter. When it is invoked with an Int32 value, then we would like RATA to discover that x is an Int32. Widening with thresholds is of no help here (there are no constants in guards) so the iterations stabilize at $I = \langle \mathsf{OpenRight}(0), \mathtt{Float64} \rangle$. A re-execution of the loop with initial state I will refine the abstract state to $\langle \mathsf{OpenRight}(0), \mathtt{Int32} \rangle$.

Re-execution is justified by Tarski's fixpoint theorem [29], which states that in a partial order $\mathsf{lfp}(F) = \sqcap \{ I \mid F(I) \sqsubseteq I \}$. So, if I is a post-fixpoint for F, then $F(I)$ is still above the least fixpoint $\mathsf{lfp}(F)$, and hence it is a sound approximation of the loop invariant. During re-execution, we refine the abstract semantics of the tests appearing in *loops* which involve inequalities where one of the operands is an Int32. For instance in the loopToN example:

$$\mathbb{N}[\![\mathtt{x} < \mathtt{n}]\!]([\mathtt{x} \mapsto \langle \mathsf{OpenRight}(0), \mathtt{Float64} \rangle]) = [\mathtt{x} \mapsto \langle \mathsf{Normal}(0, 2^{31} - 2), \mathtt{Int32} \rangle].$$

In Ex. 7 we pointed out that in general it is not sound to assume $\mathtt{x} : \mathtt{Int32}$ after a test $\mathtt{x} < \mathtt{y}$ when $\mathtt{y} : \mathtt{Int32}$. However during re-execution this is sound as there are essentially three cases why $\mathtt{x} : \mathtt{Float64}$ in I: (i) x was a Float64 at the loop entry; (ii) x may be assigned a fractional value (or NaN or an infinite) in the loop body; or (iii) the analysis of the loop could not figure out that $\mathtt{x} : \mathtt{Int32}$. In the first two cases, $F(I)$ will imply that $\mathtt{x} : \mathtt{Float64}$ (because of the definition of F). In the third case one may hope to recover some of the lost precision. In our running example:

$$\begin{aligned} F(I) &= [\mathtt{x} \mapsto \langle \mathsf{Normal}(0, 0), \mathtt{Int32} \rangle] \dot{\sqcup}_{ki} [\mathtt{x} \mapsto \langle \mathsf{Normal}(1, 2^{31} - 1), \mathtt{Int32} \rangle] \\ &= [\mathtt{x} \mapsto \langle \mathsf{Normal}(0, 2^{31} - 1), \mathtt{Int32} \rangle] \ \dot{\sqsubseteq}_{ki} \ I. \end{aligned}$$

(Recall that $\gamma_{ki}(\langle \mathsf{Normal}(0, 2^{31} - 1), \mathtt{Int32} \rangle) = \gamma_{ki}(\langle \mathsf{OpenRight}(0), \mathtt{Int32} \rangle)$).

Ignored Statements and Havoc. Ignored statements have no effect on the local state, so the analysis treats them as the identity : $\mathbb{N}[\![\mathtt{Ignored}_{\mathtt{C}}]\!](\sigma_0) = \sigma_0$. Havoc statements may have some side-effect on local variables. We abstract them by $\mathbb{N}[\![\mathtt{Havoc}_{\mathtt{C}}]\!](\sigma_0) = \langle \top_i, \top_k \rangle$.

5.2 Variation Analysis

The numerical analysis alone cannot determine that $\mathtt{c} : \mathtt{Int32}$ in bitsinbyte (Fig. 1). It discovers the loop invariant $\sigma_L = [\mathtt{m} \mapsto \langle \mathsf{Normal}(1, 512), \mathtt{Int32} \rangle, \mathtt{c} \mapsto \langle \mathsf{OpenRight}(0), \mathtt{Float64} \rangle]$ (we omit b). The invariant σ_L can be refined by the variation analysis. At the loop entry, c and m differ by one. At each iteration c is either incremented by one or it remains the same, whereas m is multiplied by 2, thus m grows faster than c. However, m bounded implies that c should be bounded too, thus we can safely refine σ_L to $\sigma_L[\mathtt{c} \mapsto \langle \mathsf{Normal}(0, 512), \mathtt{Int32} \rangle]$.

We run the variation analysis $\mathbb{V}[\![\cdot]\!]$ on a *per-loop* basis. The goal of the analysis is to compute, for each loop and each variable an interval over-approximating the increment of a variable in a *single* loop iteration. The variation analysis is similar in many aspects to the numerical analysis above, with the major difference that the initialization and the assignments are re-interpreted. An abstract state is a map from local variables to intervals. At the loop entry point, all the local variables are set to the interval $[0, 0]$ [2] (no increment). For assignments, we compute variable increments. We consider simple forms of increments and decrements, and we abstract away all the other expressions. So, we let $\mathbb{V}[\![\mathtt{x} = \mathtt{x} \pm k]\!](\sigma_0) = \sigma_0[\mathtt{x} \mapsto \pm[k, k]]$, and $\mathbb{V}[\![\mathtt{x} = \mathsf{e}]\!](\sigma_0) = \sigma_0[\mathtt{x} \mapsto [-\infty, +\infty]]$.

Once we have computed ν, the increment ranges for the variables in the loop, we use this information to refine the numerical loop invariant σ_L to σ'_L according to refinement rules that looks like:

$$\forall \mathtt{x}, \mathtt{y}.\ \mathtt{x} \neq \mathtt{y} \wedge \sigma_0 \models \mathtt{x} < \mathtt{y} \wedge \mathtt{y} \text{ is upper-bounded by } b \wedge \nu \models \mathtt{x} < \mathtt{y}$$
$$\Longrightarrow \sigma'_L(\mathtt{x}) = \sigma_L(\mathtt{x}) \dot{\sqcap}_{ki} \langle \mathsf{OpenLeft}(b), \top_k \rangle,$$

(the intuitive meaning of $\sigma_0 \models \mathtt{x} < \mathtt{y}$ is that in the k-interval σ_0, $\mathtt{x} < \mathtt{y}$ and the meaning of $\nu \models \mathtt{x} < \mathtt{y}$ is that according to ν, `x` grows slower than `y`). The rule above essentially states that if `y` is an upper bound for `x` at the entry of the loop, and `y` is bounded by b during all the executions of the loop, and `x` does not grow more than `y` in the loop, then b should be an upper bound for `x` too. We omit all the other (tedious) refinement rules, which consider the combination of the other cases (*e.g.*, $\sigma_0 \models \mathtt{x} \leq \mathtt{y}$, lower bounds, decrements and so on).

5.3 Atomic Types

The atomic types T for a function are obtained by joining together the post-states of all the statements in the function body. The reason for that is that we want to assign a *unique* atomic type at each local variable. One may wonder why we designed a flow-sensitive analysis if we were interested in a flow-insensitive property (the type of a local variable through all the function's body). Actually, in an early stage of this project we tried to avoid the joining phase by designing a flow-insensitive analysis. For instance, the abstract semantics of the sequence was $\mathbb{N}[\![\mathtt{C}_1\ \mathtt{C}_2]\!](\sigma_0) = \text{let } \sigma = \mathbb{N}[\![\mathtt{C}_1]\!](\sigma_0) \text{in } \mathbb{N}[\![\mathtt{C}_2]\!](\sigma) \dot{\sqcup}_{ki} \sigma$. We immediately realized that a flow-insensitive analysis was too imprecise for handling loops, and in particular it voided the advantages of the re-execution step and the variation analysis which we found crucial for precision. Therefore, we rejected the flow-insensitive analysis for a flow-sensitive followed by a join-all step.

6 Experiments

We have implemented RATA in our `JavaScript` engine for `.Net`. The engine itself is written in C#. It parses the `JavaScript` source, it compiles the main

[2] We use the notation $[a, b]$ to avoid confusion between the range intervals of the previous sections and the increment intervals. In the implementation we share the code, though.

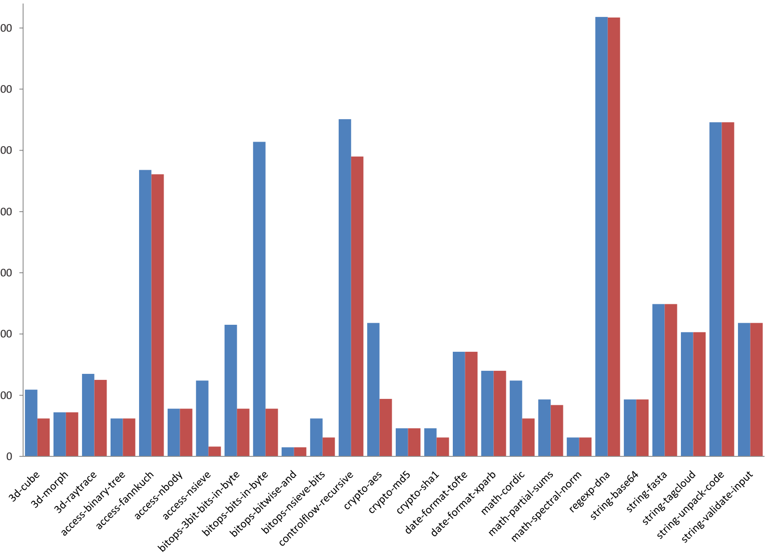


Fig. 7. The results of the optimizations enabled by a text-book type inference algorithm (blue/light bars) and RATA (red/dark bars). Times are expressed in milliseconds. On numerical intensive benchmarks RATA enables up to a 7.7× speed-up.

(global) function and it generates proxies for function invocations. When the execution encounters a function proxy, the `JavaScript` engine resolves it, and it checks if it has a specialized version in the cache which matches the actual parameters. If this is the case, then it executes the cached version. Otherwise: (i) it runs the RATA to infer the atomic types for the locals of the variables; (ii) it compiles the function in memory, performing atomic type specialization; (iii) executes the specialized function, and caches it for future needs. It is worth noting that the specialization is polymorphic: If the same function is invoked at two points of time with two actual parameters of different types, then it is analyzed and specialized twice.

We report the experience of applying RATA on the SunSpider `JavaScript` benchmarks [31]. The SunSpider benchmarks measure `JavaScript` performance for problems that presents difficulties to `JavaScript` implementations. They are designed to be balanced and to stress different areas of the language. They are commonly used to compare the `JavaScript` performance of different browsers, or different versions of the same browser. We run the experiments on a 2.1GHz Centrino Duo Laptop, 4Gbyte, under Windows 7 and `.Net` v3.5. We compared a text-book type inference algorithm [2] with RATA. The type inference algorithm determines which locals are definitely doubles, and for some expressions it can also infer that a local is an `Int32`.

The results of our experiments are in Fig. 7. Measuring the performances of managed programs is quite complex, as their runtime behavior depends on too many variables [18]. In general, when the execution time is too low, it is

```
global x;
x = 0;
while(x < 4) {
  foo(x);
  x = x + 1;
}
```

```
function zeroarray(arr) {
  var x; x = 0;
  while(x < arr.length) {
    arr[x] = 0; x = x + 1; }
}

global a; a = new Array(10);
zeroarray(a)
```

Fig. 8. Two code snippets in which it would be unsound to infer that x : Int32. In the first case, x is declared in the global scope and its value can be changed by foo. In the second case, x depends on the property arr.length which in general is a UInt32. Furthermore, JavaScript allows the user-redefinition of Array, so that we need a global analysis to determine that a is an array.

impossible to distinguish the *effective* time spent in computation from the external noise (*e.g.*, the garbage collector, the thread scheduler, network traffic, background services and so on). We run each JavaScript program in the SunSpider suite 80 times choosing the best execution time. The execution times of Fig. 7 do not include the compilation and the type inference/RATA time. The reason for that is that we observed the analysis time to be of the same order of magnitude of the experiment noise (few tenths of milliseconds). We also observed that the runtime costs of the type inference and RATA were comparable. We modified some tests so to have them run longer, reducing the external noise, and hence obtaining more meaningful measurements.

The results of Fig. 7 show that in 12 tests RATA enables the JIT to generate more optimized code, and hence to obtain significant performance improvements.

Most of the benchmarks in the 3d family benefit from Int32 type inference. The tests themselves manipulate many doubles (and arrays of doubles), but RATA manages to discover that 20 locals in 3d-cube and 11 locals in 3d-raytrace are Int32 which convey respectively a 1.75× and 1.1× speed-up over the double-only version. We inspected the results of the analysis, and we found that in the first test RATA found all the Int32 variables one may expect, and in the second test it missed three. The reason for that was in an imprecision of handling the return statement. Finally, the locals on 3d-morph depends on some global values, so nothing can be inferred about them.

The best performance improvements are in the bitops family benchmarks. RATA discovers that all the local variables are Int32 in the test bitops-bits-in-byte, which provides a 6.6× speed-up with respect to the same test when all the locals are inferred to be doubles. Similar results are observed in the bitops-3bit-bits-in-byte (2.8×) and the bitops-nsieve-bits (2×) tests, where RATA is again precise enough to infer all the Int32 locals. The test bitops-bitwise-and contains only globals, so there is no hope to statically optimize it.

Example 10. The test bitops-bitwise-and contains a main loop that looks like the first code snippet of Ex. 8. In general, it is unsound to infer that x : Int32 as foo may change the value of x. Functions are analyzed top-down: first the JIT runs RATA on the global statement, and then, at the first concrete occurrence, it invokes RATA on foo. As a consequence when inferring the type of x, RATA assumes the worst case for foo. Determining x : Int32 requires a bottom-up purity analysis or an effect analysis [4], which are out-of-the scope of the paper, and in general too expensive to be performed online.

In the access-nsieve benchmark, RATA local inference enables a significant speedup (7.7×) over the Float64-specialized version. In particular, the inner function contains two nested loops and a counter variable. Fixpoint computation with re-execution and variation analysis are cardinal to infer that all the locals involved are indeed Int32. The other two benchmarks of the access family benchmarks perform computations which either depend on globals or on very short loops.

The controlflow-recursive benchmark stresses JavaScript implementations with standard recursive-function benckmarks such as fibonacci or ackerman. RATA infers that the variables inside those functions are Int32 and thus achieves a slight performance improvement (1.12×).

The cryptographic benchmarks benefit by an aggressive type specialization. RATA infers all the Int32 locals for the crypto-aes and the crypto-sha1 benchmarks, enabling a 2.3× and 1.5× speedup. The crypto-md5 benchmark contains many functions taking an array as parameter, and iterating over its elements. The next example shows that it would be unsound to infer those locals to be Int32.

Example 11. Let us consider the zeroarray function of Fig. 8. In JavaScript, the length property of Array is a UInt32, *i.e.*, it can assume values as large as $2^{32} - 1$. As a consequence, even if we know that arr is an array, we cannot conclude x : Int32. In general, to infer that x : Int32, we should refine RATA to track that arr is an array *and* that arr.length $< 2^{31} - 1$. The JavaScript languages allows the redefinition of Array, so we need a global analysis to guarantee that the value of a is actually an array.

The execution time of date and string manipulating benchmarks is heavily dominated by the interaction with the object model, and by other non-numerical computations so that RATA is of no help here.

Math benchmarks manipulate double values, but the inference of some Int32 locals enable up to a 2× speedup in math-cordic, a slight improvement in math-partialsums. For atomic type inference, the test math-spectral-norm looks like crypto-md5, and as a consequence nothing can be statically inferred.

To sum up, RATA is precise enough to infer all (but 3) of the local variables which are Int32 in the SunSpider benchmarks. One may wonder if broadening the analysis to also consider Int64, UInt32 and so on may provide further performance gains. According to the previous experience of the second author with JScript.NET, those cases are so rare, and they complicate so much the implementation and the JIT code generation, that it seems not worthwhile to try.

7 Related Work

Just-in-time compilation is known at least from 1960. In his LISP paper [25], McCarthy sketches the dynamic compilation of functions into machine code, a process fast enough that the compiler's output does not need to be saved. Deutsch and Schiffman introduced in [14] lazy JIT compilation for Smalltalk, where functions were compiled at the first usage, and cached for further usage. The Self programming language influenced the `JavaScript` design. The first Self compiler used a data-flow analysis ("Class analysis") to compute an over-approximation of the set of possible classes that variables might hold instances of and hence to optimize dynamic dispatching [8]. Further versions of the Self compiler introduced more aggressive type analyses [30], but they did not consider the specialization of atomic types as here [1].

The implementation of popular dynamic languages as Python try to optimize the generated code by performing some kind of online static analysis. The JIT compiler of the PyPy system [28] uses "flexswitches" to perform type specialization [13]. Flexswitches are essentially a form of online partial evaluation [21]. Psyco [27] is another implementation of Python which tries to guess `Int32` variables at *runtime*. The tracing JIT generalizes the ideas of Psyco and PyPy. A tracing JIT essentially identifies frequently executed loop traces at runtime, and it dynamically generates specialized machine code [17]. RATA is complementary to a tracing JIT. In his master thesis, Cannon presented a localized atomic type inference algorithm for Python [7]. His analysis is based on the Cartesian product algorithm, and it is less precise than ours. As a consequence, it is not a surprise that his experimental results are less satisfactory than ours. In [3], Anderson *et al.* introduced an algorithm for type inference of `JavaScript` to derive the types of objects. It is unclear if their algorithm is fast enough to be used in dynamic compilation. They did not consider the inference of `Int32` variables which require reasoning on the *values* of variables. In this sense, our work is then complementary to theirs. In [20], Jensen *et al.* presented an abstract interpretation based static analysis to check the absence of common errors in `JavaScript` programs. Their analysis is more oriented to program verification than optimization. However, for numerical values their abstract domain is less precise than ours and so they are not likely to discover all the numerical properties that RATA can discover.

Abstract Interpretation is mainly applied to program verification (*e.g.*, [6,15]) and *offline* program optimization (*e.g.*, [10,5]). To the best of our knowledge this is the first work which applies full-powered Abstract Interpretation techniques (*e.g.*, infinite lattices, widenings and narrowings) to *online* program optimization. We believe that this is a promising line of work.

8 Conclusions

We have presented RATA, a new static analysis, based on abstract interpretation, for the rapid inference of atomic types in dynamic languages. The analysis is a combination of three analyses: a range analysis, a kind analysis and a variation analysis. We formalized the underlying abstract domains and we related them to the concrete values via Galois connections. We described the analysis, and we reported the results of the atomic type specialization on the SunSpider `JavaScript` benchmarks (the industrial standard for comparing `JavaScript` implementations). We observed that: (i) RATA is precise enough to infer all the `Int32` locals that one may hope to infer statically; and (ii) the `Int32`-specialization produces remarkable performance improvements in most tests (up to a 7.7× speed-up for numerical intensive ones).

For the future, we plan to extend RATA to whole program analysis, and in particular to apply it to the wider goal of program verification.

References

1. Agesen, O., Hölzle, U.: Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In: OOPSLA 1995. ACM Press, New York (1995)
2. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading (1986)
3. Anderson, C., Giannini, P., Drossopoulou, S.: Towards type inference for javascript. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 428–452. Springer, Heidelberg (2005)
4. Barnett, M., Fähndrich, M., Garbervetsky, D., Logozzo, F.: Annotations for (more) precise points-to analysis. In: IWACO 2007 (2007)
5. Blanchet, B.: Escape Analysis: Correctness proof, implementation and experimental results. In: POPL 1998 (1998)
6. 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: PLDI 2003. ACM Press, New York (2003)
7. Cannon, B.: Localized type inference of atomic types in Python. Master's thesis, California Polytechnic State University (2005)
8. Chambers, C., Ungar, D.: Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In: PLDI 1989. ACM Press, New York (1989)
9. Cousot, P.: The calculational design of a generic abstract interpreter. In: Calculational System Design. NATO ASI Series F. IOS Press, Amsterdam (1999)
10. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977 (1977)
11. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL 1979 (1979)
12. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL 1978 (1978)
13. Cuni, A., Ancona, D., Rigo, A.: Faster than C#: Efficient implementation of dynamic languages on.NET. In: ICOOOLPS 2009. ACM Press, New York (2009)
14. Deutsch, L.P., Schiffman, A.M.: Efficient implementation of the smalltalk-80 system. In: POPL1980. ACM Press, New York (1980)
15. Ferrara, P., Logozzo, F., Fähndrich, M.A.: Safer unsafe code in.Net. In: OOPSLA 2008 (2008)

16. Flanagan, D.: JavaScript, the definitive guide. O'Reilly, Sebastopol (2009)
17. Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E., Reitmaier, R., Bebenita, M., Chang, M., Franz, M.: Trace-based just-in-time type specialization for dynamic languages. In: PLDI 2009 (2009)
18. Georges, A., Eeckhout, L., Buytaert, D.: Java performance evaluation through rigorous replay compilation. In: OOPSLA 2008 (2008)
19. IEEE. IEEE standard for floating-point arithmetic. Technical report, IEEE (2008)
20. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for javascript. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 238–255. Springer, Heidelberg (2009)
21. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Prentice-Hall, Englewood Cliffs (1993)
22. Laviron, V., Logozzo, F.: Refining abstract interpretation-based static analyses with hints. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 343–358. Springer, Heidelberg (2009)
23. Laviron, V., Logozzo, F.: Subpolyhedra: a (more) scalable approach to infer linear inequalities. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 229–244. Springer, Heidelberg (2009)
24. Logozzo, F., Fähndrich, M.A.: Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. In: SAC 2008 (2008)
25. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part i. Commun. ACM 3(4), 184–195 (1960)
26. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, p. 155. Springer, Heidelberg (2001)
27. Rigo, A.: Representation-based just-in-time specialization and the psyco prototype for Python. In: PEPM 2004. ACM Press, New York (2004)
28. Rigo, A., Pedroni, S.: PyPy's approach to virtual machine construction. In: OOPSLA Companion 2006. ACM Press, New York (2006)
29. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics 5, 285–309 (1955)
30. Ungar, D., Smith, R.B., Chambers, C., Hölzle, U.: Object, message, and performance: How they coexist in self. IEEE Computer 25(10), 53–64 (1992)
31. WebKit. SunSpider JavaScript benchmarks, `http://www2.webkit.org/perf/sunspider-0.9/sunspider.html`

JReq: Database Queries in Imperative Languages

Ming-Yee Iu[1], Emmanuel Cecchet[2], and Willy Zwaenepoel[1]

[1] EPFL, Lausanne, Switzerland
[2] University of Massachusetts Amherst, Amherst, Massachusetts

Abstract. Instead of writing SQL queries directly, programmers often prefer writing all their code in a general purpose programming language like Java and having their programs be automatically rewritten to use database queries. Traditional tools such as object-relational mapping tools are able to automatically translate simple navigational queries written in object-oriented code to SQL. More recently, techniques for translating object-oriented code written in declarative or functional styles into SQL have been developed. For code written in an imperative style though, current techniques are still limited to basic queries. JReq is a system that is able to identify complex query operations like aggregation and nesting in imperative code and translate them into efficient SQL queries. The SQL code generated by JReq exhibits performance comparable with hand-written SQL code.

1 Introduction

Because of the widespread use of databases by computer programs, language designers have often sought to find natural and elegant ways for programmers to write database queries in general purpose programming languages. Although techniques have been developed to integrate database query support into functional languages, for imperative languages such as Java, current techniques are not yet able to handle complex database queries involving aggregation and nesting. Support for aggregation is important because it allows a program to calculate totals and averages across a large dataset without needing to transfer the entire dataset out of a database. Similarly, support for nesting one query inside another significantly increases the expressiveness of queries, allowing a program to group and filter data at the database instead of transferring the data to the program for processing.

We have developed an approach for allowing programmers to write complex database queries inside the imperative language Java. Queries can be written using the normal imperative Java style for working with large datasets—programmers use loops to iterate over the dataset. The queries are valid Java code, so no changes are needed to the Java language to support these complex queries. To run these queries efficiently on common databases, the queries are translated into SQL using an algorithm based on symbolic execution. We have implemented these algorithms in a system called JReq.

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 84–103, 2010.

These are the main technical contributions of this work: a) We demonstrate how complex queries can be written in Java code using loops and iterators. We call this programming style the JReq Query Syntax (JQS) b) We describe an algorithm that can robustly translate complex imperative queries involving aggregation and nesting into SQL c) We have implemented this algorithm in JReq and evaluated its performance.

2 Background

Currently, the most common interface for accessing database queries from Java is to use a low-level API like JDBC. With JDBC, queries are entirely separated from Java. They are written in the domain-specific language SQL, they are stored in strings (which must be compiled and error-checked at runtime), and programmers must manually marshal data into and out of queries (Fig. 12).

Object-oriented databases [11] and object-relational mapping tools like Hibernate, Ruby on Rails, or EJB3 provide a higher-level object-oriented API for accessing databases. Although these tools provide support for updates, error-handling, and transactions, their support for queries is limited. Traditional object-oriented operations such as navigational queries are well-supported, but relational-style queries that filter or manipulate datasets must still be encoded in strings and data must still be manually marshaled into and out of queries. Figure 1 shows an example of such a query written using the Java Persistence API [5].

```
List l = em.createQuery("SELECT a FROM Account a "
  + "WHERE 2 * a.balance < a.creditLimit AND a.country = :country")
  .setParameter("country", "Switzerland")
  .getResultList();
```

Fig. 1. A sample query written in the Java Persistence Query Language (JPQL)

In imperative languages like Java, the normal style for filtering and manipulating large datasets is for a programmer to use loops to iterate over the dataset. As a result, researchers have tried to develop systems that allow programmers to write database queries in imperative languages using such a syntax. We have previously developed a system called Queryll [8] that was able to translate simple queries written in an imperative form to SQL. The system made use of fairly ad hoc algorithms that could not be scaled to support more complex queries involving nesting or aggregation. Wiedermann, Ibrahim, and Cook [19,20] have also successfully translated queries written in an imperative style into SQL. They use abstract interpretation and attribute grammars to translate queries written in Java into database queries. Their work focuses on gathering the objects and fields traversed by program code into a single query (similar to the optimisations performed by Katz and Wong [9]) and is also able to recognise simple filtering constraints. Their approach lacks a mechanism for inferring loop invariants and hence cannot handle queries involving aggregation or complex nesting since these operations span multiple loop iterations.

An alternate approach for supporting complex database queries in imperative languages is to incorporate declarative and functional language features into the languages. Kleisli [21] demonstrated that it was possible to translate queries written in a functional language into SQL. Microsoft was able to add query support to object-oriented languages by extending them with declarative and functional extensions in a feature called Language INtegrated Query (LINQ) [15]. LINQ adds a declarative syntax to .Net languages by allowing programmers to specify SQL-style SELECT..FROM..WHERE queries from within these languages. This syntax is then internally converted to a functional style in the form of lambda expressions, which is then translated to SQL at runtime. Unfortunately, adding similar query support to an imperative programming language like Java without adding specific syntax support for declarative or functional programming results in extremely verbose queries [4].

The difficulty of translating imperative program code to a declarative query language can potentially be avoided entirely by translating imperative program code to an imperative query language. The research of Liewen and DeWitt [10] or of Guravannavar and Sudarshan [7] demonstrate dataflow analysis techniques that could be used for such a system. Following such an approach would be impractical though because all common query languages are specifically designed to be declarative because declarative query languages allow for more optimisation possibilities.

3 JReq Query Syntax

The JReq system allows programmers to write queries using normal Java code. JReq is not able to translate arbitrary Java code into database queries, but queries written in a certain style. We call the subset of Java code that can be translated by JReq into SQL code the JReq Query Syntax (JQS). Although this style does impose limitations on how code must be written, it is designed to be as unrestrictive as possible.

3.1 General Approach and Syntax Examples

Databases are used to store large amounts of structured data, and the most common coding convention used for examining large amounts of data in Java is to iterate over collections. As such, JReq uses this syntax for expressing its queries. JQS queries are generally composed of Java code that iterates over a collection of objects from a database, finds the ones of interest, and adds these objects to a new collection (Fig. 2). For each table of the database, a method exists that returns all the data from that table, and a special collection class called a QueryList is provided that has extra methods to support database operations like set operations and sorting.

```
QueryList<String> results = new QueryList<String>();
for (Account a: db.allAccounts())
   if (a.getCountry().equals("UK"))
      results.add(a.getName());
```

Fig. 2. A more natural Java query syntax

JQS is designed to be extremely lenient in what it accepts as queries. For simple queries composed of a single loop, arbitrary control-flow is allowed inside the loop as long as there are no premature loop exits nor nested loops (nested loops are allowed if they follow certain restrictions), arbitrary creation and modification of variables are allowed as long as they are scoped to the loop, and methods from a long list of safe methods can be called. At most one value can be added to the result-set per loop iteration, and the result-set can only contain numbers, strings, entities, or tuples. Since JReq translates its queries into SQL, the restrictions for more complex queries, such as how queries can be nested or how variables should be scoped, are essentially the same as those of SQL.

One interesting property of the JQS syntax for queries is that the code can be executed directly, and executing the code will produce the correct query result. Of course, since one might be iterating over the entire contents of a database in such a query, executing the code directly might be unreasonably slow. To run the query efficiently, the query must eventually be rewritten in a database query language like SQL instead. This rewriting essentially acts as an optional optimisation on the existing code. Since no changes to the Java language are made, all the code can compile in a normal Java compiler, and the compiler will be able to type-check the query statically. No verbose, type-unsafe data marshaling into and out of the query is used in JQS.

In JQS, queries can be nested, values can be aggregated, and results can be filtered in more complex ways. JQS also supports navigational queries where an object may have references to various related objects. For example, to find the customers with a total balance in their accounts of over one million, one could first iterate over all customers. For each customer, one could then use a navigational query to iterate over his or her accounts and sum up the balance.

```
QueryList results = new QueryList();
for (Customer c: db.allCustomer()) {
   double sum = 0;
   for (Account a: c.getAccounts())
      sum += a.getBalance();
   if (sum > 1000000) results.add(c);
}
```

Intermediate results can be stored in local variables and results can be put into groups. In the example below, a map is used to track (key, value) pairs of the number of students in each department. In the query, local variables are freely used.

```
QueryMap<String, Integer> students =
      new QueryMap<String, Integer>(0);
for (Student s: db.allStudent()) {
   String dept = s.getDepartment();
   int count = students.get(dept) + 1;
   students.put(dept, count);
}
```

Although Java does not have a succinct syntax for creating new database entities, programmers can use tuple objects to store multiple result values from a query (these tuples are of fixed size, so query result can still be mapped from flat relations and do not require nested relations). Results can also be stored in sets instead of lists in order to query for unique elements only, such as in the example below where only unique teacher names (stored in a tuple) are kept.

```
QuerySet teachers = new QuerySet();
for (Student s: db.allStudent()) {
   teachers.add(new Pair(
         s.getTeacher().getFirstName(),
         s.getTeacher().getLastName()));
}
```

In order to handle sorting and limiting the size of result sets, the collection classes used in JQS queries have extra methods for sorting and limiting. The JQS sorting syntax is similar to Java syntax for sorting in its use of a separate comparison object. In the query below, a list of supervisors is sorted by name and all but the first 20 entries are discarded.

```
QuerySet<Supervisor> supervisors = new QuerySet<Supervisor>();
for (Student s: db.allStudent())
   supervisors.add(s.getSupervisor());
supervisors
   .sortedByStringAscending(new StringSorter<Supervisor>() {
       public String value(Supervisor s) {return s.getName();}})
   .firstN(20);
```

For certain database operations that have no Java equivalent (such as SQL regular expressions or date arithmetic), utility methods are provided that support this functionality.

4 Translating JQS Using JReq

In the introduction, it was mentioned that imperative Java code must be translated into a declarative form in order to be executed efficiently on a database. This section explains this translation process using the query from Fig. 2 as an example.

Since JQS queries are written using actual Java code, the JReq system cannot be implemented as a simple Java library. JReq must be able to inspect and

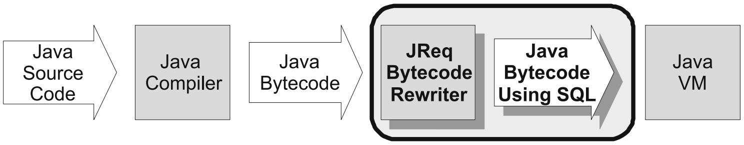


Fig. 3. JReq inserts itself in the middle of the Java toolchain and does not require changes to existing tools

modify Java code in order to identify queries and translate them to SQL. A simple Java library cannot do that. One of our goals for JReq, though, is for it to be non-intrusive and for it to be easily adopted or removed from a development process like a normal library. To do this, the JReq system is implemented as a bytecode rewriter that is able to take a compiled program outputted by the Java compiler and then transform the bytecode to use SQL. It can be added to the toolchain as an independent module, with no changes needed to existing IDEs, compilers, virtual machines, or other such tools (Fig. 3). Although our current implementation has JReq acting as an independent code transformation tool, JReq can also be implemented as a post-processing stage of a compiler, as a classloader that modifies code at runtime, or as part of a virtual machine.

The translation algorithm in JReq is divided into a number of stages. It first preprocesses the bytecode to make the bytecode easier to manipulate. The code is then broken up into loops, and each loop is transformed using symbolic execution into a new representation that preserves the semantics of the original code but removes many secondary features of the code, such as variations in instruction ordering, convoluted interactions between different instructions, or unusual control flow, thereby making it easier to identify queries in the code. This final representation is tree-structured, so bottom-up parsing is used to match the code with general query structures, from which the final SQL queries can then be generated.

4.1 Preprocessing

Although JReq inputs and outputs Java bytecode, its internal processing is not based on bytecode. Java bytecode is difficult to process because of its large instruction set and the need to keep track of the state of the operand stack. To avoid this problem, JReq uses the SOOT framework [18] from Sable to convert Java bytecode into a representation known as Jimple, a three-address code version of Java bytecode. In Jimple, there is no operand stack, only local variables, meaning that JReq can use one consistent abstraction for working with values and that JReq can rearrange instruction sequences without having to worry about stack consistency. Figure 4 shows the code of the query from Fig. 2 after conversion to Jimple form.

```
                $accounts = $db.allAccounts()
                $iter = $accounts.iterator()
                goto loopCondition
loopBody:       $next = $iter.next()
                $a = (Account) $next
                $country = $a.getCountry()
                $cmp0 = $country.equals("UK")
                if $cmp0==0 goto loopCondition
loopAdd:        $name = a$.getName()
                $results.add($name)
loopCondition:  $cmp1 = $iter.hasNext()
                if $cmp1!=0 goto loopBody
exit:
```

Fig. 4. Jimple code of a query

4.2 Transformation of Loops

Since all JQS queries are expressed as loops iterating over collections, JReq needs to add some structure to the control-flow graph of the code. It breaks down the control flow graph into nested strongly-connected components (i.e. loops), and from there, it transforms and analyses each component in turn. Since there is no useful mapping from individual instructions to SQL queries, the analysis operates on entire loops. Conceptually, JReq calculates the postconditions of executing all of the instructions of the loop and then tries to find SQL queries that, when executed, produce the same set of postconditions. If it can find such a match, JReq can replace the original code with the SQL query. Since the result of executing the original series of instructions from the original code gives the same result as executing the query, the translation is safe. Unfortunately, because of the difficulty of generating useful loop invariants for loops [3], JReq is not able to calculate postconditions for a loop directly.

JReq instead examines each loop iteration separately. It starts at the entry point to the loop and walks the control flow graph of the loop until it arrives back at the loop entry point or exits the loop. As it walks through the control flow graph, JReq enumerates all possible paths through the loop. The possible paths through the query code from Fig. 4 are listed in Fig. 5. Theoretically, there can be an exponential number of different paths through a loop since each `if` statement can result in a new path. In practise, such an exponential explosion in paths is rare. Our Java query syntax has an interesting property where when an `if` statement appears in the code, one of the branches of the statement usually ends that iteration of the loop, meaning that the number of paths generally grows linearly. The only type of query that seems to lead to an exponential number of paths are ones that try to generate “CASE WHEN...THEN” SQL code, and these types of queries are rarely used. Although we do not believe exponential path explosion to be a problem for JReq, such a situation can be avoided by using techniques developed by the verification community for dealing with similar problems [6].

Type	**Path**
Exiting	loopCondition → exit
Looping	loopCondition → loopBody →↻
Looping	loopCondition → loopBody → loopAdd →↻

Fig. 5. Paths through the loop

Path: loopCondition → loopBody → loopAdd →↻	
Preconditions	$iter.hasNext() != 0
	((Account)$iter.next()).getCountry().equals("UK") != 0
Postconditions	*$iter.hasNext()*
	$cmp1 = $iter.hasNext()
	$iter.next()
	$next = $iter.next()
	$a = (Account) $iter.next()
	((Account)$iter.next()).getCountry()
	$country = ((Account)$iter.next()).getCountry()
	((Account)$iter.next()).getCountry().equals("UK")
	$cmp0 = ((Account)$iter.next()).getCountry().equals("UK")
	((Account)$iter.next()).getName()
	$name = ((Account)$iter.next()).getName()
	$results.add(((Account)$iter.next()).getName())

Fig. 6. Hoare triple expressing the result of a path (expressions that will be pruned by liveness analysis are indented)

For each path, JReq generates a Hoare triple. A Hoare triple describes the effect of executing a path in terms of the preconditions, code, and postconditions of the path. JReq knows what branches need to be taken for each path to be traversed, and the conditions on these branches form the preconditions for the paths. Symbolic execution is used to calculate the postconditions for each path. Essentially, all variable assignments and method calls become postconditions. The use of symbolic execution means that all preconditions and postconditions are expressed in terms of the values of variables from the start of the loop iteration and that minor changes to the code like simple instruction reordering will not affect the derived postconditions. There are many different styles of symbolic execution, and JReq's use of symbolic execution to calculate Hoare triples is analogous to techniques used in the software verification community, particularly work on translation validation and credible compilation [14,12].

Figure 6 shows the different preconditions and postconditions of the last path from Fig. 5. Not all of the postconditions gathered are significant though, so JReq uses variable liveness information to prune assignments that are not used outside of a loop iteration and uses a list of methods known not to have side-effects to prune safe method calls. Figure 7 shows the final Hoare triples of all paths after pruning.

Basically, JReq has transformed the loop instructions into a new tree representation where the loop is expressed in terms of paths and various precondition

Exiting Path	
Preconditions	$iter.hasNext() == 0
Postconditions	
Looping Path	
Preconditions	$iter.hasNext() != 0 ((Account)$iter.next()).getCountry().equals(”UK”) == 0
Postconditions	$iter.next()
Looping Path	
Preconditions	$iter.hasNext() != 0 ((Account)$iter.next()).getCountry().equals(”UK”) != 0
Postconditions	$iter.next() $results.add(((Account)$iter.next()).getName())

Fig. 7. Final Hoare triples generated from Fig. 4 after pruning

and postcondition expressions. The semantics of the original code are preserved in that all the effects of running the original code are encoded as postconditions in the representation, but problems with instruction ordering or tracking instruction side-effects, etc. have been filtered out.

In general, JReq can perform this transformation of loops into a tree representation in a mechanical fashion, but JReq does make some small optimisations to simplify processing in later stages. For example, constructors in Java are methods with no return type. In JReq, constructors are represented as returning the object itself, and JReq reassigns the result of the constructor to the variable on which the constructor was invoked. This change means that JReq does not have to keep track of a separate method invocation postcondition for each constructor used in a loop.

4.3 Query Identification and Generation

Once the code has been transformed into Hoare triple form, traditional translation techniques can be used to identify and generate SQL queries. For example, Fig. 8 shows how one general Hoare triple representation can be translated into a corresponding SQL form. That particular Hoare triple template is sufficient to match all non-nested SELECT...FROM...WHERE queries without aggregation functions. In fact, because the transformation of Java code into Hoare triple form removes much of the syntactic variation between code fragments with identical semantics, a small number of templates is sufficient to handle most queries.

Since the Hoare triple representation is in a nice tree form, our implementation uses bottom-up parsing to classify and translate the tree into SQL. When using bottom-up parsing to match path Hoare triples to a template, one does have to be careful that each path add the same number and same types of data to the result collection (e.g. in Fig. 8, one needs to check that the types of the various $valA_n$ being added to $\$results$ is consistent across the looping paths). One can use a unification algorithm across the different paths of the loop to ensure that these consistency constraints hold.

Exiting Path		
	Preconditions	\$iter.hasNext() == 0
	Postconditions	*exit loop*
Looping Path$_i$		
	Preconditions	\$iter.hasNext() != 0
		...
	Postconditions	\$iter.next()
...etc.		
Looping Path$_n$		
	Preconditions	\$iter.hasNext() != 0
		$\boldsymbol{pred}_n$
	Postconditions	\$iter.next()
		\$results.add($\boldsymbol{valA}_n$, $\boldsymbol{valB}_n$, ...)
...etc.		

```
SELECT
    CASE WHEN pred_1 THEN valA_1
         WHEN pred_2 THEN valA_2
         ...
    END,
    CASE WHEN pred_1 THEN valB_1
         WHEN pred_2 THEN valB_2
         ...
    END,
    ...
FROM ?
WHERE pred_1 OR pred_2 OR ...
```

Fig. 8. Code with a Hoare triple representation matching this template can be translated into a SQL query in a straight-forward way

One further issue complicating query identification and generation is the fact that a full JQS query is actually composed of both a loop portion and some code before and after the loop. For example, the creation of the object holding the result set occurs before the loop, and when a loop uses an iterator object to iterate over a collection, the definition of the collection being iterated over can only be found outside of the loop. To find these non-loop portions of the query, we recursively apply the JReq transformation to the code outside of the loop at a higher level of nesting. Since the JReq transformation breaks down a segment of code into a finite number of paths to which symbolic execution is applied, the loop needs to be treated as a single indivisible "instruction" whose postconditions are the same as the loop's postconditions during this recursion. This recursive application of the JReq transformation is also used for converting nested loops into nested SQL queries. Figure 9 shows the Hoare triples of the loop and non-loop portions of the query from Fig. 2.

Figure 10 shows some sample operational semantics that illustrate how the example query could be translated to SQL. In the interest of space, these operational semantics do not contain any error-checking and show only how to match the specific query from Fig. 2 (as opposed to the general queries supported by JReq). The query needs to be processed three times using mappings S, F, and W to generate SQL select, from, and where expressions respectively. σ holds information about variables defined outside of a loop. In this example, σ describes the table being iterated over, and Σ describes how to look up fields of this table.

JReq currently generates SQL queries statically by replacing the bytecode for the JQS query with bytecode that uses SQL instead. Static query generation allows JReq to apply more optimisations to its generated SQL output and makes debugging easier because we can examine generated queries without running the program. During this stage, JReq can also optimise the generated SQL queries for specific databases though our prototype currently does not contain such an optimiser. In a previous version of JReq, SQL queries were constructed at

```
Hoaretriples(
  Exit(
    Pre($iter.hasNext() == 0),
    Post()
  ),
  Looping(
    Pre($iter.hasNext() != 0,
        ((Account)$iter.next()).getCountry().equals("UK") == 0),
    Post(Method($iter.next()))
  ),
  Looping(
    Pre($iter.hasNext() != 0,
        ((Account)$iter.next()).getCountry().equals("UK") != 0),
    Post(Method($iter.next()),
         Method($uk.add(((Account)$iter.next()).getName())))))
```

```
PathHoareTriple(
   Pre(),
   Post($results = (new QueryList()).addAll(
            $db.allAccounts().iterator().AddQuery())))
```

Fig. 9. The Hoare triples of the loop and non-loop portion of the query from Fig. 2. The loop Hoare triples are identical to those from Fig. 7, except they have been rewritten so as to emphasise the parsability and tree-like structure of the Hoare triple form.

runtime and evaluated lazily. Although this results in slower queries, it allows the system to support a limited form of inter-procedural query generation. A query can be created in one method, and the query result can later be refined in another method.

During query generation, JReq uses line number debug information from the bytecode to show which lines of the original source files were translated into SQL queries and what they were translated into. IDEs can potentially use this information to highlight which lines of code can be translated by JReq as a programmer types them. Combined with the type error and syntax error feedback given by the Java compiler at compile-time, this feedback helps programmers write correct queries and optimise query performance.

4.4 Implementation Expressiveness and Limitations

The translation algorithm behind JReq is designed to be able to recognise queries with the complexity of SQL92 [1]. In our implementation though, we focused on the subset of operations used in typical SQL database queries. Figure 11 shows a grammar of JQS, the Java code that JReq can translate into SQL. We specify JQS using the grammar of Hoare triples from after the symbolic execution stage of JReq. We used this approach because it is concise and closely describes what

$$\frac{\begin{array}{l} a = \texttt{Exit(Pre(\$iter.hasNext()==0), Post())} \\ b = \texttt{Looping(Pre(\$iter.hasNext()!=0, ...),} \\ \qquad\qquad\quad \texttt{Post(Method(\$iter.next())))} \\ c = \texttt{Looping(Pre(\$iter.hasNext()!=0, } d\texttt{),} \\ \qquad\qquad\quad \texttt{Post(Method(\$iter.next()), } e\texttt{))} \\ e = \texttt{Method(}\mathit{resultset}\texttt{.add(}\mathit{child}\texttt{))} \\ S \vdash \langle \mathit{child}, \sigma \rangle \Downarrow \mathit{select} \\ W \vdash \langle d, \sigma \rangle \Downarrow \mathit{where} \end{array}}{\begin{array}{l} S \vdash \langle \texttt{Hoaretriples}(a, b, c), \sigma \rangle \Downarrow \mathit{select} \\ W \vdash \langle \texttt{Hoaretriples}(a, b, c), \sigma \rangle \Downarrow \mathit{where} \end{array}}$$

$$\frac{\begin{array}{l} W \vdash \langle \mathit{left}, \sigma \rangle \Downarrow \mathit{where}_l \\ W \vdash \langle \mathit{right}, \sigma \rangle \Downarrow \mathit{where}_r \end{array}}{W \vdash \langle \mathit{left}\texttt{.equals(}\mathit{right}\texttt{)==0}, \sigma \rangle \Downarrow \mathit{where}_l \texttt{<>} \mathit{where}_r}$$

$$\frac{\begin{array}{l} W \vdash \langle \mathit{left}, \sigma \rangle \Downarrow \mathit{where}_l \\ W \vdash \langle \mathit{right}, \sigma \rangle \Downarrow \mathit{where}_r \end{array}}{W \vdash \langle \mathit{left}\texttt{.equals(}\mathit{right}\texttt{)!=0}, \sigma \rangle \Downarrow \mathit{where}_l \texttt{=} \mathit{where}_r} \qquad \frac{}{\begin{array}{l} S \vdash \langle \texttt{"UK"}, \sigma \rangle \Downarrow \textbf{"UK"} \\ W \vdash \langle \texttt{"UK"}, \sigma \rangle \Downarrow \textbf{"UK"} \end{array}}$$

$$\frac{\Sigma \vdash \langle \mathit{child}, \sigma, \textsc{Name} \rangle \Downarrow \mathit{val}}{\begin{array}{l} S \vdash \langle \mathit{child}\texttt{.getName()}, \sigma \rangle \Downarrow \mathit{val} \\ W \vdash \langle \mathit{child}\texttt{.getName()}, \sigma \rangle \Downarrow \mathit{val} \end{array}} \qquad \frac{\Sigma \vdash \langle \mathit{child}, \sigma, \textsc{Country} \rangle \Downarrow \mathit{val}}{\begin{array}{l} S \vdash \langle \mathit{child}\texttt{.getCountry()}, \sigma \rangle \Downarrow \mathit{val} \\ W \vdash \langle \mathit{child}\texttt{.getCountry()}, \sigma \rangle \Downarrow \mathit{val} \end{array}}$$

$$\frac{}{\begin{array}{l} \Sigma \vdash \langle \texttt{(Account)\$iter.next()}, \sigma, \textsc{Country} \rangle \Downarrow \sigma(\textsc{next})\textbf{.Country} \\ \Sigma \vdash \langle \texttt{(Account)\$iter.next()}, \sigma, \textsc{Name} \rangle \Downarrow \sigma(\textsc{next})\textbf{.Name} \end{array}}$$

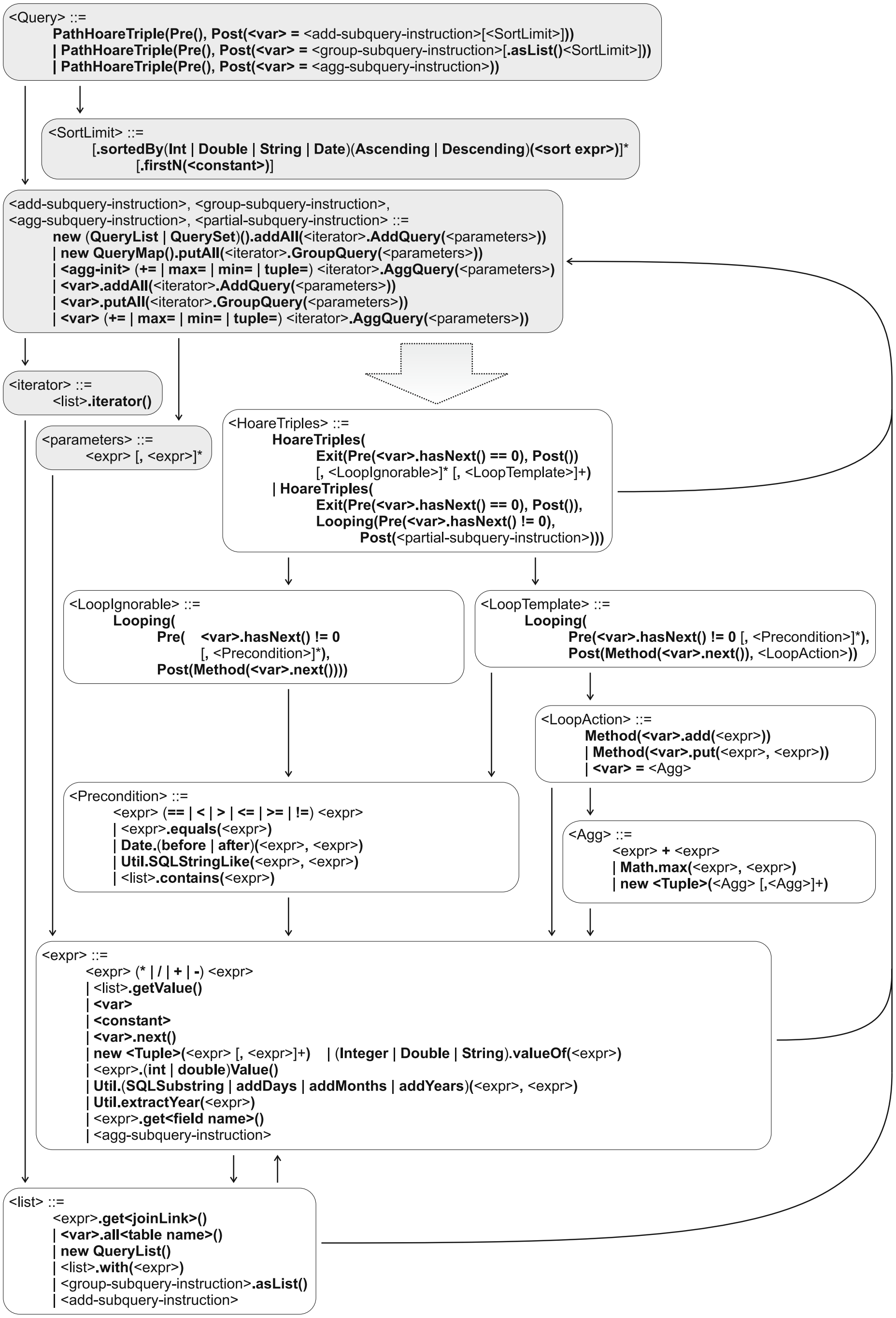
$$\frac{}{F \vdash \langle \texttt{\$db.allAccounts().iterator()}, \sigma \rangle \Downarrow \textbf{Account}}$$

$$\frac{\begin{array}{l} S \vdash \langle \texttt{HoareTriples(...)}, \sigma[\textsc{next} := \mathbf{A}] \rangle \Downarrow \mathit{select} \\ W \vdash \langle \texttt{HoareTriples(...)}, \sigma[\textsc{next} := \mathbf{A}] \rangle \Downarrow \mathit{where} \\ F \vdash \langle \mathit{iterator}, \sigma \rangle \Downarrow \mathit{from} \end{array}}{\begin{array}{l} \langle \mathit{resultset}\texttt{.addAll(}\mathit{iterator}\texttt{.AddQuery())}, \sigma \rangle \Downarrow \\ \qquad \textbf{SELECT } \mathit{select} \textbf{ FROM } \mathit{from} \textbf{ AS A WHERE } \mathit{where} \end{array}}$$

Fig. 10. Sample operational semantics for translating Fig. 9 to SQL

queries will be accepted. We have found that specifying JQS using a traditional grammar directly describing a Java subset to be too imprecise or too narrow to be useful. Because JReq uses symbolic execution, for each query, any Java code variant with the same semantic meaning will be recognised by JReq as being the same query. This large number of variants cannot be captured using a direct specification of a Java grammar subset.

In the figure, the white boxes refer to grammar rules used for classifying loops. The grey boxes are used for combining loops with context from outside of the loop. There are four primary templates for classifying a loop: one for adding



```
<Query> ::=
        PathHoareTriple(Pre(), Post(<var> = <add-subquery-instruction>[<SortLimit>]))
        | PathHoareTriple(Pre(), Post(<var> = <group-subquery-instruction>[.asList()<SortLimit>]))
        | PathHoareTriple(Pre(), Post(<var> = <agg-subquery-instruction>))

<SortLimit> ::=
        [.sortedBy(Int | Double | String | Date)(Ascending | Descending)(<sort expr>)]*
                [.firstN(<constant>)]

<add-subquery-instruction>, <group-subquery-instruction>,
<agg-subquery-instruction>, <partial-subquery-instruction> ::=
        new (QueryList | QuerySet)().addAll(<iterator>.AddQuery(<parameters>))
        | new QueryMap().putAll(<iterator>.GroupQuery(<parameters>))
        | <agg-init> (+= | max= | min= | tuple=) <iterator>.AggQuery(<parameters>)
        | <var>.addAll(<iterator>.AddQuery(<parameters>))
        | <var>.putAll(<iterator>.GroupQuery(<parameters>))
        | <var> (+= | max= | min= | tuple=) <iterator>.AggQuery(<parameters>))

<iterator> ::=
        <list>.iterator()

<parameters> ::=
        <expr> [, <expr>]*

<HoareTriples> ::=
        HoareTriples(
                Exit(Pre(<var>.hasNext() == 0), Post())
                [, <LoopIgnorable>]* [, <LoopTemplate>]+)
        | HoareTriples(
                Exit(Pre(<var>.hasNext() == 0), Post()),
                Looping(Pre(<var>.hasNext() != 0),
                        Post(<partial-subquery-instruction>)))

<LoopIgnorable> ::=
        Looping(
                Pre(    <var>.hasNext() != 0
                        [, <Precondition>]*),
                Post(Method(<var>.next())))

<LoopTemplate> ::=
        Looping(
                Pre(<var>.hasNext() != 0 [, <Precondition>]*),
                Post(Method(<var>.next()), <LoopAction>))

<LoopAction> ::=
        Method(<var>.add(<expr>))
        | Method(<var>.put(<expr>, <expr>))
        | <var> = <Agg>

<Precondition> ::=
        <expr> (== | < | > | <= | >= | !=) <expr>
        | <expr>.equals(<expr>)
        | Date.(before | after)(<expr>, <expr>)
        | Util.SQLStringLike(<expr>, <expr>)
        | <list>.contains(<expr>)

<Agg> ::=
        <expr> + <expr>
        | Math.max(<expr>, <expr>)
        | new <Tuple>(<Agg> [,<Agg>]+)

<expr> ::=
        <expr> (* | / | + | -) <expr>
        | <list>.getValue()
        | <var>
        | <constant>
        | <var>.next()
        | new <Tuple>(<expr> [, <expr>]+)    | (Integer | Double | String).valueOf(<expr>)
        | <expr>.(int | double)Value()
        | Util.(SQLSubstring | addDays | addMonths | addYears)(<expr>, <expr>)
        | Util.extractYear(<expr>)
        | <expr>.get<field name>()
        | <agg-subquery-instruction>

<list> ::=
        <expr>.get<joinLink>()
        | <var>.all<table name>()
        | new QueryList()
        | <list>.with(<expr>)
        | <group-subquery-instruction>.asList()
        | <add-subquery-instruction>
```

Fig. 11. JQS grammar

elements to a collection, one for adding elements to a map, one for aggregating values, and another for nested loops resulting in a join. Most SQL operations can be expressed using the functionality described by this grammar.

Some SQL functionality that is not currently supported by JQS include set operations, intervals, and internationalisation because the queries we were working with did not require this functionality. We also chose not to support NULL and related operators in this iteration of JQS. Because Java does not support three-value logic or operator overloading, we would have to add special objects and methods to emulate the behaviour of NULL, resulting in a verbose and complicated design. Operations related to NULL values such as OUTER JOINs are not supported as well.

JQS also currently offers only basic support for update operations since it focuses only on the query aspects of SQL. SQL's more advanced data manipulation operations are rarely used and not too powerful, so it would be fairly straight-forward to extend JQS to support these operations. Most of these operations are simply composed of a normal query followed by some sort of INSERT, DELETE, or UPDATE involving the result set of the query.

In the end, our JReq system comprises approximately 20 thousand lines of Java and XSLT code. Although JReq translations can be applied to an entire codebase, we use annotations to direct JReq into applying its transformations only to specific methods known to contain queries. Additionally, we had some planned features that we never implemented because we did not encounter any situations during our research that required them: we did not implement handling of non-local variables, we did not implement type-checking or unification to check for errors in queries, and we did not implement pointer aliasing support.

5 Evaluation

5.1 TPC-W

To evaluate the behaviour of JReq, we tested the ability for our system to handle the database queries used in the TPC-W benchmark [16]. TPC-W emulates the behaviour of database-driven websites by recreating a website for an online bookstore.

We started with the Rice implementation of TPC-W [2], which uses JDBC to access its database. For each query in the TPC-W benchmark, we wrote an equivalent query using JQS and manually verified that the resulting queries were semantically equivalent to the originals. We could then compare the performance of each query when using the original JDBC and when using the JReq system. Our JReq prototype does not provide support for database updates, so we did not test any queries involving updates. Since this experiment is intended to examine the queries generated by JReq as compared to hand-written SQL, we also disabled some of the extra features of JReq such as transaction and persistence lifecycle management.

We created a 600 MB database in PostgreSQL 8.3.0 [13] by populating the database with the number of items set to 10000. We did not run the complete TPC-W benchmark, which tests the complete system performance of web servers, application servers, and database servers. Instead, we focused on measuring the performance of individual queries instead. For each query, we first executed the query 200 times with random valid parameters to warm the database cache, then we measured the time needed to execute the query 3000 times with random valid parameters, and finally we garbage collected the system. Because of the poor performance of the getBestSellers query, we only executed it for 50 times to warm the cache and measured the performance of executing the query only 250 times. We first took the JQS version of the queries, measured the performance of each query consecutively, and repeated the benchmark 50 times. We took the average of only the last 10 runs to avoid the overhead of Java dynamic compilation. We then repeated this experiment using the original JDBC implementation instead of JQS. The database and the query code were both run on the same machine, a 2.5 GHz Pentium IV Celeron Windows machine with 1 GB of RAM. The benchmark harness was run using Sun's 1.5.0 Update 12 JVM. JReq required approximately 7 seconds to translate our 12 JQS queries into SQL.

The performance of each of the queries is shown in Table 1. In all cases, JReq is faster than hand-written SQL. These results are a little curious because one usually expects hand-written code to be faster than machine-generated code. If we look at the one query in Fig. 12 that shows the code of the original hand-written JDBC code and compares it to the comparable JQS query and the JDBC generated from that query, we can see that the original JDBC code is essentially the same as the JDBC generated by JReq. In particular, the SQL queries are structurally the same though the JReq-generated version is more verbose. What

Table 1. The average execution time, standard deviation, and difference from hand-written JDBC/SQL (all in milliseconds) of the TPC-W benchmark are shown in this table with the column JReq NoOpt referring to JReq with runtime optimisations disabled. One can see that JReq offers better performance than the hand-written SQL queries.

	JDBC		JReq NoOpt			JReq		
Query	Time	σ	Time	σ	Δ	Time	σ	Δ
getName	3592	112	3633	24	1%	2241	15	(38%)
getCustomer	8424	79	8944	57	6%	3939	24	(53%)
doGetMostRecentOrder	29108	731	88831	644	205%	8009	57	(72%)
getBook	6392	30	7347	55	15%	3491	27	(45%)
doAuthorSearch	10216	24	10414	559	2%	7306	46	(28%)
doSubjectSearch	16999	128	16898	86	(1%)	13667	120	(20%)
getIDandPassword	3706	33	3820	41	3%	2375	25	(36%)
doGetBestSellers	4472	50	4455	51	(0%)	3936	39	(12%)
doTitleSearch	27302	203	26979	418	(1%)	23985	61	(12%)
doGetNewProducts	23111	68	24447	128	6%	21086	70	(9%)
doGetRelated	6162	52	7731	92	25%	2690	34	(56%)
getUserName	3506	57	3569	13	2%	2214	11	(37%)

Original hand-written JDBC query

```
PreparedStatement getUserName = con.prepareStatement(
   "SELECT c_uname FROM customer WHERE c_id = ?");
getUserName.setInt(1, C_ID);
ResultSet rs=getUserName.executeQuery();
if (!rs.next()) throw new Exception();
u_name = rs.getString("c_uname");
rs.close(); stmt.close();
```

Comparable JQS query

```
EntityManager em = db.begin();
DBSet<String> matches = new QueryList<String>();
for (DBCustomer c: em.allDBCustomer())
   if (c.getCustomerId()==C_ID) matches.add(c.getUserName());
u_name = matches.get();
db.end(em, true);
```

JDBC generated by JReq

```
PreparedStatement stmt = null; ResultSet rs = null;
try { stmt = stmtCache.poll();
      if (stmt == null) stmt = em.db.con.prepareStatement(
          "SELECT (A.C_UNAME) AS COL0 "
        + "FROM Customer AS A WHERE (((A.C_ID)=?))");
      stmt.setInt(1, param0);
      rs = stmt.executeQuery();
      QueryList toReturn = new QueryList();
      while(rs.next()) { Object value = rs.getString(1);
                         toReturn.bulkAdd(value); }
      return toReturn;
} catch (SQLException e) { ... } finally {
   if (rs != null) try { rs.close(); } catch...
   stmtCache.add(stmt); }
```

Fig. 12. Comparison of JDBC vs. JReq on the getUserName query

makes the JReq version faster though is that JReq is able to take advantage of small runtime optimisations that are cumbersome to implement when writing JDBC by hand. For example, all JDBC drivers allow programmers to parse SQL queries into an intermediate form. Whenever the same SQL query is executed but with different parameters, programmers can supply the intermediate form of the query to the SQL driver instead of the original SQL query text, thereby allowing the SQL driver to skip repeatedly reparsing and reanalysing the same SQL query text. Taking advantage of this optimisation in hand-written JDBC code is cumbersome because the program must be structured in a certain way and a certain amount of bookkeeping is involved, but this is all automated by JReq.

Table 1 also shows the performance of code generated by JReq if these runtime optimisations are disabled (denoted as JReq NoOpt). Of the 12 queries, the performance of JReq and hand-written JDBC is identical for six of them. For the six queries where JReq is slower, four are caused by poorly formulated queries that fetched more data than the original queries (for example, they fetch entire entities whereas the original queries only fetched most of the fields of the entity). Two other queries are slower because JReq generates queries that are more verbose than the original queries thereby requiring more work from the SQL parser.

Overall though, all the queries from the TPC-W benchmark, a benchmark that emulates the behaviour of real application, can be expressed in JQS, and JReq can successfully translate these JQS queries into SQL. JReq generates SQL queries that are structurally similar to the original hand-written queries for all of the queries. Although the machine-generation of SQL queries may result in queries that are more verbose and less efficient than hand-written SQL queries, by taking advantage of various optimisations that a normal programmer may find cumbersome to implement, JReq can potentially exceed the performance of hand-written SQL.

5.2 TPC-H

Although TPC-W does capture the style of queries used in database-driven websites, these types of queries make little use of more advanced query functionality such as nested queries. To evaluate JReq's ability to handle more difficult queries, we have run some benchmarks involving TPC-H [17]. The TPC-H benchmark tests a database's ability to handle decision support workloads. This workload is characterised by fairly long and difficult ad hoc queries that access large amounts of data. The purpose of this experiment is to verify that the expressiveness of the JQS query syntax and JReq's algorithms for generating SQL queries are sufficient to handle long and complex database queries.

We extracted the 22 SQL queries and parameter generator from the TPC-H benchmark and modified them to run under JDBC in Java. We chose to use MySQL 5.0.51 for the database instead of PostgreSQL in this experiment in order to demonstrate JReq's ability to work with different backends. For this benchmark, we used a 2.5 GHz Pentium IV Celeron machine with 1 GB of RAM running Fedora Linux 9, and Sun JDK 1.5.0 Update 16.

We then rewrote the queries using JQS syntax. All 22 of the queries could be expressed using JQS syntax except for query 13, which used a LEFT OUTER JOIN, which we chose not to support in this version of JQS, as we described in Sect. 4.4. To verify that the JQS queries were indeed semantically equivalent to the original queries, we manually compared the query results between JDBC and JReq when run on a small TPC-H database using a scale factor of 0.01, and the results matched. This shows the expressiveness of the JQS syntax in that 21 of the 22 queries from TPC-H can be expressed in the JQS syntax and be correctly translated into working SQL code. JReq required approximately 33 seconds to translate our 21 JQS queries into SQL.

We then generated a TPC-H database using a scale factor of 1, resulting in a database about 1GB in size. We executed each of the 21 JQS queries from

Table 2. TPC-H benchmark results showing average time, standard deviation, and time difference (all results in seconds)

	JDBC		JReq				JDBC		JReq		
Query	Time	σ	Time	σ	Δ	Query	Time	σ	Time	σ	Δ
q1	73.5	0.4	71.9	3.4	(2%)	q12	23.4	0.5	29.7	0.2	27%
q2	145.4	2.2	146.0	1.9	0%	q14	491.7	8.9	500.8	10.1	2%
q3	37.9	0.6	38.6	0.9	2%	q15	24.9	0.7	24.8	0.6	(0%)
q4	23.0	0.5	23.8	0.2	3%	q16	21.3	0.6	> 1 hr	0.2	-
q5	209.1	4.2	206.1	3.2	(1%)	q17	2.1	0.2	11.0	3.6	429%
q6	15.2	0.3	15.8	0.3	4%	q18	> 1 hr	0.0	349.3	4.0	-
q7	79.1	0.5	83.1	1.6	5%	q19	2.8	0.1	18.1	0.4	540%
q8	48.8	1.7	51.0	1.9	4%	q20	69.4	4.3	508.4	11.4	633%
q9	682.0	97.4	690.2	97.9	1%	q21	245.5	3.2	517.0	7.1	111%
q10	47.1	1.0	47.2	0.5	0%	q22	1.1	0.0	1.6	0.0	43%
q11	41.7	0.6	41.9	0.7	1%						

TPC-H in turn using random query parameters, with a garbage collection cycle run in-between each query. We then executed the corresponding JDBC queries using the same parameters. This was repeated six times, with the last five runs kept for the final results. Queries that ran longer than one hour were cancelled. Table 2 summarises the results of the benchmarks.

Unlike TPC-W, the queries in TPC-H take several seconds each to execute, so runtime optimisations do not significantly affect the results. Since almost all the execution time occurs at the database and since the SQL generated from the JQS queries are semantically equivalent to the original SQL queries, differences in execution time are mostly caused by the inability of the database's query optimiser to find optimal execution plans. In order to execute the complex queries in TPC-H efficiently, query optimisers must be able to recognise certain patterns in a query and restructure them into more optimal forms. The particular SQL generated by JReq uses a SQL subset that may match different optimisation patterns in database query optimisers than hand-written SQL code. For example, the original SQL for query 16 evaluates a COUNT(DISTINCT) operation inside of GROUP BY. This is written in JQS using an equivalent triply nested query, but MySQL is not able to optimise the query correctly, and running the triply nested query directly results in extremely poor performance. On the other hand, in query 18, JReq's use of deeply nested queries instead of a more specific SQL operation (in this case, GROUP BY...HAVING) fits a pattern that MySQL is able to execute efficiently, unlike the original hand-written SQL. Because of the sensitivity of MySQL's query optimiser to the structure of SQL queries, it will be important in the future for JReq to provide more flexibility to programmers in adjusting the final SQL generated by JReq.

Overall, 21 of the 22 queries from TPC-H could be successfully expressed using the JQS syntax and translated into SQL. Only one query, which used a LEFT OUTER JOIN, could not be handled because JQS and JReq do not currently support the operation yet. For most of the queries, the JQS queries executed with similar performance to the original queries. Where there are differences in execution time, most of these differences can be eliminated by either improving the MySQL query optimiser, adding special rules to the SQL generator to generate patterns that are better handled by MySQL, or extending the syntax of JQS to allow programmers to more directly specify those specific SQL keywords that are better handled by MySQL.

6 Conclusions

The JReq system translates database queries written in the imperative language Java into SQL. Unlike other systems, the algorithms underlying JReq are able to analyse code written in imperative programming languages and recognise complex query constructs like aggregation and nesting. In developing JReq, we have created a syntax for database queries that can be written entirely with normal Java code, we have designed an algorithm based on symbolic execution to automatically translate these queries into SQL, and we have implemented a research prototype of our system that shows competitive performance to handwritten SQL.

We envision JReq as a useful complement to other techniques for translating imperative code into SQL. For common queries, existing techniques often provide greater syntax flexibility than JReq, but for the most complex queries, programmers can use JReq instead of having to resort to domain-specific languages like SQL. As a result, all queries will end up being written in Java, which can be understood by all the programmers working on the codebase.

References

1. American National Standards Institute: American National Standard for Information Systems—Database Language—SQL: ANSI INCITS 135-1992 (R1998). American National Standards Institute (1992)
2. Amza, C., Cecchet, E., Chanda, A., Elnikety, S., Cox, A., Gil, R., Marguerite, J., Rajamani, K., Zwaenepoel, W.: Bottleneck characterization of dynamic web site benchmarks. Tech. Rep. TR02-389, Rice University (February 2002)
3. Bradley, A.R., Manna, Z.: The Calculus of Computation: Decision Procedures with Applications to Verification. Springer, New York (2007)
4. Cook, W.R., Rai, S.: Safe query objects: statically typed objects as remotely executable queries. In: ICSE 2005: Proceedings of the 27th international conference on Software engineering, pp. 97–106. ACM, New York (2005)
5. DeMichiel, L., Keith, M.: JSR 220: Enterprise JavaBeans 3.0, http://www.jcp.org/en/jsr/detail?id=220
6. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: generating compact verification conditions. In: POPL 2001, pp. 193–205. ACM, New York (2001)
7. Guravannavar, R., Sudarshan, S.: Rewriting procedures for batched bindings. Proc. VLDB Endow. 1(1), 1107–1123 (2008)
8. Iu, M.Y., Zwaenepoel, W.: Queryll: Java database queries through bytecode rewriting. In: van Steen, M., Henning, M. (eds.) Middleware 2006. LNCS, vol. 4290, pp. 201–218. Springer, Heidelberg (2006)

9. Katz, R.H., Wong, E.: Decompiling CODASYL DML into relational queries. ACM Trans. Database Syst. 7(1), 1–23 (1982)
10. Lieuwen, D.F., DeWitt, D.J.: Optimizing loops in database programming languages. In: DBPL3: Proceedings of the third international workshop on Database programming languages: bulk types & persistent data, pp. 287–305. Morgan Kaufmann, San Francisco (1992)
11. Maier, D., Stein, J., Otis, A., Purdy, A.: Development of an object-oriented DBMS. In: OOPLSA 1986, pp. 472–482. ACM Press, New York (1986)
12. Necula, G.C.: Translation validation for an optimizing compiler. In: PLDI 2000, pp. 83–94. ACM, New York (2000)
13. PostgreSQL Global Development Group: PostgreSQL, `http://www.postgresql.org/`
14. Rinard, M.C.: Credible compilation. Tech. Rep. MIT/LCS/TR-776, Cambridge, MA, USA (1999)
15. Torgersen, M.: Language INtegrated Query: unified querying across data sources and programming languages. In: OOPSLA 2006, pp. 736–737. ACM Press, New York (2006)
16. Transaction Processing Performance Council: TPC Benchmark W (Web Commerce) Specification Version 1.8. Transaction Processing Performance Council (2002)
17. Transaction Processing Performance Council: TPC Benchmark H (Decision Support) Standard Specification Version 2.8.0. Transaction Processing Performance Council (2008)
18. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a Java bytecode optimization framework. In: CASCON 1999: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, p. 13. IBM Press (1999)
19. Wiedermann, B., Cook, W.R.: Extracting queries by static analysis of transparent persistence. In: POPL 2007, pp. 199–210. ACM Press, New York (2007)
20. Wiedermann, B., Ibrahim, A., Cook, W.R.: Interprocedural query extraction for transparent persistence. In: OOPSLA 2008, pp. 19–36. ACM, New York (2008)
21. Wong, L.: Kleisli, a functional query system. J. Funct. Program. 10(1), 19–56 (2000)

Verifying Local Transformations on Relaxed Memory Models

Sebastian Burckhardt[1], Madanlal Musuvathi[1], and Vasu Singh[2]

[1] Microsoft Research
[2] EPFL, Switzerland

Abstract. The problem of locally transforming or translating programs without altering their semantics is central to the construction of correct compilers. For concurrent shared-memory programs this task is challenging because (1) concurrent threads can observe transformations that would be undetectable in a sequential program, and (2) contemporary multiprocessors commonly use relaxed memory models that complicate the reasoning.

In this paper, we present a novel proof methodology for verifying that a local program transformation is sound with respect to a specific hardware memory model, in the sense that it is not observable in any context. The methodology is based on a structural induction and relies on a novel compositional denotational semantics for relaxed memory models that formalizes (1) the behaviors of program fragments as a set of traces, and (2) the effect of memory model relaxations as local trace rewrite operations.

To apply this methodology in practice, we implemented a semi-automated tool called Traver and used it to verify/falsify several compiler transformations for a number of different hardware memory models.

1 Introduction

Compilers perform a series of transformations that translate a high-level program into low-level machine instructions, while optimizing the code for performance. For correctness, these transformations must preserve the meaning for any input program. Proving the correctness of program transformations has been well studied for sequential programs [29,18,17,19].

However, concurrent shared-memory programs require additional caution because transformations that reorder, introduce, or eliminate accesses to shared memory may be observed by concurrent threads and can thus introduce subtle safety or liveness errors in an otherwise correct program. For example, the redundant read elimination shown in Fig. 1 is not safe because it leads to nontermination, and the branch consolidation in Fig. 2 is unsafe because it can lead to an assertion violation.

Typically, only a very small part of all memory accesses (namely the accesses that are used for synchronization purposes) are susceptible to such issues. However, in the absence of a whole-program-analysis or user-provided annotations,

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 104–123, 2010.

```
                     int X = 0;

              Transformation            Observer

int r1 = X;         int r1 = X;        |
while(X == 0);  ⇒  while(r1 == 0);    | X = 1;
```

Fig. 1. Redundant read elimination causing nontermination

```
        bool B = false, X = false, Y = false;

              Transformation              Observer

bool r = B;                    |
if(r) {                        |
  X = r; Y = !r;   bool r = B; | X = true;
} else {        ⇒  X = r;      | assert(X || Y);
  Y = !r; X = r;   Y = !r;     |
}                              |
```

Fig. 2. This branch consolidation is unsafe: the assert can fail in the transformed program, but not the original program. The reason is that the transformation changes the order of the writes to X and Y in the then-branch.

we can not distinguish between data accesses and accesses that are used for synchronization. In practice, most compilers rely on the programmer to provide special type qualifiers like 'volatile' [20] or 'atomic' [4] or on custom annotations to identify synchronization accesses. Programs that correctly convey all synchronization are called 'properly labeled' [14] or 'data-race-free' [2].

There is a general understanding on how to correctly transform data-race free programs [20,4,25]. In this paper, however, we address the more conservative problem of safely transforming general programs, including programs that contain data races, or programs that are missing the annotations or types needed to identify synchronization accesses.

It may seem at first that under this conservative restriction, very few transformations would be safe. However, we can assume that programs that are designed to work on relaxed hardware memory models are resilient to certain transformations. Clearly, there is no need for a compiler to be more conservative than the hardware executing the compiled program.

For example, consider the example in Fig. 2 again, but let the execution be on a machine that relaxes write-to-write order. Now, we may argue that the transformation is indeed correct as it does not introduce new behaviors: if write-to-write order is relaxed by the hardware, the assertion violation may occur even for the original untransformed program.

For some transformations it can be rather mind-boggling to determine whether it is safe for a given architecture. For instance, by using the methodology presented in this paper, we will prove (though not fully comprehend) that the transformation

$$\{r := A; \textbf{if } r == 0 \textbf{ then } A := 0\} \rightarrow \{r := A\}$$

is safe on a sequentially consistent machine, unsafe on a machine that relaxes write-to-read order (such as TSO), but once more safe on a machine that additionally relaxes write-to-write order (such as PSO).

Overall, we summarize our contributions as follows:

- (Section 3) We build a semantic foundation for relaxed hardware memory models. We show how many common relaxations can be explained as local rewrite operations on memory access sequences. In particular, we present a novel aggregation rule that can explain the effect of store buffers, the most common relaxation of all. Our semantics is compositional (it defines the behavior of program fragments recursively) and can model infinite executions.
- (Section 4) We present a proof methodology to verify the soundness of local program transformations over relaxed memory models, based on a notion of observations. We introduce a notion of invisible rewrite rules (Section 4.1) to reason about all possible program contexts.
- (Section 5) We show how to apply the methodology in practice by verifying/falsifying 8 program transformation for 5 different memory models (including sequential consistency), aided by a custom semi-automatic tool called Traver. Given a local program transformation and a memory model, Traver uses an automated theorem prover [11] to prove that the set of observations of the transformed program is contained in the set of observations of the original program, for all possible program contexts. Conversely, when provided with an additional falsification context, Traver can automatically show that the transformation leads to observable differences in behavior. This produces a certificate of unsoundness of the transformation.

2 Related Work

Our calculus and semantics, and in particular the handling of infinite executions, were inspired by Brookes' fully abstract denotational semantics for sequentially consistent programs [6]. Languages and semantics to study relaxed memory models have been developed before, in both operational style [5] and algebraic style [24]. Our work differs in that it (1) guarantees fairness for infinite executions and (2) relates to contemporary multiprocessor architectures and common program transformations.

Much prior work on hardware memory models focuses on the complex intricacies of axiomatic specifications and gives only partial formalizations (in particular, program syntax is generally ignored). Some work departs from the mainstream and uses an operational style [23] or an algebraic style [3,27] (where the algebraic style bears some similarity to our use of dynamic rewrite rules, but does not include the important store-load aggregation rule which is crucial to correctly model contemporary hardware memory models). Recently, researchers have proposed revised axiomatic formalizations of the x86 architecture [13,22]. Our work is orthogonal: our goal is to find simple yet precise means to reason about various common hardware relaxations, rather than fully model all details of one specific hardware architecture.

Our work was partly motivated by recent work [9,26] that demonstrated the difficulty of manually verifying compiler optimizations against memory models. It is also similar to efforts on verifying the soundness of compiler transformations for language-level models (Java, DRF) [25]. Unlike the latter, however, we define soundness of transformations relative to the hardware memory model (and are thus not susceptible to whether programs are data-race-free or not), can handle infinite executions, and provide a tool that helps to automate parts of the verification/falsification effort.

3 Semantic Foundation

In this section, we lay the foundation for understanding hardware memory models and for reasoning about them formally. We start by demonstrating how we explain typical relaxations in the hardware using dynamic rewrite operations. We then formalize this concept by defining a simple imperative language for shared-memory programs and a compositional denotational semantics. Along the way, we discuss various challenges, such as how our semantics handles infinite executions and fairness.

We start with a quick introduction to relaxed hardware memory models, revisiting classical examples [1,14]. We use special diagrams called *derivations* to explain how to understand relaxations as a consequence of dynamic rewriting of access sequences. We distinguish three types of dynamic rewrite operations: reordering, aggregation, and splitting.

Ordering relaxations allow the hardware to execute operations in a different order than specified by the program. This can speed up execution as it allows the hardware to delay the completion of operations with high latency (such as propagating stores to a global shared memory) past subsequent operations with low latency (such as reading a locally cached value). In Fig. 3 (a) and (b), we show classic "litmus tests" to illustrate the effects of ordering relaxations. These programs distinguish syntactically between processor-local registers (lowercase identifiers) and shared memory locations (capitalized identifiers).

Not all effects can be explained by simply reordering instructions. For example, the program in Fig. 3(c) is a variation of 3(b) that shows how stored values can be visible to subsequent loads by the same processor before they have been committed to shared memory. This effect is very common and often attributed to processor-local "store buffers". We explain this effect as an aggregation of the store with the following load.

More formally, let $\langle ld\ L, x\rangle$ and $\langle st\ L, x\rangle$ represent store or load accesses from/to location L, with loaded/stored value of x. Now consider the dynamic

(a)

Initially: $A = B = 0$

$P1$	$P2$
$A := 1$	$r := B$
$B := 1$	$s := A$

Eventually: $r = 1, s = 0$

(b)

Initially: $A = B = 0$

$P1$	$P2$
$A := 1$	$B := 1$
$r := B$	$s := A$

Eventually: $r = s = 0$

(c)

Initially: $A = B = 0$

$P1$	$P2$
$A := 1$	$B := 1$
$u := A$	$v := B$
$r := B$	$s := A$

Eventually: $r = s = 0, u = v = 1$

Fig. 3. (a) This outcome is possible if the stores by P1 are reordered, or if the loads by P2 are reordered. (b) This outcome (known as Dekker) is possible if the stores are delayed past the loads. (c) This outcome (a variation of Dekker) is possible if stores can be both forwarded to loads and delayed past loads.

sss (swap store-store)

$$\langle st\ L, x\rangle\langle st\ L', x'\rangle \overset{L\neq L'}{\rightarrow} \langle st\ L', x'\rangle\langle st\ L, x\rangle$$

sll (swap load-load)

$$\langle ld\ L, x\rangle\langle ld\ L', x'\rangle \rightarrow \langle ld\ L', x'\rangle\langle ld\ L, x\rangle$$

ssl (swap store-load)

$$\langle st\ L, x\rangle\langle ld\ L', x'\rangle \overset{L\neq L'}{\rightarrow} \langle ld\ L', x'\rangle\langle st\ L, x\rangle$$

sls (swap load-store)

$$\langle ld\ L, x\rangle\langle st\ L', x'\rangle \overset{L\neq L'}{\rightarrow} \langle st\ L', x'\rangle\langle ld\ L, x\rangle$$

asl (aggregate store-load)

$$\langle st\ L, x\rangle\langle ld\ L, x\rangle \rightarrow \langle st\ L, x\rangle$$

Model	Rewrite Rules				
SC	(none)				
390	**ssl**				
TSO	**ssl**	**asl**			
x86-TSO	**ssl**	**asl**			
PSO	**ssl**	**asl**	**sss**		
CLR	**ssl**	**asl**	**sll**		
RMO	**ssl**	**asl**	**sss**	**sll**$^{\not c \not d}$	**sls**$^{\not c \not d}$
Alpha	**ssl**	**asl**	**sss**	**sll**$^{\neq}$	**sls**$^{\not c \not d}$

Fig. 4. Dynamic rewrite operations employed by some commercial hardware memory models and by the CLR memory model. The symbols $\not c$, $\not d$ and $\neq$ indicate that the accesses are swapped only if they are not control dependent, not data dependent, or target a different location, respectively.

$$\underbrace{\begin{array}{ll} \langle st\ A,1\rangle\langle st\ B,1\rangle & \\ \quad\big\downarrow \textbf{sss} & \\ \langle st\ B,1\rangle\langle st\ A,1\rangle & \langle ld\ B,1\rangle\langle ld\ A,0\rangle \end{array}}_{\langle st\ B,1\rangle\langle ld\ B,1\rangle\langle ld\ A,0\rangle\langle st\ A,1\rangle}$$

$$\underbrace{\begin{array}{cc} \langle st\ A,1\rangle\langle ld\ B,0\rangle & \langle st\ B,1\rangle\langle ld\ A,0\rangle \\ \big\downarrow \textbf{ssl} & \big\downarrow \textbf{ssl} \\ \langle ld\ B,0\rangle\langle st\ A,1\rangle & \langle ld\ A,0\rangle\langle st\ B,1\rangle \end{array}}_{\langle ld\ B,0\rangle\langle ld\ A,0\rangle\langle st\ A,1\rangle\langle st\ B,1\rangle}$$

$$\underbrace{\begin{array}{cc} \langle st\ A,1\rangle\langle ld\ A,1\rangle\langle ld\ B,0\rangle & \langle st\ B,1\rangle\langle ld\ B,1\rangle\langle ld\ A,0\rangle \\ \big\downarrow \textbf{asl} & \big\downarrow \textbf{asl} \\ \langle st\ A,1\rangle\langle ld\ B,0\rangle & \langle st\ B,1\rangle\langle ld\ A,0\rangle \\ \big\downarrow \textbf{ssl} & \big\downarrow \textbf{ssl} \\ \langle ld\ B,0\rangle\langle st\ A,1\rangle & \langle ld\ A,0\rangle\langle st\ B,1\rangle \end{array}}_{\langle ld\ B,0\rangle\langle ld\ A,0\rangle\langle st\ A,1\rangle\langle st\ B,1\rangle}$$

Fig. 5. Top left: Derivation for Fig. 3(a). P1 issues two stores that get reordered by **sss** before being interleaved with the two loads by P2. Note that we could provide an alternative derivation where the loads get reordered by **sll**. **Top right**: Derivation for Fig. 3(b). Both store-load sequences are reordered by **ssl** before being interleaved. **Bottom**: Derivation for Fig. 3(c). Both processors first aggregate the stores with the first following load by **asl**, then delay it past the second load by **ssl**.

rewrite operations in Fig. 4. All of these operations preserve the semantics of single-processor programs, as long as the conditions are observed (**asl** applies only to accesses that target the same location and store/load the same value, while **sss**, **ssl**, and **sls** apply only to accesses that target different locations).

To see how these dynamic rewrite operations can explain the examples in Fig. 3, consider the derivation diagrams in Fig. 5. Each processor first produces a sequence of memory accesses consistent with the program. These sequences are *dynamic*, as they contain data that may not be known statically (such as actual addresses and values loaded or stored), and may repeat program fragments that execute in loops. The access sequences may then be locally modified by the dynamic rewrite operations. Next, the sequences of the processors are interleaved. Informally, an interleaving shuffles the various sequences while maintaining the access order within each sequence (we give a formal definition in Section 3.2). Our derivation diagrams show which sequences are being interleaved with an underbrace. At the end of the derivation (but not necessarily before), the sequence must be *value-consistent*; that is, loaded values must be equal to the latest value stored to the same location, or the initial value if there is no preceding store.

In general, it is quite difficult to establish a precise relationship between abstract memory models (described as a collection of relaxations, in the style of [1]) and official memory model specifications of commercially available multiprocessors. However, it is possible and sensible for research purposes to model just the abstract core of such models, by focusing on the behavior of regular loads and stores. Fig. 4 shows how can model the core of many commercial hardware memory models, and even the CLR memory model, using the dynamic rewrite rules defined in Fig. 3. Our main sources for constructing this table were [16] for 390, [28] for TSO, PSO and RMO, [10] for Alpha, [22] for x86-TSO, and [7,12,21] for CLR.

Beyond simple loads and stores, all of these architectures contain additional constructs (such as locked instructions, compare-and-swaps, various memory fences, or volatile memory accesses). Many of them can be formalized using custom syntax and rewrite rules. However, for simplicity, we stick to regular loads and stores in this paper, augmented only by atomic load-stores (which offer a general method to represent synchronization operations such as locked instructions or compare-and-swap) and a full memory fence. Also, we do not currently model control or data dependencies (which would require us to follow the machine language syntax much more closely, as done in [22], for example).

Some memory models (such as PPC, ARM, RC, and PC) allow stores to be split into separate components for each processor. By combining the **asl** rule with a hierarchical cache organization, our formalism can handle a limited form of store splitting that is sufficient to explain most examples (for more detail on this topic, see [8]).

To correctly handle examples that involve synchronization with spinloops (such as Fig. 1), our formalism must handle infinite executions and model fairness conditions (e.g., the store must eventually be performed). To illustrate the subtleties of infinite rewriting, consider first the program in Fig. 6(a). If we naively

(a)

Initially: $A = B = r = s = 0$

$P1$	$P2$
$A := 1$	**while** $(s == 0)$
while $(r == 0)$	$s := A$
$r := B$	$B := 1$

Eventually: $P1, P2$ do not terminate

(b)

Initially: $A = B = r = s = 0$

$P1$	$P2$
while $(r == 0)\{$	**while** $(s == 0)\{$
$r := B$	$s := A$
$A := 1$	$B := 1$
$B := 0$	$A := 0$
$\}$	$\}$

Eventually: $P1, P2$ do not terminate

Fig. 6. (a) This outcome is not possible: the store by P1 has to reach P2 eventually, and vice versa. (b) This outcome is possible: both processors repeat Dekker forever.

allow infinite applications of **ssl**, the store of A can be delayed past the infinite number of subsequent loads in the while loop. As a result, the program may not terminate, which we would like to disallow for the following reason. On actual hardware, stores are not retained indefinitely, so this program is guaranteed to terminate. Now consider Fig. 6(b). This program is essentially a "repeated Dekker" (Fig. 3(b)) and it is conceivable that both P1 and P2 keep executing forever. To explain such behavior, we need to apply **ssl** infinitely often.

To handle both these examples correctly, our denotational semantics uses *parallel* rewriting on infinite traces (to be formally defined in the next section).

3.1 A Simple Imperative Language for Shared Memory

We now proceed to formalize our description of relaxed memory models. We start by defining a simple imperative "toy" programming language that is sufficient to express the relevant concepts. It is explicitly parallel and distinguishes syntactically between shared variables (uppercase identifiers) and local variables (lowercase identifiers). All variables are mutable and lexically scoped, and must be initialized. For example, the litmus test in Fig. 3(a) looks as follows:

$$\begin{aligned}&\textbf{share } A = 0 \textbf{ in } (\textbf{share } B = 0 \textbf{ in}\\ &\qquad\qquad (\textbf{local } r = 0 \textbf{ in } (\textbf{local } s = 0 \textbf{ in}\\ &\qquad\qquad\qquad ((A := 1;\ \ B := 1) \parallel (r := B;\ \ s := A)))))\end{aligned}$$

The formal syntax is shown in Fig. 7. We let $\mathcal{L}$ be the set of shared variables (locations in shared memory), $\mathcal{R}$ be the set of processor-local variables (registers), $\mathcal{V} = \mathcal{L} \cup \mathcal{R}$ be the set of all variables, and $\mathcal{X}$ be the set of values assumed by the variables.

The (load) and (store) statements move values between local and shared variables. The (assign) statement performs computation, such as addition, on local variables. The (compare-and-swap) statement compares the values of L and r_c, stores r_n to L if they are equal, and assigns the original value of L to r_r. Note that our language does not contain lock or unlock instructions, as there is in fact no

$$
\begin{array}{lcll}
L & \in & \mathcal{L} & \text{(shared variable)} \\
r & \in & \mathcal{R} & \text{(local variable)} \\
x & \in & \mathcal{X} & \text{(value)} \\
f & : & \mathcal{X}^n \to \mathcal{X} & \text{(local computation)}, n \geq 0 \\
s & ::= & \mathbf{skip} & \text{(skip)} \\
 & \mid & r := L & \text{(load)} \\
 & \mid & L := r & \text{(store)} \\
 & \mid & r := f(r_1, \ldots, r_n) & \text{(assign)}, n \geq 0 \\
 & \mid & r_r := \mathit{cas}(L, r_c, r_n) & \text{(compare and swap)} \\
 & \mid & \mathit{fence} & \text{(full memory fence)} \\
 & \mid & \mathit{get}\ r & \text{(read from console)} \\
 & \mid & \mathit{print}\ r & \text{(write to console)} \\
 & \mid & s;\ s & \text{(sequential composition)} \\
 & \mid & s_1 \parallel \cdots \parallel s_n & \text{(parallel composition)}, n \geq 2 \\
 & \mid & \mathbf{if}\, r\, \mathbf{then}\, s\, \mathbf{else}\, s & \text{(conditional)} \\
 & \mid & \mathbf{while}\, r\, \mathbf{do}\, s & \text{(loop)} \\
 & \mid & \mathbf{local}\ r = x\ \mathbf{in}\ s & \text{(local variable declaration)} \\
 & \mid & \mathbf{share}\ L = x\ \mathbf{in}\ s & \text{(shared variable declaration)}
\end{array}
$$

Fig. 7. Syntax of program snippets s

blocking synchronization at the hardware level (blocking synchronization can be implemented using spinloops and compare-and-swap). We also include a (fence) statement to enforce a full memory fence.

The statements (get) and (print) represent simple I/O in the form of reading from or writing to an interactive console. The statements (sequential composition), (conditional) and (loop) have their usual meaning (we let the special value 0 denote false, and all others denote true). The statement (parallel composition) executes its components concurrently, and waits for all of them to finish before completing. The statements (local) and (shared) declare *mutable* variables and initialize them to the given value. Compared to *let*, as used in functional languages, they differ by (1) allowing mutation of the variable, and (2) strictly restricting the scope and lifetime to the nested snippet.

To enforce that local variables are not accessed concurrently, we define the free variables as in (Fig. 8) and call a snippet *ill-formed* if it contains a parallel composition $s_1 \parallel \cdots \parallel s_n$ such that for some i, j, we have $(FV(s_i) \cap FV(s_j) \cap \mathcal{R}) \neq \emptyset$, and *well-formed* otherwise. We let $\mathcal{S}$ be the set of all well-formed snippets.

Finally, we define a *program* to be a well-formed snippet s with no free variables. We let $\mathcal{P}$ be the set of all programs.

Note that conventional hardware memory models consider only a restricted shape of programs (a single parallel composition of sequential processes). Our syntax is more general, as it allows arbitrary nesting of declarations and compositions. This (1) simplifies the definitions and proofs, (2) lets us perform local reasoning (because we can delimit the scope of variables), and (3) allows us to explore the implications of hierarchical memory organizations.

$$\begin{aligned}
FV(\mathbf{skip}) &= \emptyset \\
FV(r := L) &= \{r, L\} \\
FV(L := r) &= \{L, r\} \\
FV(r_0 := f(r_1 \dots r_n)) &= \{r_0, r_1, \dots r_n\} \\
FV(r_r := cas(L, r_c, r_n)) &= \{L, r_r, r_c, r_n\} \\
FV(fence) &= \emptyset \\
FV(get\ r) &= \{r\} \\
FV(print\ r) &= \{r\} \\
FV(s; s') &= FV(s) \cup FV(s') \\
FV(s_1 \parallel \dots \parallel s_n) &= FV(s_1) \cup \dots \cup FV(s_n) \\
FV(\mathbf{if}\ r\ \mathbf{then}\ s\ \mathbf{else}\ s') &= \{r\} \cup FV(s) \cup FV(s') \\
FV(\mathbf{while}\ r\ \mathbf{do}\ s) &= \{r\} \cup FV(s) \\
FV(\mathbf{local}\ r = x\ \mathbf{in}\ s) &= FV(s) \setminus \{r\} \\
FV(\mathbf{share}\ L = x\ \mathbf{in}\ s) &= FV(s) \setminus \{L\}
\end{aligned}$$

Fig. 8. Definition of the set of free variables $FV(s)$ of s

3.2 Denotational Semantics

Our semantics mirror the ideas behind the derivation diagrams used in the previous section. Informally speaking, each processor generates a set of potential traces. These traces are concatenated by sequential composition, interleaved by parallel composition, and modified by the dynamic rewrite operations of the memory model. They are then filtered by requiring value consistency (*after* being interleaved and reordered).

To capture the semantics of a program or snippet more formally, we first define a set $\mathcal{B}$ of behaviors; We then recursively define the semantic function $[\![]\!]_M$ to map any snippet s onto the set $[\![s]\!]_M \subset \mathcal{B}$ of its behaviors for a given memory model M. We represent the memory model M as a set of dynamic rewrite operations, and model its effect on behaviors as a closure operator.

To capture behaviors locally, we use a combination of state valuations (to capture local state) and event traces (to capture externally visible events and accesses to shared variables). Let Q be the set of local states, defined as functions $\mathcal{R} \to \mathcal{X}$, and let Evt be the set of events e of the form

$$e ::= \langle ld\ \ L, x\rangle \mid \langle st\ \ L, x\rangle \mid \langle ldst\ \ L, x_l, x_s\rangle \mid \langle fence\rangle \mid \langle get\ \ x\rangle \mid \langle print\ \ x\rangle.$$

We let Evt^* be the set of finite event sequences (containing in particular the empty sequence, denoted ϵ), we let Evt^ω be the set of infinite event sequences, and we let $Evt^\infty = Evt^* \cup Evt^\omega$ be the set of all event sequences. For two sequences $w \in Evt^*$ and $w' \in Evt^\infty$, we let $ww' \in Evt^\infty$ be the concatenation as usual. For a sequence of finite sequences $w_1, w_2, \dots \in Evt^*$, we let $w_1 w_2 \dots \in Evt^\infty$ be the concatenation (which may be finite or infinite).

We then define the set of *behaviors*

$$\mathcal{B} = (Q \times Q \times Evt^*) \cup (Q \times Evt^\infty).$$

A triple (q, q', w) represents a terminating behavior that starts in local state q, ends in local state q', and emits the finite event sequence w. A pair (q, w)

represents a nonterminating behavior that starts in local state q and emits the (finite or infinite) event sequence w. For a set $B \subseteq \mathcal{B}$ and states $q, q' \subseteq Q$ we define the projections $[B]_{qq'} = \{w \mid (q, q', w) \in B\}$ and $[B]_q = \{w \mid (q, w) \in B\}$.

To specify dynamic rewrite operations formally, we use *rewrite rules* (as in Fig. 4) of the form $p \xrightarrow{\varphi} q$ where p and q are symbolic event sequences (that is, sequences of events where locations and values are represented by variables) and where φ (if present) is a formula over the variables appearing in p and q which describes conditions under which the rewrite rule applies. We let $\mathcal{T}$ be the set of all such rewrite rules.

Definition 1. *A* memory model *is a finite set $M \subset \mathcal{T}$ of rewrite rules.*

Definition 2. *For a rewrite rule $t = p \xrightarrow{\varphi} q$, let $g_t \subset Evt^* \times Evt^*$ be the set of pairs (w_1, w_2) such that there exists a valuation of the variables in p, q for which $p = w_1$, $q = w_2$ and φ is true. Then, define the operator $t : \mathcal{P}(Evt^*) \rightarrow \mathcal{P}(Evt^*)$ to map a set A of finite event sequences to the set*

$$t(A) = \{ww_2w' \mid w, w' \in Evt^* \wedge (w_1, w_2) \in g_t \wedge ww_1w' \in A\}$$

For a set of rewrite rules $M \subset \mathcal{T}$ and a set of finite sequences $A \subset Evt^*$, we define the result of applying M to A as $M(A) = A \cup \bigcup_{t \in M} t(A)$. In order to apply M to infinite sequences as well, we first introduce a definition for parallel rewriting. We generalize the notation for sequence concatenation to sets of sequences as usual (elementwise): for example, for $S \subseteq Evt^*$ and $S' \subseteq Evt^\infty$ we let $SS' = \{ss' \mid s \in S, s \in S'\}$.

Definition 3. *Let $f : \mathcal{P}(Evt^*) \rightarrow \mathcal{P}(Evt^*)$. Then we define the operators $P_f : \mathcal{P}(Evt^*) \rightarrow \mathcal{P}(Evt^*)$ and $\widehat{P_f} : \mathcal{P}(Evt^\infty) \rightarrow \mathcal{P}(Evt^\infty)$ by*

$$P_f(A) = \bigcup \{\, f(A_1) \cdots f(A_n) \mid A_i \subset Evt^* \text{ such that } A_1 \cdots A_n \subseteq A\}$$
$$\widehat{P_f}(\hat{A}) = \bigcup \{\, f(A_1)f(A_2)f(A_3) \cdots \mid A_i \subset Evt^* \text{ such that } A_1A_2A_3 \cdots \subseteq \hat{A}\}$$

Note that $\widehat{P_f}(\hat{A})$ may contain infinite sequences even if $\hat{A}$ does not.[1] We now show how to construct fixpoints for the effect of memory models $M \subseteq \mathcal{T}$ on behaviors.

Definition 4. *Let M be a memory model. We define $M^* : \mathcal{P}(Evt^*) \rightarrow \mathcal{P}(Evt^*)$ and $M^\infty : \mathcal{P}(Evt^\infty) \rightarrow \mathcal{P}(Evt^\infty)$ by*

$$M^*(A) = \bigcup_{k \geq 0} M^k(A) \qquad\qquad M^\infty(\hat{A}) = \bigcup_{k \geq 0} (\widehat{P_{M^*}})^k(\hat{A})$$

Moreover, for a set $B \subseteq \mathcal{B}$ of behaviors, define the closure $B^M =$

$$\{(q, q, w) \mid q, q' \in Q \text{ and } w \in M^*([B]_{qq'})\} \cup \{(q, w) \mid q \in Q \text{ and } w \in M^\infty([B]_q)\}.$$

[1] for example, consider $\hat{A} = \{\epsilon\}$ and $M = \{\epsilon \rightarrow 0\}$. Then $\widehat{P_M}(\hat{A})$ contains the infinite sequence $000 \cdots$.

We can show that this is indeed a closure operation, namely, that $(B^M)^M = B^M$ (see our tech report [8] for a proof). Note that our use of parallel rewriting applies the rewrite rules in a "locally finite" manner, which is important to handle infinite executions correctly.[2]

Definition of the Semantics. Using the notations listed in the next paragraph, Fig. 9 shows our recursive definition of the semantic function $[\![.]\!]_M : \mathcal{S} \to \mathcal{P}(\mathcal{B})$ that assigns to each snippet s the set of behaviors $[\![s]\!]_M$ that s may exhibit on memory model M. It computes behaviors of snippets from the inside out, applying the rewrite rules at each step. Sequential composition appends the behaviors of its constituents, while parallel composition interleaves them. The behaviors of a load include all possible values it could load (because the actual value depends on the context which is not known at this point). Value consistency is enforced at the level of the shared-variable declaration, at which point we also project away accesses to that variable.[3] Fences are modeled as events that do not participate in any rewrite rules, thus enforcing ordering.

Notations used. For $q \in Q$, $r \in \mathcal{R}$ and $x \in \mathcal{X}$ we let $q[r \mapsto x]$ denote the function that maps r to x, but is otherwise the same as the function q. For a shared variable $L \in \mathcal{L}$, let $Evt(L) \subseteq Evt$ be the set of memory accesses to L. For $w \in Evt^\infty$ and $i \in \mathbb{N}$, let $w[i] \in Evt$ be the event at position i (starting with 1). Let $dom\ w \subseteq \mathbb{N}$ be the set of positions of w. For two sequences $w, w' \in Evt^\infty$ we define the set of fair interleavings $(w \# w') \subseteq Evt^\infty$ to consist of all sequences $u \in Evt^\infty$ such that there exist strictly monotonic functions $f : dom\ w \to dom\ u$ and $g : dom\ w' \to dom\ u$ satisfying $rg\ f \cap rg\ g = \emptyset$ and $rg\ f \cup rg\ g = dom\ w$, and such that $w[i] = u[f(i)]$ and $w'[i] = u[g(i)]$ for all valid positions i. Note that the interleaving operator # is commutative and associative. For a subset of events $C \subseteq Evt$, we define the projection function $proj_C : Evt^\infty \to Evt^\infty$ to map a sequence to the largest subsequence containing only events in C. We write $proj_{-L}$ short for the function $proj_{Evt \setminus Evt(L)}$ (which removes all accesses to L). We call a sequence $w \in Evt^\infty$ *value-consistent* with respect to a shared variable $L \in \mathcal{L}$ and an initial value $x \in \mathcal{X}$ if for each load of L appearing in w, the value loaded matches the value of the rightmost store to L that precedes the load in w, or the initial value x if there is no such store. We let $Cons(L, x) \subseteq Evt^\infty$ be the set of all sequences that are value-consistent with respect to L and x. Similarly, we let $Cons(L, x, x') \subseteq Evt^*$ be the set of finite sequences that are value-consistent with respect to initial and final values x and x' of L, respectively. For simplicity, we assume $\mathcal{X} = \mathbb{Z}$.

[2] For example, consider the operation **ssl** in Fig. 4 which represents the effect of stores being delayed in a buffer; while there is no bound on how long stores can be delayed, they must be eventually performed. Our formalism reflects this properly, as follows (using digits 0,1 instead of load and store events for illustration purposes). Let $A = \{1010\dots\}$ and $M = \{10 \to 01\}$. Then $0^k1010\dots$ is in $M^\infty(A)$, but $000\cdots$ is not.

[3] This behavior is similar to "hide" operators in process algebras. It implies that the behaviors of a program (unlike the behaviors of snippets) contain only external events.

$$
\begin{aligned}
\llbracket \mathbf{skip} \rrbracket_M &= \{(q, q, \epsilon) \mid q \in Q\}^M \\
\llbracket r :=_h L \rrbracket_M &= \{(q, q[r \mapsto x], \langle ld\ L, x \rangle) \mid q \in Q, x \in \mathcal{X}\}^M \\
\llbracket L :=_h r \rrbracket_M &= \{(q, q, \langle st\ L, q(r) \rangle \mid q \in Q\}^M \\
\llbracket r_0 := f(r_1 \ldots r_n) \rrbracket_M &= \{(q, q[r_0 \mapsto f(q(r_1) \ldots q(r_n)], \epsilon) \mid q \in Q\}^M \\
\llbracket r_r := cas_h(L, r_c, r_n) \rrbracket_M &= \begin{pmatrix} \{(q, q[r_r \mapsto q(r_c)], \langle ldst\ L, q(r_c), q(r_n) \rangle) \mid q \in Q\} \\ \cup \{(q, q[r_r \mapsto x], \langle ldst\ L, x, x \rangle) \mid q \in Q, x \in \mathcal{X}, x \neq q(r_c)\} \end{pmatrix}^M \\
\llbracket get\ r \rrbracket_M &= \{(q, q[r \mapsto x], \langle get\ x \rangle) \mid q \in Q, x \in \mathcal{X}\}^M \\
\llbracket print\ r \rrbracket_M &= \{(q, q, \langle print\ q(r) \rangle \mid q \in Q\}^M
\end{aligned}
$$

$$
\llbracket s_1; s_2 \rrbracket_M = \\
\begin{pmatrix}
\{(q, q', w) \mid \text{there exist } (q, q'', w_1) \in \llbracket s_1 \rrbracket_M \text{ and } (q'', q', w_2) \in \llbracket s_2 \rrbracket_M \text{ with } w = w_1 w_2\} \\
\cup \{(q, w) \mid (q, w) \in \llbracket s_1 \rrbracket_M\} \\
\cup \{(q, w) \mid \text{there exist } (q, q', w_1) \in \llbracket s_1 \rrbracket_M \text{ and } (q', w_2) \in \llbracket s_2 \rrbracket_M \text{ with } w = w_1 w_2\}
\end{pmatrix}^M
$$

$$
\llbracket s_1 \parallel \cdots \parallel s_n \rrbracket_M = \\
\begin{pmatrix}
\{(q, q', w) \mid \text{there exist } (q, q_i, w_i) \in \llbracket s_i \rrbracket_M \text{ for all } 1 \leq i \leq n \text{ such that} \\
\qquad w \in w_1 \# \ldots \# w_n \text{ and such that } q'(r) = q_i(r) \text{ for all } r \in FV(s_i) \text{ and} \\
\qquad q'(r) = q(r) \text{ for all } r \notin FV(s_1) \cup \ldots FV(s_n)\} \\
\cup \{(q, w) \mid \text{ there exist } w_1, \ldots, w_n \in Evt^\infty \text{ and a nonempty subset } D \subseteq \{1, \ldots, n\} \\
\qquad \text{such that for all } j \in D, \text{ we have a behavior } (q, w_j) \in \llbracket s_j \rrbracket_M, \\
\qquad \text{and for all } j \notin D, \text{ we have a behavior } (q, q_j, w_j) \in \llbracket s_j \rrbracket_M \text{ for some } q_j, \\
\qquad \text{and } w \in w_1 \# \ldots \# w_n\}
\end{pmatrix}^M
$$

$$
\llbracket \mathbf{if}\ r\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \rrbracket_M = \\
\begin{pmatrix}
\{(q, q', w) \mid (q(r) \neq 0 \wedge (q, q', w) \in \llbracket s_1 \rrbracket_M) \vee (q(r) = 0 \wedge (q, q', w) \in \llbracket s_2 \rrbracket_M)\} \\
\cup \{(q, w) \mid (q(r) \neq 0 \wedge (q, w) \in \llbracket s_1 \rrbracket_M) \vee (q(r) = 0 \wedge (q, w) \in \llbracket s_2 \rrbracket_M)\}
\end{pmatrix}^M
$$

$$
\llbracket \mathbf{while}\ r\ \mathbf{do}\ s \rrbracket_M = \\
\begin{pmatrix}
\{(q_0, q_n, w_1 \cdots w_n) \mid \text{there exist } n \geq 0 \text{ and } q_0, \ldots, q_n \text{ such that } (q_i, q_{i+1}, w_{i+1}) \in \llbracket s \rrbracket_M \\
\qquad \text{for } 0 \leq i < n, \quad \text{and } q_0(r) \neq 0, \ldots, q_{n-1}(r) \neq 0, \text{ and } q_n(r) = 0\} \\
\cup \{(q_0, w_1 w_2 \cdots) \mid \exists q_1, q_2, \ldots : (q_i, q_{i+1}, w_{i+1}) \in \llbracket s \rrbracket_M \text{ and } q_i(r) \neq 0\} \\
\cup \{(q_0, w_1 \cdots w_n) \mid \text{there exist } n \geq 1 \text{ and } q_0, \ldots, q_{n-1} \text{ such that } q_i(r) \neq 0 \text{ for all } i \text{ and} \\
\qquad (q_i, q_{i+1}, w_{i+1}) \in \llbracket s \rrbracket_M \text{ for } 0 \leq i < n-1 \text{ and } (q_{n-1}, w_n) \in \llbracket s \rrbracket_M\}
\end{pmatrix}^M
$$

$$
\llbracket \mathbf{local}\ L = x\ \mathbf{in}\ s \rrbracket_M = \\
\begin{pmatrix}
\{(q, q', w) \mid \text{ there exists a behavior } (q[r \mapsto x], q'', w) \in \llbracket s \rrbracket_M \\
\qquad \text{such that } q' = q''[r \mapsto q(r)]\} \\
\cup \{(q, w) \mid \text{ there exists a behavior } (q[r \mapsto x], w) \in \llbracket s \rrbracket_M\}
\end{pmatrix}^M
$$

$$
\llbracket \mathbf{share}\ L = x\ \mathbf{in}\ s \rrbracket_M = \\
\begin{pmatrix}
\{(q, q', w) \mid \text{ there exists a behavior } (q, q', w') \in \llbracket s \rrbracket_M \\
\qquad \text{such that } w' \in Cons(L, x) \text{ and } w = proj_{-L}(w')\} \\
\cup \{(q, w) \mid \text{ there exists a behavior } (q, w') \in \llbracket s \rrbracket_M \\
\qquad \text{such that } w' \in Cons(L, x) \text{ and } w = proj_{-L}(w')\}
\end{pmatrix}^M
$$

Fig. 9. Denotational Semantics of our Calculus, parameterized by a set M of dynamic rewrite rules. An empty set M represents the standard semantics (sequential consistency).

$$\begin{array}{cccc}
p_1 & p_2 & p_3 & p_4 \\
\left[\begin{array}{l}\textbf{local } r = 1 \textbf{ in} \\ \textbf{local } s = 2 \textbf{ in} \\ (\mathit{print}\ r) \parallel (\mathit{print}\ s)\end{array}\right] &
\left[\begin{array}{l}\textbf{local } r = 1 \textbf{ in} \\ \textbf{local } s = 2 \textbf{ in} \\ \mathit{print}\ r; \\ \mathit{print}\ s\end{array}\right] &
\left[\begin{array}{l}\textbf{local } r = 1 \textbf{ in} \\ \textbf{while } r \textbf{ do} \\ \quad \mathit{print}\ r\end{array}\right] &
\left[\begin{array}{l}\textbf{local } r = 0 \textbf{ in} \\ \mathit{get}\ r; \\ \textbf{while } r \textbf{ do} \\ \quad \textbf{skip}; \\ \mathit{print}\ r\end{array}\right]
\end{array}$$

Fig. 10. Four example programs. p_1 and p_2 always terminate, p_3 never terminates, and p_4 sometimes terminates. p_1 can be soundly transformed to p_2, but not vice versa.

4 Verifying Local Program Transformations

In this section, we present our methodology for verifying the soundness of local program transformations on a chosen hardware memory model. We start with a general definition of what can be observed about a program execution. Next, we show how to prove that a local, static program transformation is unobservable (and thus sound) if its effect on dynamic traces can be captured by *invisible rewrite rules* on those traces (Section 4.1) . For each memory model, we present a list of invisible rewrite rules and describe how we proved invisibility.

For our purposes, the observable behavior of a program includes (1) whether the program terminates or diverges, and (2) the sequence of externally visible events (that is, interactions of the program with the environment). We formalize this by defining the subset $\mathit{Ext} \subset \mathit{Evt}$ of externally visible events and the set $\mathcal{O}$ of observations as

$$\begin{aligned}
\mathit{Ext} &= \{\langle \mathit{get}\ n\rangle \mid n \in \mathbb{Z}\} \cup \{\langle \mathit{print}\ n\rangle \mid n \in \mathbb{Z}\} \\
\mathcal{O} &= \{u \mid u \in \mathit{Ext}^*\} \cup \{\nabla u \mid u \in \mathit{Ext}^\infty\}
\end{aligned}$$

An observation of the form u represents a terminating execution that produces the finite event sequence u; an observation of the form ∇u represents a nonterminating execution that produces the (finite or infinite) sequence u. For example, the program p_1 in Fig. 10 has two possible observations, $\langle \mathit{print}\ 1\rangle\langle \mathit{print}\ 2\rangle$ and $\langle \mathit{print}\ 2\rangle\langle \mathit{print}\ 1\rangle$; the program p_2 has one possible observation, $\langle \mathit{print}\ 1\rangle\langle \mathit{print}\ 2\rangle$; the program p_3 has one possible observation, $\nabla\langle \mathit{print}\ 1\rangle^\omega$; and the program p_4 has the set $\{\langle \mathit{get}\ 0\rangle\langle \mathit{print}\ 0\rangle\} \cup \{\nabla\langle \mathit{get}\ n\rangle \mid n \neq 0\}$ of observations.

Using the semantics established in the previous section, we now formally define the set of observations of a program p on a memory model M as follows:

$$\begin{aligned}
\mathit{obs}_M(p) = \{u \mid \exists(q, q', w) \in [\![p]\!]_M : u = \mathit{proj}_{\mathit{Ext}}(w)\} \\
\cup\ \{\nabla u \mid \exists(q, w) \in [\![p]\!]_M : u = \mathit{proj}_{\mathit{Ext}}(w)\}
\end{aligned}$$

For programs $p, p' \in \mathcal{P}$, we let $\langle p \Rightarrow p'\rangle$ represent the *global transformation* of p into p'. We then define a global transformation $\langle p \Rightarrow p'\rangle$ to be *sound* for memory model M if it does not introduce any new observations, that is, $\mathit{obs}_M(p') \subseteq \mathit{obs}_M(p)$.

Note that we consider it acceptable if the transformed program has *fewer* observations than the original one. For example, we would consider it o.k. to

$$\begin{array}{rl}
\textbf{edl} \text{ (eliminate double load)} & : \langle ld\ L, x\rangle\langle ld\ L, x\rangle \rightarrow \langle ld\ L, x\rangle \\
\textbf{eds} \text{ (eliminate double store)} & : \langle st\ L, x\rangle\langle st\ L, x'\rangle \rightarrow \langle st\ L, x'\rangle \\
\textbf{ecs} \text{ (eliminate confirmed store)} & : \langle st\ L, x\rangle\langle ld\ L, x\rangle \rightarrow \langle st\ L, x\rangle \\
\textbf{asl} \text{ (aggregate store-load)} & : \langle st\ L, x\rangle\langle ld\ L, x\rangle \rightarrow \langle st\ L, x\rangle \\
\textbf{iil} \text{ (invent irrelevant load)} & : \epsilon \rightarrow \langle ld\ L, *\rangle \\
\textbf{eil} \text{ (eliminate irrelevant load)} & : \langle ld\ L, *\rangle \rightarrow \epsilon
\end{array}$$

Fig. 11. A list of rewrite rules that are invisible for certain memory models. The last two contain wildcards; the meaning is that those rules apply to sets of behaviors, rather than indidvidual behaviors.

transform program p_1 to program p_2 in Fig. 10, which essentially reduces the nondeterministic choices available to the scheduler in scheduling the two print statements. An external entity interacting with the program cannot conclusively detect that a transformation took place. The reason is that schedulers are free to favor certain schedules over others (as long as the schedules themselves are fair). Therefore, an observer can not tell whether the reduction in schedules is caused by the transformation or by a whim of the scheduler.

In this work, we focus on local transformations, that is, transformations of components whose context is not known. See Fig. 12 for 8 examples of local transformations. More formally, we define a *program context* to be a "program with a hole [ ]", defined syntactically as follows:

$$\begin{array}{llllllllll}
c ::= & [\,] & | & c\ ;\ s & | & s\ ;\ c & | & \textbf{local}\ r = x\ \textbf{in}\ c & | & \textbf{share}\ L = x\ \textbf{in}\ c \\
 & & | & \multicolumn{3}{l}{\textbf{while}\ r\ \textbf{do}\ c} & | & \textbf{if}\ r\ \textbf{then}\ c\ \textbf{else}\ s & | & \textbf{if}\ r\ \textbf{then}\ s\ \textbf{else}\ c \\
 & & | & \multicolumn{7}{l}{s_1 \parallel \cdots \parallel s_{k-1} \parallel c \parallel s_{k+1} \parallel \cdots \parallel s_n \qquad (\text{where } 1 \leq k \leq n)}
\end{array}$$

For a context c and snippet s, we let $c[s]$ be the snippet obtained by replacing the hole in c with s. For two snippets $s, s' \in \mathcal{S}$, we let $\langle s \rightarrow s'\rangle$ be a *local transformation*. We say a local transformation $\langle s \rightarrow s'\rangle$ *induces* a global transformation $\langle p \Rightarrow p'\rangle$ if there exists a context c such that $p = c[s]$, $p' = c[s']$, and we say a local transformation is *sound* if all induced global transformations are sound.

4.1 Invisible Rewrite Rules

To determine whether a local transformation $\langle s \rightarrow s'\rangle$ (such as shown in Fig. 12) is sound, we can compare the set of behaviors $[\![s]\!]_M$ and $[\![s']\!]_M$. Because our denotational semantics is defined recursively, it is quite obvious that $[\![s']\!]_M = [\![s]\!]_M$ implies $obs_M(c[s']) = obs_M(c[s])$ in any context c, and thus that the transformation is sound. Unfortunately, not all transformations are that simple to prove, because a transformation can be sound even if $[\![s']\!]_M \neq [\![s]\!]_M$ (our semantics is not fully abstract).[4]

[4] For example, consider the "redundant read-after-read elimination" transformation from Fig. 12, and consider $M = SC = \emptyset$. Clearly, the sets $[\![s']\!]_M$ and $[\![s]\!]_M$ are not the same and not contained in each other (all behaviors of $[\![s']\!]_M$ contain one fewer load). Nevertheless, this transformation is actually safe, because the removal of the read can not be observed by any context.

$$\begin{array}{rl}
\text{(load reordering)} & \{\mathbf{if}\ r\ \mathbf{then}\ \{s := A;\ t := B\}\ \mathbf{else}\ \{t := B;\ s := A\}\} \\
& \to \{s := A;\ t := B\} \\
\text{(store reordering)} & \{\mathbf{if}\ r\ \mathbf{then}\ \{A := s;\ B := t\}\ \mathbf{else}\ \{B := t;\ A := s\}\} \\
& \to \{A := s;\ B := t\} \\
\text{(irrelevant read elim.)} & \{\mathbf{local}\ r = 0\ \mathbf{in}\ \{r := A;\ \mathbf{if}\ r\ \mathbf{then}\ \{B := s\}\ \mathbf{else}\ \{B := s\}\}\} \\
& \to \{B := s\} \\
\text{(irrelevant read introd.)} & \{\mathbf{if}\ r\ \mathbf{then}\ \mathbf{local}\ s = 0\ \mathbf{in}\ \{s := A;\ B := s\}\} \\
& \to \{\mathbf{local}\ s = 0\ \mathbf{in}\ \{s := A;\ \mathbf{if}\ r\ \mathbf{then}\ B := s\}\}\}
\end{array}$$

$$\begin{array}{rl}
\text{(redundant read-after-read elim.)} & \{r := A;\ b := A\} \to \{r := A;\ b := r\} \\
\text{(redundant read-after-write elim.)} & \{A := r;\ s := A\} \to \{A := r;\ s := r\} \\
\text{(redundant write-before-write elim.)} & \{A := r;\ A := s\} \to \{A := s\} \\
\text{(redundant write-after-read elim.)} & \{r := A; \mathbf{if}\ r == 0\ \mathbf{then}\ A := 0\} \to \{r := A\}
\end{array}$$

Fig. 12. Some examples of local transformations [26]. The snippets follow the syntax defined in §3.1, with $\mathcal{L} = \{A, B, \dots\}$ and $\mathcal{R} = \{r, s, t, \dots\}$.

To handle a larger generality of transformations, we introduce the concept of "invisible" rewrite rules on dynamic traces. Essentially, we show that certain dynamic rewrite operations never alter the set of observations. In particular, any rewrite rule that is already part of the memory model is invisible. In general, there can be many more such rules, however. Consider the rules shown in Fig. 11. All of these rules are "invisible" on at least some of the memory models.

More formally, we say a local transformation $\langle s \to s' \rangle$ is *covered* by a set of rewrite rules D if

$$[\![s']\!]_M \subseteq f_D([\![s]\!]_M),$$

where the operator $f_D : \mathcal{P}(\mathcal{B}) \to \mathcal{P}(\mathcal{B})$ on behaviors is defined as parallel rewriting[5]

$$[f_D(B)]_{qq'} = P_D([B]_{qq'}) \qquad [f_D(B)]_q = \widehat{P_D}([B]_q).$$

The following definition and theorem relate how invisibility provides the means to prove the soundness of a local transformation by showing that it is covered by some set D of invisible rules.

Definition 5 (Invisibility). *Let D be a set of rewrite rules, and let M be a memory model. We say D is* invisible *on M if it is the case that any local transformation that is covered by D is sound for M. We say an individual rule d is invisible on M if the set $\{d\}$ is invisible on M.*

Theorem 1. *The dynamic rewrite rules **edl**, **eds**, **ecs**, **asl**, **eil**, and **iil** are invisible on SC, the rules **edl**, **eds**, **eil**, and **iil** are invisible on TSO,* 390 *and PSO, the rules **edl**, **eds**, **eil**, and **iil** are invisible on CLR, and the set $\{\textbf{eds}, \textbf{ecs}\}$ is invisible on PSO.*

[5] Recall our earlier definition of $[X]_{qq'} = \{w \mid (q, q', w) \in X\}$ and $[X]_q = \{w \mid (q, w) \in X\}$ for a set $X \subset \mathcal{B}$ of behaviors.

The proof of Thm. 1 is based on structural induction, and is available in our tech report [8]. However, walking through the entire proof whenver we wish to enlarge the list of rules or memory models in Thm. 1 is unpractical. Thus, we have broken out a set of conditions that are sufficient to prove invisibility, and can be checked with relative ease.

Theorem 2 (Simple Conditions for Invisibility). *Let $M \subseteq \mathcal{T}$ be a memory model, and let $D \subseteq \mathcal{T}$ be a set of rewrite rules. Then the following conditions are sufficient to guarantee that D is invisible on M:*

1. ***(Commutativity).*** *$m(P_D(A)) \subseteq P_D(M^*(A))$ for all $m \in M$ and $A \subseteq Evt^*$.*
2. ***(Atomicity).*** *if $(S_1, S_2) \in G_d$ for some $d \in D$, then all sequences in S_2 are of length 0 or 1.*
3. ***(Value Consistency)*** *if $(S_1, S_2) \in G_d$ for some $d \in D$, and $w_2 \in S_2 \cap Cons(L, x, x')$ for some L, x, and x', then there exists a $w_1 \in S_1 \cap Cons(L, x, x')$ such that $proj_{-L}(\{w_2\}) \in D(proj_{-L}(\{w_1\}))$.*
4. ***(External Consistency)*** *if $(S_1, S_2) \in G_d$ for some $d \in D$, then $proj_{Ext}(S_2) \subseteq proj_{Ext}(S_1)$.*

We illustrate the use of these conditions by walking through one case, namely $M = 390 = \{\mathbf{ssl}\}$ and $D = \{\mathbf{edl}\}$. Atomicity is immediate (the right-hand side of **edl** is a single event). Value Consistency is straightforward because the left- and right-hand side of **edl** are functionally equivalent, and the projection $proj_{-L}$ will either map them both to ϵ or both to themselves. External Consistency is trivial as **edl** does not contain external events. Commutativity requires some work. To show that $\mathbf{ssl}(P_{\mathbf{edl}}(A)) \subset P_{\mathbf{edl}}(\mathbf{ssl}^*(A))$ for all A, we examine $\mathbf{ssl}(P_{\mathbf{edl}}(A))$ and think about all possible scenarios where **ssl** rewrites modified positions of a parallel application of **edl** (if it rewrites only unmodified positions, it clearly commutes with $P_{\mathbf{edl}}$). Thinking about this scenario (matching the left-hand side of **ssl** with the right-hand side of **edl**), we can single out the following situation

$$\begin{aligned} \langle st\ L, x\rangle\langle ld\ L', x'\rangle\langle ld\ L', x'\rangle &\in A \\ \langle st\ L, x\rangle\langle ld\ L', x'\rangle &\in P_D(A) \\ \langle ld\ L', x'\rangle\langle st\ L, x\rangle &\in \mathbf{ssl}(P_{\mathbf{edl}}) \end{aligned}$$

Now we understand that starting with the same first line, we can get to the same last line by first applying **ssl** *twice* and then applying $P_{\mathbf{edl}}$:

$$\begin{aligned} \langle st\ L, x\rangle\langle ld\ L', x'\rangle\langle ld\ L', x'\rangle &\in A \\ \langle ld\ L', x'\rangle\langle st\ L, x\rangle\langle ld\ L', x'\rangle &\in \mathbf{ssl}(A) \\ \langle ld\ L', x'\rangle\langle ld\ L', x'\rangle\langle st\ L, x\rangle &\in \mathbf{ssl}(\mathbf{ssl}(A)) \\ \langle ld\ L', x'\rangle\langle st\ L, x\rangle w' &\in P_{\mathbf{edl}}(\mathbf{ssl}(\mathbf{ssl}(A))) \end{aligned}$$

which implies the claim.

5 Application

To simplify the task of proving or refuting soundness, we automated some parts of the proof by developing a tool called Traver, written in F# and using the

transformation name (see Fig. 12)	*SC*	390	*TSO*	*PSO*	*CLR*
(load reordering)	×	×	×	×	√
(store reordering)	×	×	×	√	×
(irrelevant read elim.)	√ **(eil)**	√ **(eil)**	√ **(eil)**	√ **(eil)**	√ **(eil)**
(irrelevant read intr.)	√ **(iil)**	√ **(iil)**	√ **(iil)**	√ **(iil)**	√ **(iil)**
(red. read-after-read elim.)	√ **(edl)**	√ **(edl)**	√ **(edl)**	√ **(edl)**	√ **(edl)**
(red. wr.-bef.-wr. elim.)	√ **(eds)**	√ **(eds)**	√ **(eds)**	√ **(eds)**	√ **(eds)**
(red. read-after-wr. elim.)	√ **(asl)**	×	√	√	√
(red. wr.-after-read elim.)	√ **(ecs)**	×	×	√ **(eds, ecs)**	×

Fig. 13. Soundness results for the examples from Fig. 12. For sound transformations (marked by √), we list the set D of invisible rules employed by the proof. For unsound transformations (marked by ×), we show example derivations in Fig. 14. All results were validated by our tool.

automated theorem prover Z3 [11]. It operates in one of two modes, verification or falsification.

- In verification mode, Traver takes as input a local transformation $\langle s \rightarrow s' \rangle$, a memory model M, and a set D of invisible rewrite rules supplied by the user. It then executes both s and s' symbolically to obtain symbolic representations of their behaviors, and attempts to prove that D covers $\langle s \rightarrow s' \rangle$ by computing the closure of $[\![s]\!]_M$ under D and checking whether it contains $[\![s']\!]_M$. If successful, soundness is established. Otherwise, the result is inconclusive,and Traver reports a behavior in the set difference to the user (which can be inspected to find new candidates for invisible rules that may help to prove soundness, or provide ideas on how to falsify the transformation).
- In falsification mode, Traver takes as input a local transformation $\langle s \rightarrow s' \rangle$, a memory model M, and a context c (which may contain several threads). It then computes the closure of $c[s']$ and $c[s]$ under interleavings under M, and solves for a behavior of $c[s']$ that is not observationally equivalent to any behavior in $c[s]$ (assuming that *all* initial and final values of all variables are being observed) . If such a behavior is found, soundness has been successfully refuted. Otherwise, the result is inconclusive.

For both modes, the snippets s, s' are supplied to Traver using a sugared syntax, which makes it very easy to try out many different local transformations (however, we currently support loop-free snippets without parallel composition only). The model M is specified by selecting a subset of the rewrite rules in Fig. 4 and Fig. 11 (not including rewrite rules that are conditional on control or data dependencies).

Using our tool, we successfully proved or refuted soundness of the 8 transformations in Fig. 12 for the memory models SC, 390, TSO,[6] PSO, and CLR as defined in Fig.4. The total time needed by the tool to prove/refute all examples is about 15 seconds. The results are shown in Fig. 13.

[6] Note that the results for TSO also apply for x86-TSO and for x86-IRIW

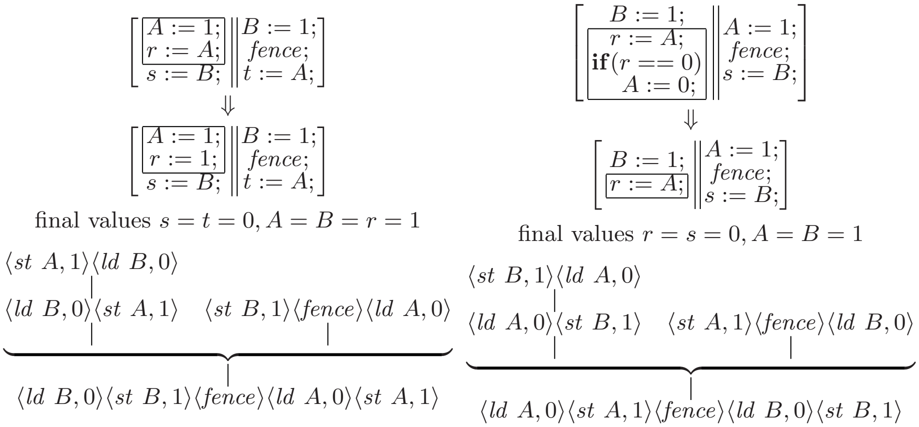

A := 1; r := A; s := B; ∥ B := 1; fence; t := A;
⇓
A := 1; r := 1; s := B; ∥ B := 1; fence; t := A;
final values s = t = 0, A = B = r = 1
⟨st A,1⟩⟨ld B,0⟩
⟨ld B,0⟩⟨st A,1⟩
⟨st B,1⟩⟨fence⟩⟨ld A,0⟩
⟨ld B,0⟩⟨st B,1⟩⟨fence⟩⟨ld A,0⟩⟨st A,1⟩
B := 1; r := A; if(r == 0) A := 0; ∥ A := 1; fence; s := B;
⇓
B := 1; r := A; ∥ A := 1; fence; s := B;
final values r = s = 0, A = B = 1
⟨st B,1⟩⟨ld A,0⟩
⟨ld A,0⟩⟨st B,1⟩
⟨st A,1⟩⟨fence⟩⟨ld B,0⟩
⟨ld A,0⟩⟨st A,1⟩⟨fence⟩⟨ld B,0⟩⟨st B,1⟩

Fig. 14. (Left.) Derivation showing that the redundant-read-after-write-elimination is not sound on 390. **(Right.)** Derivation showing that the redundant-write-after-read-elimination is not sound on 390, on *TSO*, and on *CLR*. **(Both.)** We show the original program, the transformed program, and an execution of the transformed program that is not possible on the original program. All shared variables and registers are initially zero.

As expected, the first two transformations (load-reordering, store-reordering) are unsound for all models except models that specifically relax load-load order or store-store order.

The next four transformations (irrelevant-read-elimination, irrelevant-read-introduction, redundant-read-after-read-elimination, and redundant-write-before-write-elimination) are sound for all memory models. The last two transformations proved more interesting. For both, we were able to prove that they are sound on *SC*. However, they exhibit some surprising behavior on relaxed memory models.

- The redundant-read-after-write-elimination is unsound on 390. Fig. 14 (left) shows a derivation to explain this effect. Intuitively, the sequence $\{A := r;\ s := A\}$ has a fence-like effect on 390 which is lost by the transformation. However, on memory models that also support store-load forwarding (**asl**), this transformation is sound.
- The redundant-write-after-read elimination is unsound on 390, *TSO*, and *CLR*, but sound on *PSO*. Fig. 14 (right) shows a derivation to explain this effect. Intuitively, the reason is that because the transformed snippet is a simple load, it can be swapped with a preceding store if the rule **ssl** is part of the memory model. This would not be possible with the original code unless the memory model also contains the rule **sss** which in turn sheds some light on why this transformation is sound for *PSO*.

We believe it would have been very difficult to correctly determine soundness of these transformations (in particular the last two) or to discover the derivations that explain the effects without our proof methodology.

6 Conclusion and Future Work

Our experience with Traver has successfully demonstrated the power of formalism and automation in discovering corner cases where normal intuition fails. We believe that the proof methodology and the tool presented in the paper have many more uses in the future. Of particular interest are (1) verifying translations involving different memory models (between different architectures, or between different intermediate representations), and (2) extending our methodology to transformations involving higher-level synchronization such as locks, semaphores, or sending and receiving messages on channels.

References

1. Adve, S., Gharachorloo, K.: Shared memory consistency models: a tutorial. Computer 29(12), 66–76 (1996)
2. Adve, S., Hill, M.: A unified formalization of four shared-memory models. IEEE Trans. Parallel Distrib. Syst. 4(6), 613–624 (1993)
3. Arvind, Maessen, J.-W.: Memory model = instruction reordering + store atomicity. In: ISCA, pp. 29–40 (2006)
4. Boehm, H.-J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: Programming Language Design and Implementation (PLDI), pp. 68–78 (2008)
5. Boudol, G., Petri, G.: Relaxed memory models: an operational approach. In: Principles of Programming Languages, POPL (2009)
6. Brookes, S.: Full abstraction for a shared variable parallel language. In: LICS, pp. 98–109 (1993)
7. Brumme, C.: cbrumme's weblog, `http://blogs.gotdotnet.com/cbrumme/archive/2003/05/17/51445.aspx`
8. Burckhardt, S., Musuvathi, M., Singh, V.: Verification of compiler transformations for concurrent programs. Technical Report MSR-TR-2008-171, Microsoft Research (2008)
9. Cenciarelli, P., Sibilio, E.: The java memory model: Operationally, denotationally, axiomatically. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 331–346. Springer, Heidelberg (2007)
10. Compaq Computer Corporation. Alpha Architecture Reference Manual, 4th edn. (January 2002)
11. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
12. Duffy, J.: Joe Duffy's Weblog, `http://www.bluebytesoftware.com/blog/2007/11/10/CLR20MemoryModel.aspx`
13. Sarkar, S., et al.: The semantics of x86-CC multiprocessor machine code. In: Principles of Programming Languages, POPL (2009)
14. Gharachorloo, K.: Memory Consistency Models for Shared-Memory Multiprocessors. PhD thesis, University of Utah (2005)
15. Intel Corporation. Intel 64 Architecture Memory Ordering White Paper (August 2007)
16. International Business Machines Corporation. z/Architecture Principles of Operation, 1st edn. (December 2000)

17. Klein, G., Nipkow, T.: A machine-checked model for a java-like language, virtual machine, and compiler. ACM Transactions on Programming Languages and Systems 28(4), 619–695 (2006)
18. Lerner, S., Millstein, T., Chambers, C.: Automatically proving the correctness of compiler optimizations. In: Programming Language Design and Implementation (PLDI), pp. 220–231 (2003)
19. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Principles of programming languages (POPL), pp. 42–54 (2006)
20. Manson, J., Pugh, W., Adve, S.: The Java memory model. In: Principles of Programming Languages (POPL), pp. 378–391 (2005)
21. Morrison, V.: Understand the impact of low-lock techniques in multithreaded apps. MSDN Magazine 20(10) (October 2005)
22. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO (extended version). Technical Report UCAM-CL-TR-745, Univ. of Cambridge (2009)
23. Park, S., Dill, D.L.: An executable specification, analyzer and verifier for RMO (relaxed memory order). In: Symposium on Parallel Algorithms and Architectures (SPAA), pp. 34–41 (1995)
24. Saraswat, V., Jagadeesan, R., Michael, M., von Praun, C.: A theory of memory models. In: PPoPP 2007: Principles and practice of parallel programming, pp. 161–172 (2007)
25. Sevcik, J.: Program Transformations in Weak Memory Models. PhD thesis, University of Edinburgh (2008)
26. Sevcik, J., Aspinall, D.: On validity of program transformations in the Java memory model. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 27–51. Springer, Heidelberg (2008)
27. Shen, X., Arvind, Rudolph, L.: Commit-reconcile & fences (crf): A new memory model for architects and compiler writers. In: ISCA, pp. 150–161 (1999)
28. Weaver, D., Germond, T. (eds.): The SPARC Architecture Manual Version 9. PTR Prentice Hall, Englewood Cliffs (1994)
29. Young, W.D.: A mechanically verified code generator. Journal of Automated Reasoning 5(4), 493–518 (1989)

Practical Extensions to the IFDS Algorithm

Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez

University of Waterloo, Canada
{nanaeem,olhotak,j2rodrig}@uwaterloo.ca

Abstract. This paper presents four extensions to the Interprocedural Finite Distributive Subset (IFDS) algorithm that make it applicable to a wider class of analysis problems. IFDS is a dynamic programming algorithm that implements context-sensitive flow-sensitive interprocedural dataflow analysis. The first extension constructs the nodes of the supergraph on demand as the analysis requires them, eliminating the need to build a full supergraph before the analysis. The second extension provides the procedure-return flow function with additional information about the program state before the procedure was called. The third extension improves the precision with which ϕ instructions are modelled when analyzing a program in SSA form. The fourth extension speeds up the algorithm on domains in which some of the dataflow facts subsume each other. These extensions are often necessary when applying the IFDS algorithm to non-separable (i.e. non-bit-vector) problems. We have found them necessary for alias set analysis and multi-object typestate analysis. In this paper, we illustrate and evaluate the extensions on a simpler problem, a variation of variable type analysis.

1 Introduction

The Interprocedural Finite Distributive Subset (IFDS) algorithm [15] is an efficient and precise, context-sensitive and flow-sensitive dataflow analysis algorithm for the class of problems that satisfy its restrictions. Although this class includes the classic bit-vector dataflow problems, the original IFDS algorithm is not directly suitable for more interesting problems for which context- and flow-sensitivity would be useful, particularly problems involving objects and pointers. The algorithm can be extended to solve this larger class of problems, however, and in this paper, we present four such extensions.

The IFDS algorithm is an efficient dynamic programming instantiation of the functional approach to interprocedural analysis [19]. The fundamental restrictions of the algorithm, which we do not seek to eliminate in this paper, are that the analysis domain must be a powerset of some finite set D, and that the dataflow functions must be distributive. We present a detailed overview of the IFDS algorithm in Section 2, and further illustrate the algorithm with a running example variable type analysis in Section 3.

A more practical restriction is that the set D must be small, because the algorithm requires as input a so-called exploded supergraph, and the number of nodes in this supergraph is approximately the product of the size of D and the number of instructions in the program. Our first extension, presented in Section 4, removes the restriction on the size of D by enabling the algorithm to compute only those parts of the supergraph that are actually reached in the analysis. This allows the algorithm to be used for problems

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 124–144, 2010.

in which D is theoretically large, but only a small subset of D is encountered during the analysis, which is typical of analyses modelling objects and pointers.

A second practical restriction of the original IFDS algorithm is that it provides limited information to flow functions modelling return flow from a procedure. For many analyses, mapping dataflow facts from the callee back to the caller requires information about the state before the procedure was called. In Section 5, we extend the IFDS algorithm to provide this information to the return flow function.

A third limitation of many standard dataflow analysis algorithms, IFDS included, is that they can be less precise on a program in Static Single Assignment (SSA) form [2] than on the original non-SSA form of the program. When an instruction has multiple control flow predecessors, incoming dataflow facts are merged before the flow function is applied; this imprecisely models the semantics of ϕ instructions in SSA form. In Section 6, we present an example that exhibits this imprecision, and we extend the IFDS algorithm to avoid it, so that it is equally precise on SSA form as on non-SSA form programs. SSA form is not only a convenience; in prior work, we showed that SSA form can be used to improve running time and space requirements of analyses such as alias set analysis [13].

Finally, the IFDS algorithm does not take advantage of any structure in the set D. In many analyses of objects and pointers, some elements of D subsume others. In Section 7, we present an extension that exploits such structure to reduce analysis time.

We have implemented the IFDS algorithm with all four of these extensions, as well as the running example variable type analysis. In Section 8, we report on an empirical evaluation of the benefits of the extensions. We survey related work in Section 9 and conclude in Section 10.

2 Background: The Original IFDS Algorithm

The IFDS algorithm of Reps et al. [15] is a dynamic programming algorithm that computes a merge-over-all-valid paths solution to interprocedural, finite, distributive, subset problems. The merge is over *valid* paths in that procedure calls and returns are correctly matched (i.e. the analysis is context sensitive). The algorithm requires that the domain of dataflow facts be the powerset of a finite set D, with set union as the merge operator. The data flow functions must be distributive over set union: $f(a) \cup f(b) = f(a \cup b)$.

The algorithm follows the summary function approach to context-sensitive interprocedural analysis [19], in that it computes functions in $\mathcal{P}(D) \rightarrow \mathcal{P}(D)$ that summarize the effect of ever-longer sections of code on any given subset of D. The key to the efficiency of the algorithm is the compact representation of these functions, made possible by their distributivity. For example, suppose the set $S = \{a, b, c\}$ is a subset of D. By distributivity, $f(S)$ can be computed as $f(S) = f(\{\}) \cup f(\{a\}) \cup f(\{b\}) \cup f(\{c\})$. Thus every distributive function in $\mathcal{P}(D) \rightarrow \mathcal{P}(D)$ is uniquely defined by its value on the empty set and on every singleton subset of D. Equivalently, the function can be defined by a bipartite graph $\langle D \cup \{\mathbf{0}\}, D, E\rangle$, where E is a set of edges from elements of $D \cup \{\mathbf{0}\}$ to elements of (a second copy of) D. The graph contains an edge from d_1 to d_2 if and only if $d_2 \in f(\{d_1\})$. The special **0** vertex represents the empty set: the edge $\mathbf{0} \rightarrow d$ indicates that $d \in f(\{\})$. The function represented by the graph is defined to be

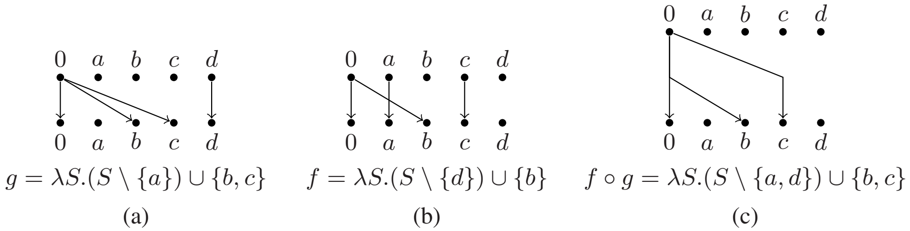

0
a
b
c
d
0
a
b
c
d
$g = \lambda S.(S \setminus \{a\}) \cup \{b, c\}$
(a)
0
a
b
c
d
0
a
b
c
d
$f = \lambda S.(S \setminus \{d\}) \cup \{b\}$
(b)
0
a
b
c
d
0
a
b
c
d
$f \circ g = \lambda S.(S \setminus \{a, d\}) \cup \{b, c\}$
(c)

Fig. 1. Compact representation of functions and their composition

$f(S) = \{b : (a, b) \in E \wedge (a = \mathbf{0} \vee a \in S)\}$. For example, the graph in Figure 1(a) represents the function $g(S) = \{x : x \in \{b, c\} \vee (x = d \wedge d \in S)\}$, which can be written more simply as $g(S) = (S \setminus \{a\}) \cup \{b, c\}$.

The composition $f \circ g$ of two functions can be computed by combining their graphs, merging the nodes of the range of g with the corresponding nodes of the domain of f, then computing reachability from the nodes of the domain of g to the nodes of the range of f. That is, a relational product of the sets of edges representing the two functions gives a set of edges representing their composition. An example is shown in Figure 1. The graph in Figure 1(c), representing $f \circ g$, contains an edge from x to y whenever there is an edge from x to some z in the representation of g in Figure 1(a) and an edge from the same z to y in the representation of f in Figure 1(b).

We have reproduced the original IFDS algorithm [15] in Figure 2. The input to the algorithm is a so-called exploded supergraph that represents both the program being analyzed and the dataflow functions. The supergraph is constructed from the interprocedural control flow graph (ICFG) of the program by replacing each instruction with the graph representation of its flow function. Thus the vertices of the supergraph are pairs $\langle l, d\rangle$, where l is a label in the program and $d \in D \cup \{\mathbf{0}\}$. The supergraph contains an edge $\langle l, d\rangle \rightarrow \langle l', d'\rangle$ if the ICFG contains an edge $l \rightarrow l'$ and $d' \in f(\{d\})$ (or $d' \in f(\{\})$ when $d = \mathbf{0}$), where f is the flow function of the instruction at l. For each interprocedural call or return edge in the ICFG, the supergraph contains a set of edges representing the flow function associated with the call or return. The flow function on the call edge typically maps facts about actuals in the caller to facts about formals in the callee. The merge-over-all-valid paths solution at label l contains exactly the elements d of D for which there exists a valid path from $\langle s, \mathbf{0}\rangle$ to $\langle l, d\rangle$ in the supergraph. The dataflow analysis therefore reduces to valid-path reachability on the supergraph.

The IFDS algorithm works by incrementally constructing two tables, PathEdge and SummaryEdge, representing the flow functions of ever longer sequences of code. The PathEdge table contains triples $\langle d, l, d'\rangle$, indicating that there is a path from $\langle s_p, d\rangle$ to $\langle l, d'\rangle$, where s_p is the start node of the procedure containing l. These triples are often written in the form $\langle s_p, d\rangle \rightarrow \langle l, d'\rangle$ for clarity, but the start node s_p is uniquely determined by l, so it is not stored in an actual implementation. The SummaryEdge table contains triples $\langle c, d, d'\rangle$, where c is the label of a call site. Such a triple indicates that $d' \in f(\{d\})$, where f is a flow function summarizing the effect of the procedure called at c. These triples are often written $\langle c, d\rangle \rightarrow \langle r, d'\rangle$, where r is the instruction

```
declare PathEdge, WorkList, SummaryEdge: global edge set
algorithm Tabulate($G^{\sharp}_{IP}$)
begin
  Let $(N^{\sharp}, E^{\sharp}) = G^{\sharp}_{IP}$
  PathEdge:={ $\langle s_{main}, \mathbf{0}\rangle \longrightarrow \langle s_{main}, \mathbf{0}\rangle$ }
  WorkList:={ $\langle s_{main}, \mathbf{0}\rangle \longrightarrow \langle s_{main}, \mathbf{0}\rangle$ }
  SummaryEdge:= $\emptyset$
  ForwardTabulateSLRPs()
  foreach $n \in N^{\sharp}$ do
    $X_n$ := { $d_2 \in D | \exists d_1 \in (D \cup \{\mathbf{0}\})$ s.t. $\langle s_{procOf(n)}, d_1\rangle \longrightarrow \langle n, d_2\rangle \in$ PathEdge
  od
end
procedure Propagate(e)
begin
  if $e \notin$ PathEdge then Insert $e$ into PathEdge; Insert $e$ into WorkList; fi
end
procedure ForwardTabulateSLRPs()
begin
  while WorkList $\neq \emptyset$ do
    Select and remove an edge $\langle s_p, d_1\rangle \longrightarrow \langle n, d_2\rangle$ from WorkList
    switch $n$
      case $n \in Call_p$ :
        foreach $d_3$ s.t. $\langle n, d_2\rangle \longrightarrow \langle s_{calledProc(n)}, d_3\rangle \in E^{\sharp}$ do
          Propagate($\langle s_{calledProc(n)}, d_3\rangle \longrightarrow \langle s_{calledProc(n)}, d_3\rangle$)
        od
        foreach $d_3$ s.t. $\langle n, d_2\rangle \longrightarrow \langle returnSite(n), d_3\rangle \in (E^{\sharp} \cup$ SummaryEdge$)$ do
          Propagate($\langle s_p, d_1\rangle \longrightarrow \langle returnSite(n), d_3\rangle$)
        od
      end case
      case $n \in e_p$ :
        foreach $c \in callers(p)$ do
          foreach $d_4$,$d_5$ s.t. $\langle c, d_4\rangle \longrightarrow \langle s_p, d_1\rangle \in E^{\sharp}$ and
                                         $\langle e_p, d_2\rangle \longrightarrow \langle returnSite(c), d_5\rangle \in E^{\sharp}$ do
            if $\langle c, d_4\rangle \longrightarrow \langle returnSite(c), d_5\rangle \notin$ SummaryEdge then
              Insert $\langle c, d_4\rangle \longrightarrow \langle returnSite(c), d_5\rangle$ into SummaryEdge
              foreach $d_3$ s.t. $\langle s_{procOf(c)}, d_3\rangle \longrightarrow \langle c, d_4\rangle \in$ PathEdge do
                Propagate($\langle s_{procOf(c)}, d_3\rangle \longrightarrow \langle returnSite(c), d_5\rangle$)
              od
            fi
          od
        od
      end case
      case $n \in (N_p - Call_p - \{e_p\})$ :
        foreach $\langle m, d_3\rangle$ s.t. $\langle n, d_2\rangle \longrightarrow \langle m, d_3\rangle \in E^{\sharp}$ do
          Propagate($\langle s_p, d_1\rangle \longrightarrow \langle m, d_3\rangle$)
        od
      end case
    end switch
  od
end
```

Fig. 2. Original IFDS Algorithm reproduced from [15]

following c. For convenience, Reps's presentation of the IFDS algorithm [15] assumes that in the ICFG, every call site c has a single successor, a no-op "return site" node r.

The PathEdge and SummaryEdge tables are interdependent. Consider the edge $\langle s_p, d_1\rangle \rightarrow \langle e_p, d_2\rangle$ added to PathEdge, in which e_p is the exit node of some procedure p. This edge means that $d_2 \in f_p(\{d_1\})$, where f_p is the flow function representing the effect of the entire procedure p. As a result, for every call site c calling procedure p, a corresponding triple must be added to SummaryEdge indicating the newly-discovered effect at that call site. In fact, several such triples may be needed for a single edge added to PathEdge, since the effect of a procedure at c is represented not just by f_p, but by the composition $f_r \circ f_p \circ f_c$, where f_c and f_r are the flow functions representing the function call and return. This composition is computed by combining the graphs representing f_c and f_r from the supergraph with the newly discovered edge $\langle d_1, d_2\rangle$ of f_p. That is, for each d_4 and d_5 such that $\langle d_4, d_1\rangle \in f_c$ and $\langle d_2, d_5\rangle \in f_r$, $\langle c, d_4, d_5\rangle$ is added to SummaryEdge. This is performed in lines 23 to 25 of the algorithm.

Conversely, consider a triple $\langle c, d_4, d_5\rangle$ added to SummaryEdge, indicating a new effect of the call at c. As a result, for each d_3 such that there is a path from $\langle s, d_3\rangle$ to $\langle c, d_4\rangle$, where s is the start node of the procedure containing c, there is now a valid path from $\langle s, d_3\rangle$ to $\langle r, d_5\rangle$, where r is the successor of c. Thus $\langle s, d_3\rangle \rightarrow \langle r, d_5\rangle$ must be added to PathEdge. This is performed in lines 26 to 28 of the algorithm.

3 Running Example: Type Analysis

The extensions to the IFDS algorithm presented in this paper were originally motivated by context-sensitive alias set analysis [13] and multi-object typestate analysis [12]. The same extensions are applicable to many other kinds of analyses. In this paper, we will use a much simpler analysis as a running example to illustrate the IFDS extensions.

The example analysis is a variation of Variable Type Analysis (VTA) [21] for Java. The analysis computes the set of possible types for each variable. This information can be used to construct a call graph or to check the validity of casts. At each program point p, the analysis computes a subset of D, where D is defined as the set of all pairs $\langle v, t\rangle$, where v is a variable in the program and t is a class in the program. The presence of the pair $\langle v, t\rangle$ in the subset indicates that the variable v may point to an object of type t.

For the sake of the example, we would like the analysis to analyze only the application code and not the large standard library. The analysis therefore makes conservative assumptions about the unanalyzed code based on statically declared types. For example, if `m()` is in the library, the analysis assumes that `m()` could return an object of the declared return type of `m()` or any of its subtypes. To this end we amend the meaning of a pair $\langle v, t\rangle$ to indicate that v may point to an object of type t or any of its subtypes.

The unanalyzed code could write to fields in the heap, either directly or by calling back into application code. To keep the analysis sound yet simple, we make the conservative assumption that a field can point to any object whose type is consistent with its declared type. We model a field read `x = y.f` with the pair $\langle \mathtt{x}, t\rangle$, where t is the declared type of `f`. We make these simplifications because the analysis is intended to illustrate the extensions to the IFDS algorithm, not necessarily as a practical analysis.

When the declared type of a field is an interface, the object read from it could be of any class that implements the interface. For a read from such a field, we generate multiple pairs $\langle \mathrm{x}, t_i\rangle$, where the t_i are all classes that implement the interface. If class A extends B and both implement the interface, it is redundant to include $\langle \mathrm{x}, B\rangle$ since $\langle \mathrm{x}, A\rangle$ already includes all subclasses of A, including B. For efficiency, we generate only those pairs $\langle \mathrm{x}, t_i\rangle$ where t_i implements the interface and its superclass does not.

The analysis is performed on an intermediate representation comprising the following kinds of instructions, in addition to procedure calls and returns: $s ::= x \leftarrow y \mid y.f \leftarrow x \mid x \leftarrow y.f \mid x \leftarrow \textbf{null} \mid x \leftarrow \textbf{new}\ T \mid x \leftarrow (T)y$. The instructions copy pointers between variables, store and load objects to and from fields, assign **null** to variables, create new objects and cast objects to a given type, respectively. We use $[\![s]\!]_P : \mathcal{P}(D) \to \mathcal{P}(D)$ to denote the transfer function for the type analysis. The IFDS algorithm requires the transfer function to be decomposed into its effect on each individual element of D and on the empty set. We decompose it as $[\![s]\!] : D \cup \{\mathbf{0}\} \to \mathcal{P}(D)$ and define $[\![s]\!]_P(P) \triangleq [\![s]\!](\mathbf{0}) \cup \bigcup_{d \in P} [\![s]\!](d)$. The decomposed transfer function $[\![s]\!]$ is defined in Figure 3.

$$[\![x \leftarrow y]\!](\langle v,t\rangle) \triangleq \begin{cases} \{\langle x,t\rangle, \langle y,t\rangle\} & \text{if } v = y \\ \{\langle v,t\rangle\} & \text{if } v \neq y \text{ and } v \neq x \\ \emptyset & \text{if } v \neq y \text{ and } v = x \end{cases}$$

$$[\![y.f \leftarrow x]\!](\langle v,t\rangle) \triangleq \{\langle v,t\rangle\}$$

$$[\![x \leftarrow \textbf{null} | \textbf{new}\ T | y.f]\!](\langle v,t\rangle) \triangleq \begin{cases} \{\langle v,t\rangle\} & \text{if } v \neq x \\ \emptyset & \text{otherwise} \end{cases}$$

$$[\![x \leftarrow \textbf{new}\ T]\!](\mathbf{0}) \triangleq \{\langle x,T\rangle\}$$

$$[\![x \leftarrow y.f]\!](\mathbf{0}) \triangleq \{\langle x,c\rangle : c \in \text{implClasses}(\mathit{type}(f))\}$$

$$[\![x \leftarrow (T)y]\!](\langle v,t\rangle) \triangleq \bigcup_{c \in \text{implClasses}(T)} \mathit{cast}(x,y,c)(\langle v,t\rangle)$$

$$\mathit{cast}(x,y,t_2)(\langle v,t_1\rangle) \triangleq \begin{cases} \{\langle v,t_1\rangle\} & \text{if } v \neq x \text{ and } v \neq y \\ \emptyset & \text{if } v = x \text{ and } v \neq y \\ \{\langle x,t_1\rangle, \langle y,t_1\rangle\} & \text{if } v = y \text{ and } t_1 <: t_2 \\ \{\langle x,t_2\rangle, \langle y,t_2\rangle\} & \text{if } v = y \text{ and } t_2 <: t_1 \\ \emptyset & \text{if } v = y \text{ and } t_1 \text{ and } t_2 \text{ are unrelated} \end{cases}$$

$$[\![s]\!](\mathbf{0}) \triangleq \emptyset \text{ if } s \neq x \leftarrow y.f \text{ and } s \neq x \leftarrow \textbf{new}$$

Fig. 3. Intraprocedural flow functions for the running example type analysis

The flow function for a copy instruction ($x \leftarrow y$) applied to a pair $\langle v,t\rangle$ requires three cases. When v is the same as y, the pair $\langle v,t\rangle$ is preserved and, since the value of y is copied to x, a new pair $\langle x,t\rangle$ is created. If v is neither x nor y, the value of v is unaffected by the copy and the pair is therefore preserved. If v is x, and x and y are distinct, then since the existing value of x is overwritten by the new value, the existing pair $\langle v,t\rangle$ describing the old value of v is discarded, and the result is the empty set.

The store instruction ($v.f \leftarrow x$) has no effect on the values of local variables, and its flow function is therefore the identity. The flow function for an assignment to x

via a load, **new** or **null** does not affect $\langle v, t\rangle$, unless v is x, in which case the existing value of x is overwritten, so the pair is dropped from the set. An allocation instruction $x \leftarrow$ **new** T generates the new pair $\langle x, T\rangle$. A load instruction $x \leftarrow y.f$ creates the pair $\langle x, t\rangle$, if the type of the field f is a class t, or the set of pairs $\langle x, t_i\rangle$, if the type of the field f is an interface, where the t_i are all of the classes implementing the interface, as explained earlier. The helper function implClasses(t) computes this set of classes.

The most interesting case is the cast instruction $(x \leftarrow (T)y)$. The first complication is that T could be an interface. Such a cast is treated as casts to all classes implementing T. The flow function is the union of the flow functions modelling casts to these classes, reflecting the fact that the cast to the interface type succeeds if the cast to at least one of the implementing classes succeeds. For the simpler case of a cast to a type t_2 that is a class, not an interface, there are still several cases. The cast instruction has no effect on $\langle v, t_1\rangle$ when v is neither x nor y. When v is x, the pair is dropped because the cast overwrites the existing value of x. When v is y and $t_1 <: t_2$, indicating that we already know that y points to an object whose type is a subtype of t_2, the cast acts as a copy and the new pair $\langle x, t_1\rangle$ is generated. When v is y and $t_2 <: t_1$, indicating that y is being cast to a more restrictive type than the type it is already known to point to, we generate the new pair $\langle x, t_2\rangle$, indicating that x must point to a subtype of the more restrictive cast type. The original pair $\langle y, t_1\rangle$ can also be changed to the more precise pair $\langle y, t_2\rangle$, since if control flow proceeds after the cast, the cast must have succeeded, and therefore y must point to an object whose type is a subtype of the cast type. For the purposes of the example, we assume that a failing cast terminates the program rather than being caught by an exception handler; catching class cast exceptions is rare in practice.

4 Demand Construction of the Supergraph

The number of nodes in the exploded supergraph $G^\sharp$ is $|\text{Inst}| \times (|D| + 1)$, where $|\text{Inst}|$ is the number of instructions in the program and $|D|$ is the size of D. In many analyses, D, though finite, is very large. For example, in an alias set analysis [13], D is a union of the powersets of the sets of variables of all procedures, and therefore exponential in the number of variables in a procedure. In our example variable type analysis, $D = |\text{Var}| \times |\text{Class}|$, where Var is the set of all variables in the program and Class is the set of all classes in the program, so $|D|$ is over one million even for a moderate program with a thousand variables and a thousand classes. Constructing and storing a graph that is a million times larger than the ICFG is not practical. In practice, only a small subgraph of $G^\sharp$ is reachable by valid paths from $\langle s_{main}, \mathbf{0}\rangle$ and therefore explored by the algorithm. Unfortunately, we cannot know exactly which subgraph this is before running the IFDS algorithm, since determining which nodes are reachable is exactly what the IFDS algorithm does. Therefore, our first extension to the IFDS algorithm modifies it to request only those parts of the supergraph that it encounters, instead of requiring the whole supergraph as input.

The extended IFDS algorithm with all four of our extensions is shown in Figure 4. Parts of the algorithm that were changed from the original or added are underlined.

The input to the extended algorithm is a function that, given a supergraph node $n^\sharp$, computes all of the edges leaving that node (i.e. the flow function of the desired analysis). For clarity of presentation, we have split this function into four separate functions:

```
declare PathEdge, WorkList, SummaryEdge, Incoming, EndSummary: global
algorithm Tabulate(flow, passArgs, returnVal, callFlow)
    ⋮
procedure ForwardTabulateSLRPs()
begin
  while WorkList $\neq \emptyset$ do
    Select and remove an edge $\langle s_p, d_1 \rangle \xrightarrow{\pi} \langle n, d_2 \rangle$ from WorkList
    switch $n$
      case $n \in Call_p$ :
        foreach $d_3 \in$ passArgs($\langle n, d_2 \rangle$) do
          Propagate$\Big( \langle s_{calledProc(n)}, d_3 \rangle \xrightarrow{0} \langle s_{calledProc(n)}, d_3 \rangle \Big)$
          Incoming $\big[ \langle s_{calledProc(n)}, d_3 \rangle \big] \cup = \langle n, d_2 \rangle$
          foreach $\langle e_p, d_4 \rangle \in$ EndSummary $\big[ \langle s_{calledProc(n)}, d3 \rangle \big]$ do
            foreach $d_5 \in$ returnVal($\langle e_p, d_4 \rangle$ , $\langle n, d_2 \rangle$) do
              Insert $\langle n, d_2 \rangle \rightarrow \langle returnSite(n), d_5 \rangle$ into SummaryEdge
            od
          od
        od
        foreach $d_3$ s.t. $d_3 \in$ callFlow($\langle n, d2 \rangle$) or
                        $\langle n, d_2 \rangle \rightarrow \langle returnSite(n), d_3 \rangle \in$ SummaryEdge do
          Propagate$\Big( \langle s_p, d_1 \rangle \xrightarrow{n} \langle returnSite(n), d_3 \rangle \Big)$
        od
      end case
      case $n \in e_p$ :
        EndSummary $[\langle s_p, d_1 \rangle] \cup = \langle e_p, d_2 \rangle$
        foreach $\langle c, d_4 \rangle \in$ Incoming $[\langle s_p, d_1 \rangle]$ do
          foreach $d_5 \in$ returnVal($\langle e_p, d_2 \rangle$ , $\langle c, d_4 \rangle$) do
            if $\langle c, d_4 \rangle \rightarrow \langle returnSite(c), d_5 \rangle \notin$ SummaryEdge then
              Insert $\langle c, d_4 \rangle \rightarrow \langle returnSite(c), d_5 \rangle$ into SummaryEdge
              foreach $d_3$ s.t. $\langle s_{procOf(c)}, d_3 \rangle \rightarrow \langle c, d_4 \rangle \in$ PathEdge do
                Propagate$\Big( \langle s_{procOf(c)}, d_3 \rangle \xrightarrow{c} \langle returnSite(c), d_5 \rangle \Big)$
              od
            fi
          od
        od
      end case
      case $n \in (N_p - Call_p - \{e_p\})$ :
        foreach $m, d_3$ s.t. $n \rightarrow m \in$ CFG and $d_3 \in$ flow($\langle n, d_2 \rangle$ , $\pi$) do
          Propagate$\Big( \langle s_p, d_1 \rangle \xrightarrow{n} \langle m, d_3 \rangle \Big)$
        od
      end case
    end switch
  od
end
```

Fig. 4. Extended IFDS Algorithm

- flow$(n^\sharp)$ computes all intraprocedural edges.[1]
- passArgs$(n^\sharp)$ computes call-to-start edges when $n^\sharp$ is at a call site.
- returnVal$(n^\sharp)$ computes exit-to-return-site edges when $n^\sharp$ is at the exit of a procedure.[2]
- callFlow$(n^\sharp)$ computes call-to-return-site edges when $n^\sharp$ is at a call site. These edges model procedure-local information that is not affected by the called procedure.

The original IFDS algorithm queries the edges of the supergraph $E^\sharp$ in five places. The queries on lines 14, 17 and 34, and the second query on line 23 can simply be replaced by calls to passArgs, callFlow, flow, and returnVal, respectively.

However, the first query on line 23 asks to evaluate the inverse of the flow function: find all call nodes $\langle c, d_4 \rangle$ from which an edge leads to the procedure start node $\langle s_p, d_1 \rangle$. This would require computing the inverse of the flow function, which can be difficult for many analyses. Moreover, even though $\langle s_p, d_1 \rangle$ is reachable in $G^\sharp$, many of its predecessors in $E^\sharp$ may not be, and enumerating them may be intractable. For example, for an alias set analysis, the number of predecessors for most nodes is $2^{|\text{Var}|-1}$, where $|\text{Var}|$ is the number of variables in the calling procedure. The extended algorithm therefore maintains a set Incoming$[\langle s_p, d_1 \rangle]$ that records nodes that the analysis has observed to be reachable and predecessors of $\langle s_p, d_1 \rangle$. Whenever the call to passArgs$(\langle n, d_2 \rangle)$ in line 14 returns $\langle s_p, d_3 \rangle$, $\langle n, d_2 \rangle$ is added in line 15.1 to Incoming$(\langle s_p, d_3 \rangle)$.

An obvious issue with querying the set of nodes already observed to be predecessors of $\langle s_p, d_1 \rangle$ is what must be done when a new predecessor is observed later. The solution is to keep track of exit nodes for which a given value of Incoming has been queried (line 21.1). Then, whenever a new predecessor is observed, those exit nodes are reprocessed to reflect the new predecessor. A simple way to reprocess the exit nodes correctly is to add them to the worklist. However, this approach is very inefficient, because whenever a new predecessor is added at one call site, the effect of the procedure is reprocessed for all predecessors at all call sites of the procedure. This intuitively poor performance was confirmed by our experience with the initial implementation of the algorithm.

A better way to reprocess the exit node is to recognize that when a new predecessor of $\langle s_p, d_1 \rangle$ is observed, the predecessor tells us the relevant call site. Instead of adding the corresponding exit node to the worklist, we can immediately process that exit node, but do only the work necessary for that one predecessor. Concretely, we duplicate the effect of lines 24 through 29 after line 15.1. The effect of lines 24, 25 and 29, adding the appropriate edge to SummaryEdge, is done in lines 15.3 through 15.5. The effect of lines 26 through 28 is already done by lines 17 through 19 of the original algorithm.

5 Return Flow Functions

In the original IFDS algorithm, the return flow function is modelled by interprocedural edges in the exploded supergraph that lead from the exit of a procedure to the call site that called the procedure. In the callee, each flow fact is represented in terms of the local scope of the callee. For many analyses, it is necessary to map information

[1] In Figure 4, flow has a second parameter π, which will be explained in Section 6.

[2] In Figure 4, returnVal has a second node parameter, which will be explained in Section 5.

in the callee back to the caller. For example, in the code on the right, the cast inside `ensureCircle` succeeds only if the object pointed to by `z`, which is also pointed to by `x` and `y`, is of type `Circle` or its subtype. Therefore, if `ensureCircle` returns normally, we know that `x` cannot point to an arbitrary `Shape`, but only to a `Circle`. However, the original IFDS algorithm cannot discover this fact: although it determines that at the exit of `ensureCircle`, `z` points to an object of type `Circle`, there is no way in the supergraph to associate `z` in the callee with `x` in the caller.

Yet with a small extension, this reverse mapping can be recovered. The fact that `z` points to a subtype of `Circle` is expressed by the edge $\langle s_{\texttt{ensureCircle}}, \langle \texttt{y}, \texttt{Shape}\rangle\rangle \rightarrow \langle e_{\texttt{ensureCircle}}, \langle \texttt{z}, \texttt{Circle}\rangle\rangle$ in PathEdge. This edge means that at the beginning of the procedure, there was an object pointed to by `y`, and at the exit of the procedure, the same object is pointed to by `z` and we know it is of type `Circle`. In addition, Incoming$[\langle s_{\texttt{ensureCircle}}, \langle \texttt{y}, \texttt{Shape}\rangle\rangle]$ contains $\langle c, \langle \texttt{x}, \texttt{Shape}\rangle\rangle$. This means that the object passed in through `y` from the call site c was pointed to by `x` in the caller scope. We can combine the context information provided by Incoming with the intraprocedural information computed in PathEdge to determine that the object pointed to by `x` at the call site is known to be of type `Circle` after the call.

```
void ensureCircle(Shape y){
    Shape z = y;
    (Circle) z;
}
Shape x = ...;
ensureCircle(x);
```

This extension appears in the extended algorithm in Figure 4 on line 23. The returnVal function takes, in addition to the node d_2 at the exit instruction e_p, a second node d_4 at the call site c. These arguments indicate not only that the node d_2 is reachable at e_p, but that it is reachable from some node d_1 at the start instruction s_p of the procedure, and that a passArgs edge leads to the latter node from node d_4 at the call site c. Thus the returnVal function can use the caller-side state from the time the procedure was invoked to map the callee-side state at the exit of the procedure back to the caller-side context.

This extension is not merely an extension of the IFDS algorithm, but an extension of the exploded supergraph abstraction that the algorithm is based on. In the supergraph, for every pair of nodes d_2 at an exit node and d_5 at a return site, there either is or is not an edge from d_2 to d_5; if there is such an edge, the algorithm adds a SummaryEdge from $\langle c, d_4\rangle$ to $\langle \mathit{returnSite}(c), d_5\rangle$ for every call site c calling the procedure and for every reachable node d_4 at c. However, the extended algorithm gives the analysis designer more flexbility, in that the decision to add the SummaryEdge is additionally dependent on the specific call-site node $\langle c, d_4\rangle$ being considered. It is as if the supergraph edge $\langle e_p, d_2\rangle \rightarrow \langle \mathit{returnSite}(c), d_5\rangle$ can both exist and not exist, depending on which call site node $\langle c, d_4\rangle$ is being taken on the path used to reach $\langle e_p, d_2\rangle$.

6 Static Single Assignment (SSA) Form

Static Single Assignment (SSA) form [2] is a popular intermediate representation that makes many program analyses simpler and more efficient. Standard dataflow analysis algorithms such as the original IFDS can be applied unchanged to programs in SSA form, but without appropriate extensions, such an analysis may be less precise than when the same analysis is done on the original, non-SSA version of the program. In this section, we discuss the reasons for the precision loss and propose an extension to the IFDS algorithm that fully restores the lost precision. The extended algorithm analyzes a program in SSA form as precisely as in its original, non-SSA form.

The defining feature of SSA form is that every variable is written to in only one instruction in the program. To convert a program to SSA form, every variable is renamed at each of its definitions, so each definition writes to a fresh, unique variable. Every use of a variable must also be renamed to match the reaching definition. A problem arises when multiple definitions reach a use: to which of the new names should the variable at the use be renamed? The solution is to add ϕ pseudo-instructions to select the reaching definition based on the control flow path taken. A ϕ instruction at a control flow merge point defines a new variable whose value is selected from among the reaching definitions depending on the edge taken into the merge point. Thus only the ϕ definition of the variable reaches the instructions following the merge.

The ϕ pseudo-instruction differs from normal instructions in two ways. First, if multiple variables require ϕ assigments at a given merge point, the ϕ assignments are performed simultaneously, in parallel. The set of ϕ instructions at the merge point defines, for each incoming control-flow edge, a permutation of the variables. Thus it is clearer to group all of the ϕ instructions at a given merge point into a single multi-variable ϕ instruction. Multiple instructions in sequence would suggest that the operations are performed one after the other, which is an incorrect semantics for ϕ instructions.

Second, unlike other instructions, the effect of a ϕ depends on the control-flow edge taken to reach the instruction. This causes many dataflow analysis algorithms, including the original IFDS, to lose precision when analyzing a program in SSA form, compared to analyzing the same program in its original non-SSA form. We will present an example program that exhibits such precision loss in Section 6.1. In most dataflow analyses, at a control flow merge point, the analysis first merges the dataflow facts from the incoming edges, then passes the merged value to the flow function of the instruction after the merge (i.e. $\mathit{out}[s] = f_s(\bigcup_{p \in \mathit{pred}(s)} \mathit{out}[p])$). Merging before applying the flow function reflects the structure of the control flow graph, and is appropriate when the merge is followed by a non-ϕ instruction. When the merge is followed by a ϕ instruction, however, the merge preceding the flow function application makes it impossible for the flow function f_s to depend on the control flow predecessor that its input came from, since the inputs from all the predecessors have been merged into a single dataflow value. Most dataflow analyses treat a ϕ instruction such as $x_3 = \phi(x_1, x_2)$ as an assignment from both x_1 and x_2 to x_3, ignoring the control flow edges on which those values of x_1 and x_2 arrived.

To analyze SSA-form code as precisely as non-SSA-form code, the merge must be delayed until *after* the ϕ instruction. That is, the ϕ flow function is applied separately to the dataflow value on each incoming control flow path, and the merge is performed on the *outputs* of the ϕ flow function, not on its input. As a result, the incoming control flow edge associated with each dataflow value can be made available to the flow function f_ϕ modelling the ϕ instruction. Formally, $\mathit{out}[\phi] = \bigcup_{p \in \mathit{pred}(\phi)} f_\phi(p, \mathit{out}[p])$.

Extending the IFDS algorithm to perform dataflow merges after ϕ instructions instead of before them requires two modifications. First, every edge added to PathEdge is annotated with a control flow predecessor. The edge $\langle s_p, d_1 \rangle \xrightarrow{n} \langle m, d_2 \rangle$ indicates that there is a path in the supergraph starting at the dataflow fact d_1 at the start node s_p, leading to the dataflow fact d_2 at node m, and that the second-last node on the path is at node n. In other words, the dataflow fact d_2 reaches m along the incoming control flow edge from n. Two PathEdge edges that differ only in the control flow predecessor are considered to be distinct. The PathEdge edges created in lines 18, 27, and 35 of the algorithm are annotated with the control flow predecessor, shown above the arrow. The PathEdge edge created in line 15 corresponds to the empty path from $\langle s, d_3 \rangle$ to itself, so there is no control flow predecessor to record. We therefore use a dummy predecessor, which we write as 0. However, the target of this edge is the start node of the procedure, which is never a ϕ instruction, so the predecessor will never be needed for this node.

Second, the flow function is extended with a second parameter, and when the function is called in line 34, the control-flow predecessor π of the PathEdge edge currently being processed is passed in. Thus the flow function for the ϕ instruction can depend on the control-flow predecessor π associated with the dataflow value d_2 reaching n.

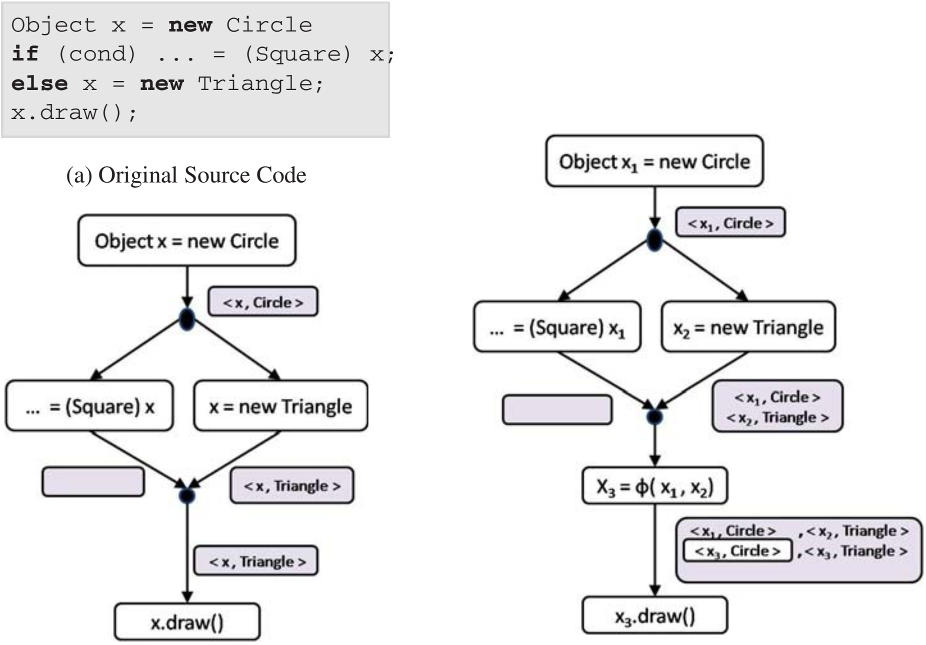
An obvious optimization is to annotate only those edges $\langle s_p, d_1 \rangle \rightarrow \langle m, d_2 \rangle$ in which m is a ϕ instruction, and leave all other edges unannotated. We do this in our implementation, but have not shown it in Figure 4 to avoid cluttering the algorithm.

6.1 Example of Precision Loss

An example of how merging dataflow information before rather than after a ϕ instruction reduces precision is shown in Figure 5. The original non-SSA source code of the example program is in Figure 5(a). A variable x is initialized as a `Circle`. In the left branch of the conditional, x is cast to `Square`. In the right branch, x is redefined as a `Triangle`. Figure 5(b) shows the results of running the type analysis on the code. The flow function for the cast operation kills the flow fact $\langle x, \texttt{Circle} \rangle$, since a `Circle` cannot be successfully cast to a `Square`. Therefore, the type analysis indicates that the only possible type for receiver x at instruction `x.draw()` is `Triangle`. This is sound since the cast operation can never succeed and therefore a program executing the left branch can never reach the `draw` call. Conversely, if the program reaches the `draw` call it must have taken the right branch and the receiver must be a `Triangle`.

Figure 5(c) shows the same code after SSA conversion. The receiver x_3 for the call x_3`.draw()` is x_1 when the path follows the left branch and x_2 when the path follows the right branch, as reflected in the ϕ function. The left predecessor of the ϕ function has no flow facts because the cast kills $\langle x_1, \texttt{Circle} \rangle$ as before. The right predecessor has the facts $\langle x_1, \texttt{Circle} \rangle$ and $\langle x_2, \texttt{Triangle} \rangle$. The original IFDS algorithm would first merge the incoming flow facts from the two branches, then apply the flow function that models the ϕ as a copy from both x_1 and x_2. At the call to x_3`.draw()`, the analysis would compute the facts $\langle x_3, \texttt{Circle} \rangle$ and $\langle x_3, \texttt{Triangle} \rangle$, which is less precise than the non-SSA version of the analysis that was able to rule out x being a `Circle`.

In the extended IFDS algorithm, the merge is not performed before the flow function of the ϕ instruction, so the flow function has information about the control flow edge on which each dataflow fact arrives. For facts coming in from the left edge, it models a copy from x_1 to x_3; for facts coming in from the right edge, it models a copy from x_2 to x_3. Thus only the fact $\langle x_2, \texttt{Triangle} \rangle$ coming from the right edge leads to a new



```
Object x = new Circle
if (cond) ... = (Square) x;
else x = new Triangle;
x.draw();
```

(a) Original Source Code

(b) Non-SSA Type Results

(c) SSA Type Results

Fig. 5. The effect on precision due to the choice of merge strategy at ϕ nodes

fact $\langle x_3, \texttt{Triangle}\rangle$. The fact $\langle x_1, \texttt{Circle}\rangle$ does not give rise to $\langle x_3, \texttt{Circle}\rangle$, as it did before, because it comes in from the right edge, which is not associated with a copy from x_1 to x_3. Thus the extended IFDS algorithm achieves the same precision on the SSA-form version of the program as on the original non-SSA-form version.

7 Exploiting Structure in the Set D

The IFDS algorithm requires that the dataflow domain be the powerset of a finite set D. The elements of D are treated independently and equally. The algorithm does not assume or take advantage of any relationships between the elements of D. This is appropriate for bit-vector dataflow problems. For example, the liveness of variable x at some program point implies nothing about the liveness of a different variable y.

However, some domains have more structure in the form of subsumption relationships between elements. In the example type analysis, the fact $\langle x, \texttt{Circle}\rangle$ subsumes the fact $\langle x, \texttt{Shape}\rangle$, since knowing that x points to an object whose type is some subtype of Circle implies that its type is also a subtype of Shape. Therefore, if the analysis computes, for some program point, the set $\{\langle x, \texttt{Circle}\rangle, \langle x, \texttt{Shape}\rangle\}$, which means that x points to a subtype of Circle or that x points to a subtype of Shape, then this set provides no additional information compared to the smaller set $\{\langle x, \texttt{Shape}\rangle\}$ that could have been computed; the two sets are equivalent.

Formally, we can define for an arbitrary analysis the partial order $a \leq b$, meaning that a subsumes b (for example, $\langle x, \texttt{Circle} \rangle \leq \langle x, \texttt{Shape} \rangle$). We require all of the dataflow functions to be monotone in the partial order: $a \leq b \implies \text{flow}(a) \leq \text{flow}(b)$. We consider two sets computed by the analysis to be equivalent, written $D_1 \sim D_2$, if every element of each set is subsumed by some element of the other set:

$$\begin{aligned} D_1 \leq D_2 &\iff \forall d_1 \in D_1 \exists d_2 \in D_2 \text{ s.t. } d_1 \leq d_2 \\ D_1 \sim D_2 &\iff D_1 \leq D_2 \wedge D_2 \leq D_1 \end{aligned}$$

The original IFDS algorithm handles analyses in which D has structure correctly but not as efficiently as possible. Because it ignores the subsumption relationship, it compute $\{\langle x, \texttt{Circle} \rangle, \langle x, \texttt{Shape} \rangle\}$ instead of the equivalent smaller set $\{\langle x, \texttt{Shape} \rangle\}$. We have extended the algorithm to use subsumption relationships in D to find smaller equivalent sets. The extension reduces the size not only of the final result, but of the intermediate sets during execution of the algorithm. The performance improvement is cumulative since smaller intermediate sets require less further processing.

The extended algorithm is as precise as the original IFDS algorithm in the sense that if the algorithms compute dataflow facts D_{ext} and D_{orig}, respectively, for a given program point, then $D_{ext} \sim D_{orig}$.

Extending the algorithm to exploit subsumption requires two steps. First, the Propagate function is changed to only add an edge to PathEdge if it does not subsume any already existing edge, as shown in Figure 6.[3] Any edges in PathEdge and in the WorkList subsuming the newly-added edge are redundant and can be removed in line 9.1.

```
     procedure Propagate($\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$)
     begin
9      if $\nexists \langle s_p, d_1 \rangle \rightarrow \langle n, d_3 \rangle \in$ PathEdge s.t. $d_2 \leq d_3$ then
           Insert e into PathEdge; Insert e into WorkList; fi
9.1    Remove all edges $\langle s_p, d_1 \rangle \rightarrow \langle n, d_3 \rangle$ s.t. $d_3 \leq d_2$ from PathEdge and from WorkList
     end
```

Fig. 6. Extended Propagate Procedure

Second, the worklist is modified so that subsumed elements are processed before subsuming ones. Without an appropriate worklist ordering, the algorithm might do the work of constructing the full sets and only afterwards discover an element that the existing elements subsume, making the existing elements unnecessary. Thus only after all of the work was done would the algorithm discover that the work was not necessary.

To define a suitable worklist ordering, we define an estimate function mapping each element of D to an integer with the property that $d_1 \leq d_2 \implies \text{estimate}(d_1)$

[3] Though it may seem counterintuitive, it is correct to only add elements that do not subsume an existing element, rather than elements not themselves subsumed by an existing element. The interpretation of the PathEdge set is a disjunction of the possible types for each variable: any element in the set is a possible abstraction of runtime behaviour. If a subsumes b, then adding a to a disjunction already containing b does not change the meaning of the disjunction.

$\leq$ estimate(d_2). For all analyses we have encountered, we have found it easy to define such an estimate. For the example type analysis, we use the following estimate: the class `Object` has estimate 0, and the estimate of each other class is one less than the estimate of its superclass. For a given estimate function, the worklist is implemented as a priority queue that makes the algorithm process edges with the highest estimate first.

This ordering heuristic does not completely guarantee that the algorithm will never call Propagate with an edge that makes a previous edge unnecessary, but it does ensure this property in most cases, and works well in practice. Recall that each flow function is monotonic, so that $a \leq b \implies \text{flow}(a) \leq \text{flow}(b)$. We can be sure to compute flow(b) and flow(a) in the correct order (that is, flow(b) first) by following the ordering heuristic to remove b from the worklist before a. However, at a control flow merge point, it is possible that a and b appear at two different control flow predecessors p, p', which are modelled by different flow functions. There is no guarantee that $a \leq b \implies \text{flow}_p(a) \leq \text{flow}_{p'}(b)$, so we cannot guarantee that it is more efficient to compute flow$_{p'}(b)$ before flow$_p(a)$.

8 Empirical Evaluation

We have performed experiments on the variable type analysis to measure the following:

- How large is the supergraph, and what fraction of it is reachable along valid paths?
- How does taking advantage of subsumption relationships in D reduce the number of dataflow facts that must be processed and the running time of the IFDS algorithm?

We implemented the extended IFDS algorithm and the example type analysis in Scala [14] using Soot [22] as a front-end to parse and convert Java classes into 3-address code and construct a control flow graph (CFG). Both normal Java control flow and control flow due to exceptions was represented by edges in the CFG. We ran the extended algorithm on the DaCapo Benchmark Suite, version 2006-10-MR2 [1]. Since most of the benchmarks use reflection, we provided Soot with summaries of uses of reflection obtained by instrumenting the benchmarks using ProBe [11] and *J [5].[4] Statistics about the benchmarks are presented in Table 1. The Methods column shows the number of methods in the part of the call graph analyzed by the IFDS analysis; since the type analysis does not analyze the library, we cut off the call graph at any call into the library. Not analyzing the library is a characteristic of our example analysis, and not a limitation of the IFDS algorithm in general. In earlier work [12], we evaluated two IFDS analyses that successfully analyze the whole program including the standard library. The Variables column shows the number of SSA variables in the analyzed methods. The Instructions column shows the number of instructions after conversion to the intermediate representation presented in Section 3. The Possible Types column shows the number of concrete classes in the benchmark. These are the classes that could appear as the type associated with a variable in the analysis results.

We first measured the size of the complete exploded supergraph. In general, the number of nodes in the exploded graph is given by $|\text{Inst}| \times (|D|+1)$ where $D = \text{Var} \times \text{Class}$,

[4] We excluded the Eclipse benchmark because it makes such heavy use of reflection that Soot is unable to process it.

Table 1. Benchmark Characteristics

Benchmark	Methods	Variables	Instructions	Possible Types
antlr	949	10839	16621	257
bloat	3142	33727	46550	623
chart	9419	91280	129850	2292
fop	13556	131901	185129	3400
hsqldb	768	8004	11552	443
jython	5487	56090	74031	1079
luindex	1306	12519	18131	617
lusearch	1633	14850	21368	676
pmd	3643	33945	49640	998
xalan	786	7708	11084	451

Var is the set of all variables in the program and Class is the set of all classes. However, when analyzing a given method, only the local variables of that method need to be tracked. Thus a much smaller exploded supergraph can be constructed of size $\sum_{m \in \text{Methods}} |\text{Var}_m| \times |\text{Class}| \times |\text{Inst}_m|$, where $|\text{Var}_m|$ and $|\text{Inst}_m|$ are the numbers of variables and instructions in method m. We measured the size of this smaller, more reasonable exploded supergraph. In addition to the number of nodes, we computed the number of edges in the exploded supergraph. To do this, we applied the flow function to every node of the exploded supergraph and counted the number of outgoing edges. The sizes of the exploded supergraph are shown using diamond shapes in Figure 7. The sizes range from 138 million to 21 billion nodes. On average (geometric mean), each exploded supergraph has 1.16 times as many edges as nodes. The largest exploded supergraphs took over 24 hours to enumerate.

We also measured the sizes of the reachable part of the supergraph that is explored when the IFDS algorithm has been extended with demand supergraph construction. These sizes are shown as horizontal lines in Figure 7. The number of edges in the reachable part of the supergraph is 1.09 times the number of nodes. On average (geometric mean), the complete supergraph contains 2081 times as many nodes as the reachable part of the supergraph. Constructing the supergraph on demand rather than exhaustively is key to analyzing benchmarks of this size in reasonable time and memory bounds.

Next, we measured the effect of using subsumption relationships in D to avoid propagating dataflow facts that subsume existing facts. We ran the type analysis three times. In the first run, the subsumption extension from Section 7 was turned off, so all dataflow facts were propagated regardless of their subsumption relationships. In the second run, the subsumption extension was turned on, but the original first-in first-out (FIFO) worklist was used. In the third run, both the subsumption extension and the subsumption-aware worklist ordering from Section 7 were used. For each case, we measured the running time of the analysis and the total number of pairs $\langle v, t \rangle$ computed (i.e. the sum over all instructions of the number of $\langle v, t \rangle$ pairs for that instruction). The results are shown in Table 2. Empty cells in the table indicate that the analysis did not complete within 10000 seconds of CPU time and 10 GB of memory. The subsumption-extended analysis completed on all of the benchmarks, but the unextended analysis completed on only five benchmarks within these time and memory limits. Columns 2 and 3 show

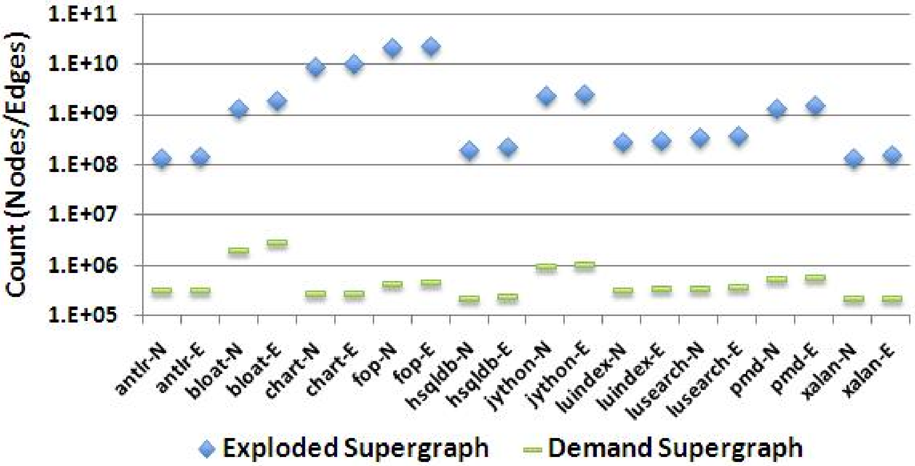


Fig. 7. Number of nodes and edges in the exploded supergraph and its reachable subgraph. The letters N and E after each benchmark name designate nodes and edges, respectively.

Table 2. Effect of taking advantage of subsumption relationships in D

	Facts (x10^3)		Time (s)		
Benchmark	w/o subs.	w subs.	w/o subs.	w subs., w/o PQ	w subs., w PQ
antlr	546	309	179	44	45
bloat		2037		1544	1518
chart		2817		3377	3197
fop		4408		3247	2847
hsqldb	1758	224	4720	60	60
jython		1015		1225	697
luindex	2900	326	9860	75	70
lusearch	3432	356	9776	78	68
pmd		556		241	211
xalan	1809	218	4813	61	60

the number of $\langle v, t\rangle$ pairs without and with the subsumption extension (this number is independent of the worklist ordering). Columns 4, 5, and 6 show the running time of the three runs of the analysis. On the five benchmarks on which all algorithms ran to completion, the unextended analysis had to compute 6.3 times as many pairs as the extended analysis, so the unextended analysis took 55 times as long as the extended analysis (geometric mean). In the extended analysis, the subsumption-aware priority queue worklist reduced the running time by 10% (geometric mean over all benchmarks). Extremes were jython, where the reduction was 43%, and antlr, where the running time increased by 2% due to the higher cost of maintaining a priority queue compared to a FIFO list. The subsumption extension presented in Section 7 is very important for the speed of the analysis and for its ability to analyze programs of significant size.

9 Related Work

Sharir and Puneli [19] extended Kildall's framework of intraprocedural dataflow analysis [9, 10] to two frameworks of context-sensitive interprocedural dataflow analysis, which they called the *call-strings* approach and the *functional* approach. The two frameworks compute a merge-over-all-valid-paths solution, where a valid path is one in which procedure calls and returns are correctly matched. The call-strings approach treats calls and returns from a procedure like all other control flow but restricts propagation to valid paths by tagging propagated dataflow facts with a call string (an abstraction of the active call stack). In the functional approach, the effects of each procedure are summarized by a summary function $f_p : \mathcal{D} \rightarrow \mathcal{D}$, where $\mathcal{D}$ is the dataflow analysis domain. The summary function is then used at each call site of the procedure to model the effect of the call. The key operation in the functional approach is function composition. For example, to compute the summary function f_r of a caller procedure that contains a call site to a callee procedure, the summary function f_e of the callee procedure must be composed with functions representing the intraprocedural effects of the caller procedure. Although the functional approach has the potential to be more precise and more efficient than the call strings approach, a key challenge is devising efficient representations of the summary functions that are amenable to function composition.

The IFDS framework [15] provides such an efficient representation of summary functions for the functional approach, as discussed in Section 2. When the dataflow domain is $\mathcal{P}(D)$ for a finite set D and all of the dataflow functions are distributive, they can be compactly represented using bipartite graphs with $O(D)$ nodes. Function composition can be computed efficiently in this representation, and the composition of distributive functions is also distributive. Thus the IFDS algorithm makes the functional approach practical for the class of dataflow analyses satisfying these restrictions. 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.

The IDE [18] algorithm generalizes IFDS to a wider class of dataflow analyses. Whereas in IFDS, the dataflow facts are elements of $\mathcal{P}(D)$, the IDE algorithm allows dataflow facts that are maps drawn from $D \rightarrow L$, where D is a finite set and L is a finite-height semi-lattice.[5] The IDE algorithm has been used to express copy-constant propagation and linear constant propagation [18]. The IDE literature calls elements of $D \rightarrow L$ environments, so the flow functions that are composed in the algorithm are environment transformers drawn from $(D \rightarrow L) \rightarrow (D \rightarrow L)$. Provided these transformers are distributive, they can be represented efficiently using graphs similar to those used in the IFDS algorithm, with additional labels on the edges of the graph describing the effect of the edge on elements of L. Whereas the IFDS problem computes reachability along valid paths, the IDE algorithm additionally evaluates functions $L \rightarrow L$ along those paths. The overall structure of both algorithms is very similar, however. All of the extensions presented in this paper are equally applicable to the IDE algorithm as well as to the IFDS algorithm. We have implemented the extensions in both algorithms.

[5] The domain $\mathcal{P}(D)$ is isomorphic to $D \rightarrow L$ if L is chosen to be the two-point lattice.

Demand-driven variations of the IFDS and IDE algorithms have been thoroughly studied [3, 4, 8, 16, 18]. These algorithms differ from the exhaustive algorithms in that rather than computing all nodes reachable from the start node, they determine whether a given node n is reachable. These algorithms can be faster when only a small number of nodes are queried. The algorithms work by exploring reverse paths along the supergraph from the given node n, by evaluating inverses of the dataflow functions. The demand-driven computation of reachability implemented by these algorithms is distinct from and complementary to the demand-driven exploded supergraph construction that we presented in Section 4. The purpose of demand supergraph construction is to avoid constructing the whole supergraph, which may be much larger than its reachable subgraph; the demand-driven reachability algorithms do require the whole exploded supergraph to be constructed ahead of time. Although our extended IFDS algorithm constructs the exploded supergraph on demand, it then exhaustively computes all nodes reachable along valid paths, rather than answering reachability queries for specific notes. An interesting direction for future work would be to combine demand supergraph construction with demand-driven reachability queries. Such an algorithm appears to be challenging to design and to tune, however. The key difficulty that we had to overcome in constructing the exploded supergraph on demand was the need, on line 23 of the original IFDS algorithm, to evaluate the inverse of the dataflow function. The demand-driven supergraph reachability algorithms require much more evaluation of inverse dataflow functions.

Others have noticed limitations of the original IFDS algorithm, and mention implementing extensions similar to some of those that we have presented here [6, 7, 17, 20, 23]. Fink et al. [6, 7] used the IFDS algorithm to verify typestate properties of objects. To verify that an object respects a temporal property, they build precise abstractions of the objects in the program and aliasing between them. The analysis computes an object abstraction containing sets of access paths that must or must-not reference an object. This abstraction is computed using the IFDS algorithm with extensions for exceptional control flow and polymorphic dispatch. Though their presentation focuses on the typestate analysis rather than specifics of their extensions to the IFDS algorithm, their implementation depends on constructing the exploded supergraph on demand, providing call-site information to return flow functions, and exploiting subsumption between elements of D. Shoham et al. [20] apply the infrastructure of Fink et al. [6, 7], along with its IFDS extensions, to statically extract finite-state automata of sequences of API calls.

Some shape analyses that have been implemented as instances of the IFDS algorithm construct the supergraph on demand for scalability. Rinetzky et al. [17] present an efficient shape analysis for the class of cutpoint-free programs, in which at each procedure call, the subgraph of the heap reachable in the callee can only be reached in the caller through arguments of the call. Yang et al. [23] present a different shape analysis that works for general programs. Both of these analyses are instances of the IFDS algorithm, and both implementations construct only the reachable part of the supergraph.

Several of the analyses just mentioned [6, 7, 17, 20, 23] use partial joins, an extension similar to subsumption in the analysis domain D that we discussed in Section 7. Whereas a partial join enables the analysis designer to sacrifice precision for efficiency, exploiting subsumption does not change analysis precision. A partial join may make the analysis output depend on the order of exploration; exploiting subsumption does not. A partial join operator $\overset{\dagger}{\sqcup}$ is a partial function $\overset{\dagger}{\sqcup} : D \times D \nrightarrow D$ with the property that if $a \overset{\dagger}{\sqcup} b = d$, then each of a and b subsume d. Whenever the partial join IFDS algorithm encounters both a and b in a given set, it replaces them with d, reducing the size of the set. This operation is sound, since if each of a and b subsume d, then so does their disjunction. However, it may reduce precision. For example, if we also define $a \overset{\dagger}{\sqcup} c = d$, it becomes impossible for the analysis to distinguish $\{a, b\}$ from $\{a, c\}$, even though neither set subsumes the other (i.e. $\{a, b\} \not\leq \{a, c\}$). Our subsumption extension can be implemented using the following definition of a partial join: if $a \leq b$, then $a \overset{\dagger}{\sqcup} b = b \overset{\dagger}{\sqcup} a = b$, else $a \overset{\dagger}{\sqcup} b$ is undefined.

Our previous work [12] on verifying temporal properties of groups of interacting objects also uses the IFDS and IDE algorithms. Verifying typestate-like properties of multiple objects requires two separate abstractions and analyses: an alias-set abstraction to track the objects in the program and a second abstraction of the typestate of groups of objects. We used the IFDS algorithm to compute these abstractions. We used the IDE algorithm to compute the set of events that might trigger a violation of a temporal property. In later work [13] we improved the alias-set analysis using properties of programs in SSA form. Our extension to the IFDS algorithm to precisely handle ϕ instructions, as presented in Section 6, was essential to obtaining precise alias set information.

10 Conclusions

We presented four extensions to the IFDS algorithm that make it applicable to a wider class of interprocedural dataflow analysis problems, in particular analyses of objects and pointers. The extended algorithm does not require an exploded supergraph as input, but builds it on demand, only for those dataflow facts for which it is actually needed. The extended algorithm provides caller-side context information from before a procedure call to the flow function that maps callee-side state back to the caller after the call. The extended algorithm analyzes programs in SSA form as precisely as programs not in SSA form. The extended algorithm takes advantage of structure in the dataflow analysis domain to significantly speed up analyses exhibiting such structure. We illustrated our extensions on a variation of variable type analysis, and we have applied them to more complicated analyses including alias set analysis [13] and multi-object typestate analysis [12]. The extensions apply not only to the IFDS algorithm but also to the more general IDE algorithm.

References

1. 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., Van Drunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA 2006, pp. 169–190 (2006)
2. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: An efficient method of computing static single assignment form. In: POPL 1989, pp. 25–35 (1989)

3. Duesterwald, E., Gupta, R., Soffa, M.L.: Demand-driven computation of interprocedural data flow. In: POPL 1995, pp. 37–48 (1995)
4. Duesterwald, E., Gupta, R., Soffa, M.L.: A practical framework for demand-driven interprocedural data flow analysis. ACM Trans. Program. Lang. Syst. 19(6), 992–1030 (1997)
5. Dufour, B.: Objective quantification of program behaviour using dynamic metrics. Master's thesis, McGill University (June 2004)
6. Fink, S., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. In: ISSTA 2006, pp. 133–144 (2006)
7. Fink, S.J., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. ACM Trans. Softw. Eng. Methodol. 17(2), 1–34 (2008)
8. Horwitz, S., Reps, T., Sagiv, M.: Demand interprocedural dataflow analysis. In: SIGSOFT FSE 1995, pp. 104–115 (1995)
9. Kam, J.B., Ullman, J.D.: Monotone data flow analysis frameworks. Acta Inf. 7, 305–317 (1977)
10. Kildall, G.A.: A unified approach to global program optimization. In: POPL 1973, pp. 194–206 (1973)
11. Lhoták, O.: Comparing call graphs. In: PASTE 2007, pp. 37–42 (2007)
12. Naeem, N.A., Lhoták, O.: Typestate-like analysis of multiple interacting objects. In: OOPSLA 2008, pp. 347–366 (2008)
13. Naeem, N.A., Lhoták, O.: Efficient alias set analysis using SSA form. In: ISMM 2009, pp. 79–88 (2009)
14. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima Press (2008)
15. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL 1995, pp. 49–61 (1995)
16. Reps, T.W.: Solving demand versions of interprocedural analysis problems. In: Fritzson, P.A. (ed.) CC 1994. LNCS, vol. 786, pp. 389–403. Springer, Heidelberg (1994)
17. 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)
18. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theoretical Computer Science 167(1-2), 131–170 (1996)
19. 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)
20. Shoham, S., Yahav, E., Fink, S., Pistoia, M.: Static specification mining using automata-based abstractions. In: ISSTA 2007, pp. 174–184 (2007)
21. 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, pp. 264–280 (2000)
22. 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)
23. 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)

Using Ownership to Reason about Inherent Parallelism in Object-Oriented Programs

Andrew Craik and Wayne Kelly

Queensland University of Technology
2 George St GPO Box 2434
Brisbane QLD 4001 Australia
a.craik@qut.edu.au, w.kelly@qut.edu.au

Abstract. With the emergence of multi-cores into the mainstream, there is a growing need for systems to allow programmers and automated systems to reason about data dependencies and inherent parallelism in imperative object-oriented languages. In this paper we exploit the structure of object-oriented programs to abstract computational side-effects. We capture and validate these effects using a static type system. We use these as the basis of sufficient conditions for several different data and task parallelism patterns. We compliment our static type system with a lightweight runtime system to allow for parallelization in the presence of complex data flows. We have a functioning compiler and worked examples to demonstrate the practicality of our solution.

1 Introduction

Imperative programming languages have an inherently sequential semantics, but programs in these languages may contain sections which can be safely executed concurrently. The problem of automatically detecting and exploiting this inherent parallelism is long-standing but still beyond the current state-of-the-art for general programs. The emergence of multi-core computing into the mainstream has only increased the need for solutions. Rapid growth in the number of cores per chip is projected and so scalability of proposed solutions is becoming a key concern. Given the difficulty of the problem, we must find a way to reformulate it so that it becomes more tractable even if we loose some precision. We seek a solution that yields sufficient conditions for parallelism that are permissive enough to be useful while allowing programmers and automated systems to easily reason about inter-procedural data dependencies and inherent parallelism in large complex applications.

Parallelism research has traditionally focused on scientific applications where data-flow analysis has tended to be used to solve complicated array index expressions and pointer may-alias questions. We believe that these traditional approaches have met with limited success outside the realm of scientific applications for two main reasons: (1) the analyses are too fine grained and (2) they do

Funding provided by Microsoft Research and the Queensland State Government

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 145–164, 2010.

not facilitate abstraction and composition. These traditional approaches employ very complex and detailed dependence analyses which do not support abstraction. This lack of abstraction hinders their ability to reason across method and component barriers. At the heart of the problem is the fact that, traditionally, method signatures provide no information about side-effects. This makes it impossible to reason about inter-procedural dependencies without examining method implementations and all those which may be called. The pervasive use of dynamic linking and late binding in modern componentized software systems further exacerbates this problem.

Current approaches to parallelizing applications tend to follow one of two main schools of thought: (1) statically determine potential conflicts and prevent them from occuring or (2) allow conflicts to occur and incurr a runtime penalty to resolve them. Both approaches have different strengths and weaknesses and have been used to solve different types of problems. In this work we have chosen to use static analysis, but there is also valuable and interesting work in the field of runtime conflict resolution. Ultimately, some combination of these approaches may prove the best compromise.

We address the problem of reasoning about inherent parallelism in the specific context of imperative object-oriented languages for two reasons. Firstly, the emergence of multi-cores means that parallelism will now enter the domain of general purpose desktop and server applications; imperative object-oriented languages dominate this development space. Secondly, the object-oriented programming model provides structure to the memory allocated by the program and we seek to exploit this structure to facilitate reasoning at higher levels of abstraction. The use of these higher levels of abstraction allow our techniques to scale across large and complex applications unlike traditional data flow analysis techniques.

Capturing the side-effects of methods is difficult as they may, directly or indirectly, access a virtually uncountable number of memory locations with no easily describable structure. To simplify reasoning about data dependencies we abstract these effects by exploiting the hierarchical “ownership” relationships which inherently exist within object-oriented programs. Objects contain other objects as part of their representation and we view this as providing a large tree structure to all of the objects in the program’s heap. We can, therefore, summarize method side-effects in terms of the subtrees which may be accessed or modified. To reason that two computations are independent (i.e. can be executed in parallel) we can reason about the parts of this ownership tree that could, potentially, be accessed instead of reasoning about the individual memory locations themselves. If the sub-trees accessed are disjoint then there can be no data dependencies. This approach sacrifices some precision so that we can perform inter-procedural dependency analyses in a scalable and composable manner.

We have developed a static type system based on Ownership Types[1–6] which is a type system formulated to capture this “ownership” tree. Our system captures, computes, and validates computation side-effects in terms of these ownerships. Because of the hierarchical nature of ownership, we can describe side-effects at different levels of granularity. The side-effects in turn can be used to statically reason about the presence of inherent parallelism.

Complex inter-procedural data flows in addition to dynamic linking and late binding reduce our ability to statically determine the relationship between some contexts at compile time. Because of this, we have complemented our static type system with a runtime representation of these ownership relationships that allow us to determine the disjointness of effects at runtime in $O(1)$ time. Such runtime tests result in conditional parallelism and allow us to parallelize more cases.

To help demonstrate the efficiency and effectiveness of our system for real applications we have created an extension of the C♯ language with support for ownership and effect annotations (the same could easily be done using Java as the base language). We have implemented a compiler for this extended language that performs type checking and generates parallelized C♯ source code as output. Complete source code for our compiler and runtime system is available from our web site [7] together with some examples that we have applied our system to. Snippets from one of those examples are presented in Section 7 together with runtime results.

The reader is asked to note that this paper only addresses the question of *where* inherent parallelism can be found. We address neither the question of *which* parallelism should be exploited nor *how* best to exploit it.

Our specific contributions in this paper are:

– Application of Ownership Types to the problem of automatically parallelizing programs; the use of which has been suggested, by several authors, but there have been no experiments performed to determine if these reasoning systems work in practice for detecting inherent parallelism [3, 6–8].
– Sufficient conditions for the safe parallelization of data parallel `foreach` loops and several task parallelism patterns based on our framework for abstracting and reasoning about side-effects and data dependencies.
– A lightweight runtime ownership system which allows our techniques to operate in the presence of complex data flows. Our runtime implementation provides effect disjointness tests in constant time.

2 Background

To facilitate discussion of our parallelism analyses in subsequent sections, we first provide the reader with background information on Ownership Types and our static type system. We begin by providing a brief introduction to Ownership Types.

2.1 Introduction to Ownership Types

Consider the following code snippet:

```
private Object[] signers;
public Object[] getSigners() {...return signers;}
```

Note that despite the `private` annotation on the `signers` field, it is possible for the `getSigners` method to return the object referenced by this field. The `private` annotation on the field only protects the name of the field and not the data it contains. This code was the source of the infamous `getSigners` bug in Java 1.1.1 for precisely this reason [9]. Ownership Types [1–6] is one of the systems originally proposed to enforce this kind of protection in a rigorous manner.

Enforcing encapsulation requires each object to track: (1) which object's representation it is part of and (2) which objects are part of its representation. In Ownership Types this tracking is achieved through the notions of *ownership* and *object contexts* (here after referred to as contexts). As Clark, Noble, and Potter eloquently described it, "Each object owns a context, and is owned by a context that it resides within" [1]. This definition creates a tree of ordered contexts rooted in the top context called *world.* Each object has a context in which it may store its representation (the object's *this* context). Encapsulation enforcement in these systems is achieved by only permitting the object itself to name its *this* context. If one is unable to name a context one cannot name the type of a reference to an object in that context.

In the `getSigners` example, the `signers` field would have been denoted as owned by the *this* context which would have prevented its contents being returned and directly accessed by external components. Such invariants are useful from the perspective of parallelism analysis because we can reason that others are not accessing the protected data; that is we have some means of containing the scope of effects.

2.2 Ownership Syntax

Our system's ownership syntax is similar to that used by Effective Ownership Types [6]. Ownership Types are a form of constructed type similar to the idea of generic types. While generic types are constructed by providing a list of actual type parameters, Ownership Types are constructed by providing a list of contexts. Methods are normally parameterized by data values. In generic types, methods can also be parameterized by types; in a similar manner, we allow methods to be parameterized by context parameters. In the case of class definitions, the first formal context parameter in the list, by convention, represents the context that owns the object. Any other formal context parameters, if they exist, can be used as actuals to construct other types used within the class. In our extended C♯ language we support both generics and ownership types, so a class can have both type parameters and context parameters. Our syntax for ownership types uses square brackets for delineating the list of formal context parameters and vertical bars for delineating the list of actual context parameters. Whilst contexts are associated with objects, we cannot refer to the context of arbitrary objects, the only contexts that we can name are the special contexts *this* and *world* and formal context parameters visible in the current lexical scope. Below is an example showing this syntax:

```
class LinkedList<T>[x] {
  private Link<T>|this| head;
  ...
}
class Link<T>[y] {
  private Link<T>|y| next;
  private T dt;
  ...
}
```

In the above example, the head node is part of the representation of the linked-list and so is owned by the *this* context of the linked-list. The next field of the node class is defined recursively to also be owned by the same linked-list object. Simply having a private reference to an object does not imply that you own it. It is up to the programmer to decide the logical ownership relationships.

We also allow an objects' state to be subdivided into a set of named subcontexts to allow effects to be described at a sub-object level of granularity. Aldrich and Chambers were the first to propose such a subdivision of an object as part of their Ownership Domains system[10]. Section 7 demonstrates how these subcontexts are used in practice.

Finally, it is important to note that the built-in value types like int, double, and string as well user defined value types in the form of structs do not have have owners because they cannot be aliased; they are passed and copied by value not by reference like classes.

2.3 Side-Effects

Our system partitions effects into stack and heap effects. Stack effects only appear as part of local data-dependency analysis. Heap effects are captures by listing the contexts read and written. Programmers are required to specify heap read and write effects as lists of contexts on method signatures. The type rules for our language enforce the invariant that if an expression or statement reads some value on the heap then the context that owns the value or one of its ancestors is included in the computed read effect set and similarly for writes. The type rules for our language can be found in our companion technical report [11].

Note that the scope of effects can be described at different levels of abstraction due to the hierarchical nature of contexts. The scope of effects can be thought of as similar to street addresses. We could describe an effect as being limited to a very precise location, for example 5^{th} Avenue, Manhattan. It is also correct, but less precise, to say that the effect is limited to New York City or indeed to the United States. If we were then to observe an effect occurring in Boston we would know that the effect in New York and the effect in Boston could not interfere because they are in different cities. If, however, we were to observe an effect occurring in New York City, we know that the effect could interfere with our effect on 5^{th} Avenue.

Consider our previous linked-list example, the following shows the syntax for declaring effects:

```
class Link<T>[o] {
  private Link<T>|o| next;
  private T data;
  public Link<T>|o| getNext() reads<this> writes<> {
    return next;
  }
  public T getNextData() reads<o> writes<> {
    return next.data;
  }
}
```

In the above example, the `getNext` method reads a field from the current object which is captured as a read of *this*. The `getNextData` method reads the current object, generating a read effect of *this*. It also causes a read of the object referenced by `next` which is owned by *o*. The read effect contains only *o* because *this* is part of *o*'s representation and so a read of *o* includes a read of *this*.

This idea and style of effect annotation has been used before by other authors for different purposes. Geenhouse and Boyland were amongst the first to propose an effect system in terms of ownership style contexts[8]. Clarke and Drossopoulou extended these ideas to show how effects could be used for the purposes of validating program properties[3]. Lu and Potter have also proposed effect systems for reasoning about programs[6]. Other authors then took these effect systems and applied them to the problem of verifying locking protocols/ordering in already parallelized programs. Examples of such systems include the work of Boyapati, Lee, and Rinard[12] and Cunningham et al.[13]. Note that this is a very a different, and we believe less interesting, problem than the problem of automatically detecting the inherent parallelism in a sequential problem.

To support legacy components written without any context or effect annotation we employ two strategies. By default, any object without an owner is assumed to be owned by *world* and any method that does not have declared effects is assumed to read and write *world*. Such code will prevent parallelization but is guaranteed to be safe. In addition, we have invented a syntactic construct that allows programmers to specify ownerships and effects for existing legacy classes. The added ownership and effect information is treated as programmer assertions which are accepted on trust rather than being verified.

2.4 Separating Ownership from Encapsulation

The original ownership type systems[1] were designed to enforce strong encapsulation. They both provide a notation for describing which objects were owned by which other objects and placed strong restrictions on which contexts could be read or written from other contexts. Our proposed use of Ownership Types can work with such restrictions; however, they are not strictly necessary for our purposes. Like many recent Ownership systems, including MOJO [2] and Jo∃[14], we choose to omit such strong encapsulation enforcement to make programming easier; we only need to track reads and writes of the heap, not restrict them.

3 Ownerships and Data Dependencies

The key idea of this paper is that we can use the overlap of the read and write effect sets of sections of code to determine if data dependencies can exist. Data dependencies can be classified as either flow, output, or anti-dependencies. If the write set of one section of code does not overlap with the read set of some other then a flow dependence cannot exist. Similarly for output and anti dependencies.

When considering the overlap of effect sets we consider stack and heap effects separately as they represent disjoint sets of memory locations. Sets of stack effects overlap if they contain the same local variable or parameter names. Determining if sets of heap effects overlap is harder because context parameters with different names do not necessarily represent disjoint subtrees of the ownership tree. One context's relationship to another can be said to be:

- equal (=) they are one and the same
- dominating (<) one context is directly or indirectly owned by that on the other
- disjoint (|) they appear on different branches of the ownership tree

Two context sets $\overline{S1}$ and $\overline{S2}$ overlap if any of the contexts in the two sets overlap:

$$\text{overlaps}\left(\,\overline{S1},\overline{S2}\,\right) = \exists s \in \overline{S2}\ \exists t \in \overline{S2}\ \neg\left(\,s \mathbin{\#} t\,\right)$$

In some cases we can statically determine that all of the relevant contexts do not overlap and so we can safely parallelize the code. Similarly, in other cases we can statically determine that the effect sets are not disjoint and so we will not try to parallelize the code. In the remaining cases we may not be able to statically determine if the relevant effects can overlap, but we can determine this dynamically with our runtime system. As we will describe in Section 4, we can compute the relationship of two arbitrary contexts in constant time. It is important to note that presence of a runtime system does not require modifications to our static type system or our sufficient conditions for parallelism.

The following section discusses how the relationship between contexts can be tested at runtime. Following our discussion of our runtime system, we will formulate sufficient conditions for parallelism based on the data dependency techniques developed in this section.

4 The Runtime Representation

Consider the following code snippet of a method parameterized with two context parameters:

```
public void method[c1,c2](...) reads<c1> writes<c2> { ... }
```

Note that the relationship between `c1` and `c2` is not known until the method is invoked. There may be some calls where `c1|c2` and others where they are not. Producing code for every possible combination of context relationships is not feasible in general and so we need to be able to ask questions about the relationship of contexts at runtime.

Our runtime system compliments the static system by allowing context relationships to be checked at runtime. Further, each individual context relationship test can be done in $O(1)$ time even though the program may have a theoretically unbounded number of memory locations in use.

4.1 Context Testing

What we are trying to do is to find the relationship between two nodes in a tree. There are three different relationships which we may want to test for: equality, domination, and disjointness. What we are trying to do is determine if one context is included in the ownership subtree rooted at a second context. This problem is analogous to trying to determine if one type is a sub-type of a second type in an object-oriented language with single inheritance; this is known as the type-extension problem[15].

We map contexts to objects at runtime; this means that an object's *this* context is represented by the object itself; the distinction between the *this* context and `this` variable is, therefore, removed at runtime. The naive implementation of a runtime system would have each object maintain a single parent pointer to its owner. Context relationship tests could then be performed by chasing pointers in the same way that Wirth performed type-extension tests in Oberon[15]. This solution consumes a constant amount of space per object and provides constant-time object creation overhead, but $O(n)$ time relationship tests where n is the height of the hierarchy.

The runtime testing of context relationships is a potentially frequently executed operation. We have, therefore, chosen to use Cohen's solution to the type-extension problem[16] which uses Dijkstra's views[17]; each object maintains an array of pointers to its ancestors. This solution allows us to perform relation tests in $O(1)$ time at the cost of $O(n)$ creation time and space per object, where n is the height of the hierarchy. Fortunately, the maximal depth of ownership hierarchies tends to be low according to recent studies applying ownership to larger programs [18]. Alternative hybrid approaches, like the use of skip lists [19], could be used to provide implementations with time and space performance between these extremes.

4.2 Static Test Minimization

Even with the efficient runtime system outlined, it is necessary to minimize the number of disjointness tests required. We use two techniques to achieve this:

Static Reasoning. At compile time there are a limited number of context relationships which are statically known for any given class:

- An object's *this* context is dominated by its owning context
- All of the declared subcontexts are dominated by the *this* context
- All subcontexts of an object are disjoint from one another

This information can be used to make some parallelization decisions at compile time without runtime tests.

Context Constraints. We have added syntax to our language which allow programmers to statically constrain the relationship between context parameters on classes or methods, similar to C♯'s constraints on generic type parameters. The programmer can specify the relationship between contexts to be domination ($<$) or independence ($|$). The constraints are preserved by the type system during type extension, abstraction, and overriding. The compiler statically enforces these constraints during type checking. The example below shows a class with such constraints; specifically that o is dominated by d and d is independent of context t.

```
class Foo[o,d,t] where o < d where d | t
```

5 Task Parallelism

Imperative programs are composed out of sequence, selection, and repetition constructs. Selection is an inherently sequential operation in the absence of speculative execution so we focus on the parallelization of sequence and repetition constructs.

If a programmer has a set of operations that need to be performed, the imperative paradigm requires them to be listed in some arbitrary sequence thereby imposing a total order on the them. In actuality, the data dependencies between operations may only imply a partial order to the steps. The difference between this partial order and the total order represents potential for parallelism. We can construct the partial order by computing the data dependencies between operations. Our effects system allows us to build a Data Dependency Graph (DDG) easily to allows to detect and exploit this parallelism.

6 Loop Parallelism

Repetition parallelism can take many different forms. In this paper we focus on data parallel loops; those in which the structure of any available parallelism is based around the data. In C♯ data parallel loops most commonly take the form of the `foreach` loop which are the only type of loop considered in this section. We will present sufficient conditions for two parallelism patterns for such loops: (1) *data parallelism* where loop iterations execute independently and are distributed across multiple processors and (2) *pipelining* where the execution of a loop iteration is divided up into stages and distributed across multiple processors.

6.1 Loop Parallelism

The data parallelism pattern can only be safely applied if there are no inter-iteration dependencies. We begin by considering the following simple loop:

```
foreach (T|c| element in collection)
  element.operation();
```

We now state informal conditions which are sufficient to ensure there are no such dependencies:

- **Loop Condition 1:** There are no control dependencies which would prevent loop parallelization.
- **Loop Condition 2:** The objects traversed by the iterator are all different. Note that they all share the same owner so this implies their contexts are all disjoint.
- **Loop Condition 3:** The `operation` only mutates the representation of its "own" element and does not read the state owned by any of the other elements.

Detecting the control dependencies which are the subject of Loop Condition 1 is a much simpler problem than detecting data dependencies; we do not claim any new contribution with respect to detecting control dependencies in this paper. Loop condition 2 can be satisfied in one of two ways. Either we can dynamically test the uniqueness condition just prior to loop execution or we can have the programmer assert the uniqueness condition. In the case of a programmer assertion we have the option of verifying the uniqueness invariant at runtime or turning off such assertion checking in order to improve efficiency. If checked, such a uniqueness invariant could be verified either when an insertion takes place or just prior to when the invariant actually needs to hold. The uniqueness assertion can be made by annotating either the collection itself or its enumerator (a collection may contain duplicates, but if its enumerator only returns unique elements then the condition is still effectively met). The uniqueness annotation could be placed on the collection class, or just on specific instances of that collection class. Which of the above possibilities is used to ensure loop condition 2 is met will depend on programmer preferences and performance considerations - we therefore do not stipulate a single mechanism.

Loop Condition 3 says that the write set of `operation` can contain at most *this*. The read set can contain *this*, but it may also contain other contexts r, provided that we know r to be disjoint from c.

We now more formally state our sufficient conditions for parallelism: Let $\overline{R}$ and $\overline{W}$ represent the read and write effects of `operation`:

1. **Loop Condition 2:** The values in the collection traversed by the iterator are asserted to be `unique` meaning that
   $\forall i \in 1..|\texttt{iterated\_values}|\ \forall j \in 1..|\texttt{iterated\_values}|\ i \neq j \Rightarrow$
   $\texttt{iterated\_values[i]} \neq \texttt{iterated\_values[j]}$
2. **Loop Condition 3:**
   $\forall w \in \overline{W}\ w \leqslant \mathit{this} \wedge \forall r \in \overline{R}\ r \leqslant \mathit{this} \vee \big(r \neq c\big)$

If any one of the conditions is known not to hold, then we must execute the original sequential loop to preserve program correctness. We may not be able to decide if conditions 2 and 3 hold at compile-time depending on the contexts concerned. Context relationships may not be known until runtime and so conditionally parallel code is emitted when this is the case:

```
if (/*runtime test: all r's are disjoint from c*/)
  parallel.foreach (element in collection) { element.op(); }
else
  foreach (element in collection) { element.op(); }
```

Facilitating Upward Data Access. So far we have formulated a sufficient condition for data parallel loops designed to allow reading of disjoint and descendent contexts. We now look at facilitating access to ancestor contexts.

Figure 1 illustrates the ownership tree we would like to be able to support. We have a collection of elements $d_1...d_n$ which are owned by some object c. From context c we wish to read data from context r. If context r is not in scope (ie we cannot name it) then we must access r through context b, an upward access.

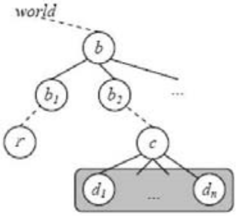


Fig. 1. Ownership relationships between contexts at runtime used for example of capturing context disjointness

Abstracting a safe read of the disjoint context r to be a read of b suddenly makes the read unsafe in our current scheme. To avoid this problem, we introduce the notion of sub-contexts to allow us to partition contexts like b.

With sub-contexts, context b would "own" a finite number of named sub-contexts b_1 and b_2. We only permit the *this* context to be subdivided into sub-contexts. Using these sub-contexts reading r could be summarized as a read of b_1 rather than b itself. If the elements returned by the enumerator are located in sub-context b_2, then we could safely allow the read of b_1 as it is disjoint from c. The idea of sub-contexts has been presented previously by other authors including Clarke and Drossopoulou who used them to provide more precise effect information[3].

Within each class, the programmer can decide if they wish to declare sub-contexts and if they do, they can declare as many as they desire. In the extreme case, each private field might be given its own sub-context, but programmers would more commonly create a sub-context to encapsulate a group of related private fields. The more sub-contexts, the more information that needs to be passed as context arguments on types; the creation of sub-contexts is a trade-off between precision and complexity. Sub-contexts are limited in scope to their class of declaration. To children they look like any other context passed down from the parent while to parents they appear to be part of the owning class' representation.

Loop Body Re-writing. Now that we have explored the sufficient conditions for the parallelization of a simple data parallel loop, the question of how to generalize these conditions to handle arbitrary `foreach` loop bodies arises.

Consider the following loop:

```
class Foo[o] {
  foreach (T|e| elem in collection)
    // sequence of statements possibly including local variable defs
}
```

Fortunately, generalization to arbitrary loop bodies is a natural extension of our existing techniques. We can conceptually re-write the loop body as:

```
class Foo[o] {
  foreach (T|e| elem in collection)
    elem.loopBody|o|(this);
}
```

where `o` is the owner of the class containing the loop and conceptually becomes a method of the element type T:

```
class T {
  void loopBody[c](Foo|c| me) {
    // same sequence of statements replacing all elem by this
    // and all this by me
}}
```

6.2 Pipelining

The data parallelism pattern for loop parallelization can only be applied to loops without inter-iteration dependencies. Consider the following loop:

```
foreach (T|o| elem in collection) {
  S_A; S_B; S_C; S_D;
}
```

This loop may, for example, have both intra- and inter-loop iteration dependencies as depicted in Figure 2. Despite the presence of the dependencies it is possible, for example, to execute iteration 1 of `S_B` in parallel with iteration 2 of `S_A` (provided iteration 1 of `S_A` has already completed execution).

The only form of dependence that we must rule out is a dependence from an iteration p of statement S_i to a later iteration q of some statement S_j where

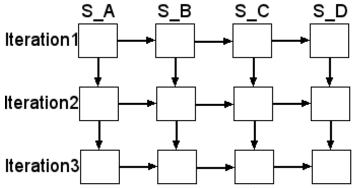


Fig. 2. Diagram showing the permitted data dependencies between stages and iterations. S_A through S_A represent four pipeline stages and $Iteration1$ through $Iteration3$ represent three iterations.

$j < i$. In Figure 2, these dependencies would take the form of diagonal edges moving down and to the left.

To determine which dependencies exist we must first compute the loop body's "virtual" effects of each statement within the loop body using the techniques from Section 6.1. So if a statement reads or writes any part of the representation of the loop iteration variable `elem` then that will show-up as *this* in the virtual effect set of that statement.

We now formalize the sufficient conditions for the safe pipelining of a data parallel loop with stages $S_1..S_n$:

1. the enumerated values are asserted to be unique which means that $\forall i \in 1..|\texttt{iterated\_values}|\ \forall j \in 1..|\texttt{iterated\_values}|\ i \neq j \Rightarrow \texttt{iterated\_values[i]} \neq \texttt{iterated\_values[j]}$
2. There does not exist a dependence (flow, output, or anti) from S_i to S_j where $j < i$. The presence of such dependencies is determined as described previously based on the disjointness of the read and write virtual effect sets of the statements in question.

A number of techniques for detecting and scheduling loops for pipelined execution have been developed over the years [20]. All of these techniques consume a data dependency graph (DDG), like that used for the separation of code blocks into sequences for concurrent execution (see Section 5).

As with full loop parallelization, pipelining relies on specific relationship between the contexts being read and written. If these relationships cannot be statically determined, both the sequential and pipelined versions of the loop can be produced and the choice of which to execute deferred until runtime.

6.3 Data Parallel for Loops

Our techniques only work on data parallel which typically take the form of foreach loops in C♯. We cannot handle arbitrary for loops as they provide no means of associating iterations with distinct data elements. There are, however, some loops expressed as for loops, which are data parallel in nature and could conceptually be converted to foreach. This is not done in many cases due to the semantic restrictions of foreach loops. Specifically, foreach loops only give us access to the value of each element, but do not allow you to change the elements of the collection in place. Further, for loops are often used we need not just the value of each element, but also the index of the element within the collection:

```
for (int i = start; i < list.Count; ++i)
  list[i] = func(i, list[i], ...);
```

To support such cases, we have extended the syntax and semantics of foreach loops, over collections which support indexing, to address both of these problems. The following shows our syntax for expressing such a loop as a foreach loop:

```
foreach (ref ElemType e at Index i in list)
  e = func(i, e, ...);
```

One remaining problem with foreach loops is how to efficiently execute them in parallel across multiple processors. In the case of a for ranging over index values, it is relatively easy to express the subranges to be assigned to each processor. A common approach to parallelizing such loops to use a preliminary inspector phase which sequentially extracts each of the elements prior to the actual loop which then processes partitions of these elements in parallel. In specific cases, the inspector phase can be avoided by using custom collection traversal code. In the case of a list, this produces the same code as the equivalent parallel for loop.

The above syntax can only be used on collections which have specific support for such enhanced iteration. This support can be added to existing classes using C♯'s extension method mechanism and `ref` call parameters (source available on our website[7].

6.4 Proof of Correctness

Finally, in this section we present a proof that loop conditions 1, 2 and 3 as presented in Section 6.1 are sufficient to safely parallelize a `foreach` loop without synchronization. Proofs of the correctness of the sufficient conditions for the other patterns we have presented are very similar and straightforward.

As demonstrated in our technical report [11], our static type system guarantees that if a code fragment directly or indirectly writes a field of an object then the owning context of the object, or one of the contexts which dominates it will appear in the computed write set of the expression. Similarly for read sets.

A loop can be parallelized provided no data or control dependencies exist between iterations. Loop Condition 1 tells us that no problematic control dependencies, such as exceptions, exist. Data dependencies take one of three forms as previously described: output dependencies, flow dependencies, and anti-dependencies.

Assume, by way of contradiction, that an output dependence exists between iterations. The collection must contain two separate elements e_1 and e_2 such that e_1`.operation()` writes to a field of some object x and e_2`.operation()` writes to that same field of x.

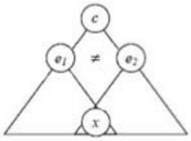


Fig. 3. The relationships between e_1, e_2, and x

The write set of `operation()` may contain only *this*, so we know that e_1`.operation()` can only write to objects that are either e_1 or strictly dominated by e_1. Similarly, e_2`.operation()` can only write object that are either e_2 or strictly dominated by e_2. Figure 3 shows this set of relationships.

Each object is owned by a unique context. If x is dominated by e_1 and e_2, it must be the case that either e_1 dominates e_2 or e_2 dominates e_1. But, e_1 and e_2 are directly owned by the same owner c and $e_1 \neq e_2$ by Loop Condition 1 which provides the contradiction.

Assume now, by way of contradiction, that a flow dependence exists. The collection must contain two elements e_1 and e_2 such that e_1`.operation()` writes to some field x and e_2`.operation()` reads that same field x. We know from the previous step of the proof above that there is no x that is part of both e_1's and e_2's representation.

The only other source of such a flow dependence would be if e_2`.operation()` reads the same field x via some context r such that r is disjoint with respect to e_1's and e_2's owning context c. Figure 4 shows the relationship between c, e_1, e_2, and x.

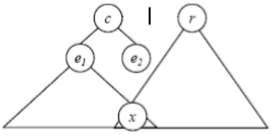


Fig. 4. Relationship of e_1,e_2,c,r, and x and the separation of c and r for the proof of the absence of flow dependencies

So, x is dominated by e_1 which is dominated by c. But x must also be dominated by r which is not possible as $c \# r$. Therefore, no flow dependence can exist. A mirror argument can be made to prove the absence of anti-dependencies.

7 Worked Example

In this section we will present an example of the parallelization of a ray tracing application. This much example demonstrates inter-procedural effect analysis and conditional parallelization based on runtime context relationship testing. The original application was released by Microsoft as part of its Samples for Parallel Programming with the .NET Framework 4 [21]. Note that this example has already been manually parallelized by Microsoft programmers. We are not trying to do a better job of parallelizing the application, we are simply trying to demonstrate that our system can automatically detect the known data parallel loops. Traditional data dependency analysis based systems would struggle to handle the inter-procedural data dependencies found in the program.

The key source of parallelism in this application is the loop in the Render method, where the color of each pixel is determined by tracing rays from the light sources in the scene to the camera. Each ray only reads the state of the scene as its path and color are computed which allows us to trace multiple rays at the same time. The original formulation of this loop is presented below:

```
internal void Render(Scene scene, Color[] scr) {
  Camera camera = scene.Camera;
  for (int y = 0; y < screenHeight; y++) {
    int stride = y * screenWidth;
    for (int x = 0; x < screenWidth; x++) {
      scr[x + stride].color = TraceRay(new Ray(camera.Pos,
        GetPoint(x, y, camera)), scene, 0);
}}}
```

The first problem is that the loop is in not in the form of a foreach loop. Note that the loop is actually iterating over the elements of the `scr` array and so can be transformed using our modified foreach loop syntax. We then add the ownership annotations to this modified loop to produce the following code:

```
internal void Render[s,t](Scene|s| scene, Color[]|t| scr)
  reads<this,s,t> writes<t> {
    Camera|s| camera = scene.Camera;
    foreach(ref Color pixel at int Index in scr) {
      pixel = TraceRay|s|(new Ray(camera.Pos,GetPoint|s|(
        Index % screenWidth, Index / screenWidth, camera)), scene, 0);
}}
```

The power of our system becomes evident when we attempt to determine if the loop can be safely parallelized. In traditional systems the required data dependence analysis would be very complicated because of the aliasing possibilities and number of methods invoked. Our system, on the other hand, allows us to look at the declared effects of the `TraceRay` and `GetPoint` methods. They read contexts `s t` and write nothing. Our compiler and the type system it implements ensures that the method body effects are consistent with the declared effects; the method body cannot cause side-effects not listed in the declared effects. For example, the `Normal` method indirectly called by `TraceRay`:

```
abstract class SceneObject[o] {
  public abstract Vector Normal(Vector pos) reads<this> writes<>;
}
```

```
class Sphere[o] : SceneObject|o| {
 public override Vector Normal(Vector pos) reads<this> writes<>...
}
class Plane[o] : SceneObject|o| {
 public override Vector Normal(Vector pos) reads<this> writes<>...
}
```

Note that our compiler also enforces effect consistency in the presence of overriding so that we do not need to determine which `SceneObject` implementation is being used. The read effect of `this` is abstracted to become context `s` (the owner of the scene) in `TraceRay`.

From the loop body's effects of reading contexts `this`, `s` and `t` and writing `t` and our sufficient conditions, the sufficient conditions for safely executing the foreach loop in parallel are (1) the elements of `scr` are unique, (2) `this` is disjoint from or a child of `t`, and (3) `s` is disjoint from or a child of `t`. Because the relationship between the relevant contexts is not known until runtime, the loop is conditionally parallelized by the compiler subject to these being true.

Table 1. The number of frames rendered per second by the sample ray tracing application. The original sequential and parallel values were obtained from the unmodified application. The enhanced foreach value was obtained from the ownership annotated code.

Implementation	**Frames / sec**
original sequential	0.325
original parallel	0.707
enhanced foreach	0.609

Table 1 shows the average number of frames per second rendered with each of the three implementations. The original parallel and sequential values were obtained using the original code supplied by Microsoft, while the enhanced foreach value was obtained using the automatically parallelized version of the application. The performance runs were conducted on an Intel Core 2 Duo T5800 with 4GB of RAM running Windows 7 Professional 64bit Edition and the .NET 4 Beta 1 runtime. Overall the results show that the performance of the automatically parallelized application is very close to that of the hand parallelized version. There is some performance degradation which can be attributed to a combination of the overheads added by the runtime ownership tracking system and by the overheads incurred in the use of the new enhanced foreach loop; these overheads can be reduced with further development and optimization. Overall, our system is lightweight and efficient at identifying exploitable parallelism.

To annotate the program, after applying the loop transformation discussed earlier in this section, we had to modify 99 lines out of 619 lines of the original application (15%). The majority of these changes were adding method effect lists to method signatures and the addition of context parameters to types. While somewhat burdensome, smart defaulting and ownership inference could reduce this annotation overhead while still providing the benefits outlined previously.

8 Related Work

As was highlighted in the introduction we are not the first to propose capturing effects using ownership contexts nor are we the first to propose many of the language features discussed in this paper. We are the first to apply Ownership Types to the problem of automatically parallelizing existing imperative programs. Others have applied Ownership Types to the simpler, and we feel less interesting, problem of verifying lock ordering in parallelized programs to prevent deadlocks and data races [12, 13]. We now discuss a few of the most directly related works in this area of automated parallelization.

A large body of work has been previously published on the use of local dataflow analysis to extract parallelism from complex iterative algorithms expressed in imperative languages. One example of such work is that of Rus, Pennings and Rauchwerger [22]. These techniques proved very good at extracting fine-grained parallelism from the complex iterative numerical kernels common in the HPC workloads at which they were targeted. Our work has focused on extracting a more course-grained parallelism which scales across method and component boundaries. This kind of course-grained parallelism will become increasingly important as the number of cores per chip grows and more general purpose applications need to exploit parallelism.

Marron, Stefanovic, Kapur, and Hermenegildo proposed techniques for reasoning about data dependencies in Java [23]. Their approach is to perform complex analyses at compile-time on unmodified programs and not try to facilitate programmer reasoning directly as we do. They employ static analysis to determine data dependencies, but need to examine the implementation of any method called to compute dependencies. They do memoize their method analyses, but do not provide the consistency guarantees on overriding like we do and do not make their effects part of the programmers conceptualization. They do, however, handle looping constructs other than `foreach` loops. It is unclear how their techniques would work for large programs.

Various parallel programming languages have also been developed, but they are largely designed for expressing parallelism rather than facilitating the process of parallelizing an existing application by automatically detecting inherent parallelism. X10 [24] is a programming language under development by IBM as part of the DARPA HPCS program which is designed to support parallel programming. Its syntax and features are inspired by Java, but a number of different parallelization and synchronization mechanisms have been purposely included in the language syntax. Our language and X10 serve different purposes. X10 helps programmers familiar with parallelism write and debug parallel applications. We aim to provide a framework for programmers and automated tools to reason about inherent parallelism in sequential programs. Our work and X10 can be viewed as complimentary.

9 Conclusions and Future Work

In this paper we have presented an effects system based on topological ownerships which allow us to reason about the data dependencies in modern imperative object-oriented languages. We have presented sufficient conditions for the safe exploitation of several different patterns of task and data parallelism. We have demonstrated the need for a complimentary runtime ownership system and how such a system can be efficiently implemented.

In the future we hope to expand our techniques to include other looping patterns and continuing to loosen the sufficient conditions for parallelization. To reduce the burden on the programmer we would like to explore automated ownership inference. It would be interesting to explore how our techniques can be applied to C♯'s pointers, possibly similar to Cyclone [25]. We hope to continue to add our annotations to further real applications to gain further understanding in how our extensions affect program development and how much of the available parallelism we are able to successfully exploit. We do not claim that our system is ready for production use, but we feel that some kind of framework to facilitate reasoning about inherent parallelism is necessary. We hope that this work will stimulate further exploration of this space.

References

1. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: 13th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 48–64. ACM Press, New York (1998)
2. Cameron, N., Drossopoulou, S., Noble, J., Smith, M.: Multiple ownership. In: 22nd annual ACM SIGPLAN conference on Object-Oriented Programming Systems and Applications, pp. 441–460. ACM Press, New York (2007)
3. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA 2002: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 292–310. ACM, New York (2002)
4. Potanin, A., Noble, J., Clarke, D., Biddle, R.: Generic ownership for generic java. In: OOPSLA 2006: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pp. 311–324. ACM, New York (2006)
5. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: OOPSLA 2002: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 311–330. ACM, New York (2002)
6. Lu, Y., Potter, J.: Protecting representation with effect encapsulation. In: POPL 2006: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 359–371. ACM, New York (2006)
7. Craik, A., Kelly, W.: Mquter parallelism research (2009), `http://www.mquter.qut.edu.au/par`
8. Geenhouse, A., Boyland, J.: An object-oriented effects system. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, p. 205. Springer, Heidelberg (1999)

9. Sun Microsystems, Jdk 1.1.1 signing flaw (March 1997)
10. Aldrich, J., Chambers, C.: Ownership domains: Separating aliasing policy from mechanism. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 1–25. Springer, Heidelberg (2004)
11. Craik, A.: Ownership types for reasoning about parallelism - type system and semantics. Technical report, QUT ePrints, Queensland University of Technology (2009), `http://eprints.qut.edu.au/`
12. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: Preventing data races and deadlocks. In: 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 211–230. ACM Press, New York (2002)
13. Cunningham, D., Drossopoulou, S., Eisenbach, S.: Universes for race safety. In: 1st International Workshop on Verification and Analysis of Multi-Threaded Java-like Programs (2007)
14. Cameron, N., Drossopoulou, S.: Existential quantification for variant ownership. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 128–142. Springer, Heidelberg (2009)
15. Wirth, N.: Type extensions. ACM Trans. Program. Lang. Syst. 10(2), 204–214 (1988)
16. Cohen, N.H.: Type-extension type test can be performed in constant time. ACM Transactions on Programming Languages and Systems (TOPLAS) 13(4), 626–629 (1991)
17. Dijkstra, E.W.: Recusive programming. Numerische Mathematik 2(1), 312–318 (1960)
18. Abi-Antoun, M., Aldrich, J.: Compile-time views of execution structure based on ownership. In: International Workshop on Aliasing, Confinement, and Ownership in Object-Oriented Programming 2007 (2007)
19. Pugh, W.: Skip lists: A probabilistic alternative to balanced trees. Communications of the ACM 33(6), 668–676 (1990)
20. Allan, V.H., Jones, R.B., Lee, R.M., Allan, S.J.: Software pipelining. ACM Comput. Surv. 27(3), 367–432 (1995)
21. Microsoft Corporation, Samples for parallel programming with the .net framework 4 (May 2009)
22. Rus, S., Pennings, M., Rauchwerger, L.: Sensitivity analysis for automatic parallelization on multi-cores. In: ICS 2007: Proceedings of the 21st annual international conference on Supercomputing, pp. 263–273. ACM, New York (2007)
23. Marron, M., Stefanovic, D., Kapur, D., Hermenegildo, M.: Identification of heap-carried data dependence via explicit store heap models. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 94–108. Springer, Heidelberg (2008)
24. Saraswat, V., Nystrom, N.: Report on the experiment language x10. Technical Report 1.7.5, IBM (2009)
25. Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., Cheney, J.: Region-based memory management in cyclone. In: ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, 2002, pp. 282–293. ACM Press, New York (2002)

Punctual Coalescing

Fernando Magno Quintão Pereira[1] and Jens Palsberg[2]

[1] Universidade Federal de Minas Gerais, Belo Horizonte
[2] University of California, Los Angeles

Abstract. Compilers use register coalescing to avoid generating code for copy instructions. For architectures with register aliasing such as x86, Smith, Ramsey, and Holloway (2004) presented a polynomial-time approach, while Scholz and Eckstein (2002) presented an optimal, exponential-time approach together with a near-optimal, quadratic-time heuristic. Both methods scale poorly after aggressive live range splitting, especially for programs in elementary form where live ranges are split at every program point. In contrast, we mentioned in a previous paper (2008), without giving details, that we have a scalable, linear-time heuristic for programs in elementary form. In an effort to formalize that heuristic, we discovered an even better algorithm, called Punctual Coalescing, which we present here. Punctual Coalescing is scalable, linear time, locally optimal in general, close to globally optimal for straight-line code, and proven correct with the Twelf theorem prover. We define global optimality with an ILP-formulation and we show via experiments that Punctual Coalescing compares well to this and two other approaches.

1 Introduction

Register allocation is the problem of mapping program variables to physical locations, which are either registers or memory. Compared to mapping all variables to memory, a good register allocator can improve the speed of the generated code on a RISC architecture by 250% [22]. We will focus on a combination of three important challenges for register allocation, namely *live-range splitting*, *coalescing* and *aliasing*, which we recall next.

To keep more variables in registers, compiler writers use live-range splitting [2,6,17,26,31]: split the live range of a variable y by (1) introducing a fresh variable name x, (2) inserting the copy instruction $x = y$ somewhere in y's live range, and (3) using the name x instead y after that copy instruction. After the split, the register allocator has the opportunity to map x to a register and y to memory, or vice versa. Coalescing [9,11,12,15,16,17,23,30] is the dual of live-range splitting: eliminate copy instructions of the form $x = y$ by mapping both x and y to the same register. Intuitively, the more we do live-range splitting, the more we need coalescing to eliminate unnecessary copy instructions. The third challenge, aliasing, is a property of architectures such as ARM, PowerPC, Sparc v8/v9, and x86: quoting Smith *et al.*, "two registers alias when assigning a value to one may change the value of the other" [33]. Open until now is the problem of designing a scalable, high-quality, and provably correct register allocator

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 165–184, 2010.

that after aggressive live-range splitting does coalescing for an architecture with aliasing. Let us briefly summarize the most closely related previous works.

There exist register allocation algorithms that deal with aliasing. An example is the integer linear programming (ILP) approach of Kong and Wilken [19]. Scholz and Eckstein (2002) [32] have addressed aliasing with partitioned boolean quadratic programming (PBQP). They presented an optimal, exponential-time approach together with a near-optimal, quadratic-time heuristic. Smith, Ramsey, and Holloway [33] have generalized graph coloring register allocation to incorporate aliased registers. Also based on graph coloring allocation, Minwook *et al.* [1] have described an optimistic coalescing algorithm that is competitive with Smith *et al.*'s iterative approach. These methods scale poorly after aggressive live range splitting. Intuitively, aggressive live-range splitting enables a high number of variables to be mapped to registers, but it also overwhelms the register allocator with copy instructions. We will show how to deal with the high number of copy instructions by adopting a particular program representation and then developing a new coalescing algorithm.

We will work with Appel and George's idea from 2001 [2] of "ultimate" live-range splitting that splits every live-range at every program point, that is, between every pair of consecutive instructions. The result is a program in what we call elementary form. A compiler can convert any program to elementary form in polynomial time, and the elementary program requires at most as many registers as its original version. We use the notion of elementary form because it allows us to avoid a difficult problem. The problem of finding the minimal number of registers that is needed to compile straight-line code to an architecture with aliasing is NP-complete [21], while for a program in elementary form, the problem can be solved in linear time by a puzzle solver [26]. Our goal is to add coalescing to the linear-time puzzle solver without changing the time complexity.

In a previous paper [26] we mentioned, without giving details, that we have a scalable puzzle solver that embodies a heuristic for coalescing. In other words, that unpublished heuristic goes a long way toward solving the open problem. In an effort to formalize that heuristic, we discovered an even better algorithm, called *Punctual Coalescing*, which we present here.

Punctual Coalescing is scalable, runs in linear time, and is a form of biased coloring [9] that uses only local information. The puzzle solver with Punctual Coalescing traverses the dominator tree of the source program finding at each program point a register assignment that minimizes the number of variables sent to memory. The assignment is guided by the assignment found at the most-recently visited program point. Punctual coalescing is well suited for just-in-time compilers such as TraceMonkey [14], and tree-scan-based allocators such as Braun and Hack's [8]. We have proved the correctness of Punctual Coalescing, and in particular we have proved the main lemma with the Twelf theorem prover [28].

In general, punctual coalescing is locally optimal for straight-line code, and close to globally optimal. Our experiments with compiling SPEC CPU 2000 to x86 show that punctual coalescing finds a locally optimal solution for 89% of the program points in our benchmarks. We define global optimality with an

ILP-formulation that combines ideas from papers by Kong and Wilken [19], who showed how to handle aliasing, and by Grund and Hack [16], who showed how to handle coalescing. During the compilation of the SPEC CPU 2000 benchmark suite to x86, only one copy was inserted per 14 instructions in the original program. These copies were typically used to insert fixing code between basic blocks, and to avoid conflicts with pre-allocated registers, as we discuss in Section 6.

We have done an experimental comparison of four register assignment approaches: register allocation via coloring of chordal graphs [25], the heuristics used in the original puzzle solver [26], the punctual coalescing algorithm and the ILP formulation – the last two algorithms are introduced in this paper. To overcome scalability issues with the ILP approach we derived long program traces from SPEC CPU 2000, that is, long sequences of code that were executed in order. For those program traces our experiments show that Punctual Coalescing is considerably better than the other approaches and close to globally optimal.

In the next section we briefly review register allocation by puzzle solving, and illustrate the coalescing problem with an example. In Section 3 we describe Punctual Coalescing, in Section 4 we describe our ILP-formulation of global optimality, in Section 5 we show experimental results, in Section 6 we discuss limitations of punctual coalescing and in Section 7 we conclude the paper.

2 Background

A *program point* is any point in between two consecutive instructions, or in between two consecutive basic blocks. The program in Figure 1 has five program points, numbered 2 to 6. A variable v is alive at program point p if there is a path from p to an instruction that uses v that does not cross a definition of v. For instance, in Figure 1, variable a is alive at program points 2, 3, 4, 5 and 6. The program points where variable v is alive form v's live range. We can split the live range of a variable inserting a copy instruction at some program point in the live range, and doing variable renaming. Many register allocators use live range splitting to keep more variables in registers [2,17,26,31,34]. The *elementary form* is a program representation introduced by Appel and George [2] in which the live ranges of variables are split at each program point. If P is a program with V variables and I instructions, and P' is P converted to elementary form, then P' contains $O(I \times V)$ variables. Many register allocators are at least $O(V^2)$ – in particular, aliasing aware methods such as Scholz and Eckstein [32]'s PBQP approach and Smith *et al.*'s [33] extensions for Chaitin style algorithms. Hence, these algorithms run in at least $O(I^2 \times V^2)$ when applied to elementary programs.

Register allocation by puzzle solving: Register allocation by puzzle solving [26] relies on elementary form to minimize register usage. In this paradigm, registers are modeled as a puzzle board, and the live ranges of the variables as puzzle pieces. There is one puzzle per program instruction, and the challenge is to arrange the pieces on the board, so that no piece will be left out. We illustrate this method with the example given in Figure 1. The program on the left side

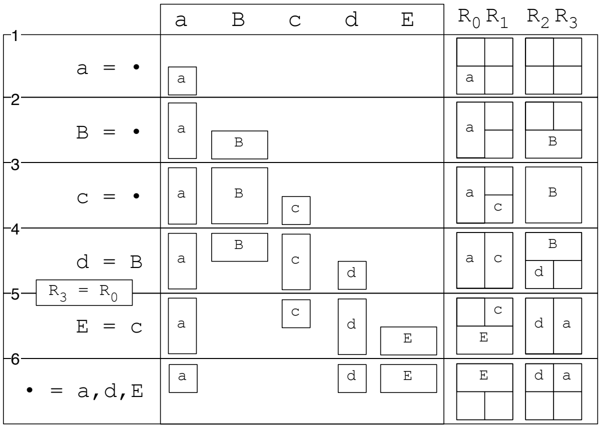


Fig. 1. An example of register allocation by puzzle solving

of the figure has six instructions and five variables, a, B, c, d and E. The live ranges of the variables are shown in the middle of the figure. We assume a target architecture with two registers, each one with two aliases. Such architectures are called *T1*, for type 1 puzzle. The type of a puzzle is determined by the number of columns in each board area: a puzzle *Tn* has 2^n columns per area. Lower case letters denote single precision values, whereas upper case letters denote double precision values. We can store two single precision values or one double precision value in one register. The opcode of each instruction is not relevant to our explanations, so we use •'s for "don't care's". The right side of Figure 1 shows a solution to this instance of the register allocation problem.

In this paper we provide coalescing algorithms for T1 puzzles. These puzzles model registers that have two independent aliases, such as the general purpose registers found in x86 (`AX, BX, CX` and `DX`), and the floating point registers found in ARM and PowerPC. It subsumes T0 puzzles, which we find in integer registers of PowerPC and ARM. T1 puzzles have three types of pieces: X, Y and Z. X pieces, such as a, d and E in puzzle six of Figure 1 can only be placed on the upper half of a board area. On the other hand, Z pieces such as B in puzzle two are only placed on the lower half of an area. Y pieces such as a and B in puzzle three occupy the upper and lower part of an area. A T1 puzzle piece may have width one or two. Size one pieces such as a, c and d in Figure 1 fit in one column of an area; they represent eight bit variables in x86, or single precision floating point values in ARM and PowerPC. Size two pieces, such as B and E span two columns. They represent 16 or 32 bit values in x86, or double precision numbers in ARM and PowerPC. We will be working with *padded* puzzles, that is, our puzzle solver expects that the area of the pieces will equal the area freely available on the board. We pad a puzzle by adding to its original set of pieces as many size one X and Z pieces as needed. A puzzle has solution if, and only if, the padded version does [26, Lemma 26].

Register coalescing: The register assignment in Figure 1 is optimal in two senses. First, it uses the minimal number of registers – it is not possible to compile this program with only one register divided into two aliases. Second, it uses the minimal number of copies to split the live ranges of variables. In order to obtain the minimal register assignment, we had to move variable a from register R_0 to register R_3. This split is performed by a register move inserted at program point five. This solution is globally optimal – the minimal register assignment requires the insertion of one copy instruction into the source code. In general, inserting copies to avoid mapping variables to memory leads to faster programs [26]; however, ideally we would like to minimize the number of copies inserted into the final program – an optimization known as coalescing. We distinguish two variations of coalescing: global and punctual, which we define below:

- Global Coalescing
  Instance: a program P in elementary form that can be compiled with K registers.
  Problem: find a register assignment for P, using K registers, that minimizes the number of instructions between puzzles.

- Punctual Coalescing
  Instance: two consecutive puzzles p_1 and p_2, such that p_1 is already solved.
  Problem: find a solution of p_2 that minimizes the number of copies inserted between p_1 and p_2. We call puzzle p_1 the *guider*, and puzzle p_2 the *follower*.

In the definition of global coalescing we assume that the input program is *greedy K-colorable* [6, p.18], that is, it is possible to find an allocation of variables to registers using at most K registers. K colorability ensures that spilling plays no role in the coalescing problem. This is the principle behind many register allocators based on live range splitting [2,17,18,25,26,31]. These algorithms are divided into two phases [6]. Initially a spilling phase removes variables, mapping them to memory, in order to ensure K colorability. Subsequently, a coloring phase finds a valid mapping of variables to registers using the available registers.

Global coalescing has a natural description as a graph coloring problem. The *interference graph* of a program is the interference graph of the live-ranges of the variables in the program. That is, given a program P, if G is its interference graph, then G has one vertex for each variable in P, and two nodes are adjacent if, and only if, they correspond to variables with overlapping live ranges. In the global coalescing problem we consider a second type of edge called *affinity edge*. There exists an affinity edge between two nodes v_1 and v_2 if P contains a copy instruction $v_1 = v_2$. The global coalescing problem asks for a coloring of G with at most K colors that maximizes the number of affinity related nodes that get the same color. In the presence of register aliasing, we consider each color as an integer number, so that some nodes must receive two consecutive colors. Figure 2 shows the graph coloring representation for the coalescing problem in Figure 1. Dashed lines represent affinity edges, grey nodes represent variables that fit into a full register, and white nodes represent variables that fit in half a register.

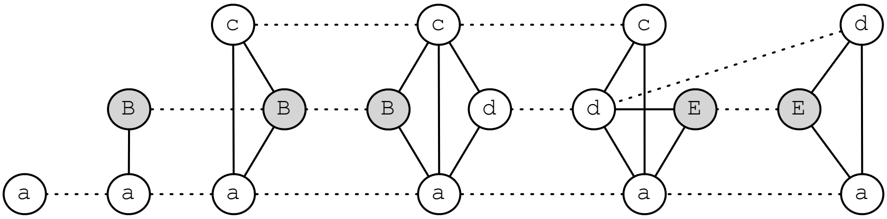


Fig. 2. A graph coloring representation for the global coalescing problem in Figure 1

Global coalescing is the version traditionally studied in the compiler literature. This problem is NP-complete [6]; thus, it is normally solved by heuristics, such as Chaitin's aggressive algorithm [12], or Brigg's conservative algorithm [9]. In Section 4 we give an optimal solution to this problem, in a T1 architecture, via integer linear programming. When restricted to program traces, global coalescing has polynomial time solution for T0 register banks, and is NP-complete for T1 register banks [21]. The problem is NP-complete if some variables are forced to be in particular registers [4], even restricted to program traces in T0 settings. Punctual coalescing has polynomial time solution for T0 and T1 architectures, as we show in Section 3. The complexity of this problem in the context of higher order register banks, or when pre-coloring is allowed is left open.

A sequence of optimal solutions to the punctual coalescing problem might produce a solution to its global counterpart, as in Figure 1. However, that is not always the case, as we show in Figure 3. The figure contains three instances of the punctual coalescing problem, one for each point between two consecutive puzzles. Each of these instances is optimally solved, and a copy is inserted between puzzles two and three. However, there is a register assignment that does not require copies between instructions, shown in the right column of the figure.

3 An Efficient Punctual Coalescing Algorithm

In this section we describe a strategy for solving the punctual coalescing problem. Our strategy is optimal for settings with initially empty follower boards. By optimal we mean that, if an instance of punctual coalescing has a solution with at most n copies inserted, then our algorithm will find it.

If a piece v fills the bottom of area a in the guider's board, we say that a is the *preferred area* for v in the follower's board. For instance, in Figure 1, R_1 is the preferred area for piece c in puzzle four, because the bottom part of R_1 is holding c in puzzle three. Also, R_2/R_3 are the preferred areas of piece B in puzzle three, for these areas are the location of B in puzzle two. In general, X and Y pieces have preferred areas, whereas Z pieces never have it.

We extend the notation introduced in [26] to include preferences between pieces and board areas. If a piece has no preference, we call it anonymous, in contrast with labeled pieces, which have preference for some area. Anonymous pieces are marked with the symbol •, and labeled pieces are given the name

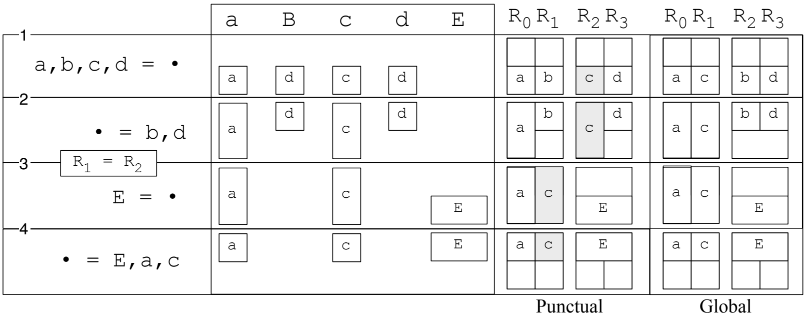
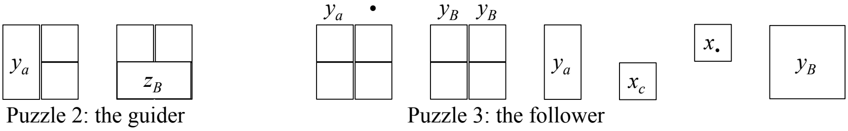


Fig. 3. An example where a sequence of optimal punctual coalescings is worse than global coalescing

of the variable that they represent. Each column of the board area now has a label, which is the name of the piece with a preference for that column. There are eight ways, up to symmetry, to label a T1 area. These patterns are shown in Figure 4. The shaded areas are not part of the pattern; they only illustrate where the preferred pieces should stay. Each area of the follower board has one of these patterns. Going back to the running example from Figure 1, area R_0/R_1 of puzzle two has pattern (h). However, the same area in puzzle five has pattern (g), with a preference for pieces a and c, as the registers R_0 and R_1 contain these pieces in puzzle four. As another example, we illustrate puzzle three below:



Our puzzle solving algorithm is given in Figure 5. This algorithm, written in a visual language, solves puzzles by pattern matching. It has eight *statements*, one for each possible pattern of preferences that can be found in an area. Each statement is composed by one or more *rules*, which specify how an area must be filled with pieces. Syntactically, a rule is a two-by-two diagram formed by a *pattern* and a *strategy*. The pattern is one of the eight configurations given in Figure 4. A strategy is a description of how to complete the area, including which pieces to use and where to put them. We say that the pattern of a rule *matches* an area a if the pattern contains the same sequence of preferences as a. For a rule r and an area a where the pattern of r matches a:

- the application of r to a *succeeds* if the pieces needed by the strategy of r are available; the result is that these pieces are placed in a;
- the application of r to a *fails* otherwise.

The complexity of solving a puzzle with A areas is $O(A)$. The rules of a statement are tried in order. If one of them succeeds, then the statement succeeds. If no

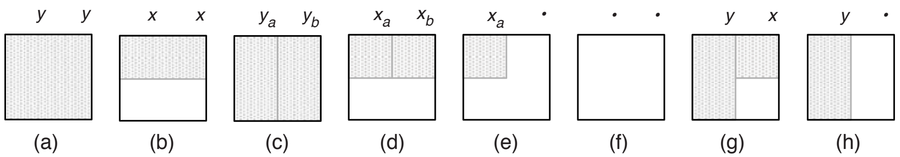
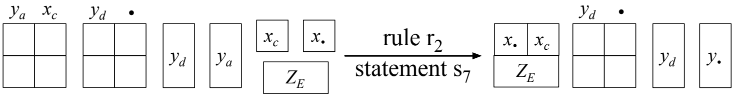


Fig. 4. Patterns of preferences. The shaded areas are not part of the notation; they only emphasize where the preferred pieces should stay.

rule succeeds, then the statement fails. The solution of a puzzle is found by successive applications of statements on empty board areas, as follows:

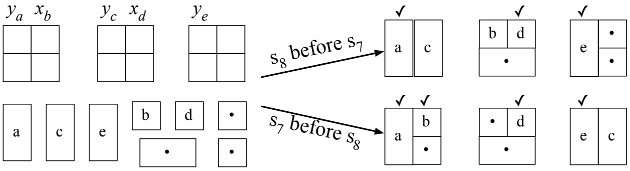
```
For each $i$ from 1 to 8:
    For each empty area $a$ such that the pattern of $s_i$ matches $a$:
        – apply $s_i$ to $a$
        – if the application of $s_i$ to $a$ fails,
          then terminate the entire execution and report failure.
```

If the preferred area of a piece v is filled with a piece other than v, and v is still available to fill other areas, we remove the name of v and mark it as an anonymous piece. We illustrate this step in the figure below, which uses puzzle five from Figure 1 as an example:



The piece $x_\bullet$ was added to pad the puzzle. This example shows the application of the second rule of statement seven of our solving algorithm. After the application, the piece y_a can no longer be allocated into its preferred spot, so we relabel it to an anonymous piece $y_\bullet$.

The algorithm in Figure 5 determines an order in which areas must be filled with pieces. Part of the ordering that we chose is arbitrary, e.g, any ordering between statements one to seven would preserve the optimality of the solution. However, some choices are essential to guarantee the optimal solution of punctual coalescing. For instance, the figure below illustrates a case in which we get more copies if we switch the precedence between statements seven and eight:



Similarly, the figure below illustrates a case in which we get worse results if we invert the order of rules inside statement five:

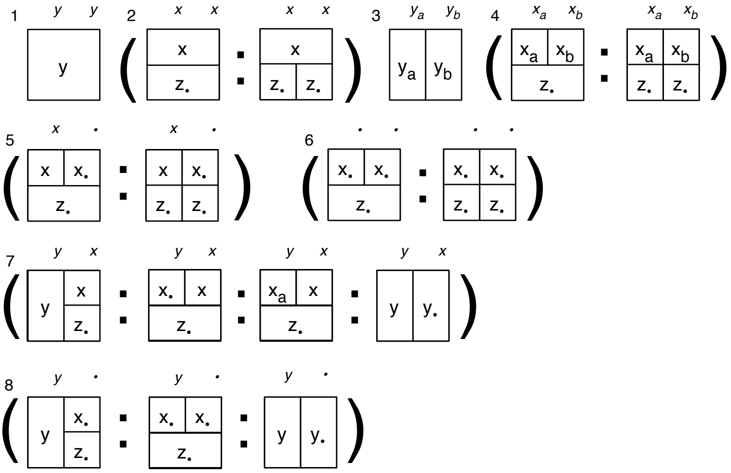
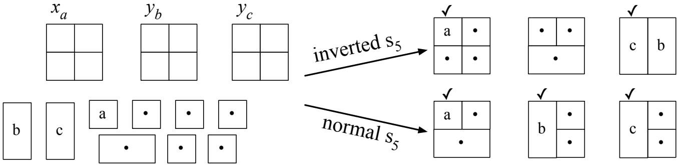


Fig. 5. Program P_c that solves punctual coalescing for empty follower boards



Correctness. We have proven that the algorithm in Figure 5 solves a puzzle with an initially empty board, if, and only if, that puzzle has solution [24, ch.4]. This result comes from the combination of two lemmas: progress (Lemma 1) and preservation (Lemma 2). In particular, we give a mechanical proof of Lemma 2 using the Twelf Meta Theorem prover [28].

Lemma 1. ***(Progress)*** *If P is a solvable puzzle, then there is a rule r in the algorithm from Figure 5 that applies to P.*

Lemma 2. ***(Preservation)*** *If P is a solvable puzzle, and the algorithm from Figure 5 applies rule r to P to produce P', then P' is solvable.*

We also show the optimality of our solution, which we state as Theorem 1 below. To state optimality we need to define the number of displaced pieces. The number n of pieces displaced in a solution of a type-1 puzzle, as found by the algorithm in Figure 5, is determined uniquely by the types of patterns and the number of size 2 Z pieces in the puzzle. There are eight different patterns, shown in Figure 4. We let Z_2 be the number of size 2 Z pieces, and we let $P_i, i \in \{a, \ldots, h\}$ be the number of patterns i in the puzzle board. The algorithm to compute n is given below:

```
– let $n_d = Z_2 - (P_b + P_d + P_e + P_f)$
– if $n_d \leq 0$
    • then $n \leftarrow 0$
    • else if $P_h \geq n_d$
        ∗ then $n \leftarrow n_d$
        ∗ else $n \leftarrow P_h + 2 \times (n_d - P_h)$
```

Theorem 1. ***(Optimality)*** *If P is solvable with n displaced pieces, and rule r is applied on P producing P' and causing k displaced pieces, then P' is solvable with at most $n - k$ displaced pieces.*

The proofs of progress, preservation and optimality are given in [24, ch.4].

4 ILP Formulation

We use a 0/1 integer linear programming (ILP) formulation to find a solution to the global coalescing problem. Our ILP model uses three sets: puzzle areas R, puzzle pieces V and a set N of puzzles with one element for each split point in the source program. The set R contains $3m$ elements, where m is the number of columns in the puzzle board. We assume that, for all $i, 0 \leq i \leq m$, areas $2i$ and $2i + 1$ alias area $i + 2m$. Figure 6 gives an example. In this case we have two puzzle areas, labeled four and five. Area four is divided into columns zero and one, and area five is divided into columns two and three. We define binary variables p_{nvr} ranging on these three sets. Each p_{nvr} is 1 if piece p has been allocated to the area r of the puzzle n, and is 0 otherwise. Notice that p_{nvr} only exists if the piece v has the same width as area r. For instance, in Figure 6, piece a of puzzle five produces the variables $p_{5a0}, p_{5a1}, p_{5a2}$ and p_{5a3}, but not p_{5a4}, because piece a has width one, and area four has width two.

Following Grund *et al.* [16], we define *affinity variables*. The affinity variable a_{ijvr} is 0 if the puzzle pieces p_{ivr} and p_{jvr} have the same value. This happens when the pieces representing variable v have been assigned to the same puzzle area r across two consecutive puzzles i and j. Affinity variables model the control flow graph of the source program. Thus, due to affinity edges, our ILP model finds an optimal solution to register coalescing for the whole program, and not only for a single program block. The objective function consists in minimizing the sum of the affinity variables:

$$\min f = \sum_{i,j,v,r} a_{ijvr}$$

Our formulation uses three basic types of constraints:

1. Each puzzle piece must be allocated to just one area. That is, given piece v at puzzle n, for each area r with the same width as v we have that:

$$\sum_{r} p_{nvr} = 1$$

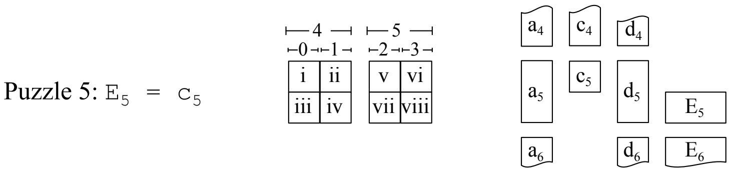


Fig. 6. Puzzle five from Figure 1

i	$p_{5a0} + p_{5c0} + p_{5d0} \leq 1$	v	$p_{5a2} + p_{5c2} + p_{5d2} \leq 1$
ii	$p_{5a1} + p_{5c1} + p_{5d1} \leq 1$	vi	$p_{5a3} + p_{5c3} + p_{5d3} \leq 1$
iii	$p_{5a0} + p_{5d0} + p_{5E4} \leq 1$	vii	$p_{5a2} + p_{5d2} + p_{5E5} \leq 1$
iv	$p_{5a1} + p_{5d1} + p_{5E4} \leq 1$	viii	$p_{5a2} + p_{5d2} + p_{5E5} \leq 1$

Fig. 7. Constraints asserting that a puzzle area can contain only one piece

2. Each puzzle area must contain at most one piece. That is, given an area, we define four inequalities, one for each region where a piece can be placed. For all p_{nvr} that can be placed on the same region, and all $0 \leq i \leq m$, we have the equations below, where the double summation is due to the double aliasing of T1 puzzles:

$$\sum_v p_{nv(2i)} + \sum_v p_{nv(2m+i)} \leq 1 \text{ and also } \sum_v p_{nv(2i+1)} + \sum_v p_{nv(2m+i)} \leq 1$$

3. Each affinity edge a_{ijvr} must be greater than or equal the absolute value of $p_{ivr} - p_{jvr}$.

4.1 Example

As an example, we model the constraints that are produced by the puzzle in Figure 6, i.e, puzzle five from Figure 1. We have numbered the puzzle areas using roman numerals to help our explanation. Also, we have added indices to the variable names, to distinguish those that are part of puzzle five from those that are part of other puzzles. For each of the four quadrants of an area we have a constraint that forces the piece stored in that location to be unique. These constraints are given in Figure 7. Notice that the constraint that refers to an area uses only the variables that may be allocated in that area. In this way, the constraint of area i mentions only pieces a_5, c_5 and d_5.

Figure 8 shows the constraints used to guarantee that each piece will receive a puzzle area. There are four such constraints, one for each variable.

Finally, the affinity edges add 44 equations to our model. These equations are described by the expressions below:

$$\forall(r \in \{0,1,2,3\}, v \in \{a,c,d\}), f_{45vr} \geq p_{4vr} - p_{5vr} \text{ and also } f_{45vr} \geq p_{5vr} - p_{4vr}$$

$$\forall(r \in \{0,1,2,3\}, v \in \{a,d\}), f_{56vr} \geq p_{5vr} - p_{6vr} \text{ and also } f_{56vr} \geq p_{6vr} - p_{5vr}$$

$$\forall(r \in \{4,5\}), f_{56Er} \geq p_{5Er} - p_{6Er} \text{ and also } f_{56Er} \geq p_{6Er} - p_{5Er}$$

a	$p_{5a0} + p_{5a1} + p_{5a2} + p_{5a3} = 1$	d	$p_{5d0} + p_{5d1} + p_{5d2} + p_{5d3} = 1$
c	$p_{5c0} + p_{5c1} + p_{5c2} + p_{5c3} = 1$	E	$p_{5E4} + p_{5E5} = 1$

Fig. 8. Constraints asserting that a piece must be placed on only one area

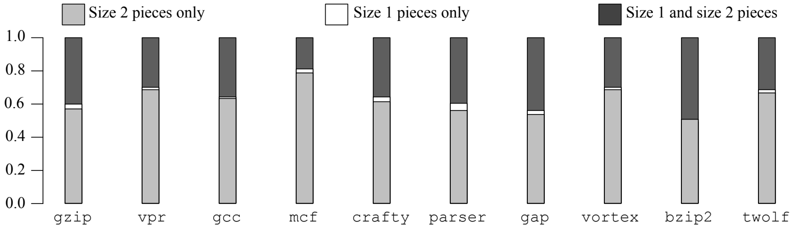
5 Experimental Results

This section empirically validates our punctual coalescing approach. In order to ensure reproducibility, the material used in these experiments is available at `http://homepages.dcc.ufmg.br/~fpereira/projects/puzzles/punctual/`.

Punctual Coalescing in x86. We have implemented our punctual coalescing algorithm on top of the original puzzle solver [26], running on LLVM 2.2 [20]. When compiling SPEC CPU 2000, our implementation is 4% slower than LLVM's default register allocator, an extended version of linear scan [29]. We emphasize that our implementation is a research artifact, whereas LLVM's is an industrial quality software that does not convert the input program into elementary form.

In terms of number of copies, results are very good: no copy was required between two consecutive puzzles in which the follower had an empty puzzle board during the compilation of SPEC CPU 2000. These puzzles account for 89% of the instructions in the source programs. The puzzle solver inserted approximately one copy per each group of 14 puzzles; however, these copies were used to implement fixing code between basic blocks (63% of copies), and to avoid conflicts between program variables and pre-allocated registers (37% of copies); we discuss these issues in Section 6. These results mean that we have not found a pattern such as that in Figure 1 in our benchmarks. However, x86 is an "easy" target for punctual coalescing, because it contains only four aliased registers (`AX, BX, CX` and `DX`). Moreover 67% of the puzzles that we found contain only pieces of the same size, in which case it is possible to find a solution for punctual coalescing requiring zero copies [24, ch.4]. Thus, to verify the behavior of our algorithm in a larger puzzle board and with more diverse inputs, we tested it in an artificial architecture, as we describe in the next section.

Punctual versus Global Coalescing. We have seen, in Section 2, that a sequence of optimal solutions to punctual coalescing may be worse than an optimal solution to global coalescing, even for straight line programs. The objective of this section is to measure this difference and to compare the punctual coalescer with other polynomial-time algorithms. In these experiments, we use LLVM [20] to compile SPEC CPU 2000 to an artificial architecture. LLVM uses a typed intermediate representation, in which integer values have a well known bit width: 1, 8, 16 or 32 bits. We assume a T1 architecture with 32-bit registers, each of them divided into two 16-bit aliases. A register may contain one 32-bit value, or two 1, 8, or 16-bit values. LLVM's IR does not use any form of pre-allocated registers; thus, all the puzzle instances produced have an empty register board.



	gzip	vpr	gcc	mcf	crafty	parser	gap	vortex	bzip2	twolf
#traces	97	297	2182	26	79	321	833	922	73	190
#puzzles	3954	12907	87062	1412	8172	9257	47920	38108	2596	15401
longest trace	201	224	608	238	728	255	480	320	188	420

Fig. 9. Puzzle distribution obtained from LLVM's intermediate representation. **#traces**: the total number of traces produced. **#puzzles**: the total number of puzzles produced. **longest trace**: size of longest trace, in number of puzzles.

The nature of the data produced. We use program traces in these experiments, as they are small enough for our ILP solver to handle. A trace is a set of instructions that are executed in sequence. We build traces by concatenating successive basic blocks. For each function in SPEC CPU 2000, we compile the longest trace that we obtain given a depth first traversal of the function's control flow graph. Our longest trace, taken from `186.crafty`, contains 728 puzzles. For each trace, we assume that our target architecture contains exactly the minimal number of registers necessary to compile all its puzzles. This number, called *T1 register pressure*, has a simple formula for puzzles with initially empty boards. In the formula below, Y is the number of size two Y pieces, and y is the number of size one Y pieces; similar notation applies to Z, z, X and x:

$$\text{T1 register pressure} = \lceil (2Y + y + \max((2X + x), (2Z + z)))/2 \rceil \qquad (1)$$

By equaling available registers and register pressure, we ensure that an optimal allocator can find a register assignment without causing spills. Spilling plays no role in the experiments, because the four register assignment algorithms that we compare fit the model explained in Section 2, which decouples register assignment from register spilling [6].

Given the scenario previously described, we obtained the puzzle distribution detailed in Figure 9. We have produced 5,020 traces from the ten integer SPEC CPU 2000 programs that LLVM is able to compile in our system. Together, these traces contain 226,789 puzzles. We distinguish three groups of puzzles: (i) those with all the pieces having size two, (ii) those with all the pieces having size one and (iii) those having pieces of both sizes. We notice that size one pieces are rare: puzzles of group (ii) correspond to less than 2% of all the puzzles, and over 60% of our puzzles are in group (i), thus containing only size two pieces. This discrepancy is due to most C programmers seldom using the `char` and `short` data types, recurring instead to `int`, even to represent boolean variables.

The Competing Coalescers. We compare four register assignment algorithms. Two of them are the punctual coalescer of Section 3 and the ILP formulation of Section 4. The other two algorithms are polynomial-time register assignment heuristics: a coalescing oblivious allocator based on the coloring of chordal graphs [25], and the register assignment heuristics used in the original puzzle based allocator [26]. The ILP algorithm uses CPLEX, the two punctual approaches – the optimal and the heuristic – are implemented in C++, and the chordal based allocator is written in Java.

Register allocation via coloring of chordal graphs follows from the fact that programs in *static single assignment* (SSA) [13] form have chordal interference graphs, and thus, can be optimally colored in polynomial time [5,10,18]. This property also applies to elementary programs, which are in SSA form [26]. In this experiments, we use the register allocator introduced by Pereira and Palsberg [25]. This chordal allocator is not guaranteed to deliver optimal results in the presence of aliasing. If we fail to find an allocation with n registers, where n is the $T1$ register pressure of the input program, then we re-run the algorithm with $n + 1$ registers. None of our traces has caused such an iteration.

We have included the chordal based approach in these experiments to show how bad a coalescing oblivious algorithm can do compared to an optimal allocator. There exists effective coalescing heuristics for chordal based allocators. Good examples are given by Bouchez *et al.* [7] and Hack *et al.* [17]. However, we do not use these sophisticated coalescing methods. Instead, after color assignment is performed, we use a very simple coalescing heuristics. If we let $G = (V, E, A)$ be an interference graph with a set V of vertices, a set E of interference edges, and a set A of affinity edges, our heuristics is:

```
$\forall$ affinity edge $(u, v) \in A$ such that $(u, v) \notin E$
    if $\exists$ color $c$ such that $c$ is not assigned to any neighbor of $u$ or $v$,
        assign $c$ to $u$ and $v$
```

The original puzzle solving heuristics [26] was the inspiration for the punctual algorithm described in Section 3. The original placement rules are shown in Figure 10. The main difference between this program and the program shown in Figure 5 is the arbitrary choice of pieces for areas without preferences. In Figure 5 we use $p_\bullet$ to denote a piece that has no preference for any area, and we use p_a to denote a piece that has preference for a given area a. In Figure 10 we write $p_?$ to indicate that we do not take the preference of piece p into consideration when choosing an area to place it.

Results for SPEC CPU 2000 traces. Figure 11 compares the number of copies inserted by the coalescing algorithms. The ILP solver did not finish running on four traces, given a two hours time limit. In total we run the CPLEX solver for 5+ days in order to find solutions to all the traces. In contrast the punctual coalescer, implemented in C++, took 33 seconds to find a register allocation for all the traces, and the original heuristics took 30 seconds. The chordal based algorithm runs for 6+ hours; however, we point that this is a Java program implemented with no concern for fast running time. For any practical purposes, the ILP and the two punctual approaches generate a very small number of copies, hence causing negligible increase in code size. Furthermore, for straight line programs,

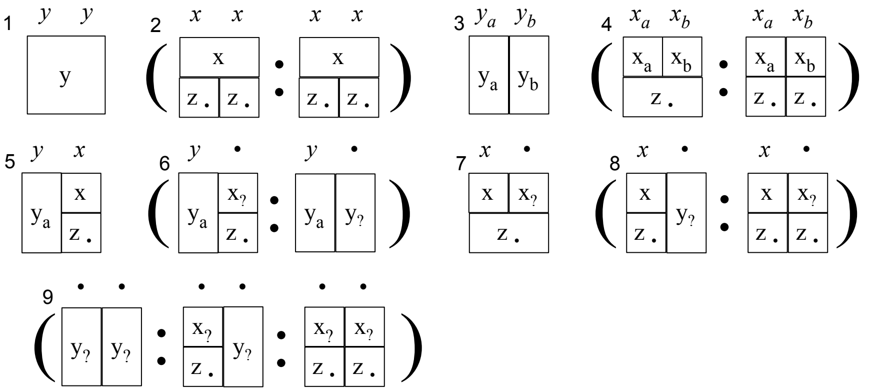


Fig. 10. The puzzle solving program for empty boards used by Pereira and Palsberg [26]

Benchmark	gzip	vpr	gcc	mcf	crafty	parser	gap	vortex	bzip2	twolf
Chordal	13,471	47,677	329,783	4,757	46,182	27,082	174,633	199,355	10,581	101,816
Original	13	32	241	1	21	19	135	79	12	44
Punctual	0	10	17	0	1	5	33	1	0	0
ILP	0	2	4	0	0	0	3	0	0	0

Fig. 11. Number of copies inserted by: (chordal) the coalescing oblivious register allocator via coloring of chordal graphs. (original) the coalescing heuristics used in the original puzzle solver [26], (punctual) the algorithm from Section 3, (ILP) the ILP formulation from section 4.

the optimal punctual approach delivers results that are very close to the ILP method. For instance, our punctual coalescing algorithm required 17 copies to solve the 87,000 puzzles of gcc. This is less than one copy per 5,000 puzzles! Only the punctual algorithms – optimal and heuristic – are implemented in LLVM, an there is no runtime performance difference between them. Based on the results of Hack and Goos [17], we speculate that there will be no measurable differences among the four algorithms when targeting x86.

The influence of variable widths on the performance of punctual coalescing. We have observed that the width of the variables found in the traces plays an important role on the quality of the solution produced by punctual coalescing. The width of a variable determines if it fits in half a register, or if it demands a full register. In order to support this observation, we define two types of register pressures: T0 and T1. The T1 register pressure is computed by Equation 1. The T0 register pressure is the register pressure computed assuming a register bank without aliasing, and it is calculated by Equation 2, where X, Y, Z, x, y and z are defined as in Equation 1.

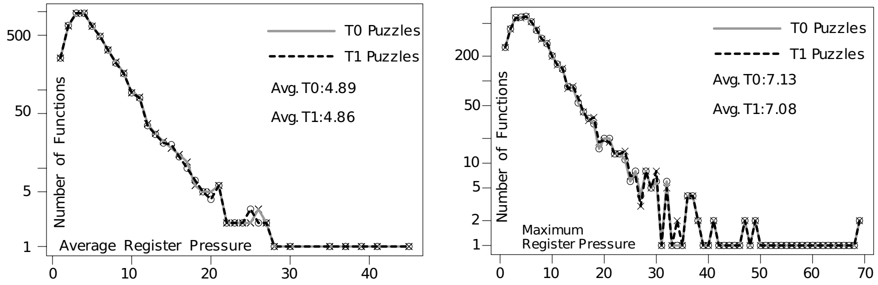


Fig. 12. (Left) Histogram of average register pressures. (Right) Histogram of maximum register pressures.

$$\text{T1 register pressure} = Y + y + \max((X + x), (Z + z)) \tag{2}$$

For instance, in the example of Figure 1, the average T1 register pressure is 1.83, and the maximum T1 register pressure is 2. On the other hand, the average T0 register pressure is 2.5, and the maximum T0 register pressure is 3. Figure 12 gives a histogram in which the traces produced from SPEC CPU 2000 are grouped according to the T0 and T1 register pressures. Both numbers are very similar in our benchmarks. On the average, each of our puzzles could be solved with 7.13 registers, assuming no aliasing, and with 7.08, given T1 aliasing. Furthermore, 95.2% of all the traces could be compiled with 16 registers of type T0, whereas 95.4% of the functions could be compiled assuming a T1 target architecture. These numbers are similar because programmers tend to use 32 bit types such as `int` instead of smaller types.

Punctual coalescing tends to produce better results when the T1 pressure is close to the T0 pressure. The intuition behind this fact is simple: for T0 puzzles, if the number of registers is greater than or equal to the maximum register pressure in the trace, then there is a register assignment that requires no copy, and the punctual coalescing strategy discussed in Section 3 trivially finds it. As an illustration, we have inverted the proportion of size one and size two variables

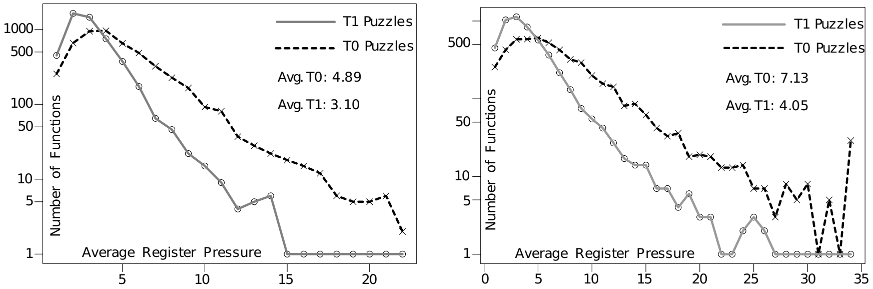


Fig. 13. Histograms obtained by inverting the proportion of size one and size two pieces in our benchmarks. (Left) average register pressures. (Right) maximum register pressures.

Benchmark	gzip	vpr	gcc	mcf	crafty	parser	gap	vortex	bzip2	twolf
Chordal	15,160	48,878	337,608	4,746	55,468	26,894	176,097	204,325	10,581	101,747
Original	282	1,025	5,554	91	683	486	2,128	3,686	182	1,456
Punctual	25	108	516	20	29	47	251	288	13	111
ILP	0	2	39	0	1	9	21	0	2	8

Fig. 14. Number of copies inserted by different allocators compiling the traces from Figure 13

presented in Figure 9, obtaining the histograms in Figure 13. In this artificial setting, we have more size one than size two variables, resulting in a conspicuous difference between the T1 and T2 register pressures. The results of global and punctual coalescing in this new context are given in Figure 14. Our punctual technique inserts 20 times more copies than before; however, this number is still negligible given the amount of puzzles solved: one copy per each 160 puzzles.

6 Limitations of Punctual Coalescing

The punctual coalescing algorithm of Section 3 may not give optimal results in two situations: settings with two or more guiding puzzles, and settings with non-empty follower boards.

Two or more guiding puzzles stems from a merge in the control-flow graph of the input program. Figure 15 shows an example. The program in Figure 15(a) contains four basic blocks. Three of these blocks – L_1, L_2 and L_4 – form the program trace in Figure 1. If our punctual coalescer traverses this trace first, then it will produce one copy instruction, moving variable a from register R_0 into register R_3, as seen in Figure 1, and shown again in Figure 15(b). However, when performing register assignment in the trace formed by basic block L_3, our coalescer will not take into consideration the mapping of variables to registers

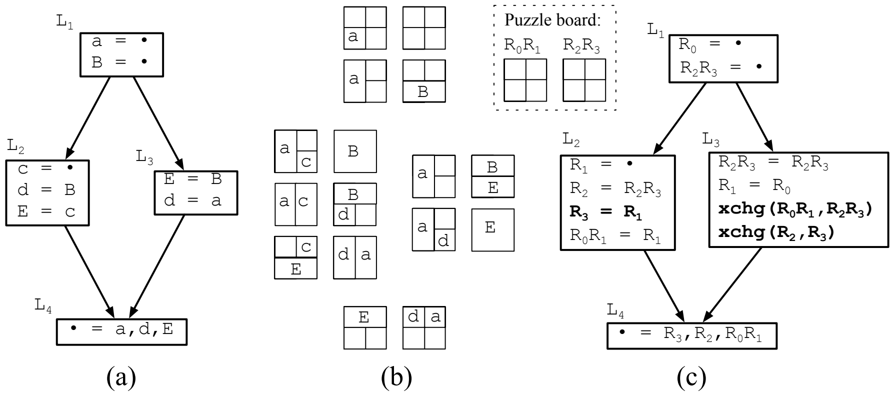


Fig. 15. The complete example, from puzzle solving to code generation

in block L_4, previously visited. Thus, it may be necessary to insert fixing code between basic blocks L_3 and L_4. The insertion of this code is analogous to SSA elimination after register allocation, and there are standard algorithms to perform it [27]. Figure 15(c) shows the final assembly program produced; fixing code is shown in bold face. We borrowed the `xchg` instruction, that swaps the contents of two registers, from the x86 lexicon.

The problem of maximizing coalescing in a setting with two or more guiding puzzles is NP-complete. The reduction is from the Global Pinning problem, defined by Rastello *et al.* [30]. However, we have observed that in practice, at least in the x86 architecture, the punctual coalescer produces good results: SSA elimination after register allocation adds approximately 5% more instructions to the final assembly program, and has negligible impact on the run time of compiled programs [26]. As a future work, we will couple register assignment with the static branch prediction technique of Ball and Larus [3] to increase the likelihood that our puzzle solver will traverse hot program paths first.

Non-empty follower boards stem from constraints in the target architecture's instruction set. For instance, x86's `div` instruction always produces a result in register `AX`. Thus, the puzzle board created for a `div` instruction contains the area that corresponds to `AX` initially taken. Punctual coalescing is not guaranteed to deliver optimal results if the follower board contains pre-allocated pieces. Pre-assignment may take away the preferred spot of Y and X pieces. When faced with pre-allocation we use the original puzzle solving algorithm [26] to eliminate areas containing pre-assigned pieces, and then apply the punctual coalescing program from Figure 5 on the remaining areas. In the x86 experiments, move and swap instructions due to pre-coloring increased the final assembly program in about 2%. Optimal punctual coalescing in face of pre-assignment is an open-problem.

7 Conclusion

This paper has presented punctual coalescing, a technique for reducing the number of copy instructions inserted by tree-scan register allocators that rely on live range splitting to lower register pressure. In addition, this paper gave an optimal solution to global coalescing in register banks with aliasing. A comparison between these two techniques showed that the linear time punctual approach is very close to the exponential time global algorithm for straight line programs. We are currently adapting our punctual algorithm to run on a trace compiler[14].

References

1. Ahn, M., Lee, J., Paek, Y.: Optimistic coalescing for heterogeneous register architectures. SIGPLAN Notices 42(7), 93–102 (2007)
2. Appel, A.W., George, L.: Optimal spilling for CISC machines with few registers. In: PLDI, pp. 243–253. ACM, New York (2001)
3. Ball, T., Larus, J.R.: Branch prediction for free. In: PLDI, pp. 300–313. ACM, New York (1993)

4. Biró, M., Hujter, M., Tuza, Z.: Precoloring extension. I. interval graphs. Discrete Mathematics 100(1-3), 267–279 (1992)
5. Bouchez, F.: Allocation de registres et vidage en mémoire. Master's thesis, ENS Lyon (October 2005)
6. Bouchez, F.: A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases. PhD thesis, ENS Lyon (2008)
7. Bouchez, F., Darte, A., Rastello, F.: Advanced conservative and optimistic register coalescing. In: CASES, pp. 147–156. ACM, New York (2008)
8. Braun, M., Hack, S.: Register spilling and live-range splitting for SSA-form programs. In: de Moor, O., Schwartzbach, M.I. (eds.) CC 2009. LNCS, vol. 5501, pp. 174–189. Springer, Heidelberg (2009)
9. Briggs, P., Cooper, K.D., Torczon, L.: Improvements to graph coloring register allocation. TOPLAS 16(3), 428–455 (1994)
10. Brisk, P., Dabiri, F., Jafari, R., Sarrafzadeh, M.: Optimal register sharing for high-level synthesis of SSA form programs. TCAD 25(5), 772–779 (2006)
11. Chaitin, G.J.: Register allocation and spilling via graph coloring. In: Symposium on Compiler Construction, vol. 17(6), pp. 98–105 (1982)
12. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register allocation via coloring. Computer Languages 6, 47–57 (1981)
13. 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)
14. Gal, A., Eich, B., Shaver, M., Anderson, D., Kaplan, B., Hoare, G., Mandelin, D., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E., Reitmair, 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. George, L., Appel, A.W.: Iterated register coalescing. Transactions on Programming Languages and Systems (TOPLAS) 18(3), 300–324 (1996)
16. Grund, D., Hack, S.: A fast cutting-plane algorithm for optimal coalescing. In: Krishnamurthi, S., Odersky, M. (eds.) CC 2007. LNCS, vol. 4420, pp. 111–125. Springer, Heidelberg (2007)
17. Hack, S., Goos, G.: Copy coalescing by graph recoloring. In: PLDI, pp. 227–237. ACM, New York (2008)
18. 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)
19. Kong, T., Wilken, K.D.: Precise register allocation for irregular architectures. In: MICRO, pp. 297–307. IEEE, Los Alamitos (1998)
20. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO, pp. 75–88. IEEE, Los Alamitos (2004)
21. Lee, J.K., Palsberg, J., Pereira, F.M.Q.: Aliased register allocation. Theoretical Computer Science 407(1-3), 258–273 (2008)
22. Nandivada, V.K., Pereira, F., Palsberg, J.: A framework for end-to-end verification and evaluation of register allocators. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 153–169. Springer, Heidelberg (2007)
23. Park, J., Moon, S.-M.: Optimistic register coalescing. In: IEEE PACT, pp. 196–204 (1998)
24. Pereira, F.M.Q.: Register Allocation by Puzzle Solving. PhD thesis, University of California, Los Angeles (2008)

25. Pereira, F.M.Q., Palsberg, J.: Register allocation via coloring of chordal graphs. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 315–329. Springer, Heidelberg (2005)
26. Pereira, F.M.Q., Palsberg, J.: Register allocation by puzzle solving. In: PLDI, pp. 216–226. ACM, New York (2008)
27. Pereira, F.M.Q., Palsberg, J.: SSA elimination after register allocation. In: de Moor, O., Schwartzbach, M.I. (eds.) CC 2009. LNCS, vol. 5501, pp. 158–173. Springer, Heidelberg (2009)
28. Pfenning, F., Schürmann, C.: Twelf - a meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)
29. Poletto, M., Sarkar, V.: Linear scan register allocation. TOPLAS 21(5), 895–913 (1999)
30. Rastello, F., de Ferriére, F., Guillon, C.: Optimizing translation out of SSA using renaming constraints. Technical Report 03-35, École Normale Supérieure de Lyon (2003)
31. Sarkar, V., Barik, R.: Extended linear scan: an alternate foundation for global register allocation. In: LCTES/CC, pp. 141–155. ACM, New York (2007)
32. Scholz, B., Eckstein, E.: Register allocation for irregular architectures. In: LCTES/SCOPES, pp. 139–148. ACM, New York (2002)
33. Smith, M.D., Ramsey, N., Holloway, G.: A generalized algorithm for graph-coloring register allocation. In: PLDI, pp. 277–288. ACM, New York (2004)
34. Traub, O., Holloway, G.H., Smith, M.D.: Quality and speed in linear-scan register allocation. In: PLDI, pp. 142–151. ACM, New York (1998)

Strategies for Predicate-Aware Register Allocation

Gerolf F. Hoflehner

Intel Corporation
2200 Mission College Blvd
Santa Clara, CA 95054
gerolf.f.hoflehner@intel.com

Abstract. For predicated code a number of predicate analysis systems have been developed like PHG, PQA or PAS. In optimizing compilers for (fully) predicated architectures like the Itanium® 2 processor, the primary application for such systems is global register allocation. This paper classifies predicated live ranges into four types, develops strategies based on classical dataflow analysis to allocate register candidates for all classes efficiently, and shows that the simplest strategy can achieve the performance potential provided by a PQS-based implementation. The gain achieved in the Intel® production compiler for the CINT2006 integer benchmarks is up to 37.6% and 4.48% in the geomean.

Keywords: Register Allocation, Predication, Compiler, Itanium processor, EPIC, PQS.

1 Introduction

Register allocation solves the decision problem which symbolic register ("candidate") should reside in a machine register. A symbolic register represents a user variable or a temporary in a compiler internal program representation. Register assignment solves the decision problem which specific machine registers to assign a given symbolic register. Solutions of both problems must take into account constraints between symbolic registers. A coloring allocator abstracts the allocation problem to coloring an undirected interference graph with K colors, which represent K machine registers. Then a coloring is a mapping of a large number of symbolic register candidates to a small, finite set of physical registers. Chaitin describes - in "broad brush strokes" - the fundamental building blocks of coloring allocators [6]. Eminent is the interference graph that encodes the information when two symbolic registers *cannot* be assigned the same physical register. In this case, they are said to *interfere*. A node in this undirected graph is a live range, which represents a candidate and the program points at which the candidate could be allocated a register. A live range is typically modeled by the outcome of two dataflow algorithms: a backward live variable analysis and a forward available variable (or reaching definition) analysis. The live range consists of all program points where the symbolic register is both *live* and *available*. To allocate as many symbolic registers to machine registers as possible the allocator must determine the start of a live range and its interferences precisely. Both problems are harder to solve on predicated architectures like the Itanium processor ("IA-64").

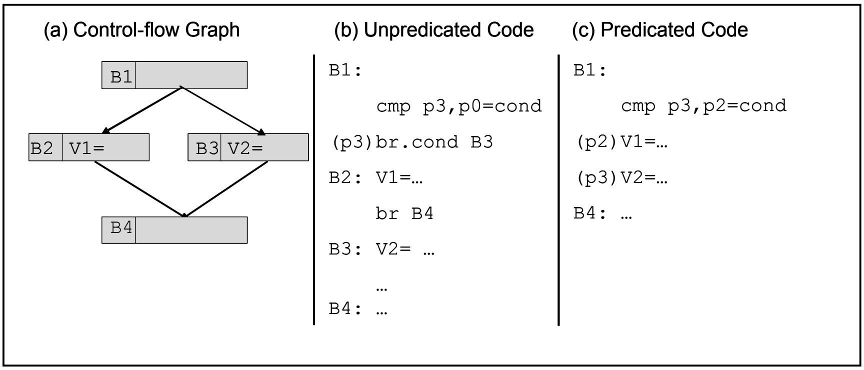
R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 185–204, 2010.

1.1 Predication

Predication is the conditional execution of an instruction guarded by a qualifying predicate. For example, on IA-64 the qualifying predicate is a binary ("predicate") register that holds a value of 1 (=True) or 0 (=False). The (qualifying) predicate register is encoded in the instruction. When its value is 1 at run-time, the predicate is set. When the value is 0 at run-time, the predicate is clear. On IA-64 almost all instructions are predicated. As (almost) fully predicated architecture IA-64 supports if-conversion. If-conversion is a compiler optimization that can eliminate conditional forward branches and their potential branch mis-prediction penalty (Fig. 1). The instructions dependent on the branch are predicated up to a merge point in the original control-flow graph. This eliminates the conditional branch and converts control dependencies (instructions dependent on the branch) into data dependencies (between qualifying instruction predicates) (Allen et al. [2]). As a result if-conversion transforms a control-flow region into a linear ("predicated") code region ("hyperblock"). The paths in the control-flow region become execution traces in the predicated code. In the predicated region all paths of the original region overlap and predicated instructions make it harder for the register allocator to find the start of a live range and determine its precise interferences.

1.2 Overview

The rest of the paper is structured as follows: section 2 gives the background on coloring allocators and the Itanium architecture. Section 3 presents register allocation for predicated code. This section is the core of the paper and presents four classes of predicated live ranges, two methods of precise live tracking interference tracking, and three implementation strategies. Section 4 discusses measurement setup and results. Section 5 reviews related work. Section 6 has conclusions.



```
B1:
    cmp p3,p0=cond
(p3)br.cond B3
B2: V1=…
    br B4
B3: V2= …
    …
B4: …
```

```
B1:
    cmp p3,p2=cond
(p2)V1=…
(p3)V2=…
B4: …
```

Fig. 1. Example with control-flow graph, non-predicated and predicated code

2 Background

This section gives the background on basic coloring allocator and the Itanium architecture.

2.1 Chaitin-Style Register Allocation

A Chaitin-style graph-coloring algorithm has six phases (Fig. 2): “renumber”, “build”, “coalesce”, “simplify”, “spill” and “select”. At the start of the algorithm each symbolic register corresponds to a single register candidate node (“renaming”). This phase may split disjoint definition-use chains of a single variable into multiple disjoint candidates. It also ensures contiguous numbering of candidates which reduces memory requirements for dataflow-analysis and interference graph. Node interference relies on dataflow analysis to determine the live range of a node. The live range of a node consists of all program points where the candidate is both live and available. Dataflow analysis is necessary only once, not at each build step. The “build” phase constructs the interference graph. The nodes in the interference graph represent register candidates. Two nodes are connected by an interference edge when they cannot be assigned the same register. This is the case when their live ranges intersect. The number of edges incident with a node is the degree of the node. Building the interference graph is a two pass algorithm. In the first pass, starting with the live out information, node interference is determined by a backward sweep over the instructions in each basic block. Interference is a symmetric relation stored in a triangular matrix. This is usually a large, sparse bit matrix inadequate for querying the neighbors of a given node. To remedy this for each node an adjacency vector is allocated in a second pass. The length of the vector is the degree of the node, and the vector lists all neighbors of the node.

The next phase, “coalescing” (aka “subsumption”, “node fusion”), is an optimization that is not needed for solving the register allocation problem, but is part of the original Chaitin allocator. It fuses the source and destination node of a move instruction when the nodes do not interfere. This reduces the size of the interference graph

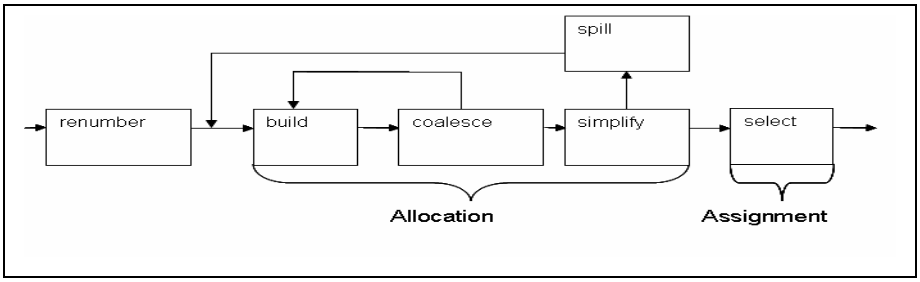


Fig. 2. Chaitin’s model for a coloring allocator

and eliminates the move instruction, since source and destination get assigned the same register. In Chaitin's original implementation any possible pair of nodes is coalesced. This form of coalescing is called "aggressive coalescing". After possibly several iterations of "coalesce" and (re-) "build", the simplification phase iterates over the nodes in the interference graph using simple graph theory to filter candidates that can be certainly allocated to machine registers: when a register candidate has fewer than K interference edges (a *low* degree node that has fewer than K neighbors), then it can always be assigned a color. Low degree nodes and their edges are removed from the graph ("simplify") and pushed on a stack ("coloring stack"). Node removal may produce new low degree nodes. When only *high* degree ("significant") nodes that have K or more neighbors are left simplification is in a blocking state. It transitions out of a blocking state using a heuristic based cost function that determines the "best node" to be removed from the graph. A node that is removed from the graph in blocking state is "spilled" and appended to a spill list. The edges of a spilled node are removed from the graph, so new low degree nodes can get exposed and simplification continues until all nodes have been pushed to the coloring stack or appended to the spill list. The cost function that decides on the "best node" estimates the execution time increase caused by spill code normalized by the degree of the node. The higher the degree the less likely a node will be allocated a register.

2.2 Itanium® 2 Processor Architecture

The Itanium processor family is a commercially available implementation of the EPIC ("Explicitly Parallel Instruction Computing") computing paradigm. In EPIC the compiler has the job of extracting instruction level parallelism (ILP) and communicating it to the processor. Instructions are grouped in fixed-size bundles. These are simple structures that contain three instructions and information about the functional unit each instruction must be issued. IA-64 is a 64bit computer architecture distinguished by a fully predicated instruction set, dynamic register stack, rotating registers and support for control- and data speculation. Predication and speculation allow the compiler to remove or break two instruction dependence barriers: branches and stores. Predicates enable the compiler to remove branches ("branch barrier removal"), control speculation allows it to hoist load instruction across branches ("breaking the branch barrier"), and data speculation makes it possible to hoist load instruction across stores ("breaking the store barrier"). Using predication and rotating registers the compiler can generate kernel-only pipelined loops. The dynamic register stack gives the compiler fine-grain control over register usage. In general exploiting instruction-level parallelism using Itanium features increases register pressure and poses new challenges for the register allocator. To support the EPIC paradigm the Itanium processor provides a large number of architected registers. Relevant for register allocation are the 128 general (integer) registers r0-r127, 128 floating point register f0-f127, 64 predicate registers p0-p63 and 8 branch registers b0-b7. The floating point and predicate register files contain rotating registers f32-f127 and p16-p63 respectively. Unique for Itanium are the 96 stacked integer registers, r32-r127, which are controlled by a special processor unit, the Register Stack Engine (RSE).

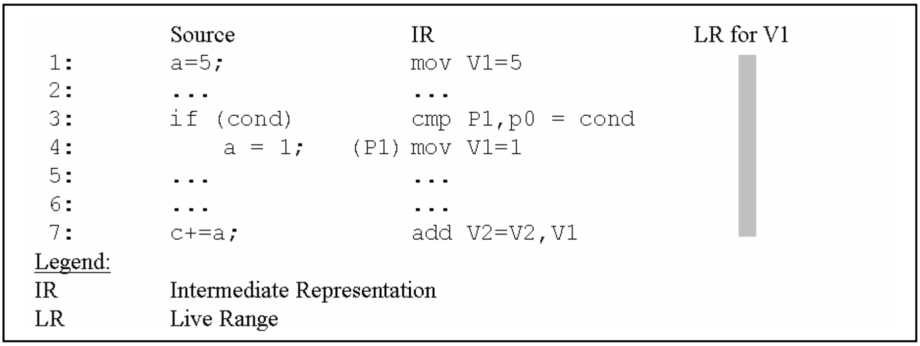
3 Register Allocation for Predicated Code

The IA-64 architecture is a fully predicated architecture with 64 predicate registers [10]. Each instruction (with some exceptions like the alloc instruction) is guarded by a qualifying predicate. A fully predicated architecture supports if-conversion, an optimization that can eliminate forward branches (Allen et al. [2]). If-conversion transforms a region of the control-flow graph to linear ("predicated") code. In this predicated region all execution paths of the original control-flow region overlap. The proper representation of a predicated region is a hyperblock, which is a predicated single entry multiple exit region. The compiler picks a single entry acyclic control-flow region as a candidate region for if-conversion. It may have multiple exits. Basic blocks where the paths originating from the single entry meet, are merge points and mark a potential end node for the region former. Typically the compiler has a threshold for the number of blocks in a region. The decision whether or not to if-convert a candidate region is driven by a predication oracle. It computes the estimated execution times of the predicated and control-flow version of the candidate region. When the estimated execution time for the predicated region is faster than the estimated execution time for the original control-flow regions, the candidate region is if-converted. The register allocator must handle predicated code formed from control-flow graph regions. This section investigates the impact of predicated code on the register allocator, discusses the predicate query system (PQS), classifies predicated live ranges and interference tracking, and presents a family of predicate-aware register allocators. The allocator based on "use-and-partition tracking" is equivalent to a PQS-based allocator, but simpler allocators are investigated as well.

3.1 Impact of Predicated Code

On a predicated architecture completeness and soundness of live variable and disjoint live range information are harder problems than for non-predicated code. For example, for live variables, completeness means at any program point where a variable is actually live, liveness computation reports it as live. Soundness means that at any point where the liveness computation reports a variable as live, it is actually live. The remainder of this section discusses live range extension, interference graph construction for predicated live ranges and global disjointness information.

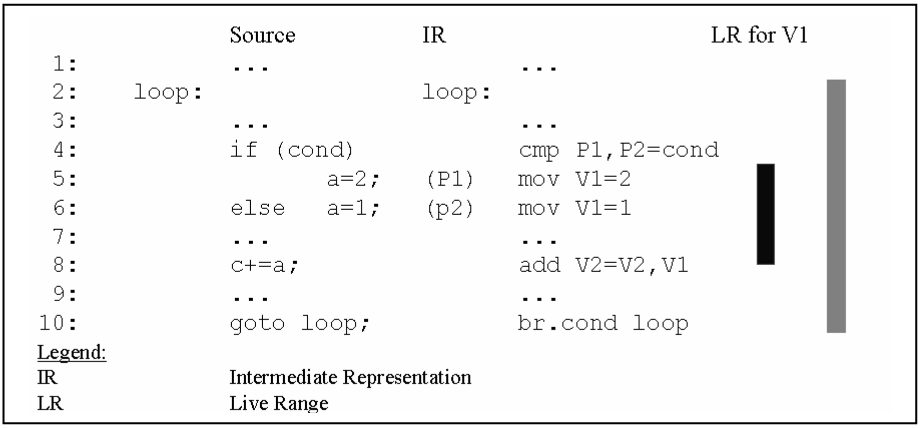
In non-predicated code a definition is the start of a live range. This is not necessarily true for predicated code. In Fig. 3 the predicated live range for virtual register V1, which corresponds to variable "a" in the source code, must span lines 1 to 7. This shows that a predicated definition of V1 (line 4) is not necessarily the start of the live range. Otherwise, the live range for V1 would extend incorrectly from line 4 to 7 in the predicated code.



```
        Source            IR                        LR for V1
  1:    a=5;              mov V1=5
  2:    ...               ...
  3:    if (cond)         cmp P1,p0 = cond
  4:        a = 1;   (P1) mov V1=1
  5:    ...               ...
  6:    ...               ...
  7:    c+=a;             add V2=V2,V1
Legend:
IR      Intermediate Representation
LR      Live Range
```

Fig. 3. Example with source code, predicated code and predicated live range. The bar at the right represents the actual live range of V1.

On the other hand, when no predicated definition is the start of a live range, predicated live ranges would extend to more program points than necessary increasing register pressure and consumption. All predicated live ranges would behave like live ranges of undefined or partially defined variables in non-predicated code. A variable is partially defined if there is (at least) one definition-free path from function entry to a use and (at least) another path that contains a definition. Depending on the run-time execution path the variable is defined or undefined. In cyclic code, the live range of a partially defined variable typically spans the entire loop nest that contains the definition. This is a result of the dataflow algorithms that determine a live range. Available variable analysis (forward) propagates the *is_available* property to every point in the loop nest. Live variable analysis (backwards) propagates the *is_live* property from the use to every point in the loop nest. Thus the live range, which consists of all program points where a variable is both available and live, spans the entire loop nest. This must be so since the variable could be defined in the first iteration and used in all subsequent iterations. Since the variable is live across the entire loop nest, it interferes with all variables in the loop. Therefore it will not be "destroyed" after being defined in the first iteration. In acyclic code, live range extension cannot occur because a live range cannot extend to a program point where it is not available. Fig. 4 illustrates live range extension in cyclic code for the predicated live range of V1: unless a predicated definition is recognized as the start of the live range for V1, it will extend across the entire loop nest (the large live range from line 2-10). The correct live range is the small live range from line 5, the first predicated definition of V1, to line 8, the use of V1. Live range extension for predicated code could also cause non-termination of a coloring allocator: When the allocator spills a predicated live range, it introduces one or more new predicated live ranges replacing the original. But the new live ranges share the same predicate. Due to live range extension the interferences in the next allocation round may actually increase. Since the new live ranges introduced for spilling are marked as non-spillable the allocator may no longer find spill candidates in the simplification phase. At this point the allocator would have to give up. So there is not only a potential run-time performance loss due to extra register pressure, but also a stability reason why a predicate-aware allocator must recognize the start of



```
                    Source               IR                          LR for V1
 1:                 ...                        ...
 2:     loop:                          loop:
 3:                 ...                        ...
 4:                 if (cond)                  cmp P1,P2=cond
 5:                         a=2;      (P1)     mov V1=2
 6:                 else    a=1;      (p2)     mov V1=1
 7:                 ...                        ...
 8:                 c+=a;                      add V2=V2,V1
 9:                 ...                        ...
10:                 goto loop;                 br.cond loop
Legend:
IR                  Intermediate Representation
LR                  Live Range
```

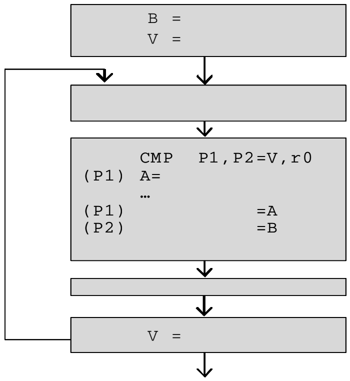
Fig. 4. Example for live range extension in predicated code. Short bar represents exact live range, while the long bar represents the extended live range of V1.

predicated live ranges. This impacts live range analysis (in particular, live variable analysis) and interference graph construction, which happens in the “build” phase of the allocator.

In non-predicated code interference is a function of “liveness”. In predicated code interference is a function of liveness and predicate disjointness. Disjoint live ranges do not interfere and can be assigned the same register. The allocator queries a Predicate Data Base (PDB) for disjointness information when it constructs the interference graph: if the sets of predicates that guard two live ranges L1 and L2 are disjoint, no interference edge needs to be added between them.

Interference graph construction for predicated code is similar to non-predicated code: it is a backward scan of the instructions in a basic block with a single, non-predicated live vector initialized with the live-at-exit candidates. For each candidate the guarding predicates (=qualifying predicates of the instruction that references the candidate) are recorded as predicate sets associated with the live range in a separate table. The compiler routine checking for interference takes the qualifying predicate, candidate and the live vector as arguments. For each live candidate in the live vector it checks if the qualifying predicate is disjoint from all predicates in its predicate set. Interference is recorded accordingly in the interference graph. A defined variable is removed from the live vector when it is recognized as the start of its live range. Each candidate used in the current instruction is added to the live vector and the qualifying predicate is recorded in its predicate set. This is the set of predicates under which a candidate is live and is kept in the table mentioned above. The live vector representation in the predicate-aware allocator does not (need to) change and can remain predicate-*unaware*.

Finally, a predicate-aware live variable analysis *must* treat predicated live ranges conservatively across back edges. For example, in Fig. 5 variable B would be live under P2 and variable A under predicate P1. In one scenario, P1 could be true in the first iteration and false in the second. If B is live under P1, the allocator would recognize A and B as disjoint and could assign them the same physical register. In this case



```
    B =
    V =

    CMP  P1,P2=V,r0
(P1) A=
    …
(P1)          =A
(P2)          =B

    V =
```

- P1 and P2 are disjoint
- Assume P1 is true in first iteration
- Assume P2 is true in second iteration
- If A and B are assigned same register, then *use* of B under P2 actually uses value of A from the previous iteration.

→ Propagate liveness property under p0 across back edges. In irreducible graphs propagate liveness property under p0 across retreat edges.

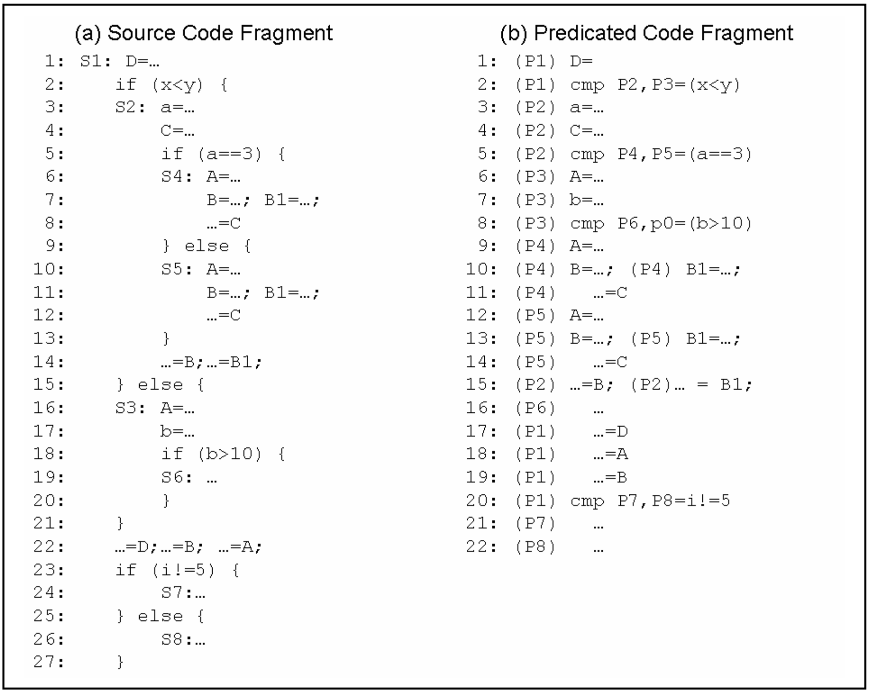
Fig. 5. Propagating liveness property under p0 on back edge

the assignment to A in the first iteration would overwrite B, which is used in the second iteration. When the live range of B becomes live under p0, which is the True predicate, then – since p0 interferes with every predicate – the live ranges for A and B are no longer disjoint.

For general graphs, global disjointness can be represented like interference in a triangular matrix of size $O(|B|^2)$, where |B| is the number of basic blocks in the routine. Global disjointness calculation is reaching-definition analysis for block predicates on the acyclic control-flow graph. This graph is derived from the original control-flow graph by removing back edges. Attention must be paid to irreducible graphs, which have retreat edges that are not back edges. Removing retreat edges gives an acyclic graph, but global disjointness is not necessarily consistent with local disjointness in an arbitrary acyclic region of the original graph.

3.2 Predicate Partition Graph (PPG) and Query System (PQS)

In predicated code live variable analysis and interference graph construction must reason about predicates. Both must find the start of a live range. Interference calculation must also recognize disjoint live ranges. The predicate query system (PQS) as described in Gillies et al. [8] provides predicate information to solve both problems. It is a set of predicate query routines on the predicate partition graph (PPG). The PPG is a directed acyclic graph whose nodes represent predicates and whose labeled edges represent partition relations between predicates. A partition $P = P1 \mid P2$ is represented by labeled edges $P \xrightarrow{r} P1$ and $P \xrightarrow{r} P2$. The common label indicates both edges belong to the same partition. The partitions represent execution paths and are derived by a single traversal of the control flow graph. For each block in the graph that has two or more successors or predecessors the partition $P = P1 \mid P2 \mid \ldots \mid PN$ is added to the graph, where P is the block predicate of the block and predicates Pi (i=1,…,N) are the block predicates corresponding to the successors (predecessors).



(a) Source Code Fragment

```
S1: D=…
    if (x<y) {
    S2: a=…
        C=…
        if (a==3) {
        S4: A=…
            B=…; B1=…;
            …=C
        } else {
        S5: A=…
            B=…; B1=…;
            …=C
        }
        …=B;…=B1;
    } else {
    S3: A=…
        b=…
        if (b>10) {
        S6: …
        }
    }
    …=D;…=B; …=A;
    if (i!=5) {
        S7:…
    } else {
        S8:…
    }
```

(b) Predicated Code Fragment

```
(P1) D=
(P1) cmp P2,P3=(x<y)
(P2) a=…
(P2) C=…
(P2) cmp P4,P5=(a==3)
(P3) A=…
(P3) b=…
(P3) cmp P6,p0=(b>10)
(P4) A=…
(P4) B=…; (P4) B1=…;
(P4)   …=C
(P5) A=…
(P5) B=…; (P5) B1=…;
(P5)   …=C
(P2) …=B; (P2)… = B1;
(P6)   …
(P1)   …=D
(P1)   …=A
(P1)   …=B
(P1) cmp P7,P8=i!=5
(P7)   …
(P8)   …
```

Fig. 6. Example to illustrate liveness under predicate sets. C source code and corresponding predicated code in an intermediate representation.

There are two preparation steps before the partition graph is built: first, the control flow graph is completed. Completion is necessary for the uniqueness of the predicate partitions and preciseness of disjointness. Completion is a single pass over the control-flow graph and inserts empty basic blocks on critical edges. A critical edge is defined as follows: If basic block B1 has two or more successors and basic block B2 has two or more predecessors, then the edge B1 —> B2 is critical. The inserted block is referred to as JS ("Join-Split") block. Second, a block predicate is assigned to each basic block. For this, the compiler uses the RK algorithm (Park and Schlansker [16]). The characteristic of the RK algorithm is that it assigns the same block predicate to a set of control-equivalent basic blocks. Two basic blocks B1 and B2 are control-equivalent if B1 executes whenever B2 executes and vice versa. Using a single predicate for a class of control-equivalent blocks results in a more compact representation of the predicate relations derived from the control-flow graph in the PPG.

We use a more elaborate version of the example in Johnson and Schlansker 12] to illustrate the predicate partition graph and PQS. Fig. 6 shows source code and predicated code of our example, while Fig. 7 has the control-flow graph including block predicates and PPG. Control-equivalent basic blocks are assigned the same block predicates. The edge from Block 3 to Block 8 is critical, and the completion phase inserted JS-Block B6' on the edge. The acyclic predicate partition graph corresponding to the control-flow graph fragment is shown in Fig. 7. Partitions $P1 = P2 \mid P3$ (edges "a"), $P1 = P7 \mid P8$ (edges "e") and $P3 = P6' \mid P6$ (edges "c") are

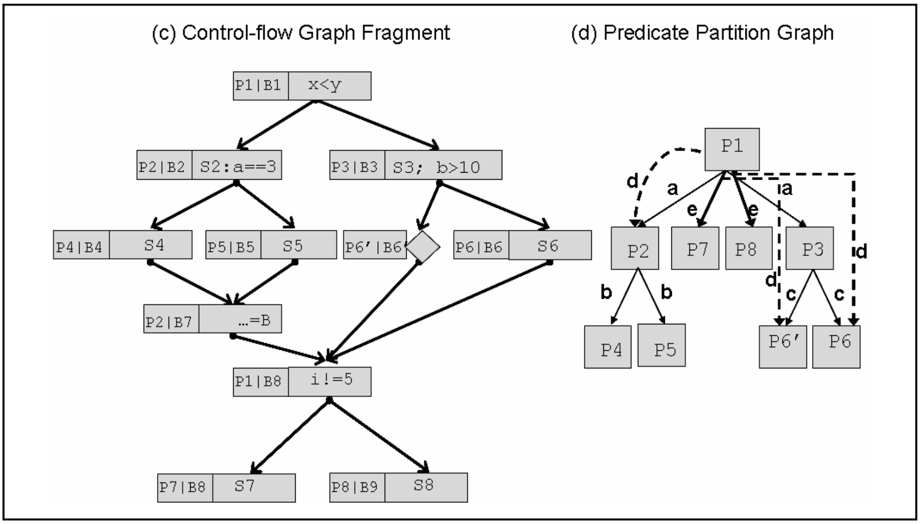


Fig. 7. Control-Flow Graph and PPG for the example in Fig. 6

forward partitions, partition P1 = P2 | P6' | P6 (edges "d") is a backward partition and P2 = P4 | P5 (edges "b") is both, a forward and a backward partition.

Both live variable analysis and interference calculation use PQS queries that walk the predicate partition graph (PPG) to compute accurate liveness information at each instruction. Fig. 10 shows two predicated live ranges A and B and their predicated sets during a PQS-based backward traversal of instructions 1-20 in the predicated code fragment of the example. The interference graph construction uses this backward traversal to find and record live range interferences.

PQS is powerful, but it comes at a cost. First, it requires the construction of the predicate partition graph, which is linear in space and time of the number of basic blocks. Second, unlike classical live variable analysis, which operates on basic blocks, the granularity for PQS-based predicated live variable dataflow is an instruction, so customized dataflow routines are required. Finally, the PQS queries are invoked at every predicated instruction. These cost factors motivate alternative solutions.

3.3 A Family of Predicate-Aware Register Allocators

This section assumes all predicated code is compiler generated. We propose a family of predicate-aware allocation schemes based solely on classical register allocation techniques. This is based on the observation that computing partitions based on PQS at every instruction during interference graph construction and live variable analysis is not necessary for all live ranges. Specifically, PQS partitions are not necessary when the qualifying predicates for the definitions and uses of a live range match or a definition dominates all uses. In other cases, partitions can be pre-computed at each use on demand. Building and repeatedly querying the PPG is not necessary. In particular we will identify four types of predicated live ranges – match, dominate, partition, overlap - two methods for interference modeling – “simple” and “complex”, and three implementation strategies. In Strategy 1, only match and dominate live ranges are modeled precisely. Strategy 2 models all simple live ranges precisely, and Strategy 3 models all live range precisely, which is equivalent to a PQS-based implementation. The four types of predicated live ranges are recognized in the original control-flow region.

There are four fundamental relations between predicated definitions and uses (Fig. 8). Predicated live ranges are classified based on the original control-flow region the predicated code is derived from. It is important to keep the correspondence between blocks and qualifying predicates in mind. A definition is clearly the start of the live range when the qualifying predicates of the definition and use match. This is also the case when the qualifying predicate of the definition dominates the qualifying predicates of the uses. When multiple definitions reach a use, two cases are possible. First, when definitions form a partition, the qualifying predicates of the definitions are mutually disjoint and the first definition in the hyperblock is the start of the live range. In this case the allocator gets precise disjointness by tracking liveness under all predicates that reach the use. Therefore it can track liveness under all definition predicates reaching a use rather than the qualifying predicate of the use (instruction). For example, in the partition case in Fig. 8, instead of tracking liveness under P3, recording liveness under P1 and P2 would give precise disjointness information. In this scenario, the definition of V under P2 (or P1) would kill liveness under P2 (or P1). Any subsequent – in the backward traversal – variables (defined or used) under P2 (or P1) do not interfere with V. This would not be the case if the live range were tracked using P3, unless a system like PQS partitioned P3 at the definition of V qualified under P2. Second, when definitions don’t form a partition (“overlap”), recording liveness under the reaching predicates would find the start of the live range, but disjointness would be conservative. For example, variables under qualifying predicate P2 could interfere with V, although V might have been killed under P2, since V would be live under P1, too (see “overlap” I n Fig. 8).

The live range for a variable defined in the region *and* live is completed (=made strict relative to the region) by adding pseudo definitions into region blocks based on two rules: first, if the variable V is live at entry of two successors, follow both paths. Second, if block B1 has two successors, B2 and B3, and variable V is live at entry in B2, but dead at entry in B3, insert a pseudo definition at the beginning of B2. The pseudo definition does not start the live range, but is used to form a partition.

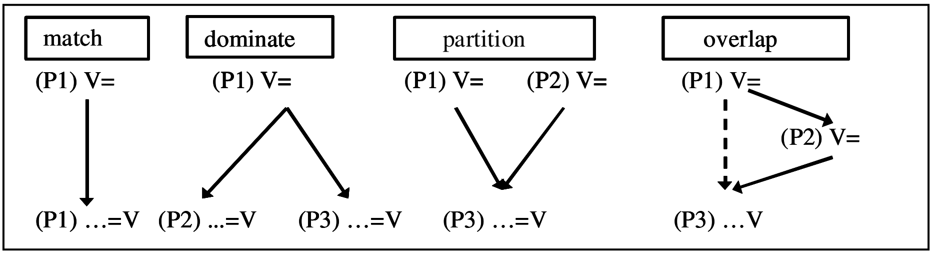


Fig. 8. Classification of predicated live ranges

Predicated Code	Predicate Sets For Variables			
	Partition		PQS	
	A	B	A	B
1: (P1) D=	{}	{P3}	{}	{P6',P6}
2: (P1) cmp P2,P3=(x<y)	{}	{P3}	{}	{P6',P6}
3: (P2) a=…	{}	{P3}	{}	{P6',P6}
4: (P2) C=…	{}	{P3}	{}	{P6',P6}
5: (P2) cmp P4,P5=(a==3)	{}	{P3}	{}	{P6',P6}
6: (P3) A=…	{}	{P3}	{}	{P6',P6}
7: (P3) b=…	{P3}	{P3}	{P6',P6}	{P6',P6}
8: (P3) cmp P6,p0=(b>10)	{P3}	{P3}	{P6',P6}	{P6',P6}
9: (P4) A=…	{P3}	{P3}	{P6',P6}	{P6',P6}
10: (P4) B=…; (P4) B1=…;	{P3,P4}	{P3}	{P4,P6',P6}	{P6',P6}
11: (P4) …=C	{P3,P4}	{P3,P4}	{P4, P6',P6}	{P4,P6',P6}
12: (P5) A=…	{P3,P4}	{P3,P4}	{P4,P6',P6}	{P4,P6',P6}
13: (P5) B=…; (P5) B1=…;	{P3,P4,P5}	{P3,P4}	{P1}	{P4,P6',P6}
14: (P5) …=C	{P3,P4,P5}	{P3,P4,P5}	{P1}	{P1,P2}
15: (P2) …=B; (P2)… = B1;	{P3,P4,P5}	{P3,P4,P5}	{P1}	{P1,P2}
16: (P6) …	{P3,P4,P5}	{P3,P4,P5}	{P1}	{P1}
17: (P1) …=D	{P3,P4,P5}	{P3,P4,P5}	{P1}	{P1}
18: (P1) …=A	{P3,P4,P5}	{P3,P4,P5}	{P1}	{P1}
19: (P1) …=B	{}	{P3,P4,P5}	{}	{P1}
20: (P1) cmp P7,P8=i!=5	{}	{}	{}	{}

Fig. 9. Example from Fig. 6 and two variables A and B and their predicate sets as seen by interference graph construction for a) partition based tracking and b) PQS-based tracking

Our example Fig. 6 illustrates the four fundamental relations. Live ranges D, a, and b are defined and used under a single predicate ("match"). In live range C the definition under (P2) dominates the uses under (P4) and (P5) ("dominate"). In live range A the qualifying predicates of the definitions (P3, P4 and P5) form a partition for the use under (P1) ("partition"). Live range B has uses under (P1) and (P2). The qualifying predicates (P4) and (P5) form a partition for (P2) ("partition"). For (P1) there is no partition. Since B is live at the entry of the predicated region, there is a pseudo definition of B in block 3 under (P3), since B is live at entry in block 3, but dead at entry in block 2.

The start of predicated live ranges can be found performing live variable analysis before if-conversion. After a region is if-converted, the first definition of a variable (= start of its live range) can be marked in a forward sweep over all instructions in the (linear) if-converted region, using the live-at-entry vector and recording definitions: at a given predicated instruction if the variable defined is not live-at-entry and no other definition of the variable has been seen, the first definition of the variable has been found.

Based on the types of predicated live ranges, three strategies for predicate-aware register allocation can be defined that model predicate disjointness with increasing accuracy:

Strategy 1: Dominate-or-Match

The qualifying predicates of instructions that use a variable form the predicate set of that variable. For live ranges with matching qualifying predicates for definition and uses, interference is precise. This is true also when the definition predicate dominates all use predicates.

Strategy 2: Partition Tracking
In addition to Strategy 1, live ranges are recorded under qualifying predicates of the -possibly pseudo- definitions that reach a use, if this set is a partition and the qualifying predicate of the use post-dominates each definition predicate. The qualifying predicates at the definitions are either in the partition or -in case the predicate is from a pseudo definition- dominate a partition predicate.

Fig. 9 has the predicated code from our example and considers two live ranges A and B to illustrate Strategy 2. For live range A, {P3, P4, P5} reach the use under (P1). Since P3, P4 and P5 are mutually disjoint and the use post-dominates the definitions, this partition is the predicate set at the use of A. Live range B is similar to live range A, except that B is completed by a (implicit) pseudo definition at the entry of block 3. Completion ensures that all live ranges with uses in the region are strict and enables partition formation at uses. A live range is strict when there is a definition on every path to a use.

After if-conversion each read operand (“use) is augmented with a list of qualifying predicates that represent the qualifying predicates of its reaching definitions. Strategy 2 relies on reaching definition analysis per predicated region: when more than one definition predicate reaches a use and the predicates are disjoint, record the partition that represents reaching qualifying predicates at each use.

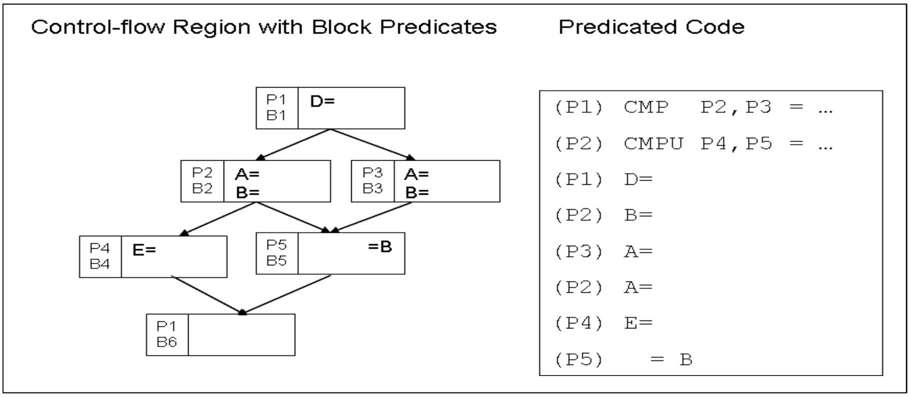
The following theorem lists the live ranges whose interferences can be modeled precisely by Strategy 2.

Theorem 1. (Characterization of Simple Live Ranges)

Strategy 2 can model interferences precisely for the following live ranges:

- Definition and use predicate match
- Definition predicate dominates use predicate
- Definition predicates form a partition. Use predicate post-dominates each partition predicate.
- Two definition predicates reaching a use are on at most one execution trace (or execution path in the original control-flow region).

Proof
Precise interference means for each predicated live range L:
When L is recognized as live at a given program point, L is actually live.
When L is recognized as dead, it is actually dead.
When L is recognized as disjoint from another live range L’, it is actually disjoint.
When L is recognized as interfering with L’, it is actually interfering.
The theorem is clear for the simple cases, match and dominate. In these cases the predicate set of a live range consists of the qualifying predicates seen at its uses. The matching or dominating definition stops the live range (Strategy 1). For the remaining cases we need to develop some intuition first. Tracking a live range under the qualifying predicate of the use ensures that disjointness in the predicated region is precise with respect to instructions (variables) that are not on a path to the use in the original control-flow graph. In case the definition predicates form a partition and the use predicate post-dominates all partition predicates, then all paths starting at definitions end at the use. There cannot be an off path instruction in the predicated region that would introduce an interference that is not visible in the original control-flow graph.



```
(P1) CMP  P2,P3 = ...
(P2) CMPU P4,P5 = ...
(P1) D=
(P2) B=
(P3) A=
(P2) A=
(P4) E=
(P5)   = B
```

Fig. 10. Tracking live variable of B under partition P2|P3 would result in extra interference with E in the predicated code. Tracking B under P5 does not cause this interference since P4 and P5 are disjoint.

Therefore tracking the live range under partition predicates cannot introduce extra interferences. The case of two definitions that don't from a partition and reach a use can be reduced to the partition case. Since there is only one path that contains both definitions, there must exist a basic block with a predicate disjoint to the "second" definition predicate (= block predicate of block containing the "second" definition). Since the region is assumed to be complete (=JS blocks inserted as needed), the block can be chosen so that the block containing the "first" definition predicate dominates it. Inserting a pseudo definition in the selected block ensures the partition property: since the qualifying predicate of the pseudo definition is disjoint from the qualifying predicate of the second definition, they form a partition at the use. In this case one partition predicate (from the pseudo definition) will not be a definition predicate, but dominated by it (the definition predicate of the first definition). This proves the theorem for two definitions. The general case of N definitions can be reduced to this special case.

Consider the example code in Fig. 10 to visualize an imprecise interference when the use does not post-dominate definitions. P2|P3 form a partition for live range B, but the use under P5 does not post-dominate P2. In the predicated code there could be an off-path instruction like the definition of E under P4 "before" the use of B under P5. If liveness of B were tracked under partition predicates P2|P3, E and B would interfere, since P2 and P4 are not disjoint. On the other hand, if liveness of B is tracked under P5, E and B cannot interfere, since clearly P4 and P5 are disjoint. This scenario cannot happen when the use post-dominates all partition predicates, since there cannot be an "off-path" instruction on the execution trace.

The remaining live ranges require a more sophisticated method to model interferences precisely. There are two cases left: first, the use does not post-dominate the partition predicates. Second, two or more definitions overlap on more than one execution trace (or execution path in the original control-flow graph). The first case can be handled by tracking the live range under the use and the partition predicates. The second case is reduced to partition, dominate or match live ranges by splitting. Splitting is described below.

Strategy 3: Use-and-partition Tracking
In addition to Strategy 2, track live variables under use-and-partition predicates and "split" live ranges when two definitions overlap on more than one path.

When the use does not post-dominate the definition, the use predicate (=qualifying predicate of the instruction containing the use) gets associated with the partition predicates. This is necessary for precise disjointness information: when the use does not post-dominate all predicates in a partition (of two or more predicates), disjointness could be conservative. Therefore, the live range is tracked under the qualifying predicate of the use *and* the partition predicates. Since the partition predicates represent disjoint portions of execution traces, precise disjointness is due to the following rule used during interference calculation: at any given instruction, if the qualifying predicate of a definition of live range L1 interferes with the use predicate of live range L2, but *not* with any of its associated partition predicates, then live ranges L1 and L2 are (actually) disjoint at this point. This gives precise disjointness: First, when the use does not post-dominate the definitions, there can be instructions on the execution trace that are not on any path from the definition to the use. Since the qualifying predicate of the use is disjoint from the qualifying predicate of these instructions, no imprecise interference can be encountered. Second, false interferences could be recorded with variables in instructions on paths to a definition, but the rule above is preventing this, since at every definition the qualifying predicate is removed from the partition.

In case definitions overlap on more than one execution path, additional live range splitting is necessary. This is achieved by inserting an identity move under the qualifying predicate of the definition. This move only changes predicate tracking for the live range.

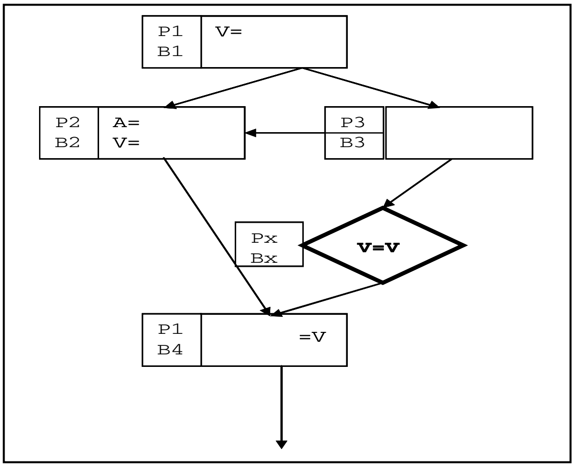


Fig. 11. Complex live range tracking and splitting for live range of variable V. Only the control-flow graph of the region is shown before if-conversion. An identical move for variable V in basic block Bx with block predicate Px splits the original complex live range for V into a partition and a dominate live range.

Fig. 11 illustrates a live range V in a control-flow graph snippet. The definitions for V overlap on more than one path to the use. There are two definitions of V in blocks B1 and B2. The use in block B4 post-dominates the definitions, but the definitions in blocks B1 and B2 overlap on paths $1 \to 2$ and $1 \to 3 \to 2$. The disjoint partition for the definition in block B4 is P2| Px, where Px is a JS block. Since P1 does neither dominate nor match Px, tracking under Px would not find the start of the live range. The trick is adding identical move in block Bx, which is inserted by control-flow graph completion. The use in block B4 is recorded under P2 and Px, which form a partition. At the definition in block Bx the predicate set for V contains Px together with the associated partition P1|P1', which corresponds to definitions of V reaching the use. The original live range has been split into a "partition" live range (see Fig. 8), which is covered by Strategy 2, and a "dominate" live range.

From the discussion is it clear that Strategy 3 models interference precisely when a live range has two definitions on more than one execution path. The identity move can be inserted where the two definitions merge. Note that translating out of an SSA representation (15]) before if-conversion would yield the moves. The general case is

Theorem 2. (Characterization of Complex Live Ranges)

Strategy 3 can model interferences precisely for the following live ranges:

- Use predicate does not post-dominate partition predicates (1)
- When definitions overlap on more than one path, the live range can be split and handled by Strategy 2 or the case above. (2)

Proof

Preciseness for first case (1) is clear: The qualifying predicate of the use avoids interferences with an off-path instruction, which could be on the trace from a definition to the use. Partition tracking asserts that there is no conservative (false) interference with variables in instructions on the path from the entry code to the definition. Interference calculation is modified: if, at a given instruction in the interference graph construction, a qualifying predicate interferes with the use predicate from a live candidate, but not with the associated partition predicates, the live range under the qualifying predicate is (actually) disjoint from the live candidate.

Overlapping live ranges on multiple paths can be split into to simpler live ranges: Assume the live range has n > 2 definitions. Like in Fig. 11 an identical move can be inserted at a merge point of any two definitions. The live range section with the two definitions and the use in the identical "mov" is either a partition live range or can be modeled by use-and-partition tracking. This splitting technique can be applied iteratively until a split live range has only two definitions. This completes the proof of the theorem since it holds in the case n=2. □

We identified four types of predicated live ranges – match, dominate, partition and overlap – and two methods for interference modeling – "simple" and "complex", and three implementation strategies. The simple method covers all "match" and "dominate", and some partition and overlap live ranges ("simple live ranges"), while the complex method handles remaining partition and overlap live ranges ("complex live ranges"). Strategy 1 models only "match" and "dominate" live ranges precisely. Strategy 2 models all simple live ranges precisely, and finally Strategy 3 models all live range precisely. Strategy 3 is equivalent to a PQS-based implementation.

4 Results

We obtained the performance data on a 1.6 GHz Montecito processor using the Intel Fortran/C++ optimizing compiler (version 11.1). The detailed configuration is listed in Table 1. The benchmark suite is CINT2006, a popular industry-standardized CPU-intense suite used by OEMs for stressing a system's processor, memory subsystem and compiler. We only show CINT2006 data, since the CFP2006 data are similar.

Table 1. Experimental Setup

Processor	Intel Itanium 2 (Montecito) Processor, 1.6 GHz
Compiler	Intel Fortran/C++ Compiler (Version 11.1)
Memory	4 Gb Main, 16 K L1D, 16KB L1I , 256K L2D, 1M L2I , 12M L3 D+I
OS	Red Hat Enterprise Linux AS Release 4 (Kernel 2.6.9-36.EL #1 SMP)

The gains from predicate-aware register allocation are for two different implementations. In section 3.3 we classified four types of predicated live ranges: match, dominate, partition, and overlap and showed that simple live range tracking gives precise interference for match, dominate, as well as some partition and overlap live ranges. When a use predicate does not post-dominate all partition predicates or definitions overlap on many (= two or more) paths, complex live range tracking techniques must be employed for precise interference modeling. The basic implementation that tracks liveness under the qualifying use predicates and marks first predicate definitions (Strategy 1) gives practically identical run-time performance as the PQS-based implementation. The difference ("Delta") between the methods shown in Table 2 is within the run-to-run variation of the benchmarks.

Table 2. Performance Gains from Predicate-Aware Allocation on CINT2006

Benchmark	Basic allocation	PQS allocation	Delta
400.perlbench	37.67%	37.67%	0.00%
401.bzip2	5.01%	5.01%	0.00%
403.gcc	1.71%	1.45%	-0.26%
429.mcf	0.93%	0.93%	0.00%
445.gobmk	2.91%	2.91%	0.00%
456.hmmer	1.20%	1.20%	0.00%
458.sjeng	8.01%	8.01%	0.00%
462.libquantum	0.00%	0.37%	0.37%
464.h264ref	0.00%	1.03%	1.03%
471.omnetpp	0.40%	0.40%	0.00%
473.astar	0.97%	0.00%	-0.96%
483.xalancbmk	0.00%	0.00%	0.00%
Geomean	4.48%	4.50%	0.01%

The outlier in Table 2 is the 37.67% gain in 400.perlbench. This is due to reduced RSE traffic in S_regmatch, the hottest (and self-recursive) function of the benchmark. Without a predicate-aware allocator all 96 register on the register stack get allocated. With a predicate-aware allocator only about half the number of registers is used. The increase in register pressure without the predicate-aware allocator is explained with live range extensions in loops (see Fig. 4).

The performance data suggest that basic predicate-awareness in the coloring allocator reaps the performance benefits. The performance gain from the simplest predicate-aware allocator (Strategy 1) and PQS-based predicate-aware allocator match. The basic predicate-aware allocator models precise interference only for match and dominate live ranges, but is conservative for all partition and overlap live ranges. For the experiment, live ranges were first completed in the original control-flow graph of the candidate region. Then a region-based reaching definition analysis was performed. Together with dominator information, this is sufficient to classify predicated live ranges within the region. When a live range falls into multiple classes, only the "most complex" class (overlap > partition > dominate > match) gets accounted for. The data for predicated live ranges distribution is in Table 3. There are only two benchmarks (402.bzip2 and 471.omnetpp) that have more than 10% (11.44% and 12.01%) partition and overlap live ranges. For all other benchmarks this number is below 10%. The data in the tables were collected for all predicated live ranges in all predicated regions of a benchmark. Since only relatively few partition and overlap live ranges exist, a system like PQS or complex live range tracking is not necessary for precise interference modeling. The data and code analysis suggests that the complex tracking cases are very rare (less than 2.5% for all benchmarks). If conservative disjointness for partition and overlap live ranges is a concern, Strategy 2 models more than 97.5% of the predicated live ranges precisely.

Table 3. Distribution of Predicated Live Ranges

Benchmark	match	dominate	partition	overlap
400.perlbench	73.92%	16.16%	9.46%	0.46%
401.bzip2	63.27%	22.68%	11.44%	2.61%
403.gcc	71.59%	20.78%	6.80%	0.83%
429.mcf	72.64%	20.46%	6.42%	0.48%
445.gobmk	77.94%	18.84%	2.91%	0.31%
456.hmmer	69.86%	22.67%	6.68%	0.79%
458.sjeng	73.68%	18.26%	7.37%	0.69%
462.libquantum	74.51%	16.67%	8.39%	0.44%
464.h264ref	74.70%	16.32%	8.87%	0.11%
471.omnetpp	66.77%	21.04%	12.01%	0.17%
473.astar	75.73%	18.63%	4.70%	0.94%
483.xalancbmk	69.81%	20.98%	6.82%	2.39%

5 Related Work

There are a number of approaches to represent predicate relations in a compiler. The IMPACT compiler uses the Predicate Hierarchy Graph (PHG) (Mahlke et. al. [13]). For each definition of a predicate the PHG tracks the predicates that guard the definition. It can also handle OR-expressions and is applied to code in hyperblock regions. A hyperblock is a predicated superblock, which is an acyclic single entry multiple exit region. A more sophisticated approach than the PHG is the predicate query system (PQS) (Gillies et al. [8], Johnson and Schlansker [12]). It uses the predicate partition graph to determine predicate relations. PQS can determine accurately predicate relations that can be expressed as logical partitions. Here two predicates P2 and P3 form a predicate partition P1 when P1 is the union of P2 and P3, and P2 and P3 cannot both be true simultaneously. Both PHG and PQS use approximations in the analysis of already predicated code. In this case the code is not derived from the control-flow graph, but instead supplied by the user (in the case of assembly code) or an earlier compiler phase. When the code is derived from the control-flow graph, PQS can represent predicate disjointness information accurately in acyclic regions. More subtle predicate analysis methods that derive accurate predicate relations for already predicated code have been developed also. Eichenberger [7] represents logical predicate relations, so called P-facts, and determines predicate relations in a logic solver. He applies this information for register allocation in hyperblocks. Sias et al. [18] developed the predicate analysis system (PAS), which is as accurate as Eichenberger, but can determine predicate relations globally using a BDD solver. In contrast to the approaches described in literature, this paper makes no attempt to address the general predicate relation problem. It makes the assumption that predicated code is derived from acyclic control-flow graph regions. This holds in general for compilers. Our paper shows that classical interference calculation can be extended in various degrees of accuracy to model interference of predicated live ranges precisely. PQS-based allocation is used for reference.

6 Conclusions

This paper classified predicated live ranges into four types: match, dominate, partition and overlap. It described implementation strategies based on classical dataflow analysis to allocate register candidates for all classes efficiently and precisely. We implemented a basic predicate-aware allocator that models match and dominate live ranges precisely, and partition and overlap live ranges conservatively. We compared the basic allocator to a PQS-based allocator and found practically no performance difference on CINT2006. In this case investing in a sophisticated predicate database and query system for a predicate-aware allocator is not necessary. The reason is that only a small portion of live ranges in predicated code require complex live range tracking for precise interference modeling. The gain achieved from predicate-aware register allocation on the CINT2006 integer benchmarks is up to 37.6% and 4.48% in the geomean for the SPEC base options of the Intel Itanium production compiler.

References

1. Aho, V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, & Tools, 2nd edn. Addison Wesley, Reading (2007)
2. Allen, J.R., Kennedy, K., Porterfield, C., Warren, J.D.: Conversion of Control Dependence to Data Dependence. In: Proceedings of the 10th ACM Symposium on Principle of Programming Languages, POPL 1983, January 1983, pp. 177–189 (1983)
3. Bharadwaj, J., Chen, W.J., Chuang, W., Hoflehner, G., Menezes, K., Muthukumar, K., Pierce, J.: The Intel IA-64 Compiler Code Generator. IEEE Micro, 44–52 (September/October 2000)
4. Briggs, P., Cooper, K.D., Torczon, L.: Improvements to Graph Coloring Register Allocation. ACM Transactions on Programming Languages and Systems 16(3), 428–455 (1994)
5. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.: Register allocation via coloring. Comp. Lang. 6(1), 47–57 (1981)
6. Chaitin, G.J.: Register Allocation and Spilling via Graph Coloring. In: Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction, pp. 98–105 (1982)
7. Eichenberger, A., Davidson, E.S.: Register allocation for predicated code. In: Proceedings of the 28th Annual International Symposium on Microarchitecture, MICRO-28, December 1995, pp. 180–191 (1995)
8. Gillies, D.M., Ju, R.D.-C., Johnson, R., Schlansker, M.S.: Global Predicate Analysis and its Application to Register Allocation. In: Proceedings of the 29th International Symposium on Microarchitecture, MICRO-29, December 1996, pp. 114–125 (1996)
9. Huck, J., Morris, D., Ross, J., Knies, A., Mulder, H., Zahir, R.: Introducing the IA-64 Architecture. IEEE Micro, 12–22 (September/October 2000)
10. Intel Corporation, Intel® Itanium® Architecture Software Developer's Manual, Revision 2.2, vol. 1-3 (January 2006), `http://developer.intel.com/design/itanium/manuals/iiasdmanual.htm`
11. Intel Corporation, Intel® Itanium® 2 Processor Reference Manual (May 2004), `http://download.intel.com/design/Itanium2/manuals/25111003.pdf`
12. Johnson, R., Schlansker, M.S.: Analysis techniques for predicated code. In: Proceedings of the 29th International Symposium on Microarchitecture, MICRO-29, December 1996, pp. 100–113 (1996)
13. Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E., Bringmann, R.A.: Effective compiler support for predicated execution using the hyperblock. In: Proceeding of the 25th Annual International Symposium on Microarchitecture MICRO-25, December 1992, pp. 45–54 (1992)
14. McNairy, C., Soltis, D.: Itanium 2 Processor Microarchitecture. IEEE Micro, 44–55 (March/April 2003)
15. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco (1997)
16. Park, J.C.H., Schlansker, M.S.: On predicated execution. Tech. Rep. HPL-91-58, HP Laboratories, Palo Alto, CA (May 1991)
17. Schlansker, M.S., Rau, B.R.: EPIC: Explicitly Parallel Instruction Computing. Computer, 37–45 (February 2000)
18. Sias, J.S., Hwu, W.-M.W., August, D.I.: Accurate and Efficient Predicate Analysis with Binary Decision Diagrams. In: Proceedings of the 33rd International Symposium on Microarchitecture MICRO-33, December 2000, pp. 112–123 (2000)

Preference-Guided Register Assignment

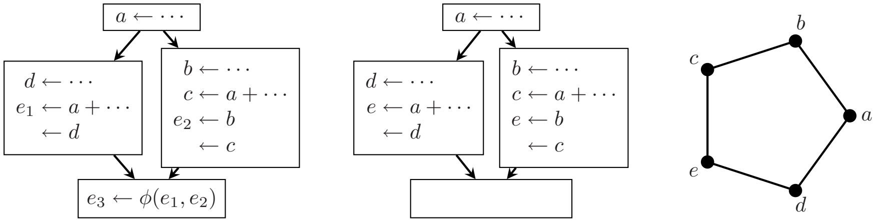
Matthias Braun[1], Christoph Mallon[2], and Sebastian Hack[2]

[1] Karlsruhe Institute of Technology
matthias.braun@kit.edu
[2] Computer Science Department
Saarland University
{mallon,hack}@cs.uni-saarland.de

Abstract. This paper deals with coalescing in SSA-based register allocation. Current coalescing techniques all require the interference graph to be built. This is generally considered to be too compile-time intensive for just-in-time compilation. In this paper, we present a biased coloring approach that gives results similar to standalone coalescers while significantly reducing compile time.

1 Introduction

The register allocation phase of a compiler maps the variables of a program to the registers of the processor. One important part of register allocation is coalescing. Coalescing is an optimization that tries to remove register-to-register move instructions by assigning the source and the target of the move the same register. One serious drawback of coalescing is that it can increase the register demand of the program. Consider the example in Figure 1. The register demand in the SSA-form program P is 2 everywhere. If we perform classical SSA destruction and coalesce the move instructions represented by the ϕ-function, that is merge the live ranges of e_1, e_2, and e_3 into one (as shown in P'), we need 3 registers for a valid register assignment, as can be verified by coloring the interference graph G of P'.



(a) Program P (b) Program P' (c) Interference graph of P'

Fig. 1. Coalescing a ϕ-function

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 205–223, 2010.

Chaitin et al. [1] express register allocation by graph coloring and show that, if one makes no assumption about coalescing, every undirected *interference graph* G corresponds to a program P for which holds: An optimal register allocation for P is an optimal coloring of G. Although this approach is very popular, it has two undesirable properties:

- Because graph coloring is NP-complete, we need a heuristic to color such a graph. Hence, we might fail to color a graph with k colors although the graph is k colorable. For register allocation this means that we unnecessarily spill variables to memory.
- For any given $n \in \mathbb{N}$ there exists a graph that has a largest clique of size l but needs $l + n$ colors for an optimal coloring. The size of the largest clique in that graph corresponds to the register pressure in the program. Hence, as in the example above, we need $l + n$ registers although there are never more than l variables alive.

Consequently, recent register allocation approaches do not allow arbitrary coalescing of live ranges: In an SSA-form program, some live ranges are split by ϕ-functions. This splitting is sufficient to overcome both drawbacks mentioned above (see [2–4] for proofs):

1. An optimal register assignment can be computed in linear time.
2. The register pressure equals the minimum number of registers needed for the program.

Live-range splitting by ϕ-functions is not the only source of move instructions in a program. Treating register constraints as they are incurred by some architectures and application binary interfaces, also provokes the insertion of move instructions: Assume a variable v is an argument to a function call and the ABI dictates that it has to be in register `R1`. Then, we need to move v to `R1` in front of the call. On the other hand, if we assigned `R1` to v in the first place, we can save this move.

All these live-range splits result in move instructions. Usually, reducing the number of move instructions is the task of the *coalescing* phase of a register allocator. However, most of the existing coalescing techniques are very compile-time intensive: They all require the interference graph to be materialized as a data structure. Some of them even perform updates on that graph. However, in just-in-time compilation, constructing and updating the interference graph is considered too costly.

1.1 Contributions

In this paper, we pursue a new approach to coalescing: We assume that spilling already took place and the register pressure everywhere in the program is $\leq k$, where k is the number of available registers. Instead of delegating coalescing to a separate phase, we make the assignment pass aware of move instructions by biasing the assignment: We try to assign sources and targets of move instructions the same register. To this end, we extend the conventional SSA register allocation algorithm by the following techniques:

- We compute *register preferences* for each variable. These preferences reflect the register constraints the variable is exposed to. Hence, instead of non-deterministically choosing a register during the assignment phase, we are able to make a more profound register choice. In doing so, we avoid many of the moves that are usually inserted due to register constraints. Section 3.1 discusses register preferences in more detail.
- When coloring the target of a move, e.g. the result of a ϕ-function, we propagate preferences for that color to the not-yet-colored sources, in this case the operands of the ϕ-function. Thus, when those variables are to be colored, we attempt to assign them the same register as the target of the ϕ-function. Section 3.3 gives a detailed discussion.
- When a variable is assigned to a register and the most preferable register is occupied by another variable, we allow for *optimistically* moving the occupying variable to a different register. Placing a variable in the preferred register from the start is often better than doing it right in front of the program point that caused the preference: If we assume that the register is occupied at that point we need two moves (one to free the register and one to move the variable to it) instead of the one needed to free the register upon the variable's definition. Details are discussed in Section 3.4.
- Based on profile data or estimated execution frequencies, we compute an order of the basic blocks in a control-flow graph that aids in removing more moves on frequently executed traces of the CFG (Section 4).

Our experimental evaluation (see Section 5) shows that coalescing in an SSA-based register allocator is important: The runtime of the benchmarks is decreased by 5% and the number of executed move instructions is decreased by 55% percent. Compared to our previous work based on graph recoloring [5], register allocation and coalescing is 2.27 times faster. Our compile-time measurements show a linear behavior of the presented algorithm.

2 SSA-Based Register Allocation

This section reviews the basics of SSA-based register allocation and describes how register constraints are treated by an SSA-based allocator.

Register allocation on the SSA form uses the live-range splitting caused by ϕ-functions. The ϕ-functions of a basic block basically act as control-flow dependent *parallel* moves (see Figure 2). This splitting and the dominance property of the SSA form[1] cause the interference graphs for SSA-form programs to be chordal (see [2–4] for proofs). Chordal graphs have two properties that make them appealing for register allocation:

1. They are optimally colorable in time $O(\omega(G) \cdot |V|)$ where $\omega(G)$ is the size of the largest clique in G and V is the set of G's nodes.
2. The size of the largest clique in the graph is equal to the minimum number of colors needed for a coloring - the graph's chromatic number.

[1] The fact that each use of a variable is dominated by its definition.

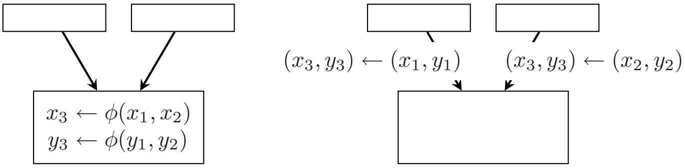

$x_3 \leftarrow \phi(x_1, x_2)$
$y_3 \leftarrow \phi(y_1, y_2)$
$(x_3, y_3) \leftarrow (x_1, y_1)$
$(x_3, y_3) \leftarrow (x_2, y_2)$

Fig. 2. ϕ-functions are parallel Moves

Furthermore, for each clique in the interference graph there is a location in the program where all the variables of the clique are alive. Thus, unlike conventional graph-coloring register allocation, lowering the register pressure to the number of available registers k results in a k-colorable interference graph. Hence, pressure-based spilling heuristics [6–8] already lead to k-colorable interference graphs.

2.1 Register Assignment

After the spilling phase has lowered the register pressure everywhere to at most k, registers can be assigned. While the interference graph is helpful to reason about, it actually never has to be built as a data structure when assigning registers. A SSA interference graph can be colored using a node elimination algorithm like the one used by Chaitin et al. [1] in their seminal paper. However, the advantage of SSA-based register allocation is that this elimination order coincides with dominance:

> Before a variable v can be eliminated, all variables that dominate v and interfere with v have to be eliminated.

Consequently, an order that colors a program point only after its dominators have been colored leads to an optimal coloring of the SSA interference graph.

Algorithm 1 shows the assignment pass for a single basic block B. This algorithm is then applied to every basic block such that the immediate dominator of B is processed before B itself (in Section 4 we propose a specific coloring order). We maintain a bit set occupied of registers used by currently live variables. We initialize this bitset with the registers of the values that are live-in at the beginning of B. Note that all live-in values already have a register assigned because:

1. The definition of a variable dominates all program points where it is alive.
2. All dominators of B have already been processed.

Then, all ϕ-functions of B are assigned. The arguments of the ϕ-functions are ignored in B because they correspond to move instructions in the predecessor blocks and hence don't represent live values in B.

The instructions inside the basic block are now processed in order: For every variable that dies at a program point, the register is put back into the pool of free registers. For every value which is defined by an instruction, a free register is chosen (function get_register) and put into the occupied set.

Algorithm 1. Coloring of a basic block

```
proc color_block(block):
    # Determine initial register occupation and color $\phi$−nodes
    occupied $\leftarrow \emptyset$
    for val in block.live_in:
        occupied $\leftarrow$ occupied $\cup$ { val.register }
    for phi in block.phi_nodes:
        phi.register $\leftarrow$ get_register(phi, occupied)
        occupied $\leftarrow$ occupied $\cup$ { phi.register }

    # Assign registers
    for insn in block.instructions:
        enforce_constraints(insn)
        for a in insn.arguments:
            if dies(a, insn):
                occupied $\leftarrow$ occupied $\setminus$ { a.register }
        for r in insn.results:
            r.register $\leftarrow$ get_register(r, occupied)
            occupied $\leftarrow$ occupied $\cup$ { r.register }

    block.processed $\leftarrow$ true
    # Create $\phi$−moves where necessary
    for pred in block.preds:
        if pred.processed:
            implement_phi_copies(pred, block)
    for succ in block.succs:
        if succ.processed:
            implement_phi_copies(block, block.succs[0])
```

2.2 Register Constraints

In practice, the instruction set architecture (ISA) and the application binary interface (ABI) impose several constraints on the registers that are allocatable for a variable *at* a program point. Most prominent and omnipresent are caller- and callee-save registers across function calls. For example, the x86 ABIs state that the contents of the registers `eax`, `ecx`, and `edx` are destroyed after a function call. The return value of a function returning an `int` is delivered in `eax`.

Traditionally, such constraints are handled by splitting the live ranges of all variables alive across such a constrained instruction by inserting a parallel move instruction. In doing so, all registers become available in front of that instruction and the assignment pass can easily compute an assignment that fulfills these constraints. In Algorithm 1 this is expressed by the function enforce_constraints

$$\begin{aligned} w\langle\texttt{R0}\rangle &\leftarrow \texttt{call}(x\langle\texttt{R1}\rangle, y\langle\texttt{R2}\rangle)\\ &\vdots\\ &\text{uses of } x, y, z, w \end{aligned}$$

(a) A constrained instruction

$$\begin{aligned} (x'\langle\texttt{R1}\rangle, y'\langle\texttt{R2}\rangle, z') &\leftarrow (x, y, z)\\ w\langle\texttt{R0}\rangle &\leftarrow \texttt{call}(x', y')\\ &\vdots\\ &\text{uses of } x', y', z', w \end{aligned}$$

(b) Splitting live ranges

Fig. 3. A call instruction with register constraints

which we do not describe in further detail here. Figure 3 gives an example of a constrained call instruction and the inserted parallel move[2].

To model register constraints, we annotate every program point ℓ with two partial functions (one for the defined and one for the used variables) that map a variable that has a register constraint at that program point to the register it is required to be in:

$$\mathit{constr}_\ell^{\mathit{use}} : \mathit{Var} \hookrightarrow \mathit{Reg} \qquad \mathit{constr}_\ell^{\mathit{def}} : \mathit{Var} \hookrightarrow \mathit{Reg}$$

where *Var* is the set of variables and $\mathit{Reg} \subset \mathbb{N}$ is the set of registers. For the example in Figure 3, we have:

$$\mathit{constr}_\ell^{\mathit{use}}(x) = \texttt{R1}, \mathit{constr}_\ell^{\mathit{use}}(y) = \texttt{R2} \qquad \mathit{constr}_\ell^{\mathit{def}}(w) = \texttt{R0}$$

2.3 Implementing Parallel Moves

The parallel move instructions are implemented *after* register assignment. Concerning the assigned registers, a parallel move corresponds to a register permutation that can be implemented with moves, swaps, xors, and so on [9]. For example, assume our architecture has four registers. Consider the following parallel move and a register allocation (indicated by the superscripts):

$$\text{Move: } (a^4, b^2, c^3) \leftarrow (d^3, e^1, f^4) \qquad \text{Permutation: } \begin{bmatrix} 2 & 3 & 4 \\ 1 & 4 & 3 \end{bmatrix}$$

This can be implemented with the following sequence of instructions:

$$\begin{aligned} &\texttt{move}\ \texttt{R2} \leftarrow \texttt{R0}\\ &\texttt{swap}\ \texttt{R3}, \texttt{R4} \end{aligned}$$

3 Coalescing with Register Preferences

In principle, Algorithm 1 can compute *any* legal register assignment for a CFG. The set of valid register allocations is basically characterized by the freedom of the function get_register: Whenever a register is assigned to a variable, get_register can choose among a set of free registers. However, regarding coalescing, not all

[2] Register constraints are indicated in angle brackets.

Algorithm 2. Choosing a register by preference

```
proc get_register(var, occupied):
    sort var.prefs by preference
    for (reg,pref) in var.prefs:
        if reg $\notin$ occupied:
            return reg
```

valid allocations are equally preferable. An allocation in which many sources and targets of moves have the same color is *better* because it will result in less shuffle code in the program. Given an oracle telling us the best register for each variable, the algorithm would produce an optimal coalescing[3].

Unfortunately the coalescing problem is NP-hard even for programs in SSA-form [3, 9]. We thus rely on a heuristic approach that is guided by register preferences which are calculated before coloring and can be updated while allocating. To this end, we introduce a *preference analysis* that computes a preference vector for every variable. Such a vector has a component for every register. The higher the value of a component, the more preferable it is to assign the variable to the corresponding register. This vector is then used by get_register to select a "good" register (see Algorithm 2). The following sections describe the preference analysis and a mechanism to adjust the preferences while assigning registers for ϕ-functions.

3.1 Register Preferences

Consider the example in Figure 4a. Assume the set of available registers when y is colored to be $\{\texttt{R0}, \texttt{R2}\}$ and assume the allocator (nondeterministically) chooses R2. Then, in front of its use, y has to be moved from R2 to R0 in order to fulfill the register constraint. If the allocator knew that y is needed in R0, it could have selected it in the first place.

To make a sensible choice in the presence of register constraints, we need to propagate information from constrained uses of variables to the point where the color selection is done.

Reconsider the live ranges in Figure 4a. When assigning registers to the variables, we first assign a color to x. Since x interferes with two variables (y, z) which have constrained definitions or uses to R1 and R0, it would be good to choose one of the other registers: R2 or R3. If we would assign x to R0, we would have to move it aside to make room for y right in front of its constrained use. Correspondingly, the variables y and z should have a strong dislike for all registers other than the ones occurring in their constraints. Furthermore, they interfere with each other, so they have an even stronger dislike for each other's

[3] Optimal for a given set of parallel moves. There are cases where a different placement of parallel moves can lead to a better overall result.

$x \leftarrow$

$y \leftarrow$

$z\langle \mathtt{R1} \rangle \leftarrow$

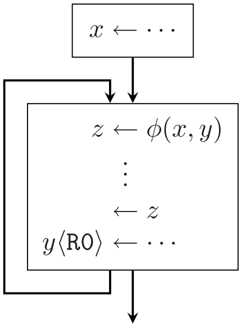
$\leftarrow y\langle \mathtt{R0} \rangle$

$\leftarrow z$

$\leftarrow x$

Var	Preference vector
x	$[-1, -1, 0, 0]$
y	$[0, -2, -1, -1]$
z	$[-2, 0, -1, -1]$

(a) Constrained live ranges


$x \leftarrow \cdots$
$z \leftarrow \phi(x, y)$
$\vdots$
$\leftarrow z$
$y\langle \mathtt{R0} \rangle \leftarrow \cdots$

(b) A ϕ-function with constrained arguments

Fig. 4. Examples for Preferences and constrained ϕ-functions

preferred register. Our analysis which is explained in the next section, computes the preference vectors shown in Figure 4a.

Thus, the allocator puts x in register R2 or R3 and leaves R0 and R1 untouched. y and z can then be directly allocated to R0 and R1 obviating any moves.

3.2 Preference Analysis

The register preference vector $\mathit{pref}(v)$ of a variable v is given by

$$\mathit{pref}(v) = \sum_{\{\ell \mid v \text{ is alive before } \ell\}} f_\ell \cdot \boldsymbol{c}_\ell^{\mathit{use}}(v) + \sum_{\{\ell \mid v \text{ is alive after } \ell\}} f_\ell \cdot \boldsymbol{c}_\ell^{\mathit{def}}(v)$$

where f_ℓ denotes the execution frequency of program point ℓ. This execution frequency can either be gathered from profile data or estimated (see e.g. [10]). For the sake of brevity, let $\Box \in \{\mathit{use}, \mathit{def}\}$. $\boldsymbol{c}_\ell^{\Box}(v)$ is the constraint vector concerning the used (defined) variables of program point ℓ for variable v:

$$\boldsymbol{c}_\ell^{\Box}(v) := \begin{cases} \boldsymbol{e}_i - \mathbf{1} & \text{if } v \in \text{ dom } \mathit{constr}_\ell^{\Box} \text{ and } i = \mathit{constr}_\ell^{\Box}(v) \\ -\sum_{i \in R} \boldsymbol{e}_i & \text{else} \qquad \text{with } R = \operatorname{ran} \mathit{constr}_\ell^{\Box} \end{cases}$$

where $\boldsymbol{e}_i$ is the vector that is one at component i and zero everywhere else. $\mathbf{1}$ is the vector containing only ones.

Thus, the preference vector of a variable contains the sum of dislikes (negative preferences) caused by register constraints of program points where the variable is alive. To calculate the preferences, we perform a backward walk over the program's basic blocks so we can keep track of live values. When we encounter a constrained definition/use we add preferences to all other variables alive at that point. This is a simple flow-insensitive analysis and can be done in a single pass over the program.

3.3 Affinity Chunks

Besides register constraints, ϕ-functions are the second source of shuffle code. A "bad" register assignment can cause a cascade of move instructions to be inserted at the end of a ϕ predecessor block. In contrast to constrained instructions, the desirable register of an operand of a ϕ-function is not fixed a priori: It depends on which registers the other operands and the result variable of the ϕ are allocated to. Therefore, we do *not* consider ϕ-functions when performing the preference analysis *but* modify the preference vectors during the assignment process. When coloring a ϕ-function, a preference for the chosen color is added to the preference vectors of the still uncolored variables of the same affinity chunk.

A second observation is that the constraints of the arguments of a ϕ-function affect the ϕ-function as well. Consider the example in Figure 4b. One variable of the affinity chunk of the ϕ-function needs to be in `R0` upon its definition. Assigning z any other register than `R0` will cause a move on the loopback edge which needs to be avoided at all costs. Hence, we *propagate* the preference for `R0` to the whole affinity chunk of y and thus try to assign x and z to `R0` as well. In general, the preferences for all members of an affinity chunk are weighted by their execution frequencies and distributed among its members.

When coloring a ϕ-function, we want to assign that register to all not-yet colored variables of the ϕ's affinity component. However, such an affinity component can exhibit interferences within itself. Thus, one usually splits up the affinity components into interference-free *chunks* by *aggressive coalescing.* Aggressive coalescing itself is an NP-complete problem; it is an instance of a minimum multi-cut (see [3, 9] for example). In practice, one is content with a heuristic that greedily tries to merge chunks. Let C and D be two chunks that we want to merge. To merge the chunk, there must not exist an interference between both chunks. If there is, the chunks cannot be merged and we "sacrifice" every affinity edge between both chunks. That means, that we no longer try to assign the same color to the variables of the move instruction, represented by the lost affinities. Of course, the order in which the chunks are merged decides on how good the results are, i.e. how many moves are introduced. This greedy heuristic requires an interference check between the two chunks. Naively, one could test each variable pair for interference, resulting in a quadratic algorithm. Recently, Boissinot et al. [11] gave a linear algorithm, exploiting SSA properties, to perform that check. However, this linear check still has to be performed whenever two chunks are to be merged.

To avoid this overhead, we do not split chunks up to the last interference edge but allow for remaining interferences within a chunk. This does not pose any correctness problems, as we use these chunks only to propagate register preferences when a ϕ-function is colored. In the worst case, we propagate preferences to a variable that interferes with that ϕ-function.

Our "approximated" chunks are computed using a union-find data structure. Whenever we encounter a ϕ-function, we check, whether the result variable of that ϕ-function and its operands interfere. This is can be done efficiently since we still have the set of live-in variables calculated by the liveness analysis. The chunk of an operand is merged with the ϕ's if the operand and the ϕ do not interfere. This can be done in hand with the preference analysis.

3.4 Optimistic Move Insertion

There is further room for reducing the number of move instructions: The fixed positions of the parallel moves aren't always optimal. A typical situation is shown in Figure 5a:

$$\begin{aligned} w\langle \mathtt{R0}\rangle &\leftarrow \mathit{call}() \\ x &\leftarrow \\ &\vdots \\ &\leftarrow x\langle \mathtt{R0}\rangle \\ &\leftarrow w \end{aligned}$$

(a) Program P

$$\begin{aligned} w\langle \mathtt{R0}\rangle &\leftarrow \mathit{call}() \\ x\langle \mathtt{R1}\rangle &\leftarrow \\ &\vdots \\ w\langle \mathtt{R2}\rangle &\leftarrow w\langle \mathtt{R0}\rangle \\ x\langle \mathtt{R0}\rangle &\leftarrow x\langle \mathtt{R1}\rangle \\ &\leftarrow x\langle \mathtt{R0}\rangle \\ &\leftarrow w\langle \mathtt{R2}\rangle \end{aligned}$$

(b) Assignment for P

$$\begin{aligned} w\langle \mathtt{R0}\rangle &\leftarrow \mathit{call}() \\ w\langle \mathtt{R1}\rangle &\leftarrow w\langle \mathtt{R0}\rangle \\ x\langle \mathtt{R0}\rangle &\leftarrow \\ &\vdots \\ &\leftarrow x\langle \mathtt{R0}\rangle \\ &\leftarrow w\langle \mathtt{R1}\rangle \end{aligned}$$

(c) Assignment with optimistic move

Fig. 5. Candidate for optimistic move insertion

Algorithm 3. Choosing a register with optimistic move insertion

```
proc get_register(var, occupied):
    sort var.prefs by preference
    for (reg,pref) in var.prefs:
        if reg $\notin$ occupied:
            return reg

        # Determine costs for moving the variable
        # which occupies the register away
        ovar $\leftarrow$ reg.current_variable
        sort ovar.prefs by preference
        for (oreg,opref) in ovar.prefs:
            if oreg $\notin$ occupied:
                other_win $\leftarrow$ opref $-$ oreg.current_pref
                break
        next_pref $\leftarrow$ preference value for next register
        win $\leftarrow$ next_pref $-$ pref
        if win $+$ other_win $>$ block.execfreq:
            create move from reg to oreg
            return reg
```

When the allocator reaches the assignment to variable x register `R0` is already occupied by w. A classical allocator would assign the next free register to x, say `R1`. A fixup would only occur before the constrained use of x. At this point however at least 2 move instructions are necessary: Variable w has to be moved away from `R0` and variable x into it. Instead, it is more beneficial to move variable w away from `R0` before the assignment to x as shown in Figure 5c compared to Figure 5c.

This situation is handled by optimistically inserting such early moves into the program: When the allocator finds that a desired output register is occupied by another variable then we determine the costs of moving that variable into another register. The cost is the sum of the preference differences when freeing the register by moving the occupying variable away and the preference differences when assigning the next possible register instead of the desired one. We compare these costs with the execution frequency of the current block. Higher costs are an indication that a move at the current position is cheaper than a later fixup. The move instruction is created optimistically. An improved version of get_register is shown in Algorithm 3.

4 Block Coloring Order

To retain the properties by SSA-based register allocation, we color basic blocks in dominance order. This still provides many valid visiting orders. We choose an order in which we color the most often executed basic blocks first while coloring paths beginning at the start block. By following the control flow along the "hot" paths, there is always one control flow predecessor colored already and we can assign ϕ-functions the same color as their operands in this predecessor.

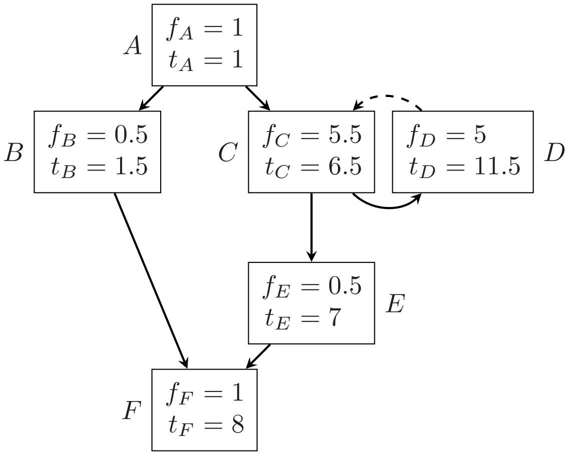


Fig. 6. A control-flow graph annotated with execution frequencies (f) and trace values (t)

Algorithm 4. Determining the block coloring order

```
proc blockorder():
    for b in reverse_postorder(blocks):
        t ← 0
        for p in control_flow_predecessors(b):
            if t < trace[p]:
                t ← trace[p]
        trace[b] ← t + frequency(b)
    order ← ∅
    for b in sort(blocks, by: trace)
        order ← add_trace(order, block)
    return order

proc add_trace(order, block):
    if not block ∈ order:
        best_trace ← 0
        best_pred ← null
        for p in preds(block)
            if backedge(p, block): continue
            if best_trace < trace[p]:
                best_trace ← trace[p]
                best_pred ← p
        if not best_pred ← null:
            order ← add_trace(order, block)
        order ← order + block
    return order
```

To determine these paths in the control flow graph, we calculate a trace value for each basic block: First we gather execution frequencies for each basic block. This can be done heuristically (cf. Wagner et al. [10]) or they can be obtained from profiling information. Using the execution frequencies, we calculate the trace value of each block: The trace value of a block is the maximum of the trace values of its control flow predecessors (disregarding back edges) plus its own execution frequency. This approximates the amount of instructions executed from the start to each block while considering that a block can be executed multiple times.

Then we select the block with the highest trace value and determine a path to the start. Before this block is colored, we color its control flow predecessor (again ignoring back edges) which has the highest trace value. In turn, we repeat this until we reach the start block. This path then is colored in reverse order. After that, we select the block with the highest trace value from the remaining uncolored blocks and again construct a path towards the start block but this time stopping at some already colored block. Again, this new path is colored in reverse order and the process is repeated until all blocks are colored. Algorithm 4 shows the procedure as pseudo code.

In the example in Figure 6 the block with the highest trace value is D, therefore we first color the path A, C, D. Of the remaining, i.e. uncolored, blocks block F has the highest trace value, so we color its path E, F (A and C are already colored). B is colored last.

5 Experimental Evaluation

We implemented the presented coalescing algorithm in the libFIRM [12] compiler. This compiler produces code for the x86 architecture and features a completely SSA-based register allocator as presented in [9]. All measurements were conducted on the integer part `CINT2000` of the `CPU2000` benchmark [13]. The program `252.eon` is missing because the compiler does not support C++. The time measurements were performed on a Core 2 Duo 2GHz PC with 2GB RAM running a Linux 2.6.24 kernel. The benchmarks mostly exercise the seven general-purpose registers of the x86. The execution frequencies were statically estimated using a Markov-chain model [10]. We compare the algorithm presented in this paper with our previous work performing colaescing by recoloring[5] after register allocation.

5.1 Compile Time

Figure 7 shows the runtime of the preference-guided assignment algorithm described in this paper running on the entire `CINT2000` benchmark set. We do not show CFGs larger than 2000 instructions because they are rare and unnecessarily scale the figure. The runtime behavior of the few CFGs not shown is consistent with those shown.

CFGs as large as 2000 instructions are processed well within 20 msecs ($\triangleq 10\mu$s per instruction) on the machine we experimented on. On average, an instruction took $6.2\mu s$ to allocate while the average speed of the recoloring approach is $14.1\mu s$. In comparison to the recoloring algorithm the approach presented here accelerates the allocation by a of factor 2.27.

5.2 Code Quality

We evaluate the quality of the produced code based on two experiments:

1. Counting the number of executed move/swap instructions in the benchmarks.
2. Measuring actual runtime of the benchmarks.

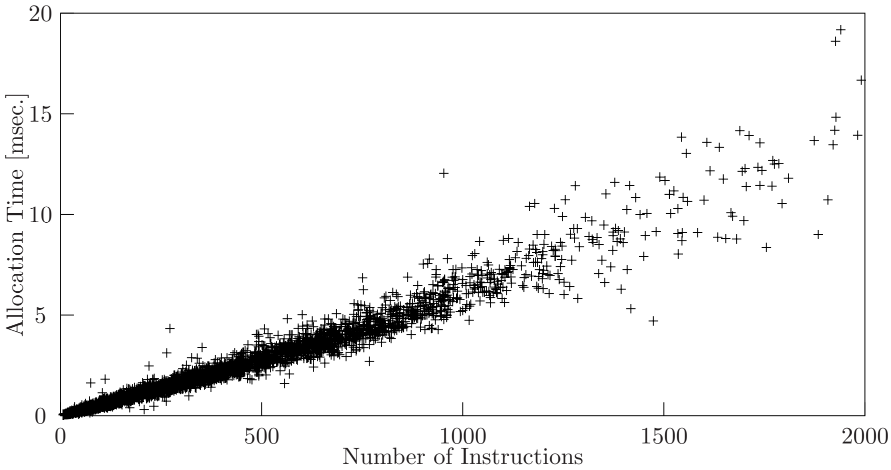


Fig. 7. Allocator runtime

Counting moves and exchanges. By instrumenting the created binaries using Valgrind [14], we counted the number of move and swap instructions in the *runs* of the benchmarks. Table 1 shows the results of counting the move/exchange instructions.

The column "No Coalescing" corresponds to not performing any sophisticated coalescing at all: For live-range splits that are due to register constraints, get_register will try to assign targets and corresponding sources at parallel moves the same register if possible. Else, no effort is made to coalesce copies.

The column "Pref. Guided" denotes the algorithm presented in this paper and "Recoloring" is the aforementioned recoloring approach. For every evaluated

Table 1. Number of executed move and swap operations in billions

Benchmark	No Coalescing			Pref. Guided			Recoloring		
	Copies	Swaps	Percent	Copies	Swaps	Percent	Copies	Swaps	Percent
164.gzip	24.1	18.3	11.76%	8.5	2.0	3.22%	5.8	0.3	1.88%
175.vpr	19.2	7.0	12.12%	11.7	1.1	6.28%	7.5	1.0	4.28%
176.gcc	16.9	7.9	14.02%	7.8	0.9	5.42%	6.5	0.4	4.37%
181.mcf	4.4	3.1	13.66%	3.3	0.0	6.67%	2.9	0.0	5.89%
186.crafty	29.0	4.9	16.30%	18.7	1.1	10.16%	17.5	1.0	9.58%
197.parser	34.4	11.9	13.19%	16.2	3.1	5.98%	13.9	1.8	4.93%
253.perlbmk	50.0	19.3	15.69%	23.3	1.4	6.23%	21.6	0.6	5.62%
254.gap	31.2	6.2	13.81%	17.2	1.4	7.40%	15.5	1.1	6.65%
255.vortex	44.0	3.8	13.11%	11.2	0.7	3.69%	9.5	0.3	3.03%
256.bzip2	34.6	9.8	14.14%	19.9	1.7	7.53%	17.0	3.1	7.01%
300.twolf	17.4	17.6	10.89%	10.3	5.6	5.25%	8.0	3.4	3.85%
Average	27.7	10.0	13.47%	13.5	1.7	5.93%	11.4	1.2	4.97%

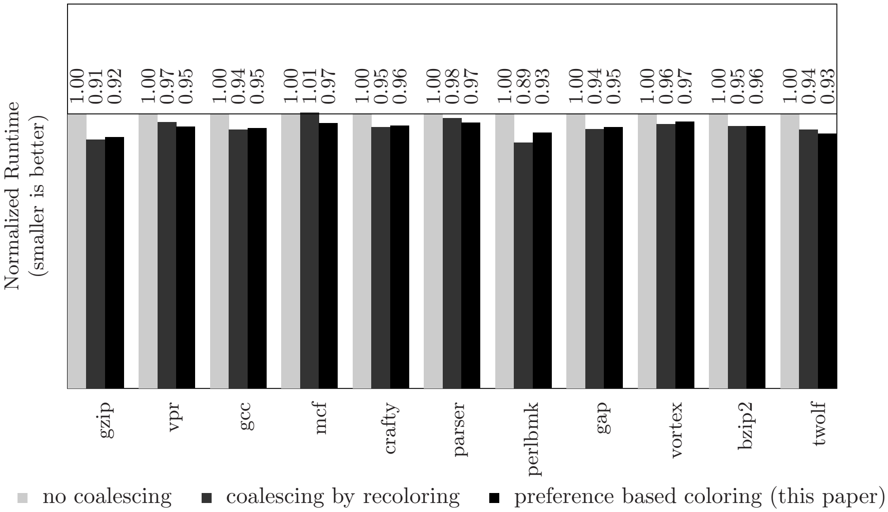


Fig. 8. SPEC `CINT2000` runtimes with different coalescing schemes

coalescing algorithm, we show the number of move/swap instructions and the percentage of all instructions being moves or exchanges. The preference-guided approach significantly reduces moves and swaps but does not reach the performance of the recoloring apporach. Performing almost no coalescing results in 13.46% of all executed instructions being moves or swaps. This number is decreased by our approach to 5.93% and to 4.97% by the recoloring technique. Hence, the code quality of the technique presented in this paper is very close to the recoloring approach which currently is one of the best conservative coalescers [5].

Runtime of the benchmarks. Figure 8 shows the runtime of the benchmarks normalized to “No Coalescing” as explained above. We see that performing coalescing *is* important and moves are not for free: The benchmark runtimes are decreased by 5%. Furthermore, the preference-guided approach is on par with the recoloring technique. Between those two, there is no clear winner. However, we suspect (without having verified this claim) that a smaller CPU with less pipelines and no out-of-order scheduling is more susceptible to register moves. Therefore, the recoloring approach might produce faster programs on such systems.

Finally, to show that our compiler produces high-quality results and the SSA-based register allocation technique is competetive, we compare the benchmark runtimes against those produced by GCC 4.2.4 and LLVM 2.5. libFirm has the smallest code base among these compilers and performs only a subset of the optimizations the others do. All compilers ran on maximum optimization level and had machine-dependent optimizations for the benchmarking machine

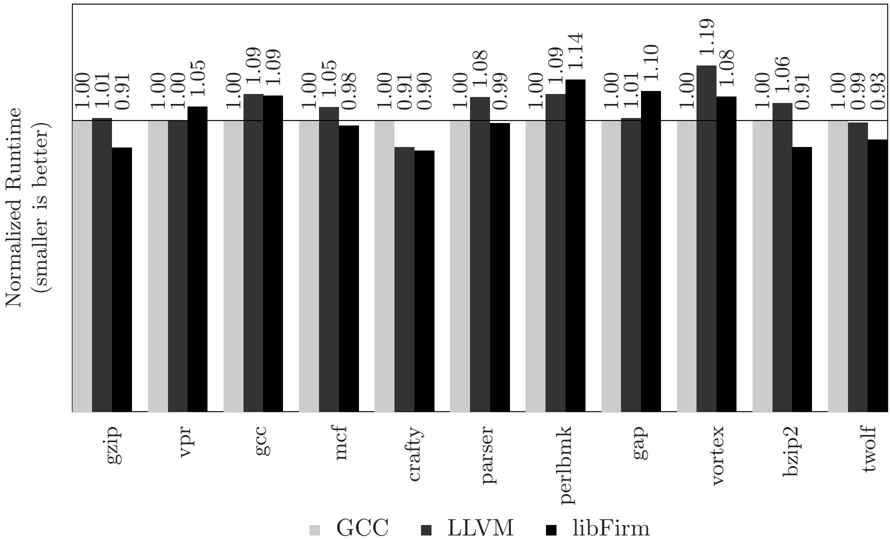


Fig. 9. SPEC `CINT2000` runtimes relative to GCC and LLVM

(see above) turned on[4]. As can be seen in Figure 9 the runtime of the benchmark programs produced by our compiler is on par with the others.

6 Related Work

Graph-based approaches. The first graph-coloring allocator due to Chaitin et al. [1] used *aggressive coalescing* and did not make any effort at all to respect the chromatic number of the graph. Since then, a lot of work was done on safe coalescing. Briggs et al. [15] introduced *conservative coalescing.* To decide whether an affinity can be coalesced, they considered the degree of the resulting coalesced node. Only if that node's degree was lower than k, the copy was coalesced. George and Appel's *iterated coalescing* [16] improves upon conservative coalescing by applying Briggs et al.'s criterion and a new one iteratively to the graph. Park and Moon [17] left the road of safe coalescing and improved upon the aggressive scheme.

Live-range splitting. Live-range splitting has often been proposed to aid coloring. To our knowledge, Fabri [18] was first to observe this. Appel and George [19] presented an ILP approach to reduce the register pressure everywhere to k by allowing every live range being split at every program point. Lueh et al.'s fusion based allocator [20] integrates live-range splitting into the register allocator. They start by building the interference graphs of certain regions (that can be basic blocks, loops, traces, etc.) that are not imposed by the allocator but can be

[4] `-O3 -fomit-frame-pointer -march=native`

chosen by the compiler writer. In a later step, the interference graphs are fused to form the complete interference graph. During this fusion process, live ranges can be split or spilled if the fused interference graph was no longer colorable. Recently, Nakaike et al. [21] proposed a dynamic approach that splits around basic blocks and uses coalescing to unify split live-ranges in hot code regions.

Linear-scan allocators. Wimmer and Mössenböck [22] give a highly tuned extension of Traub's version [23] of linear scan. Their *register hints* is a similar technique to our preference propagation for ϕ-functions. Furthermore, they can take register constraints into account. Recently, Sarkar and Barik [24] introduced more aggressive live-range splitting to linear scan allocators however without performing coalescing.

SSA-based register allocation. Budimlic et al. [25] pioneered in coalescing on SSA-form programs already using many properties that SSA-based register allocation relies on. However, they are only concerned with aggressive coalescing. In 2005, three groups [2, 4, 26] independently from each other discovered that the interference graphs of SSA-form programs are chordal. All yet published coalescing techniques tailored to SSA-based register allocation use interference graphs.

Bouchez et al. [3] investigate the theoretic background of coalescing. They show that coalescing is NP-complete concerning the number of affinities, also in the SSA-based setting. Later, Bouchez et al. proposed several extensions to conservative coalescing [27]. Brisk [28] presents a biased coloring algorithm for chordal graphs. Hack et al. [4, 5] present two approaches based on recoloring: First, the program is colored using the standard algorithm presented in Section 2. Then, the color assignment is changed by assigning move-related nodes the same color. Color clashes are resolved recursively through the graph.

Pereira and Palsberg [29] consider the problem of subregisters. In this setting, optimal allocation even inside a basic block is NP-complete. Therefore, they split live ranges after every program point and allocate each instruction separately. In doing so, they process the program points in dominance order and perform coalescing only along dominance order. Especially, moves on loop back edges are not coalesced.

7 Conclusions

In this paper, we presented an SSA-based register assignment algorithm that uses register preferences to bias the register assignment in order to reduce shuffle code. In doing so, we do not need a separate coalescing pass in the register allocator. Furthermore, building the interference graph, which is considered a red rag for just-in-time compilation, is no longer necessary. Compared to a state-of-the art coalescing technique, our algorithm gives competitive results while reducing the runtime of the register allocation by a factor of 2.27.

Acknowledgements. We thank Michael Beck, Alain Darte, Gerhard Goos, Daniel Grund, Fabrice Rastello, Jan Reineke, and Christian Würdig for several insightful discussions. Furthermore, we thank the anonymous reviewers for their valuable comments.

References

1. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register allocation via graph coloring. Journal of Computer Languages 6, 45–57 (1981)
2. Brisk, P., Dabiri, F., Jafari, R., Sarrafzadeh, M.: Optimal Register Sharing for High-Level Synthesis of SSA Form Programs. IEEE Trans. on CAD of Integrated Circuits and Systems 25(5), 772–779 (2006)
3. Bouchez, F., Darte, A., Rastello, F.: On the Complexity of Register Coalescing. In: CGO, San Jose, USA. IEEE Computer Society Press, Los Alamitos (2007)
4. 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)
5. Hack, S., Goos, G.: Copy Coalescing by Graph Recoloring. In: PLDI, pp. 227–237. ACM Press, New York (2008)
6. Braun, M., Hack, S.: Register Spilling and Live-Range Splitting for SSA-Form Programs. In: de Moor, O., Schwartzbach, M.I. (eds.) CC 2009. LNCS, vol. 5501, pp. 174–189. Springer, Heidelberg (2009)
7. Morgan, R.: Building an Optimizing Compiler. Digital Press, Newton (1998)
8. Paleczny, M., Vick, C., Click, C.: The Java HotSpot$^{\mathrm{TM}}$Server Compiler. In: Proceedings of the Java$^{\mathrm{TM}}$ Virtual Machine Research and Technology Symposium (JVM 2001) (April 2001)
9. Hack, S.: Register Allocation for Programs in SSA Form. PhD thesis, Universität Karlsruhe (October 2007)
10. Wagner, T.A., Maverick, V., Graham, S.L., Harrison, M.A.: Accurate Static Estimators for Program Optimization. In: PLDI, pp. 85–96. ACM, New York (1994)
11. Boissinot, B., Darte, A., Dupont de Dinechin, B., Guillon, C., Rastello, F.: Revisiting out-of-SSA translation for correctness, code quality, and efficiency. In: CGO, pp. 114–125. IEEE Computer Society Press, Los Alamitos (2009); Best paper award
12. The libFirm Compiler, `http://www.libfirm.org`
13. Standard Performance Evaluation Corporation: SPEC CPU2000 V1.3
14. Valgrind Instrumentation Framework for Building Dynamic Analysis Tools, `http://www.valgrind.org`
15. Briggs, P., Cooper, K.D., Torczon, L.: Improvements to Graph Coloring Register Allocation. TOPLAS 16(3), 428–455 (1994)
16. George, L., Appel, A.W.: Iterated Register Coalescing. TOPLAS 18(3), 300–324 (1996)
17. Park, J., Moon, S.M.: Optimistic Register Coalescing. ACM Transactions on Programming Languages and Systems 26(4), 735–765 (2004)
18. Fabri, J.: Automatic Storage Optimization. In: SIGPLAN 1979: Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction, pp. 83–91. ACM Press, New York (1979)

19. Appel, A.W., George, L.: Optimal Spilling for CISC Machines with Few Registers. In: ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, June 2001, pp. 243–253 (2001)
20. Lueh, G.Y., Gross, T., Adl-Tabatabai, A.R.: Fusion-based Register Allocation. ACM Transactions on Programming Languages and Systems 22(3), 431–470 (2000)
21. Nakaike, T., Inagaki, T., Komatsu, H., Nakatani, T.: Profile-based Global Live-Range Splitting. In: PLDI 2006: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 216–227. ACM Press, New York (2006)
22. Wimmer, C., Mössenböck, H.: Optimized interval splitting in a linear scan register allocator. In: VEE 2005: Proceedings of the 1st ACM/USENIX international Conference on Virtual Execution Environments, pp. 132–141. ACM Press, New York (2005)
23. Traub, O., Holloway, G., Smith, M.D.: Quality and speed in linear-scan register allocation. In: PLDI 1988: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, pp. 142–151. ACM Press, New York (1998)
24. Sarkar, V., Barik, R.: Extended linear scan: An alternate foundation for global register allocation. In: Krishnamurthi, S., Odersky, M. (eds.) CC 2007. LNCS, vol. 4420, pp. 141–155. Springer, Heidelberg (2007)
25. Budimlić, Z., Cooper, K.D., Harvey, T.J., Kennedy, K., Oberg, T.S., Reeves, S.W.: Fast copy coalescing and live-range identification. In: PLDI, pp. 25–32. ACM Press, New York (2002)
26. Bouchez, F., Darte, A., Guillon, C., Rastello, F.: Register Allocation: What Does the NP-Completeness Proof of Chaitin et al. Really Prove? Or Revisiting Register Allocation: Why and How? In: Almási, G.S., Caşcaval, C., Wu, P. (eds.) KSEM 2006. LNCS, vol. 4382, pp. 283–298. Springer, Heidelberg (2007)
27. Bouchez, F., Darte, A., Rastello, F.: Advanced Conservative and Optimistic Register Coalescing. In: CASES, pp. 147–156 (2008)
28. Brisk, P., Verma, A.K., Ienne, P.: An Optimistic and Conservative Register Assignment Heuristic for Chordal Graphs. In: CASES, pp. 209–217 (2007)
29. Pereira, F., Palsberg, J.: Register Allocation by Puzzle Solving. In: PLDI, pp. 216–226. ACM, New York (2008)

Validating Register Allocation and Spilling

Silvain Rideau[1] and Xavier Leroy[2]

[1] École Normale Supérieure, 45 rue d'Ulm, 75005 Paris, France
silvain.rideau@ens.fr
[2] INRIA Paris-Rocquencourt, BP 105, 78153 Le Chesnay, France
xavier.leroy@inria.fr

Abstract. Following the translation validation approach to high-assurance compilation, we describe a new algorithm for validating *a posteriori* the results of a run of register allocation. The algorithm is based on backward dataflow inference of equations between variables, registers and stack locations, and can cope with sophisticated forms of spilling and live range splitting, as well as many architectural irregularities such as overlapping registers. The soundness of the algorithm was mechanically proved using the Coq proof assistant.

1 Introduction

To generate fast and compact machine code, it is crucial to make effective use of the limited number of registers provided by hardware architectures. Register allocation and its accompanying code transformations (spilling, reloading, coalescing, live range splitting, rematerialization, etc) therefore play a prominent role in optimizing compilers.

As in the case of any advanced compiler pass, mistakes sometimes happen in the design or implementation of register allocators, possibly causing incorrect machine code to be generated from a correct source program. Such compiler-introduced bugs are uncommon but especially difficult to exhibit and track down. In the context of safety-critical software, they can also invalidate all the safety guarantees obtained by formal verification of the source code, which is a growing concern in the formal methods world.

There exist two major approaches to rule out incorrect compilations. *Compiler verification* proves, once and for all, the correctness of a compiler or compilation pass, preferably using mechanical assistance (proof assistants) to conduct the proof. *Translation validation* checks *a posteriori* the correctness of one run of compilation: a *validator*, conceptually distinct from the compiler itself, is given the intermediate code before and after a compilation pass, and verifies that they behave identically using static analysis or (specialized) theorem proving technology [1,2,3,4]. For additional confidence, the validator can itself be mechanically verified once and for all; this provides soundness guarantees as strong as compiler verification and reduces the amount of compiler code that needs to be proved correct, at the expense of weaker completeness guarantees [5].

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 224–243, 2010.

This paper describes a new algorithm to validate (in one pass) register allocation plus splitting, reloading, coalescing, live range splitting, dead code elimination, and enforcement of calling conventions and architectural constraints on registers. This algorithm is based on a backward dataflow analysis that refines standard liveness analysis. It comes accompanied with a machine-checked proof of soundness, conducted using the Coq proof assistant [6,7]. Our algorithm improves on an earlier algorithm by Huang, Childers and Soffa [8] because it is mechanically proved and because it can deal with overlapping registers. (See section 6 for a discussion.)

This work is part of the CompCert project, which aims at formally verifying a realistic optimizing compiler for the C language, usable in the context of critical embedded systems [9]. Currently, CompCert follows the compiler verification approach for its register allocation and spilling/reloading passes. While the verified register allocator is a state-of-the-art George-Appel graph coloring allocator [10], the spilling strategy that was proved correct is very naive: it inserts spills after every definition and reloads before every use of a temporary that could not be allocated to a register, reserving some registers specially for this purpose [11, section 11]. This strategy is adequate for a register-rich target architecture such as the PowerPC, but more sophisticated strategies are needed to retarget CompCert to a register-poor architecture like x86. Proving those sophisticated strategies is a daunting task. The verified validation algorithm presented in this paper offers an attractive alternative, reducing the amount of code that needs to be proved and enabling the use of advanced spilling strategies. Moreover, we can experiment with various register allocation algorithms and spilling strategies without having to re-do any proofs.

The remainder of this paper is organized as follows. Section 2 outlines the source and target languages for the untrusted register allocator and characterizes the code transformations it is allowed to make. Section 3 describes our validation algorithm. Section 4 sketches its proof of soundness. Section 5 discusses experience gained with a prototype implementation. Related work is reviewed in section 6, followed by concluding remarks in section 7.

2 A Bird's Eye View of Register Allocation and Spilling

2.1 Source Language

As input for register allocation, we consider the RTL intermediate language of the CompCert compiler [11, section 6]. This is a standard Register Transfer Language where control is represented by a control flow graph (CFG). Each node of a CFG carries an abstract instruction, corresponding roughly to one machine instruction but operating over variables x (also called temporaries) instead of hardware registers. Every function has an unlimited supply of variables and their values are preserved across function calls. Each variable has a machine type comprising a register class (typically, `int` or `float`) and a bit size (8, 16, 32, 64).

Control-flow graphs:

$$\begin{array}{lll} g & ::= p \mapsto I & \text{finite map} \end{array}$$

CFG nodes:

$$\begin{array}{ll} p, s & \in N \end{array}$$

RTL instructions:

$$\begin{array}{lll}
I & ::= \mathtt{nop}(s) & \text{no operation} \\
 & \mid \mathtt{op}(op, \vec{x}, x_d, s) & \text{arithmetic operation} \\
 & \mid \mathtt{load}(\kappa, mode, \vec{x}, x_d, s) & \text{memory load} \\
 & \mid \mathtt{store}(\kappa, mode, \vec{x}, x_s, s) & \text{memory store} \\
 & \mid \mathtt{call}(\tau, id, \vec{x}, x_d, s) & \text{function call} \\
 & \mid \mathtt{cond}(cond, \vec{x}, s_{true}, s_{false}) & \text{conditional branch} \\
 & \mid \mathtt{return}(x) & \text{function return}
\end{array}$$

Each RTL instruction carries the list of its successors s in the CFG. For example, $\mathtt{nop}(s)$ performs no computation and continues at node s, like an unconditional branch. $\mathtt{op}(op, \vec{x}, x_d, s)$ applies the arithmetic operation op (taken from a machine-dependent set of operators) to the values of variables $\vec{x}$, stores the result in variable x_d, and continues at s. $\mathtt{load}(\kappa, mode, \vec{x}, x_d, s)$ loads a memory quantity κ (e.g. "8-byte signed integer" or "64-bit float") from an address determined by applying addressing mode $mode$ to the values of registers $\vec{x}$, stores the result in x_d, and continues at s. $\mathtt{store}(\kappa, mode, \vec{x}, x_s, s)$ is similar, except that the value of x_s is stored at the computed address instead. $\mathtt{cond}(cond, \vec{x}, s_{true}, s_{false})$ evaluates the boolean condition $cond$ over the values of $\vec{x}$ and continues at s_{true} or s_{false} depending on the result. $\mathtt{return}(x)$ terminates the current function, returning the value of x as the result. Finally, $\mathtt{call}(\tau, id, \vec{x}, x_d, s)$ calls the function named id, giving it the values of $\vec{x}$ as arguments and storing the returned result in x_d. The τ parameter is the type signature of the call, specifying the number and types of arguments and results: this is used during register allocation to determine the calling conventions for the call. The full RTL language, described in [11], supports additional forms of function calls such as calls through a function pointer and tail calls, which we omit here for simplicity.

RTL functions:

$$\begin{array}{l}
f ::= \{\mathtt{name} = id;\ \mathtt{typesig} = \tau;\ \mathtt{params} = \vec{x}; \\
\qquad\quad \mathtt{code} = g;\ \mathtt{entrypoint} = p\}
\end{array}$$

An RTL function is defined by its name, its type signature, the list of parameter variables, a CFG, and a node in the CFG that corresponds to the function entry point.

2.2 Target Language

The purpose of register allocation is to transform RTL functions into LTL functions. LTL stands for "Location Transfer Language" and is a minor variation on RTL where variables are replaced by *locations*. A location is either a machine register r or a slot $S(\delta, n)$ in the activation record of the function; δ is the byte offset and n the byte size of the slot.

Locations:

$$\begin{array}{lll} \ell & ::= r & \text{machine register} \\ & \mid S(\delta, n) & \text{stack slot} \end{array}$$

Control-flow graphs:

$$g' ::= p \mapsto I'$$

LTL instructions:

$$\begin{array}{lll} I' & ::= \mathtt{nop}(s) & \text{no operation} \\ & \mid \mathtt{op}(op, \vec{\ell}, \ell, s) & \text{arithmetic operation} \\ & \mid \mathtt{load}(\kappa, mode, \vec{\ell}, \ell_d, s) & \text{memory load} \\ & \mid \mathtt{store}(\kappa, mode, \vec{\ell}, \ell_s, s) & \text{memory store} \\ & \mid \mathtt{call}(\tau, id, s) & \text{function call} \\ & \mid \mathtt{cond}(cond, \vec{\ell}, s_{true}, s_{false}) & \text{conditional branch} \\ & \mid \mathtt{return} & \text{function return} \end{array}$$

LTL functions:

$$\begin{array}{ll} f' ::= & \{\mathtt{name} = id;\ \mathtt{typesig} = \tau; \\ & \quad \mathtt{code} = g';\ \mathtt{entrypoint} = p\} \end{array}$$

Most LTL instructions are identical to RTL instructions modulo the replacement of variables x by locations ℓ. However, function calls and returns are treated differently: the locations of arguments and results are not marked in the `call` and `return` instructions nor in the `params` field of functions, but are implicitly determined by the type signature of the call or the function, following the calling conventions of the target platform. We model calling conventions by the following three functions:

- `arguments`(τ): the list of locations for the arguments of a call to a function with signature τ. The LTL code is responsible for moving the values of the arguments to these locations (registers or stack slots) before the `call` instruction.
- `parameters`(τ): the list of locations for the parameters of a function with signature τ. On entrance, the LTL function expects to find the values of its arguments at these locations, and is responsible for moving them to other locations if desired. `parameters`(τ) is usually identical to `arguments`(τ) modulo relocation of stack slot offsets.
- `result`(τ): the location used to pass the return value for a function with signature τ.

2.3 The Effect of Register Allocation on the Code

The essence of register allocation is to replace variables by the locations that were assigned to it in each instruction of the source RTL code, leaving the rest of the instruction unchanged. For example, the RTL instruction $\mathtt{op}(\mathtt{add}, x.y, z, s)$ can become the LTL instruction $\mathtt{op}(\mathtt{add}, \mathtt{EAX}.\mathtt{EBX}, \mathtt{EAX}, s)$ if the allocator decided to assign x and z to register `EAX` and y to register `EBX` at this program point. However, this is not the only effect of register allocation on the code: it can also insert or delete some instructions in the following cases.

- **Spilling:** a move from a register r to a stack slot is inserted at some point after an instruction that assigns r, to save the result value on the stack and free the register r for other uses.
- **Reloading:** symmetrically, a move from a stack slot to a register is inserted at some point before a use of r.
- **Coalescing:** some variable copies $\texttt{op}(\texttt{move}, x, y, s)$ present in the input RTL code may disappear if the register allocator assigned the same location to x and y. We model this deletion as replacing the $\texttt{op}(\texttt{move}, \ldots)$ instruction by a `nop` instruction.
- **Live range splitting:** if the allocator decided to split a live range of a variable x into several variables $x_1, \ldots, x_n$ connected by `move` instructions, some of these moves may remain in the generated LTL code as newly inserted instructions.
- **Enforcement of calling conventions:** additional moves may be inserted in the generated LTL code to deposit arguments to function calls and return values of functions in the locations dictated by the calling conventions, and fetch function parameters and return values from these locations.
- **Enforcement of architectural constraints:** the register allocator can also introduce move instructions to work around irregularities of the target architecture: two-address instructions, special registers, etc.
- **Dead code elimination:** the register allocator can also eliminate side effect-free instructions such as `op` and `load` whose result variables are never used. Dead code elimination can be performed in a separate pass prior to register allocation, but the availability of liveness information during register allocation makes it convenient to perform dead code elimination at the same time.

The validation algorithm we present next is able to cope with all these modifications of the code performed during register allocation. Other code transformations that sometimes accompany register allocation, such as rematerialization, are discussed in section 7.

3 The Validation Algorithm

Like intraprocedural register allocation itself, the validator proceeds function per function. It takes as input an RTL function f, the corresponding LTL function f' produced by the untrusted register allocator, and a partial map φ from the CFG nodes of f' to those of f.

The purpose of φ is to connect the computational instructions of the LTL code back to the corresponding instructions in the original RTL. Since deleted instructions are not actually removed but simply turned into LTL `nop` instructions, φ also maps these `nop` instructions back to the corresponding deleted RTL instruction. Finally, LTL `move` instructions that were inserted during register allocation are not in the domain of φ, indicating that they are new. (All these properties of φ are checked during validation.) We assume that the register allocator has been lightly instrumented to produce this mapping φ and give it as additional argument to our validator.

The validation algorithm proceeds in two steps:

– A set of structural checks (section 3.1) verifies that the computational instructions in the two CFGs match properly, that their successors agree, and that the φ mapping is consistent.
– A backward dataflow analysis (section 3.2) establishes that the same values flow in both CFGs.

The combination of these two steps suffices to ensure that the two functions f and f' behave identically at run-time (as proved in section 4).

3.1 Structural Checks

The main structural check is performed on each pair of RTL instructions and LTL instructions that match according to the φ mapping. For each mapping $p' \mapsto p$ in φ, the validator calls the following check_instr predicate:

```
check_instr($f, f', \varphi, p, p'$) =
    let $I = f.\mathtt{code}(p)$ and $I' = f'.\mathtt{code}(p')$ in
    let $s_1, \ldots, s_n$ be the successors of $I$
    and $s'_1, \ldots, s'_m$ be the successors of $I'$ in
    $I$ and $I'$ are structurally similar
    and path($f', \varphi, s_i, s'_i$) for $i = 1, \ldots, n$
```

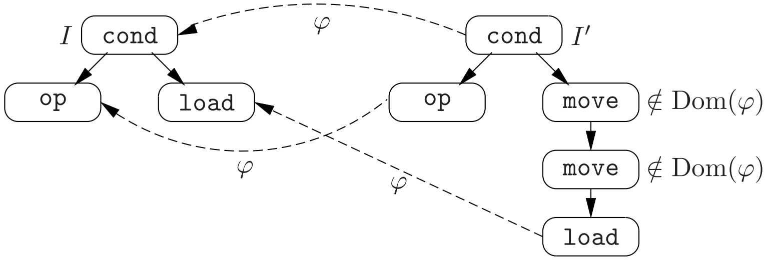
An RTL instruction I is *structurally similar* to an LTL instruction I' if they are identical modulo changes of successors and replacement of registers by locations, or if I is an op or load and I is a nop (dead code elimination). Table 1 gives a more precise definition. The $\sim$ relation (pronounced "agree") between a variable x and a location ℓ means that x and ℓ agree in register class and in size. For example, a variable of class int and size 32 bits agrees with the x86 register EAX

Table 1. Structural similarity between RTL and LTL instructions

Instruction I	Instruction I'	Condition
$\mathtt{nop}(s)$	$\mathtt{nop}(s')$	
$\mathtt{op}(op, \vec{x}, x, s)$	$\mathtt{op}(op, \vec{\ell}, \ell, s')$	if $\vec{x} \sim \vec{\ell}$ and $x \sim \ell$
$\mathtt{op}(op, \vec{x}, x, s)$	$\mathtt{nop}(s')$	
$\mathtt{load}(\kappa, mode, \vec{x}, x, s)$	$\mathtt{load}(\kappa, mode, \vec{\ell}, \ell, s')$	if $\vec{x} \sim \vec{\ell}$ and $x \sim \ell$
$\mathtt{load}(\kappa, mode, \vec{x}, x, s)$	$\mathtt{nop}(s')$	
$\mathtt{store}(\kappa, mode, \vec{x}, x, s)$	$\mathtt{store}(\kappa, mode, \vec{\ell}, \ell, s')$	if $\vec{x} \sim \vec{\ell}$ and $x \sim \ell$
$\mathtt{call}(\tau, id, \vec{x}, x, s)$	$\mathtt{call}(\tau, id)$	
$\mathtt{cond}(cond, \vec{x}, s_1, s_2)$	$\mathtt{cond}(cond, \vec{\ell}, s'_1, s'_2)$	if $\vec{x} \sim \vec{\ell}$
$\mathtt{return}(x)$	$\mathtt{return}$	

and the stack slot $S(0, 4)$, but not with the register AX (wrong size) nor with the register XMM0 (wrong class) nor with the stack slot $S(0, 8)$ (wrong size).

Besides structural similarity, check_instr also verifies the consistency of the successors of the two instructions I and I'. Naively, if φ maps the program point of I' to the program point of I, one could expect that the i-th successor of I' is mapped to the i-th successor of I. In the example below, this is the case for the left successors of the cond instructions.



This is not always the case because of the fresh move instructions that can be inserted during register allocation. However, there must exist a (possibly empty) path from the i-th successor of I' to a CFG node that is mapped to the i-th successor of I. This path must consist of move instructions that are not in the domain of φ. (See example above, right successors of the cond instructions.) This condition is checked by the auxiliary predicate path:

```
path($f', \varphi, p, p'$) =
    false if the node $p'$ was previously visited;
    true if $\varphi(p') = p$;
    path($f', \varphi, p, s'$) if $p' \notin \mathrm{Dom}(\varphi)$ and $f'$.code($p'$) = op(move, _, _, $s'$);
    false, otherwise.
```

Besides calling check_instr on each pair (p, p') of matching program points, the structural check pass also verifies that the two functions f, f' agree in name and type signature, and that there exists a valid path (in the sense above) from the entry point of f' to a point that maps to the entry point of f. (Typically, this path corresponds to move instructions that shuffle the parameters of the function.)

```
check_structure($f, f', \varphi$) =
    $f$.name = $f'$.name and $f$.typesig = $f'$.typesig
    and path($f', \varphi, f$.entrypoint, $f'$.entrypoint)
    and for each $p, p'$ such that $\varphi(p) = p'$,
        check_instr($f, f', \varphi, p, p'$)
```

There is one last family of structural checks that we omitted here: enforcement of architectural constraints on the uses of locations. In the case of a RISC load-store architecture, for instance, argument and result locations must be hardware registers for all instructions except move operations, for which one of the source and destination can be a stack slot, but not both. CISC architectures like the x86 tolerate stack slots as arguments or results of some operations, but impose other constraints such as the result location being identical to the first argument location in the case of two-address instructions. These checks can be performed either during validation or as part of a later compiler pass; we omit them for simplicity.

3.2 Dataflow Analysis

To show that the original RTL function f and the register-allocated LTL function f' compute the same results and have the same effects on memory, we use a dataflow analysis that associates to each program point p' of f' a set $E(p')$ of equations between variables and locations:

$$E(p') = \{x_1 = \ell_1; \ldots; x_n = \ell_n\}$$

The semantic meaning of these equations is that in every execution of the code, the value of x_i at point $\varphi(p')$ in f is equal to the value of ℓ_i at point p' in f'.

There are two ways to build and exploit these sets of instructions: the forward way and the backward way. For concreteness, assume that we have structurally-similar `op` instructions at points p' in f' and $p = \varphi(p')$ in f:

$$f.\mathtt{code}(p) = \mathtt{op}(op, \vec{x}, x, s) \qquad f'.\mathtt{code}(p') = \mathtt{op}(op, \vec{\ell}, \ell, s')$$

These instructions use $\vec{x}$ and $\vec{\ell}$ and define x and ℓ, respectively.

In the forward approach, we assume given a set E of variable-location equations that hold "before" points p, p'. We can then check that $\{\vec{x} = \vec{\ell}\} \subseteq E$. If so, we know that both `op` instructions are applied to the same argument values, and since the operator op is the same in both instructions, they will compute the same result value and store it in x and ℓ. To obtain the equations that hold "after" these instructions, we remove from E all equations invalidated by the parallel assignment to x and ℓ (see below for a discussion), then add the equation $x = \ell$.

In the backward approach, we are given a set E of equations that must hold "after" points p, p' for the rest of the executions of f, f' to produce identical results and effects. We first check that the assignment to x and ℓ performed by the two op instructions does not render unsatisfiable any of the equations in E. If this check succeeds, we can remove the equation $x = \ell$ from E, since it is being satisfied by the parallel execution of the two `op` instructions, then add the equations $\{\vec{x} = \vec{\ell}\}$, since these are necessary for the two `op` instructions to produce the same result value. This gives us the set of equations that must hold "before" points p, p'.

In this work, we adopt the backward approach, as it tends to produce smaller sets of equations than the forward approach, and therefore runs faster. (To build an intuition, consider a long, straight-line, single-assignment sequence of instructions: the forward approach produces sets whose cardinal grows linearly in the number of instructions, while the backward approach produces sets whose cardinal is only proportional to the length of the live ranges.)

Unsatisfiability and Overlap. We mentioned the need to check that assigning in parallel to x and ℓ does not render unsatisfiable any equation in a set E. An example of this situation is $E = \{y = \ell\}$ where $x \neq y$. The LTL-side *op* instruction overwrites ℓ with a statically-unknown value, while the RTL-side *op* instruction leaves y unchanged. Therefore, there is no way to statically ensure that $y = \ell$ after executing these two instructions. In register allocation terms, this situation typically occurs if the allocator wrongly assigned ℓ to both x and y, despite x and y being simultaneously live and not being copies of one another.

The determination of unsatisfiable equations is made more complicated by the fact that LTL locations can *overlap*, i.e. share some bits of storage. Two overlapping locations contain a priori different values, yet assigning to one changes the value of the other. Overlap naturally occurs with stack slots: for instance, the slots $S(0,8)$ (eight bytes at offset 0) and $S(4,4)$ (four bytes at offset 4) clearly overlap. Some processor architectures also exhibit overlap between registers. For example, on the x86 architecture, the 64-bit register `RAX` contains a 32-bit sub-register `EAX`, a 16-bit sub-register `AX`, and two 8-bit sub-registers `AL` and `AH`. All these registers overlap pairwise except `AL` and `AH`. In summary, for two locations ℓ_1 and ℓ_2 there are three mutually-exclusive possibilities:

- Equality (written $\ell_1 = \ell_2$): both locations always contain the same value.
- Disjointness (written $\ell_1 \perp \ell_2$): assigning to one location does not change the value of the other.
- Partial overlap (written $\ell_1 \# \ell_2$): the values of the locations are a priori different, yet assigning to one affects the value of the other.

For stack slots, we have the following definitions:

$$S(\delta_1, n_1) = S(\delta_2, n_2) \iff \delta_1 = \delta_2 \wedge n_1 = n_2$$
$$S(\delta_1, n_1) \perp S(\delta_2, n_2) \iff [\delta_1, \delta_1 + n_1) \cap [\delta_2, \delta_2 + n_2) = \emptyset$$

For registers, the precise definitions of $\perp$ depends on the target architecture.

Armed with these notions of overlap and disjointness, we can formally define the compatibility between a pair x, ℓ of destinations and a set of equations E, written $(x, \ell) \perp E$:

$$(x, \ell) \perp E \overset{\text{def}}{=} \forall (x' = \ell') \in E, \quad (x' = x \wedge \ell' = \ell) \vee (x' \neq x \wedge \ell' \perp \ell)$$

Note that if $(x, \ell) \perp E$ holds, assigning in parallel the same value to x and ℓ will satisfy the equation $x = \ell$ and preserve the satisfiability of all other equations appearing in E. (See lemma 2 in section 4.)

The Transfer Function. In preparation for a backward dataflow analysis, we now define the transfer function $\mathtt{transfer}(f, f', \varphi, p', E)$ that computes the set E' of equations that must hold "before" program point p' in order for the equations E to hold "after" point p'. Here, E and E' range over sets of equations plus the symbolic constant $\top$ denoting inconsistency, or in other words the fact that the analysis failed to validate the flow of data.

```
$\texttt{transfer}(f, f', \varphi, p', E) =$
    if $E = \top$ then $\top$                                                                    (1)
    else if $\varphi(p') = p$ then:
        if $f.\texttt{code}(p) = \texttt{nop}(\_)$ and $f'.\texttt{code}(p') = \texttt{nop}(\_)$:                                  (2)
            $E$
        if $f.\texttt{code}(p) = \texttt{op}(\texttt{move}, x_s, x_d, \_)$ and $f'.\texttt{code}(p') = \texttt{nop}(\_)$:                 (3)
            $E[x_d \leftarrow x_s]$
        if $f.\texttt{code}(p) = \texttt{op}(\_, \vec{x}, x, \_)$ or $\texttt{load}(\_, \_, \vec{x}, x, \_)$ and $f'.\texttt{code}(p') = \texttt{nop}(\_)$:   (4)
            if $(x = \_) \in E$ then $\top$ else $E$
        if $f.\texttt{code}(p) = \texttt{op}(\_, \vec{x}, x, \_)$ and $f'.\texttt{code}(p') = \texttt{op}(\_, \vec{\ell}, \ell, \_)$                 (5)
        or $f.\texttt{code}(p) = \texttt{load}(\_, \_, \vec{x}, x, \_)$ and $f'.\texttt{code}(p') = \texttt{load}(\_, \_, \vec{\ell}, \ell, \_)$:
            if $(x, \ell) \perp E$ then $(E \setminus \{x = \ell\}) \cup \{\vec{x} = \vec{\ell}\}$ else $\top$
        if $f.\texttt{code}(p) = \texttt{store}(\_, \_, \vec{x}, x, \_)$ and $f'.\texttt{code}(p') = \texttt{store}(\_, \_, \vec{\ell}, \ell, \_)$:       (6)
            $E \cup \{x = \ell\} \cup \{\vec{x} = \vec{\ell}\}$
        if $f.\texttt{code}(p) = \texttt{call}(\_, \_, \vec{x}, x, \_)$ and $f'.\texttt{code}(p') = \texttt{call}(\tau, \_)$:                 (7)
            if $(x, \texttt{result}(\tau)) \perp E$ and $E$ does not mention caller-save locations
            then $(E \setminus \{x = \texttt{result}(\tau)\}) \cup \{\vec{x} = \texttt{arguments}(\tau)\}$
            else $\top$
        if $f.\texttt{code}(p) = \texttt{cond}(\_, \vec{x}, \_, \_)$ and $f'.\texttt{code}(p') = \texttt{cond}(\_, \vec{\ell}, \_, \_)$:             (8)
            $E \cup \{\vec{x} = \vec{\ell}\}$
        if $f.\texttt{code}(p) = \texttt{return}(x)$ and $f'.\texttt{code}(p') = \texttt{return}$:                          (9)
            $\{x = \texttt{result}(f'.\texttt{typesig})\}$
    else if $p' \notin \mathrm{Dom}(\varphi)$ then:
        if $f'.\texttt{code}(p') = \texttt{op}(\texttt{move}, \ell_s, \ell_d, \_)$:                                         (10)
            $E[\ell_d \leftarrow \ell_s]$
```

Fig. 1. The transfer function for backward dataflow analysis

The transfer function is defined in figure 1. We now explain its various cases. First, inconsistency propagates up, therefore $E' = \top$ if $E = \top$ (case 1). Then, we discuss whether the instruction at p' in f' is mapped to a structurally-similar instruction at p in f (i.e. $\varphi(p') = p$) or is new (i.e. $p' \notin \mathrm{Dom}(\varphi)$).

If $\varphi(p') = p$, we examine the shape of the two similar instructions. For instructions that perform no definitions, such as `store` and `cond`, we simply add equations $\{x_i = \ell_i\}$ to E, where $x_1, \ldots, x_n$ are the uses of the RTL instruction and $\ell_1, \ldots, \ell_n$ those of the LTL instruction (cases 6 and 8). These equations must be satisfied "before" for the two instructions to behave the same.

For instructions that define a variable x or a location ℓ, such as `op` and `load`, we first check compatibility between (x, ℓ) and E, and return $\top$ if false; for in this case there is no way to ensure that the equations E will be satisfied after the assignments to x and ℓ. Otherwise, we remove the equation $x = \ell$ because the execution of the two instructions will satisfy it, then add equations $\{x_i = \ell_i\}$ before the uses as in the case of `store` or `cond` instructions.

The cases of `call` and `return` instructions are similar, except that the uses and defs of these LTL instructions are not marked in the instructions (as in RTL), but are implicitly determined from a type signature. Therefore, the uses and defs of an LTL `call`$(\tau, \ldots)$ are respectively `arguments`(τ) and `result`(τ) (case 7), and the uses of an LTL `return` are $\{\texttt{result}(f'.\texttt{typesig})\}$ (case 9). Moreover, not all registers and stack slots are preserved across an LTL function call, but only those marked as callee-save by the application binary interface used. The `call` case therefore returns $\top$ if the set E of equations "after" contains any equation $x = \ell$ where ℓ is caller-save: since the value of ℓ after the call is unpredictable, this equation cannot be satisfied.

Two cases remain that correspond to RTL instructions that were eliminated (turned into `nop`) during register allocation. Case 3 corresponds to one step of coalescing: a `move` instruction from x_s to x_d was eliminated because x_s and x_d were assigned the same location. In this case, any equation $x_d = \ell$ holds "after" provided that $x_s = \ell$ holds "before"; and any equation $x = \ell$ with $x \neq x_d$ holds after if only if it holds before. Therefore, the set E' of equations "before" is

$$E[x_d \leftarrow x_s] \stackrel{\text{def}}{=} \{(x_s = \ell) \mid (x_d = \ell) \in E\} \cup \{(x = \ell) \mid (x = \ell) \in E \wedge x \neq x_d\}$$

Case 4 corresponds to dead code elimination: an `op` or `load` instruction was removed because its destination variable x is not used later. We check that this is the case by making sure that no equation $x = \ell$ for some ℓ occurs in E, returning E if so and $\top$ if not.

Finally, let us consider the case $p' \notin \mathrm{Dom}(\varphi)$, indicating that the instruction at p' was inserted during register allocation. By our assumptions on what an allocator is allowed to do, this new LTL instruction must be a move (case 10). Let ℓ_s be its source and ℓ_d its destination. By a similar reasoning as in case 3, an equation $x = \ell_d$ is satisfied after the move if $x = \ell_s$ is satisfied before. Moreover, the move preserves satisfiability of any equation $x = \ell$ such that $\ell \perp \ell_d$. However, equations $x = \ell$ where $\ell \,\#\, \ell_d$ are not satisfiable because of overlap. The set E' of equations before point p' is, therefore:

$$\begin{aligned} E[\ell_d \leftarrow \ell_s] &= \top \text{ if there exists } (x = \ell) \in E \text{ such that } \ell \,\#\, \ell_d \\ E[\ell_d \leftarrow \ell_s] &= \{(x = \ell_s) \mid (x = \ell_d) \in E\} \cup \{(x = \ell) \mid (x = \ell) \in E \wedge \ell \perp \ell_d\} \\ &\ \text{otherwise} \end{aligned}$$

The Dataflow Analysis and Its Uses. Armed with the transfer function of figure 1, we set up backward dataflow equations of the form

$$E(p') = \bigcup \{\texttt{transfer}(f, f', \varphi, s', E(s') \mid s' \text{ successor of } p' \text{ in } f'\}$$

The unknowns are $E(p')$, the set of equations that must hold after each program point p' of the transformed function f'. By convention on $\top$, we take $\top \cup E = E \cup \top = \top$. We then solve those equations by standard fixpoint iteration, starting with $E(p') = \emptyset$ for all points p'. (In our case, we reused a generic implementation of Kildall's algorithm provided by the CompCert compiler.)

Interestingly, this dataflow analysis generalizes liveness analysis, in the following sense: if $\{x_1 = \ell_1; \ldots; x_n = \ell_n\}$ are the equations "after" inferred at a program point p' mapped to p by φ, then the first projection $\{x_1, \ldots, x_n\}$ is the set of variables live in the original function f after point p and the second projection $\{\ell_1, \ldots, \ell_n\}$ is the set of locations live in the transformed function f' after point p'.

The validator then considers the set E_0 of equations “before ” the function entry point:

$$E_0 \stackrel{\text{def}}{=} \texttt{transfer}(f, f', \varphi, f'.\texttt{entrypoint}, E(f'.\texttt{entrypoint}))$$

If $E_0 = \top$, an unprovable equation was encountered at some reachable instruction; validation therefore fails. Otherwise, we need to make sure that the equations in E_0 always hold. The only variable-location equations that hold with certainty are those between the RTL function parameters and the corresponding LTL locations:

$$E_{params} \stackrel{\text{def}}{=} \{f.\texttt{params} = \texttt{parameters}(f'.\texttt{typesig})\}$$

The validator could, therefore, check that $E_0 \subseteq E_{params}$ and signal an error otherwise. However, this check is too strong for C programs: it amounts to imposing Java’s “definite assignment” rule. Indeed, $E_0 \subseteq E_{params}$ implies that all variables live at the beginning of the RTL function are parameters of this function. This is not always the case in RTL code generated from valid C functions such as:

```
int f(int x) {
  int y;
  if (x != 0) y = 100 / x;
  if (x != 0) return y; else return -1;
}
```

Here, the local variable `y` is live at the beginning of `f`, yet the function is semantically well-defined. Performed on the corresponding RTL code and a correct LTL register allocation of this code, the dataflow analysis of our validator produces an E_0 containing the equation $\texttt{y} = \ell$ for some ℓ. (This equation arises from the use of `y` in the “then” branch of the second “if”, combined with the lack of a definition of `y` in the “else” branch of the first “if”.)

How, then, can we avoid rejecting such correct codes at validation time? We take advantage of two very reasonable assumptions:

1. The semantics of RTL, like that of C, states that a program has undefined behavior if at run-time it uses the value of an undefined variable.
2. When establishing the correctness of a run of register allocation via validation, we are only interested in RTL programs that have well-defined behavior. For source programs with undefined behaviors, the register allocator can produce arbitrary code. (Most compilers take this “garbage in, garbage out” view of optimization.)

Now, an equation $x = \ell$ at a program point where x is guaranteed to be uninitialized can safely be considered as always satisfied: since the RTL program has well-defined semantics, it is not going to use the value of x before defining it, therefore the actual value of x does not matter, and we might just as well assume that it matches the value of ℓ in the LTL code. The check performed by the validator on the initial equations E_0 is, therefore,

$$E_0 \cap f.\mathtt{params} \subseteq E_{params}$$

where the intersection $E \cap X$ between a set of equations E and a set of RTL variables X is defined as

$$E \cap X \stackrel{\text{def}}{=} \{(x = \ell) \mid (x = \ell) \in E \wedge x \in X\}$$

3.3 The Validation Algorithm

Combining the definitions of sections 3.1 and 3.2, we obtain the main validation function:

```
$\mathtt{check\_function}(f, f', \varphi) =$
    if $\mathtt{check\_structure}(f, f', \varphi) = \mathtt{false}$, return $\mathtt{false}$
    compute the solutions $E(p')$ of the dataflow equations
        $E(p') = \bigcup\{\mathtt{transfer}(f, f', \varphi, s', E(s')) \mid s'$ successor of $p'$ in $f'\}$
    let $E_0 = \mathtt{transfer}(f, f', \varphi, f'.\mathtt{entrypoint}, E(f'.\mathtt{entrypoint}))$
    check $E_0 \neq \top$ and $E_0 \cap f.\mathtt{params} \subseteq \{f.\mathtt{params} = \mathtt{parameters}(f'.\mathtt{typesig})\}$
```

Typically, this validator is combined with an untrusted implementation of a register allocator `regalloc`, as follows:

```
$\mathtt{validated\_regalloc}(f) =$
    let $(f', \varphi) = \mathtt{regalloc}(f)$ in
    if $\mathtt{check\_function}(f, f', \varphi)$ then return $f'$ else abort compilation
```

4 Soundness Proof

There are two properties of interest for a translation validator. One is *soundness*: if the validator says "yes", the transformed code behaves identically to the source code. The other is *relative completeness*: the validator never raises a false alarm; in other words, it accepts all valid instances of the code transformation considered. The completeness property is, necessarily, relative to a limited class of program transformations such as those listed in section 2.3: otherwise, validation would boil down to checking semantic equivalence between two arbitrary programs, which is undecidable.

We have formally proved the soundness of the validation algorithm presented in section 3. The proof was mechanized using the Coq proof assistant, bringing near-absolute confidence. This section gives a simplified sketch of this soundness proof. Relative completeness is difficult to even state formally, so we did not attempt to prove it. Testing shows no false alarms (see section 5). We conjecture that our validator is complete for all program transformations that can only rename variables, insert move operations, and delete operations and loads, but treat as uninterpreted (and therefore preserve) all other computations.

4.1 Dynamic Semantics

In preparation for stating and proving soundness, we need to give formal semantics to the RTL and LTL languages. The full semantics of RTL is described in [11, section 6]. Here, for simplicity, we outline the semantics of the fragment of RTL that excludes function calls and returns, and therefore is given relative to a single function f.

The semantics is presented in small-step style as a transition relation → between execution states. States are triples (p, e, m) where p is the current program point (a CFG node), e is a partial map from variables to values, and m is the memory state: a partial map from (pointer, memory quantity) pairs to values. Values are the discriminated union of integers, floating-point numbers, and pointers. (In the full semantics of RTL, the state contains additional components such as the function currently executing and an abstract call stack.)

The transition relation → between states is defined by the rules of figure 2. The rules discriminate on the instruction at the current program point p, then update the three components of the state accordingly. The partial functions $\overline{op}$, $\overline{mode}$ and $\overline{cond}$ are the semantic interpretations of operators, addressing modes and conditions as functions over values. We make no assumptions about these interpretations, except that the `move` operation is the identity: $\overline{\texttt{move}}(v) = v$. The notation $e[x \leftarrow v]$ stands for the variable environment mapping x to v and all other variables y to $e(y)$. The initial state is $(f.\texttt{entrypoint}, [f.\texttt{params} \mapsto \vec{v}_{args}], m_{init})$ where $\vec{v}_{args}$ are the values of the arguments given to function f. The final state is (p, e, m) where p points to a `return` instruction.

The semantics of LTL is essentially isomorphic to that of RTL, at least for the fragment considered here. (The full LTL treats function calls somewhat differently from RTL, to reflect the passing of function arguments and results through conventional locations.) LTL states are triples (p', e', m') of a program point p' in f', an environment e' mapping locations to values, and a memory state m'. The main difference between RTL and LTL is the update $e'[\ell \leftarrow v]$ of a location

$$\frac{f.\texttt{code}(p) = \texttt{nop}(s)}{(p, e, m) \rightarrow (s, e, m)} \qquad \frac{f.\texttt{code}(p) = \texttt{op}(op, \vec{x}, x, s) \qquad \overline{op}(e(\vec{x})) = v}{(p, e, m) \rightarrow (s, e[x \leftarrow v], m)}$$

$$\frac{f.\texttt{code}(p) = \texttt{load}(\kappa, mode, \vec{x}, x, s) \qquad \overline{mode}(e(\vec{x})) = v_{ad} \qquad m(v_{ad}, \kappa) = v}{(p, e, m) \rightarrow (s, e[x \leftarrow v], m)}$$

$$\frac{f.\texttt{code}(p) = \texttt{store}(\kappa, mode, \vec{x}, x, s) \qquad \overline{mode}(e(\vec{x})) = v_{ad} \qquad m[(v_{ad}, \kappa) \leftarrow e(x)] = m'}{(p, e, m) \rightarrow (s, e, m')}$$

$$\frac{f.\texttt{code}(p) = \texttt{cond}(cond, \vec{x}, s_1, s_2) \qquad s = \begin{cases} s_1 & \text{if } \overline{cond}(e(\vec{x})) = \texttt{true} \\ s_2 & \text{if } \overline{cond}(e(\vec{x})) = \texttt{false} \end{cases}}{(p, e, m) \rightarrow (s, e, m)}$$

Fig. 2. Transition rules for the simplified semantics of RTL

ℓ by a value v: it sets ℓ to v, but as collateral damage is also sets overlapping locations $\ell' \# \ell$ to unspecified values:

$$\begin{aligned} e'[\ell \leftarrow v](\ell) &= v \\ e'[\ell \leftarrow v](\ell') &= e'(\ell') \text{ if } \ell' \perp \ell \\ e'[\ell \leftarrow v](\ell') & \quad \text{is unspecified if } \ell' \# \ell \end{aligned}$$

Note that the values of stack locations $S(\delta, n)$ are stored in the location environment e' and not in the memory state m'. This simplifies the proof. A separate proof, detailed in [11, section 12], shows that accesses to stack locations can later be reinterpreted as memory loads and stores within the activation record.

4.2 Equation Satisfaction

The crucial invariant of the soundness proof is the following: whenever control reaches point p' in the LTL function f' and matching point $\varphi(p')$ in the RTL function f, the corresponding environments e and e' satisfy the equations E "before" point p' inferred by the validator. Equation satisfaction is written $e, e' \models E$ and defined as

$$e, e' \models E \overset{\text{def}}{=} \forall (x = \ell) \in E,\ x \in \mathrm{Dom}(e) \Longrightarrow e(x) = e'(\ell)$$

This predicate enjoys nice properties that are keys to the soundness proof. First, satisfaction implies that the argument values to matching RTL and LTL operations are identical. (This lemma is used in the parts of the soundness proof that corresponds to cases 3, 6, 7, 8 and 9 of the transfer function.)

Lemma 1. *If $e, e' \models E \cup \{\vec{x} = \vec{\ell}\}$ and $e(\vec{x})$ is defined, then $e'(\vec{\ell}) = e(\vec{x})$.*

Second, satisfaction is preserved by several kinds of parallel or unilateral assignments. (For each lemma we indicate the corresponding cases of the transfer function.)

Lemma 2 (Parallel assignment – cases 5 and 7). *If $e, e' \models E \setminus (x = \ell)$ and $(x, \ell) \perp E$ then $e[x \leftarrow v], e'[\ell \leftarrow v] \models E$*

Lemma 3 (RTL assignment to a dead variable – case 4). *If $e, e' \models E$ and $(x = \_) \notin E$ then $e[x \leftarrow v], e' \models E$*

Lemma 4 (Coalesced RTL move – case 3). *If $e, e' \models E[x_d \leftarrow x_s]$ then $e[x_d \leftarrow e(x_s)], e' \models E$*

Lemma 5 (Inserted LTL move – case 10). *If $E[\ell_d \leftarrow \ell_s] \neq \top$ and $e, e' \models E[\ell_d \leftarrow \ell_s]$ then $e, e'[\ell_d \leftarrow e'(\ell_s)] \models E$*

Finally, satisfaction holds in the initial states, taking $\vec{x} = f.\texttt{params}$ and $\vec{\ell} = \texttt{parameters}(f'.\texttt{typesig})$ and $\vec{v}$ to be the values of the function parameters.

Lemma 6. *If $E \cap \vec{x} \subseteq \{\vec{x} = \vec{\ell}\}$, then for any e' such that $e'(\vec{\ell}) = \vec{v}$, we have $[\vec{x} \mapsto \vec{v}], e' \models E$.*

4.3 Forward Simulation

The soundness proof takes the form of a forward simulation diagram relating one transition in the RTL code to one or several transitions in the LTL code, starting and ending in matching states. (The "or several" part corresponds to the execution of move instructions inserted during register allocation.)

$$\begin{tikzcd}
s_1 \arrow[r, dash, "\approx"] \arrow[d, "\text{(RTL)}"'] & s'_1 \arrow[d, dashed, "+\quad\text{(LTL)}"] \\
s_2 \arrow[r, dash, dashed, "\approx"] & s'_2
\end{tikzcd}$$

The relation $\approx$ between RTL and LTL states is defined as follows:

$$\begin{aligned}
&(p, e, m) \approx (p', e', m') \overset{\text{def}}{=} \\
&\qquad \varphi(p') = p \ \wedge \ e, e' \models \mathtt{transfer}(f, f', \varphi, p', E(p')) \ \wedge \ m = m'
\end{aligned}$$

That is, the program points must match according to the φ mapping; the variable and location environments must satisfy the dataflow equations "before" point p'; and the memory states are identical.

Theorem 1 (Forward simulation). *Assume that* $\mathtt{check\_function}(f, f', \varphi) = \mathtt{true}$. *Let* $E(p')$ *be the solutions to the dataflow equations. If* $s_1 \rightarrow s_2$ *and* $s_1 \approx s'_1$, *there exists* s'_2 *such that* $s'_1 \xrightarrow{+} s'_2$ *and* $s_2 \approx s'_2$.

The proof of this theorem proceeds in two steps. First, we show that the LTL code can make one transition from s'_1 to some state (p', e', m') that does not necessarily match s_2 (because $\varphi(p')$ can be undefined) but is such that $e, e' \models \mathtt{transfer}(f, f', \varphi, p', E(p'))$. This part of the proof proceeds by case analysis on the RTL and LTL instructions pointed to by s_1 and s'_1, and exercises all cases of the structural checks and the transfer function except the `path` check and case 10. Then, the following lemma shows that we can extend this LTL transition with zero, one or several transitions (corresponding to executions of inserted move instructions) to reach a state matching s_2.

Lemma 7 (Execution of inserted moves). *Assume* $\mathtt{path}(f', \varphi, p, p') = \mathtt{true}$ *and* $e, e' \models \mathtt{transfer}(f, f', \varphi, p', E(p'))$. *Then, there exists* p'' *and* e'' *such that* $(p', e', m) \xrightarrow{*} (p'', e'', m)$ *and* $\varphi(p'') = p$ *and* $e, e'' \models \mathtt{transfer}(f, f', \varphi, p'', E(p''))$.

From the forward simulation theorem 1, semantic preservation for whole programs (that is, agreement between the observable behaviors of the source RTL code and transformed LTL code) follows easily using the general results of [11, section 3.7].

5 Implementation and Experimental Results

We implemented the validation algorithm and a prototype register allocator within the CompCert verified compiler [9]. Like all other verified parts of this compiler, the validator is written directly in the Gallina specification language of the Coq proof assistant, in pure functional style. Sets of equations are implemented as persistent AVL trees, using the FSet standard library of Coq. This implementation supports insertion and removal of equations in $O(\log n)$ time, but the compatibility check $(x, \ell) \perp E$ requires an $O(n)$ traversal of the set E. Whether a better data structure could support compatibility check in logarithmic time is an open question.

The Gallina implementation of the validator lends itself immediately to program proof within Coq. Efficient Caml code is automatically generated from the Gallina code using Coq's program extraction facility. The generated Caml code is then linked with a register allocator hand-written in Caml.

The prototype register allocator we experimented with is a standard Chaitin-style graph coloring allocator, using George and Appel's iterated register coalescing algorithm to color the interference graph [10]. If some variables x were assigned stack slots and are used by instructions that demand a hardware register, spill and reload instructions to/from fresh temporary variables are introduced and register allocation is repeated. Two spilling strategies were experimented. The first simply inserts a reload before every use of a spilled variable and a spill after every definition. The second splits the live ranges of a spilled variable at every definition and every use, in the hope that reloaded values can stay in a register across several reloads in parts of the code where register pressure is low. (This is a less aggressive form of splitting than that considered by Appel and George [12].) Since we are targeting a register-rich architecture (the PowerPC), spilling occurs rarely. To stress the validator, we reduced the number of callee-save registers, forcing considerable spilling across function calls.

On the CompCert test suite, the validator performed as expected: it did not raise any false alarms, but found several mistakes in our implementation of the second spilling strategy. The compile-time overhead of the validator is very reasonable: validation adds 20% to the time taken by register allocation and 6% to the whole compilation time.

From a proof engineering viewpoint, the validator is a success. Its mechanized proof of correctness is only 900 lines of Coq, which is quite small for a 350-line piece of code. (The typical ratio for Coq program proofs is 6 to 8 lines of proof per line of code.) In contrast, 4300 lines of Coq proof were needed to verify the register allocation and spilling passes of the original CompCert compiler. Even this earlier development used translation validation on a sub-problem: the George-Appel graph coloring algorithm was implemented directly in untrusted Caml code, then followed by a verified validator to check that the resulting assignment is a valid coloring of the interference graph. Later, Blazy, Robillard and Appel conducted a Coq proof of the graph coloring algorithm [13]. This is a fairly large proof: in total, more than 10000 lines of proof are needed to completely verify the original CompCert register allocation and spilling passes. In summary, the translation validation approach delivers a ten-fold reduction in the proof effort compared with the compiler verification approach, while providing soundness guarantees that are just as strong. Of course, the compiler verification approach offers additional formal guarantees: not just soundness, but also completeness (register allocation never fails at compile-time). In contrast, the verified validator approach cannot rule out the possibility of a spurious compile-time error.

6 Related Work

The idea of translation validation goes back at least to Samet's 1975 Ph.D. thesis [14]. It was rediscovered and popularized by Pnueli *et al.* ten years ago [1]. Some translation validators proceed by generation of verification conditions followed by model checking or automatic theorem proving [1,15,16,17]; others rely on less powerful but cheaper and more predictable approaches based on symbolic evaluation and static analyses [2,3,4,5,8,18]. For another dividing line, some validators are general-purpose and apply to several compilation passes [2] or even to a whole compiler [3], while others are specialized to particular families of optimizations, such as software pipelining [15,19,18], instruction scheduling [5], partial redundancy elimination [4], or register allocation [8]. The present work falls squarely in the cheap, specialized, static analysis-based camp.

The earlier work most closely related to ours is that of Huang, Childers and Soffa [8]: a validator for register allocation that was prototyped within SUIF. Their validator proceeds by forward dataflow analysis and global value numbering. A nice feature of their validator, which ours lacks, is the production of meaningful explanations when an error is detected. On the other hand, their validation algorithm was not proved sound. Such a proof appears delicate because the semantic interpretation of global value numbers is difficult.

The general-purpose validators of Necula [2] and Rival [3] can also validate register allocation among other program transformations. They proceed by symbolic evaluation: variables and locations in the source and transformed code are associated symbolic expressions characterizing their values, and these expressions are compared modulo algebraic identities to establish semantic equivalence. Symbolic evaluation is a very versatile approach, able to validate many program transformations. On the particular case of register allocation and spilling, it appears no more powerful, but more costly, than the specialized techniques used by Huang *et al.* and by us.

7 Conclusions and Future Work

The validation algorithm for register allocation and spilling presented in this paper is simple enough to be integrated in production compilers and efficient enough to be invoked on every compilation run. At the same time, the mechanically-checked proof of soundness brings considerable confidence in its results.

Our validator can be improved in several directions. One is to design a more efficient data structure to represent sets of equations. As mentioned in section 5, the simple representation of equation sets as AVL trees performs compatibility checks in linear time. A more sophisticated data structure might support logarithmic-time operations over equation sets.

Another direction is to introduce additional forms of equations to enable the validation of even more code transformations related to register allocation. For example, rematerialization of constants [20] could probably be validated if we were able to keep track of equations of the form $x = \mathit{constant}$. Likewise, the parts of the function prologue and epilogue that save and restore used callee-save registers to/from stack slots (currently treated in CompCert by a separate, verified pass) could be validated along with register allocation if we had equations of the form $\texttt{init}(r) = \ell$, where $\texttt{init}(r)$ is a symbolic constant denoting the value of callee-save register r on entrance to the function.

References

1. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998)
2. Necula, G.C.: Translation validation for an optimizing compiler. In: Programming Language Design and Implementation 2000, pp. 83–95. ACM Press, New York (2000)
3. Rival, X.: Symbolic transfer function-based approaches to certified compilation. In: 31st symposium Principles of Programming Languages, pp. 1–13. ACM Press, New York (2004)
4. Tristan, J.B., Leroy, X.: Verified validation of Lazy Code Motion. In: Programming Language Design and Implementation 2009, pp. 316–326. ACM Press, New York (2009)
5. Tristan, J.B., Leroy, X.: Formal verification of translation validators: A case study on instruction scheduling optimizations. In: 35th symposium Principles of Programming Languages, pp. 17–27. ACM Press, New York (2008)
6. Coq development team: The Coq proof assistant. Software and documentation, `http://coq.inria.fr/` (1989–2010)
7. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions. EATCS Texts in Theoretical Computer Science. Springer, Heidelberg (2004)
8. Huang, Y., Childers, B.R., Soffa, M.L.: Catching and identifying bugs in register allocation. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 281–300. Springer, Heidelberg (2006)
9. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009)
10. George, L., Appel, A.W.: Iterated register coalescing. ACM Transactions on Programming Languages and Systems 18(3), 300–324 (1996)
11. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning 43(4), 363–446 (2009)
12. Appel, A.W., George, L.: Optimal spilling for CISC machines with few registers. In: Programming Language Design and Implementation 2001, pp. 243–253. ACM Press, New York (2001)
13. Blazy, S., Robillard, B., Appel, A.W.: Formal verification of coalescing graph-coloring register allocation. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 145–164. Springer, Heidelberg (2010)

14. Samet, H.: Automatically Proving the Correctness of Translations Involving Optimized Code. PhD thesis, Stanford University (1975)
15. Leviathan, R., Pnueli, A.: Validating software pipelining optimizations. In: Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2002), pp. 280–287. ACM Press, New York (2006)
16. Zuck, L., Pnueli, A., Fang, Y., Goldberg, B.: VOC: A methodology for translation validation of optimizing compilers. Journal of Universal Computer Science 9(3), 223–247 (2003)
17. Barrett, C.W., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A., Zuck, L.D.: TVOC: A translation validator for optimizing compilers. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 291–295. Springer, Heidelberg (2005)
18. Tristan, J.B., Leroy, X.: A simple, verified validator for software pipelining. In: 37th symposium Principles of Programming Languages. ACM Press, New York (to appear, 2010) (accepted for publication)
19. Kundu, S., Tatlock, Z., Lerner, S.: Proving optimizations correct using parameterized program equivalence. In: Programming Language Design and Implementation 2009, pp. 327–337. ACM Press, New York (2009)
20. Briggs, P., Cooper, K.D., Torczon, L.: Rematerialization. In: Programming Language Design and Implementation 1992, pp. 311–321. ACM Press, New York (1992)

Automatic C-to-CUDA Code Generation for Affine Programs

Muthu Manikandan Baskaran[1], J. Ramanujam[2], and P. Sadayappan[1]

[1] The Ohio State University, USA
[2] Louisiana State University, USA

Abstract. Graphics Processing Units (GPUs) offer tremendous computational power. CUDA (Compute Unified Device Architecture) provides a multi-threaded parallel programming model, facilitating high performance implementations of general-purpose computations. However, the explicitly managed memory hierarchy and multi-level parallel view make manual development of high-performance CUDA code rather complicated. Hence the automatic transformation of sequential input programs into efficient parallel CUDA programs is of considerable interest.

This paper describes an automatic code transformation system that generates parallel CUDA code from input sequential C code, for regular (affine) programs. Using and adapting publicly available tools that have made polyhedral compiler optimization practically effective, we develop a C-to-CUDA transformation system that generates two-level parallel CUDA code that is optimized for efficient data access. The performance of automatically generated code is compared with manually optimized CUDA code for a number of benchmarks. The performance of the automatically generated CUDA code is quite close to hand-optimized CUDA code and considerably better than the benchmarks' performance on a multicore CPU.

1 Introduction

Graphics Processing Units (GPUs) represent the most powerful multi-core systems currently in use. For example, the NVIDIA GeForce 8800 GTX GPU chip has a peak performance of over 350 GFLOPS and the NVIDIA GeForce GTX 280 chip has a peak performance of over 900 GFLOPS. There has been considerable recent interest in using GPUs for general purpose computing [8,13,12]. Until recently, general-purpose computations on GPUs were performed by transforming matrix operations into specialized graphics processing, such as texture operations. The introduction of the CUDA (Compute Unified Device Architecture) programming model by NVIDIA provided a general-purpose multi-threaded model for implementation of general-purpose computations on GPUs. Although more convenient than previous graphics programming APIs for developing GPGPU codes, the manual development of high-performance codes with the CUDA model is still much more complicated than the use of parallel programming models such as OpenMP for general-purpose multi-core systems. It is therefore of great interest, for enhanced programmer productivity and for software quality, to develop compiler support to facilitate the automatic transformation of sequential input programs into efficient parallel CUDA programs.

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 244–263, 2010.

There has been significant progress over the last two decades in the development of powerful compiler frameworks for dependence analysis and transformation of loop computations with affine bounds and array access functions [1,5,6,24,18,14,9,25,23,4]. For such regular programs, compile-time optimization approaches have been developed using affine scheduling functions with a polyhedral abstraction of programs and data dependencies. CLooG [4,7] is a powerful open-source state-of-the-art code generator that transforms a polyhedral representation of a program and affine scheduling constraints into concrete loop code. The Pluto source-to-source optimizer [5,6,22] enables end-to-end automatic parallelization and locality optimization of affine programs for general-purpose multi-core targets. The effectiveness of the transformation system has been demonstrated on a number of non-trivial application kernels for multi-core processors, and the system implementation is publicly available [22].

In this paper we describe an end-to-end automatic C-to-CUDA code generator using a polyhedral compiler transformation framework. We evaluate the quality of the generated code using several benchmarks, by comparing the performance of automatically generated CUDA code with hand-tuned CUDA code where available and also with optimized code generated by the Intel icc compiler for a general-purpose multi-core CPU.

The rest of the paper is organized as follows. Section 2 provides an overview of the polyhedral model for representing programs, dependences, and transformations. Section 3 provides an overview of the NVIDIA GPU architecture and the CUDA programming model. The design and implementation of the C-to-CUDA transformer is presented in Section 4. Experimental results are provided in Section 5. We discuss related work in Section 6 and conclude with a summary in Section 7.

2 Background

This section provides background information on the polyhedral model. A hyperplane in n dimensions is an $n-1$ dimensional affine subspace of the n-dimensional space and can be represented by an affine equality. A halfspace consists of all points of an n-dimensional space that lie on one side of a hyperplane (including the hyperplane); it can be represented by an affine inequality. A polyhedron is the intersection of finitely many halfspaces. A polytope is a bounded polyhedron.

In the polyhedral model, a statement s surrounded by m loops is represented by an m-dimensional polytope, referred to as an iteration space polytope. The coordinates of a point in the polytope (referred to as the iteration vector i_s) correspond to the values of the loop indices of the surrounding loops, starting from the outermost. In this work we focus on programs where loop bounds are affine functions of outer loop indices and global parameters (e.g., problem sizes). Similarly, array access functions are also affine functions of loop indices and global parameters. Hence the iteration space polytope $\mathcal{D}_s$ of a statement s can be defined by a system of affine inequalities derived from the bounds of the loops surrounding s. Each point of the polytope corresponds to an instance of statement s in program execution. Using matrix representation to express systems of affine inequalities, the iteration space polytope is defined by $D_s \begin{pmatrix} i_s \\ n \\ 1 \end{pmatrix} \geq 0$, where D_s is a matrix representing loop bound constraints and n is a vector of global parameters.

Affine array access functions can also be represented using matrices. Let $a[\mathcal{F}_{ras}(i_s)]$ be the r^{th} reference to an array a in statement s whose corresponding iteration vector is i_s. Then $\mathcal{F}_{ras}(i_s) = F_{ras} \begin{pmatrix} i_s \\ n \\ 1 \end{pmatrix}$, where F_{ras} is a matrix representing an affine mapping from the iteration space of statement s to the data space of array a. Row i in the matrix F_{ras} (often referred to as the access matrix) defines a mapping corresponding to the ith dimension of the data space. When the rank of the access matrix of an array reference is less than the iteration space dimensionality of the statement in which it is accessed, the array is said to have an order of magnitude (or higher-order) reuse due to that reference.

Given an iteration space polytope $\mathcal{D}$ and a set of array access functions $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k$ of k references to an array in the iteration space, the set of array elements accessed in the iteration space or the *accessed data space* is given by $\mathcal{DS} = \bigcup_{j=1}^{k} \mathcal{F}_j \mathcal{D}$, where $\mathcal{F}_j \mathcal{D}$ is the image of the iteration space polytope $\mathcal{D}$ formed by the affine access function $\mathcal{F}_j$ and it gives the set of elements accessed by the reference $\mathcal{F}_j$ in $\mathcal{D}$.

Dependences. There has been a significant body of work on dependence analysis in the polyhedral model [9,24,29]. An instance of statement s, corresponding to iteration vector i_s within iteration domain D_s, depends on an instance of statement t (with iteration vector i_t in domain D_t), if (1) i_s and i_t are valid points in the corresponding iteration space polytopes, (2) they access the same memory location, and (3) i_s is executed before i_t. Since array accesses are assumed to be affine functions of loop indices and global parameters, the constraint that defines conflicting accesses of memory locations can be represented by an affine equality (obtained by equating the array access functions in source and target statement instances). Hence all constraints to capture a data dependence can be represented as a system of affine inequalities/equalities with a corresponding polytope (referred to as a *dependence polytope*).

Affine Transforms. The polyhedral model has been effectively used to find good affine program transformations that are aimed at either improvement of sequential programs (source-to-source transformation) or automatic parallelization of programs or both [10,18,14,11,14,23,6].

A one-dimensional affine transformation of a statement s is represented in the polyhedral model as $\phi_s(i_s) = \mathcal{C}_s \cdot \begin{pmatrix} i_s \\ n \\ 1 \end{pmatrix}$, where $\mathcal{C}_s$ is a row vector and the affine mapping ϕ_s represents an affine hyperplane that maps each instance of statement s to a point in a dimension of the transformed iteration space. An affine transformation is valid only if it preserves the dependences in the original program. An m-dimensional affine mapping can be represented using a matrix with m rows, where each row represents a one-dimensional mapping. A set of linearly independent one-dimensional affine functions $(\phi_s^1, \phi_s^2, \ldots, \phi_s^k)$ maps each instance of statement s into a point in the multi-dimensional transformed space. The transformation matrix captures a composition of transformations like fusion, skewing, reversal and shifting.

It has been shown (in automatic transformation systems like Pluto) that key compiler transformations like tiling can be effectively performed using the polyhedral model. When tiling is performed, in the tiled iteration space, statement instances are represented by higher dimensional statement polytopes involving *supernode* or *inter-tile*

```
for (i=0;i<N;i++) {
  P: x[i]=0;
  for (j=0;j<N;j++)
    Q: x[i]+=a[j][i]*y[j];
}
```

(a) Original code

```
for ( it=0; it<=floord(N−1,32);it++) {
  for ( jt=0; jt<=floord(N−1,32);jt++) {
    if ( jt == 0) {
        for ( i=max(32*it,0);
    i<=min(32*it+31,N−1); i++) {
          P: x[i]=0;
          Q: x[i]=x[i]+a[0][i]*y[0];
      }
    }
    for ( i=max(32*it,0);
     i<=min(32*it+31,N−1); i++) {
      for ( j=max(32*jt,1);
       j<=min(32*jt+31,N−1);j++) {
        Q: x[i]=x[i]+a[j][i]*y[j];
      }
    }
  }
}
```

(b) Tiled code

$$D_Q^{orig}\cdot\begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} \geq 0 \qquad D_Q^{tiled}\cdot\begin{pmatrix} it \\ jt \\ i \\ j \\ N \\ 1 \end{pmatrix} \geq 0$$

(c) Original and tiled iteration space

Fig. 1. Example to illustrate Tiling: Transpose matrix vector multiply (tmv) kernel

iterators and *intra-tile* iterators. The code in Figure 1(b) represents the tiled version of the code in Figure 1(a). The original iteration space and the transformed iteration space are illustrated in Figure 1(c).

3 GPU Architecture and the CUDA Programming Model

In this Section, we provide an overview of the GPU parallel computing architecture, the CUDA programming interface, and the GPU execution model.

3.1 GPU Computing Architecture

NVIDIA GPUs comprises of a set of multiprocessor units called *streaming multiprocessors (SMs)*, each one containing a set of processor cores (called *streaming processors* (SPs)). The NVIDIA GeForce 8800 GTX has 16 SMs, each consisting of 8 SPs. The NVIDIA GeForce GTX280 has 30 SMs with 8 SPs in each SM. The SPs within an SM communicate through a fast explicitly managed on-chip local store, also called the *shared memory*, while the different SMs communicate through slower off-chip DRAM, also called the *global memory*. Each SM unit also has a fixed number of *registers*.

Different types of memory in the GPUs are addressable in CUDA programming model. The memories are organized in a hybrid cache and local-store hierarchy. The memories are as follows: (1) off-chip global memory (768MB on the 8800 GTX), (2) off-chip local memory, (3) on-chip shared memory (16KB per multiprocessor in 8800 GTX), (4) off-chip constant memory with on-chip cache (64KB in 8800 GTX), and (5) off-chip texture memory with on-chip cache.

The off-chip DRAM in the GPU device (i.e., the global memory) has a very high latency (about $100-200$ cycles). Hence reducing the latency in accessing data from global memory is critical for good performance. The global memory accesses in NVIDIA GPU chips are characterized by a hardware optimization – *global memory access coalescing*. Accesses from adjacent threads in a half-warp to adjacent locations (that are aligned to 4, 8, or 16 bytes) in global memory are coalesced into a single contiguous aligned memory access. Interleaved access to global memory by threads in a thread block is essential to exploit this architectural feature and is therefore an important optimization for a C-to-CUDA compiler.

The shared memory in each SM is organized into banks. When multiple addresses belonging to the same bank are accessed at the same time, bank conflict occur. Each SM has a set of registers. The constant and texture memories are read-only regions in the global memory space and they have on-chip read-only caches. Accessing constant cache is faster, but it has only a single port and hence it is beneficial when multiple processor cores load the same value from the cache. Texture cache has higher latency than constant cache, but it does not suffer greatly when memory read accesses are irregular and it is also beneficial for accessing data with 2D spatial locality. It is extremely important to reduce the number of accesses to off-chip memory and maximize utilization of the on-chip memories.

3.2 CUDA Programming Model

Programming GPUs for general-purpose applications is enabled through a C/C++ language interface exposed by the NVIDIA Compute Unified Device Architecture (CUDA) technology [20]. The CUDA programming model provides an abstraction of the GPU parallel architecture using a minimal set of programming constructs such as hierarchy of threads, hierarchy of memories, and synchronization primitives. A CUDA program comprises of a host program which is run on the CPU or host and a set of CUDA kernels that are launched from the host program on the GPU device. The CUDA kernel is a parallel kernel that is executed on a set of threads. The threads are organized into groups called *thread blocks*. The threads within a thread block synchronize among themselves through barrier synchronization primitives in CUDA and they communicate through shared memory. A kernel comprises of a *grid* of one or more thread blocks. Each thread in a thread block is uniquely identified by its thread id (threadIdx) within its block and each thread block is uniquely identified by its block id (blockIdx). The dimensions of the thread and thread block are specified at the time of launching the kernel, through the identifiers *blockDim* and *gridDim*, respectively.

Each CUDA thread has access to the different memories at different levels in the hierarchy. The threads have a private local memory space and register space. The threads in a thread block share a shared memory space. The GPU DRAM is accessible by all threads in a kernel.

3.3 GPU Execution Model

NVIDIA GPUs use a Single Instruction Multiple Threads (SIMT) model of execution. The threads in a kernel are executed in groups called *warps*, where a warp is a unit of execution. The scalar SPs within an SM share a single instruction unit and the threads of a warp are executed on the SPs. All the threads of a warp execute the same instruction and each warp has its own program counter. The SM hardware employs zero-overhead

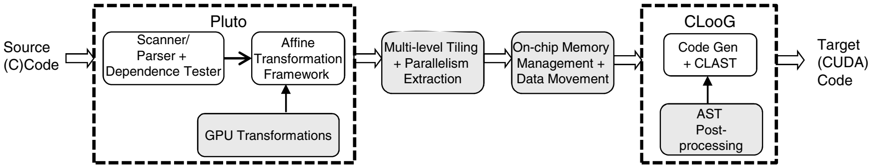

Source (C)Code
Pluto
Scanner/ Parser + Dependence Tester
Affine Transformation Framework
GPU Transformations
Multi-level Tiling + Parallelism Extraction
On-chip Memory Management + Data Movement
CLooG
Code Gen + CLAST
AST Post-processing
Target (CUDA) Code

Fig. 2. The C-to-CUDA Code Generation Framework

warp scheduling through the CUDA runtime scheduler. Any warps whose next instruction has ready operands is eligible for execution. Eligible warps are selected for execution by a prioritized scheduling policy. The warp scheduling is completely transparent to the CUDA programmer.

The computational resources on a multiprocessor unit, i.e., the shared memory and the register bank, are shared among the active thread blocks on that unit. For example, an application abstracted as a grid of 64 thread blocks can have 4 thread blocks mapped on each of the 16 multiprocessors of the NVIDIA GeForce 8800 GTX. The GeForce 8800 GTX GPU has a 16 KB shared memory space and 8192 registers. If the shared memory usage per thread block is 8 KB and the register usage is 4096, at most 2 thread blocks can be concurrently active on a multiprocessor; when one of the two thread blocks completes execution, another thread block can become active on the multiprocessor.

4 Design of C-to-CUDA Generator

In this section, we describe the C-to-CUDA code generator. Before providing details on the various transformation aspects, we first outline the general steps involved in source-to-source code generation using a polyhedral compiler framework.

1. The input program is run through a scanner and parser that constructs an abstract syntax tree (AST) for the input program. From the AST, iteration space polytopes and array access functions are extracted.
2. Data dependences are analyzed and dependence polytopes (described in Section 2) are generated.
3. After analyzing the dependences, affine statement-wise transforms are determined. The affine transforms provide the new lexicographic ordering of the statements in the transformed program.
4. When tiling has to be performed, the affine statement-wise transforms are used as tiling hyperplanes to generate higher-dimensional statements domains (involving supernode iterators and intra-tile iterators).
5. The transformed statement polytopes along with the affine transformations are provided to a polyhedral code generator such as CLooG to generate transformed code.

As described in Section 3, the GPU architecture represents a multi-level parallel architecture. It has various memory units (with different access properties) that are at different proximity with respect to the chip (on-chip and off-chip) and have very different access latencies. We now discuss the various issues that are addressed by our code generation system for generating effective CUDA code along the lines of the code generation process described above. There are several publicly available polyhedral transformation frameworks and tools. We used the Pluto [22] polyhedral parallel tiling infrastructure and CLooG [4,7], a state-of-the-art polyhedral code generator. The sequence of steps in the implemented system is shown in Fig. 2.

1. One of the key optimizations is to generate efficient access pattern for global (off-chip) memory access. Pluto finds affine transforms that are (1) communication-optimized, and (2) locality-optimized. At Step 3 of the code generation process (outlined above), our framework finds affine transforms that enable global memory coalescing in addition to being communication-optimized and locality-optimized. (detailed in [2]).
2. Two levels of parallelism must be extracted to exploit parallelism at the thread block level and the thread level for GPUs. At Step 4, we use the affine transforms determined at Step 3 to find multi-level tiled statement domains and identify and extract parallelism.
3. A critical optimization for GPUs is the utilization of on-chip memories. It is beneficial to move repeatedly reused data from off-chip memory to on-chip memory before the first use and move it back after the last use. At Step 4, our framework generates iteration space polytopes of data movement statements using polyhedral techniques, in addition to generating the transformed statement domains. (detailed in [3]).
4. At Step 5, we use the CLooG polyhedral code generator to generate the target code structure. Suitable input, in the form of a description of all statements (computation and data movement), together with their iteration spaces (as polytopes) as well as the transformations (as scheduling functions) specifying the new execution order for each statement instance, is input to the CLooG code generator. The union of all input iteration space polytopes is scanned by CLooG according to the specified scheduling functions, in order to generate loop nests in the target program that execute the statement instances in this new execution order.
5. After Step 5, the AST of the generated parallel tiled code is post processed to generate compilable CUDA code. The post processing is primarily (1) to introduce thread-centricity in the parallel code, i.e., to add thread identifier and thread block identifier, and (2) add inter-thread and inter-thread-block synchronizations at appropriate execution points.

In the rest of this section, we provide details on the following three aspects of the C-to-CUDA generator:

1. generation of multi-level tiled parallel code,
2. generation and placement of code to move data between on-chip and off-chip memories, and
3. generation of thread-centric parallel code.

4.1 Multi-level Parallel Tiled Code Generation

Tiling Hyperplanes and Tiling Legality Condition. In order to generate tiled code, Pluto finds affine transforms that satisfy the following tiling legality condition [6] in a

multi-statement imperfectly nested program and use them as tiling hyperplanes which constitute the loops in the transformed program:
A set of one-dimensional affine transformation functions (one corresponding to each statement in a imperfectly nested multi-statement program), $\{\phi_{s_1}, \phi_{s_2}, \ldots, \phi_{s_n}\}$, represents a valid tiling hyperplane if for each pair of dependent statement instances (i_{s_p}, i_{s_q}) $\phi_{s_q}(i_{s_q}) - \phi_{s_p}(i_{s_p}) \geq 0$. This condition guarantees that any inter- or intra-statement affine dependence is carried in the forward direction along the tiling hyperplane. Hence if a program is transformed using the affine transforms satisfying the above condition, then rectangular tiling is legal in the transformed program.

Affine transformations for CUDA. With CUDA, execution of a program involves distributing the computation across thread blocks and across threads within a thread block. For tiling at the outer level (at the level of thread blocks), our framework uses the affine transforms generated by Pluto. For finding tiling hyperplanes to generate tiled code at the inner level (at the level of threads), we modify Pluto to generate program transformations that enable interleaved access to global memory by threads in a thread block - this is necessary to facilitate coalesced global memory accesses that improve global memory access bandwidth. An approach to achieve this was developed in [2]. We incorporated that approach in our system by framing additional constraints to feed to Pluto while finding affine statement-wise transforms. The additional constraints are:

- If two statement instances access adjacent elements of an array (based on the actual array layout), then the statement instances are scheduled to execute at the same time; (and)
- If two statement instances access adjacent elements of an array (based on the actual array layout), then the statement instances are scheduled to execute on adjacent processors.

Extracting Parallel Loops. The affine transformations may or may not result in synchronization-free parallel tile loops (*doall* loops). If doall loops exist in the tile space, they are used as parallel loops. However when no synchronization-free parallelism exists, parallel code generation needs additional processing. There may be one or more loops that carry dependences (*doacross* loops). Since the tiling legality condition assures that the dependences are always carried in the forward direction, pipelined parallelism with synchronization can be exploited in such cases.

If $\{\phi^1, \phi^2, \ldots, \phi^n\}$ represent the doacross loops in the tile space, then the sum $\phi^1 + \phi^2 + \cdots + \phi^n$ carries all dependences that are carried by each $\phi^i, 1 \leq i \leq n$, and represents a legal wavefront of tiles such that all tiles in the wavefront are parallel [15]. In other words, the set of loops are transformed (using a unimodular skewing transformation) as follows:

$$\begin{pmatrix} \phi'^1 \\ \phi'^2 \\ \phi'^3 \\ \vdots \\ \phi'^n \end{pmatrix} = \begin{pmatrix} 1 & 1 & \ldots & 1 & 1 \\ 1 & 0 & \ldots & 0 & 0 \\ 0 & 1 & \ldots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & 0 \end{pmatrix} \begin{pmatrix} \phi^1 \\ \phi^2 \\ \phi^3 \\ \vdots \\ \phi^n \end{pmatrix}.$$

This ensures that ϕ'^1 is sequential and $\phi'^2, \phi'^3, \ldots, \phi'^n$ represent the parallel loops. This is the approach we employ to extract parallel loops at one level. A synchronization call to synchronize across the parallel units has to be placed at each iteration of the sequential loop. Handling the placement of synchronization calls is discussed later in the Section.

Pluto generates parallel code for general purpose multi-core architectures; it generates multi-level tiled code with parallelism only at the outer level. However for multi-level parallel architectures like GPGPUs, parallelism has to be extracted at multiple levels (two levels for CUDA - thread block level and thread level). Algorithm 1 provides details on the approach to generate multi-level tiled transformed statement domains (which are later fed to CLooG for code generation) along with the identification of parallel loops at thread block level and thread level.

Using CLooG for Multi-level Tiled Code Generation. As described earlier, CLooG scans a union of statement (iteration space) polyhedra using an optionally provided global lexicographic ordering specified through statement-wise scheduling functions or *scattering functions*, and generates loop nests in the target program that execute the statement instances in the new lexicographic order. CLooG does not include any data dependence information and hence the legality of scanning the statement polyhedra should be guaranteed by the user specifying the scattering functions. In our framework, the statement-wise affine transforms provided as scattering functions to CLooG ensure effective and correct execution of the transformed program. Tiled code is generated using CLooG by specifying a modified higher dimensional statement domain for each statement and also specifying the scheduling or scattering functions (using the affine statement-wise transforms) to generate the correct ordering of inter-tile and intra-tile loops.

Algorithm 1. Multi-level Parallel Tiled Code Generation

```
Input  Set of statements - $S$, Iteration Space Polytopes of all statements $\mathcal{D}_s, s \in S$, Statement-wise
    affine transforms for each level $k$: $\phi k_s^1, \phi k_s^2, \ldots, \phi k_s^n, s \in S$, Tile sizes $t_1, t_2, \ldots, t_n$ for each level
for each level do
    for each statement $s \in S$ do
        for each transform $\phi_s = C_s(i_s)$ do
            Increase the statement domain's dimensionality so that the domain includes the su-
            pernode iterators
            Add constraints involving supernode iterators ($\phi T_s$) and tile sizes that represent a
            statement instance in a supernode $t \times \phi T_s \leq C_s(i_s) \leq t \times \phi T_s + t - 1$
        end for
        Add scattering functions corresponding to supernodes. (The scattering functions are
        identity functions involving the supernode iterators)
        if level to be parallelized then
            if there exists doall loops then
                Mark them as parallel
            else
                Transform the first non-sequential loop $\phi^i$ as follows: $\phi^i \leftarrow \phi^i + \phi^{i+1} + \cdots + \phi^n$
                Mark $\phi^i$ as sequential and remaining subsequent loops in the band as parallel
            end if
        end if
    end for
end for
Output  Transformed computation statement domains and scattering functions
```

4.2 Data Movement between Off-Chip and On-Chip Memories

As discussed in Section 3, it is very important to reduce the accesses to off-chip memory and utilize the on-chip memories. Array references that have sufficient data reuse are good candidates to be copied to shared memory since the repeated accesses would be made in low-latency on-chip memory instead of off-chip memory. Array references, for which there exists no suitable affine scheduling that supports coalesced memory access, are also treated as candidates to be copied to shared memory. This is because of the fact that non-coalesced accesses incur very high memory access cost.

Given a program block or tile (having one or more statements), the data spaces accessed by array references within the block are determined using the iteration space of each statement and the array access function of each reference in each statement (as mentioned in Section 2). The data spaces accessed by the read and write references of each array are represented as separate polytopes and are then used to determine the size of storage buffer needed to host the required data. The code for data movement is then generated by scanning the data space polytopes using CLooG. The loop structure of the data movement code (copy code) is a perfect nest of n loops, where n is the dimensionality of the accessed data space. By using a cyclic distribution of the innermost loop across threads of a warp, we enable interleaved access of global memory by threads. The data movement statements are of two types: (1) those that move data in to shared memory (further referred to as copy-in statements) and (2) those that move data out of shared memory (further referred to as copy-out statements).

The target code should encompass the data movement statements and computation statements in proper order so that the parallel code results in correct program execution. At the level of thread blocks, the data movement statements are placed such that they respect the following order: *copy-in, computation, copy-out*. We utilize the scattering functions in CLooG to achieve the proper placement of data movement and computation statements. The scattering functions provide a multi-level multi-dimensional schedule. The basic idea is to introduce an additional ‘constant’ dimension in the original schedule at the level of thread blocks to define the order of statements. Suppose that in the transformed program, the computation and data movement statements are defined at the outer level by a schedule using the iterators $(c_1, c_2, \ldots, c_n)$. We modify the schedule of the copy-in, computation, copy-out statements as $(c_1, c_2, \ldots, c_n, 0)$, $(c_1, c_2, \ldots, c_n, 1)$, and $(c_1, c_2, \ldots, c_n, 2)$, respectively, to achieve the required order.

The algorithm to generate data movement statement domains and scattering functions to properly place data movement code in the target CUDA code structure is outlined in Algorithm 2.

Exploiting constant memory and registers. In addition to handling data movement to the on-chip shared memory, we handle on-chip constant memory and registers. Constant memory has an on-chip portion in the form of cache which can be effectively utilized to reduce global memory access. Access to constant memory is useful when a small portion of data is accessed by threads in such a fashion that all threads in a warp access the same value simultaneously. If threads in a warp access different values in constant memory, the requests get serialized. We determine arrays that are read-only and whose access function does not vary with respect to the loop iterators corresponding to the parallel loops used for distributing computation across threads. Such arrays

Algorithm 2. Generation and Placement of Data Movement Code

```
Input Set of statements - $\mathcal{S}$, Transformed Statement Domains of all statements $\mathcal{D}_s, s \in S$ from
    Algorithm 1, Affine array access functions
for each array $A$ do
    for all references of the array do
        Find the data space accessed by the references
    end for
    Partition the set of all data spaces into maximal disjoint sets such that each partition has
    a subset of data spaces each of which is non-overlapping with any data space in other
    partitions
    For each partition, find the convex union of its data spaces and the bounding box of the
    convex union gives the storage buffer needed for the partition
    for each statement $s \in S$ do
        for all read references of the array do
            Find the data space accessed by the references and use them as domains of copy-in
            statements
            Use "identity" scattering functions
        end for
        for all write references of the array do
            Find the data space accessed by the references and use them as domains of copy-out
            statements
            Use "identity" scattering functions
        end for
    end for
end for
Let the number of copy-in and copy-out statements be $c$ and $d$, respectively
Add a new dimension in all scattering functions (those of copy-in, computation, and copy-out
statements) with just a constant value; the constant being 0 to $c-1$ for copy-in statements, $c$
for computation statements, $c+1$ to $c+d$ for copy-out statements
Output Data movement statement domains and updated scattering functions
```

are candidates for storing in constant memory. Similarly, arrays whose access functions vary only with respect to the loop iterators corresponding to the parallel loops are considered as candidates for storing in registers in each thread.

4.3 Syntactic Post-processing

The transformed multi-level tiled computation statement domains and data movement statement domains along with the scattering functions (generated by Algorithms 1 and 2) are fed to CLooG to generate multi-level tiled code. Syntactic post processing of the multi-level tiled code generated by CLooG is needed to generate a final compilable CUDA code. The primary tasks of the post processing are (1) to generate thread-centric code and (2) to place synchronization calls for correct parallel execution.

An important aspect of CUDA code generation is thread-centric code generation, i.e. generation of code where the computation is distributed across the threads in the system. A thread in the system is uniquely identified by a combination of its "thread block identifier" and "thread identifier" within the thread block. We take a syntactic approach to introduce thread-centricity in the parallel code generated using the above technique. The CLooG tool has its own AST representation called the CLAST. The

CLAST generated for the parallel tiled code is parsed to introduce “thread block and thread identifiers” in the parallel loops (identified in Algorithm 1) such that the parallel tiles at the outer level are cyclically distributed across the *thread blocks* and that at the inner level are cyclically distributed across the *threads*. The data movement code is also parsed to place “thread identifier” in the data movement loops.

CUDA offers a synchronization primitive to synchronize across threads within a thread block, but no built-in synchronization primitives to synchronize across thread blocks. We introduce a primitive through a code segment that uses a “single-writer multiple-reader” technique to achieve synchronization across thread blocks using the global memory space. It is necessary to place barrier synchronizations at each iteration of a sequential loop (if any) that precedes parallel loops, and at the end of data movement loops. It is done syntactically by modifying the CLAST. Algorithm 3 summarizes the CUDA code generation steps after applying Algorithms 1 and 2.

It should be noted that the tile sizes used for tiling are fixed at compile time and provided by the user. The code generated by our framework represents the number of threads and thread blocks as symbolic constants, which the user sets before the actual execution. Our framework also syntactically inserts an “unroll” pragma - *#pragma unroll* $unroll\_factor$ - which enables the CUDA compiler to perform inner loop unrolling.

Algorithm 3. Parallel CUDA Code Generation

```
Input Computation statement domains, Data movement statement domains, Scattering functions
 1. Feed the computation and data movement statement domains and scattering functions to
    CLooG to generate CLAST
 2. Parse CLAST to change the lower bounds and loop increments of (outer and inner level)
    parallel loops to make them thread-centric
 3. Parse CLAST to change the lower bounds and loop increments of data movement loops to
    make them thread-centric
 4. Place barrier synchronization at each iteration of sequential loop (if any) that precedes paral-
    lel loops, and at the end of data movement loops
 5. Print the modified CLAST to generate CUDA code
Output Multi-level parallel tiled CUDA code with data movement
```

5 Experimental Results

In this section, we present experimental results to assess the effectiveness of the CUDA code generated by the implemented C-to-CUDA transformation system. We present results on seven benchmarks. Where available, we compare the performance of the automatically generated CUDA code with hand-tuned CUDA code. We also compare the performance of the generated CUDA code on the GPU with the performance of input C code (optimized by the Intel icc compiler), on a multi-core CPU.

The GPU device used in our experiments was an NVIDIA GeForce 8800 GTX GPU. The device has 768 MB of DRAM and has 16 multiprocessors (MIMD units) clocked at 675 MHz. Each multiprocessor has 8 processor cores (SIMD units) running at twice the clock frequency of the multiprocessor and has 16 KB of shared memory. The CUDA code was compiled using the NVIDIA CUDA Compiler (NVCC) to generate the device code that is launched from the CPU (host). The CPU was a 2.13 GHz Intel Core2 Duo

```
  for (t1=0; t1<VOLY; t1++) {
     for (t2=0; t2<VOLX; t2++) {
        for (t3=0;t3<NATOMS;t3++) {
           energy[zDim*VOLX*VOLY + t1*VOLX + t2] =
           atoms[3+4*t3]/ ... atoms[2+4*t3] ...
           atoms[1+4*t3] ... atoms[4*t3];
        }
     }
 }
```

Fig. 3. Original code structure for Coulombic Potential (cp) benchmark

processor with 2 MB L2 cache. The GPU device was connected to the CPU through a 16-x PCI Express bus. We used CUDA version 2.1 for our experiments.

The multi-core system used for our experiments was a quad-core Intel Core 2 Quad Q6600 CPU clocked at 2.4 GHz (1066 MHz FSB) with a 32 KB L1 D cache, 8MB of L2 cache (4MB shared per core pair), and 2 GB of DDR2-667 RAM, running Linux kernel version 2.6.22 (x86-64). ICC 10.x was the primary compiler used to compile the code on the multi-core system; it was run with -fast -funroll-loops (-openmp for parallelized code); the -fast option turns on -O3, -ipo, -static, -no-prec-div on x86-64 processors; these options also enable auto-vectorization in icc.

5.1 Coulombic Potential (cp)

This benchmark is used for the computation of electric potential in a volume containing point charges. It is one of the codes in the *parboil* benchmark suite from UIUC [21]. Fig. 4 presents the performance data - performance of the generated CUDA code with different optimizations is compared with the hand-tuned code from the parboil benchmark suite and icc optimized C code. The CUDA code generated by our framework performs better than the optimized version on general-purpose multi-core system. The

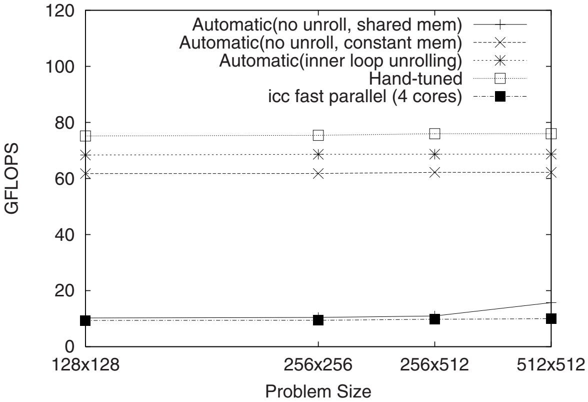


Fig. 4. Performance of cp benchmark

performance of the code generated by turning on all optimizations is very close to that of the hand-tuned code. In addition to extracting "doall' parallelism across threads and thread blocks, the code has optimized off-chip access in one of the two ways - (1) utilizing shared memory or (2) utilizing constant memory. Fig. 4 shows the performance measurements for both the cases and it can be seen that the performance when constant memory is used is significantly higher than that when shared memory is used. This is because the use of constant memory significantly reduces global memory traffic in comparison to accessing data after moving from global memory to shared memory. Inner loop unrolling was performed using NVIDIA's *#pragma unroll* option.

Figures 3 and 9 illustrate the CUDA code generation. Fig. 3 shows the structure of sequential code (along with the array accesses) for Coulombic Potential (cp) benchmark. Fig. 9 shows the structure of two-level tiled parallel code that is thread-centric where the parallelism is across thread blocks at the outer level and across threads at the inner level (Note the modified lower bounds and loop increments of parallel loops). Fig. 9 also shows the proper placement of data movement and computation statements.

5.2 N-Body Simulation (nbody)

N-body simulation is an important computation that arises in many computational science applications. It approximates the evolution of a system of bodies in which each

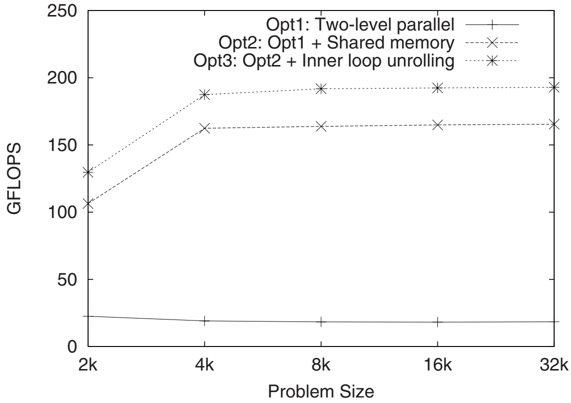


Fig. 5. Performance variation of nbody benchmark w.r.t Optimizations

Table 1. Performance of nbody benchmark (in GFLOPS)

N	Auto-CUDA	Hand-tuned	icc
2048	129.67	157.34	1.00
4096	187.41	182.31	1.10
8192	191.81	188.78	1.42
16384	192.45	198.43	1.47
32768	192.91	200.35	1.50

body continuously interacts with every other body. The CUDA code generated by our framework performs much better than the optimized version on general-purpose multi-core system and performs very comparably to the hand-tuned CUDA code, as illustrated in Table 1. The code generated by our framework exploited "doall" parallelism across threads and thread blocks. It effectively moved data from arrays that exhibited data reuse from global memory to shared memory, thereby enabling coalesced global memory access and also reduction in off-chip memory access latency, by exploiting data reuse in on-chip shared memory. Further, inner loop unrolling was performed using NVIDIA's *#pragma unroll* option. Fig. 5 depicts incremental performance improvement when different optimizations are applied. The importance of shared memory utilization and inner loop unrolling (to reduce loop overhead and dynamic loop instruction count) are illustrated by this benchmark.

Table 2. Performance of MRI-Q (in GFLOPS)

N	Auto CUDA (2)		Auto CUDA (1)		Hand	icc
	no unroll	unroll	no unroll	unroll	tuned	
32768	87.11	122.19	137.1	176.50	178.98	0.91
65536	88.27	121.87	141.7	179.32	179.12	1.14
131072	88.53	123.11	142.3	181.23	179.32	1.14
262144	89.16	122.12	142.6	183.32	180.91	1.15

Table 3. Performance of MRI-FHD (in GFLOPS)

N	Auto CUDA (2)		Auto CUDA (1)		Hand	icc
	no unroll	unroll	no unroll	unroll	tuned	
32768	57.91	90.91	112.52	142.52	143.11	1.37
65536	61.27	91.2	116.12	143.15	142.27	1.68
131072	62.13	91.6	116.22	144.21	144.39	2.19
262144	62.67	91.52	116.67	142.61	144.43	2.21

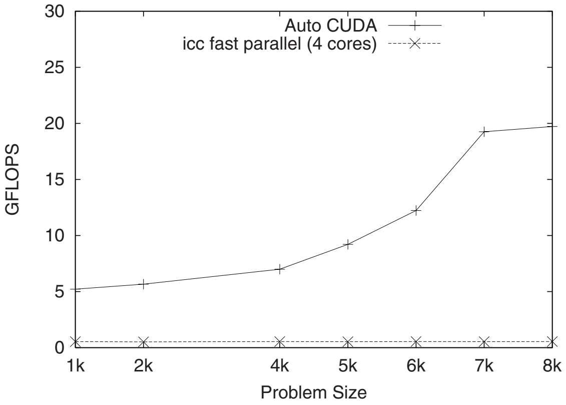


Fig. 6. Performance of 2D Jacobi

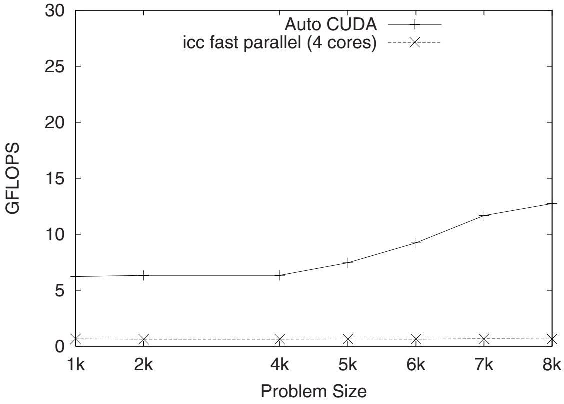


Fig. 7. Performance of 2D FDTD

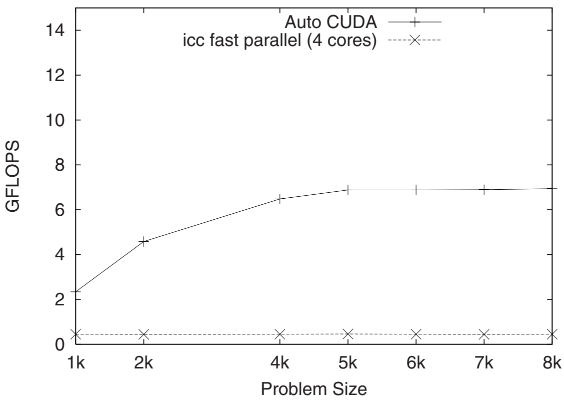


Fig. 8. Performance of Gauss Seidel

The hand-tuned version was taken from the NVIDIA CUDA SDK, the code being based on the article in [16]. The code generated by our framework represents the number of threads and thread blocks as symbolic constants, which the user sets before the actual execution.

5.3 MRI Kernels

We employed our framework to generate code for two kernels used in Magnetic Resonance Imaging, MRI-Q and MRI-FHD [21]. Both the kernels involve two computational blocks such that data computed in the first computation block is used as "read-only" data in the second computational block. The hand-tuned code from parboil optimizes the two computational blocks independently and executes them as separate GPU kernels. We used our framework to generate two versions of code for each of the two MRI kernels - version (1) in which CUDA code is generated independently for the two computational blocks (first block pre-computes data for second block) and version (2) in which unified CUDA code is generated for both blocks.

Tables 2 and 3 summarize the performance measures of the code versions of MRI-Q and MRI-FHD, respectively. The code version (1) generated as two separate GPU kernels outperforms the code version (2) generated as single GPU kernel because of the fact that in version (1) the data precomputed in the first GPU kernel is stored in constant memory and accessed in the second kernel. However both the versions identified various data arrays as candidates for constant memory and thereby optimized off-chip memory

```
int by = blockIdx.y;
int bx = blockIdx.x;
int ty = threadIdx.y;
int tx = threadIdx.x;

int t1,t2,t3,t4,t5,t6;
// Parallel loops distributed across thread blocks
// Loops modified syntactically for thread block identifiers
for (t1=by; t1<=floord(VOLY-1,16); t1+=NBLKSY) {
  for (t2=bx; t2<=floord(VOLX-1,16); t2+=NBLKSX) {
    for (t3=0;t3<=NATOMS-1;t3+=256) {
      // Data movement code
      __shared__ float atomsS[1024];
      for (t6=4*t3+THREADY*NTHRDSX+THREADX;
          t6<=min(4*NATOMS-1,4*t3+1023);
          t6+=NTHRDSX*NTHRDSY)
        atomsS[t6-4*t3] = atoms[t6];
       __syncthreads();
      // Parallel loops distributed across threads
      // Loops modified syntactically for thread identifiers
      for (t4=max(0,16*t1)+ty;
          t4<=min(VOLY-1,16*t1+15);t4+=NTHRDSY) {
        for (t5=max(0,16*t2)+tx;
            t5<=min(VOLX-1,16*t2+15);t5+=NTHRDSX) {
              ...

          // Computation code
          for (t6=t3; t6<=min(NATOMS-1,t3+255);
              t6++) {
            energy[zDim*VOLX*VOLY + t4*VOLX + t5] =
            atomsS[3+4*t6-4*t3]/ ... atomsS[2+4*t6-4*t3] ...
            atomsS[1+4*t6-4*t3] ... atomsS[4*t6-4*t3];
          }
        }
      }
    }
  }
}
```

Fig. 9. Parallel tiled code structure (with data movement) for cp benchmark

access. The code version (1) generated by our framework performs as well as the hand-tuned version.

5.4 Stencil Computation Kernels

We used two stencil computation kernels, 2D Jacobi and 2D Finite Difference Time Domain (FDTD). The code generated using our framework performs better than the optimized version on the Intel multi-core system, as illustrated in Figures 6 and 7. For these two kernels, we were unable to find any hand-tuned CUDA code to compare against. The code generated by our framework exploits parallelism across threads and thread blocks and effectively utilizes shared memory and exploits data reuse. The parallel execution of stencil computations is characterized by synchronization overhead at every time step across the processors. This overhead is particularly costly in GPUs where the thread blocks have to synchronize using the slow off-chip memory. This is the reason for the lower absolute performance of these kernels on GPUs, relative to the previous benchmarks. The performance of the stencil kernels is very low for smaller problem sizes for the same reason.

5.5 Gauss Seidel Successive over Relaxation

The Gauss Seidel benchmark illustrates the effect of exploiting wavefront or pipelined parallelism on GPUs. We achieve better performance than the optimized version on multi-core system, as illustrated in Fig. 8. However, the absolute performance is rather low because of (1) low processor utilization during the starting and draining of pipeline and (2) synchronization overhead across thread blocks at every time step.

6 Related Work

In this Section, we review prior work on optimizations and code generation for GPUs.

Ryoo et al. [27,26] presented experimental studies of program performance on NVIDIA GPUs using CUDA; they do not use or develop a compiler framework for optimizing applications, but rather perform the optimizations manually. Ryoo et al. [28] presented performance metrics such as *efficiency* and *utilization* to prune the optimization search space on a pareto-optimality basis. However, they manually generate the performance metrics data for each application they have studied. The end-to-end system described in this paper builds on our prior work [2,3] that developed some of the compiler optimizations - optimizing global memory and shared memory access, and utilizing and managing on-chip shared memory. Recently, Lee et al. [17] developed a compiler framework for automatic translation from OpenMP to CUDA. The system handles both regular and irregular programs parallelized using OpenMP primitives. Work sharing constructs in OpenMP are translated into distribution of work across threads in CUDA. However the system does not optimize data access costs for access in global memory and also does not make use on-chip shared memory. Thus the optimizations implemented in our system can complement and enhance the effectiveness of their system.

Recently, Liu et al. [19] developed a GPU adaptive optimization framework (G-ADAPT) for automatic prediction of near-optimal configuration of parameters that affect GPU performance. They take unoptimized CUDA code as input and traverse an optimization space search to determine optimal parameters to transform the unoptimized input CUDA code into an optimized CUDA code. Using our framework, a user can automatically generate CUDA code for any arbitrary input affine C code, hand-parallelization of which is very cumbersome in many cases. The user may then use G-ADAPT to further tune the CUDA code generated from our system.

7 Conclusions

In this paper, we have described an automatic source-to-source transformation framework that can take an arbitrarily nested affine input C program and generate an efficient CUDA program. Experimental results demonstrated the performance improvements achieved using the framework. We are in the process of creating a publicly available release of the C-to-CUDA transformation software.

Acknowledgments. This work was supported in part by the U.S. National Science Foundation through awards 0403342, 0508245, 0509442, 0509467, 0541409, 0811457, 0811781, 0926687 and 0926688, and by the Department of the Army through contract W911NF-10-1-0004.

References

1. Ancourt, C., Irigoin, F.: Scanning polyhedra with do loops. In: PPoPP 1991, pp. 39–50 (1991)
2. Baskaran, M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In: ACM ICS (June 2008)
3. Baskaran, 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: ACM SIGPLAN PPoPP (February 2008)
4. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT 2004, pp. 7–16 (2004)
5. Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 132–146. Springer, Heidelberg (2008)
6. Bondhugula, U., Hartono, A., Ramanujan, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: ACM SIGPLAN Programming Languages Design and Implementation, PLDI 2008 (2008)
7. CLooG: The Chunky Loop Generator, `http://www.cloog.org`
8. Fatahalian, K., Sugerman, J., Hanrahan, P.: Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In: ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, pp. 133–137 (2004)
9. Feautrier, P.: Dataflow analysis of array and scalar references. IJPP 20(1), 23–53 (1991)
10. Feautrier, P.: Some efficient solutions to the affine scheduling problem, part I: one-dimensional time. IJPP 21(5), 313–348 (1992)
11. Feautrier, P.: Automatic parallelization in the polytope model. In: Perrin, G.-R., Darte, A. (eds.) The Data Parallel Programming Model. LNCS, vol. 1132, pp. 79–103. Springer, Heidelberg (1996)
12. Govindaraju, N.K., Larsen, S., Gray, J., Manocha, D.: A memory model for scientific algorithms on graphics processors. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089. Springer, Heidelberg (2006)

13. General-Purpose Computation Using Graphics Hardware, http://www.gpgpu.org/
14. Griebl, M.: Automatic Parallelization of Loop Programs for Distributed Memory Architectures. Habilitation Thesis. FMI, University of Passau (2004)
15. Irigoin, F., Triolet, R.: Supernode partitioning. In: Proceedings of POPL 1988, pp. 319–329 (1988)
16. Nyland, L., Harris, M., Prins, J.F.: Fast N-body Simulation with CUDA. GPU Gems 3 article (August 2007)
17. Lee, S., Min, S.-J., Eigenmann, R.: Openmp to gpgpu: A compiler framework for automatic translation and optimization. In: PPoPP 2009, pp. 101–110 (2009)
18. Lim, A.: Improving Parallelism And Data Locality With Affine Partitioning. PhD thesis, Stanford University (August 2001)
19. Liu, Y., Zhang, E.Z., Shen, X.: A cross-input adaptive framework for gpu programs optimizations. In: IPDPS (May 2009)
20. NVIDIA CUDA, http://developer.nvidia.com/object/cuda.html
21. Parboil Benchmark Suite, http://impact.crhc.illinois.edu/parboil.php
22. Pluto: A polyhedral automatic parallelizer and locality optimizer for multicores http://pluto-compiler.sourceforge.net
23. Pouchet, L.-N., Bastoul, C., Cohen, A., Vasilache, N.: Iterative optimization in the polyhedral model: Part I, one-dimensional time. In: CGO 2007, pp. 144–156 (2007)
24. Pugh, W.: The Omega test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM 8, 102–114 (1992)
25. Quilleré, F., Rajopadhye, S.V., Wilde, D.: Generation of efficient nested loops from polyhedra. IJPP 28(5), 469–498 (2000)
26. Ryoo, S., Rodrigues, C., Baghsorkhi, S., Stone, S., Kirk, D., Hwu, W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: ACM SIGPLAN PPoPP 2008 (February 2008)
27. Ryoo, S., Rodrigues, C., Stone, S., Baghsorkhi, S., Ueng, S., Hwu, W.: Program optimization study on a 128-core GPU. In: The First Workshop on General Purpose Processing on Graphics Processing Units (October 2007)
28. Ryoo, S., Rodrigues, C., Stone, S., Baghsorkhi, S., Ueng, S., Stratton, J., Hwu, W.: Program optimization space pruning for a multithreaded GPU. In: CGO (2008)
29. Vasilache, N., Bastoul, C., Girbal, S., Cohen, A.: Violated dependence analysis. In: ACM ICS (June 2006)

Is Reuse Distance Applicable to Data Locality Analysis on Chip Multiprocessors?

Yunlian Jiang, Eddy Z. Zhang, Kai Tian, and Xipeng Shen

Computer Science Department
The College of William and Mary, Williamsburg, VA, USA
{jiang,eddy,ktian,xshen}@cs.wm.edu

Abstract. On Chip Multiprocessors (CMP), it is common that multiple cores share certain levels of cache. The sharing increases the contention in cache and memory-to-chip bandwidth, further highlighting the importance of data locality analysis.

As a rigorous and hardware-independent locality metric, reuse distance has served for a variety of locality analysis, program transformations, and performance prediction. However, previous studies have concentrated on sequential programs running on unicore processors. On CMP, accesses by different threads (or jobs) interact in the shared cache. How reuse distance applies to the new architecture remains an open question—particularly, how the interactions in shared cache affect the collection and application of reuse distance, and how reuse-distance–based locality analysis should adapt to such architecture changes.

This paper presents our explorations towards answering those questions. It first introduces the concept of *concurrent reuse distance*, a direct extension of the traditional concept of reuse distance with data references by all co-running threads (or jobs) considered. It then discusses the properties of concurrent reuse distance, revealing the special challenges facing the collection and application of concurrent reuse distance on CMP platforms. Finally, it presents the solutions to those challenges for a class of multithreading applications. The solutions center on a probabilistic model that connects concurrent reuse distance with the data locality of each individual thread. Experiments demonstrate the effectiveness of the proposed techniques in facilitating the uses of concurrent reuse distance for CMP computing.

1 Introduction

Because of the well-known memory wall problem, on traditional architecture, data locality has been one of the most prominent factors that determine the performance of a program. Its importance becomes even more pronounced on modern Chip Multiprocessors (CMP), where cache and memory bandwidth are shared by a growing number of cores.

In decades of locality research on uni-core architecture, two classes of metrics have been used. One is on the hardware level; an example is cache miss rates. The other is on the program level; reuse distance is a representative. Reuse distance is also called LRU stack distance [19], referring to the number of distinct data elements referenced between

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 264–282, 2010.

this and the previous accesses to the same data element [9]. Unlike hardware-level metrics, reuse distance is inherent to a program, independent to hardware configurations but applicable for the performance prediction of various hardware. It is accurate, and from point to point (from one access to another). In contrast, a cache miss rate is an average value over an interval. Furthermore, reuse distance appears to be cross-input predictable for many programs [9,36]. These features make it appealing for a wide range of uses in software refactoring [3], data reorganization [34,36], performance prediction [32,33,17], memory disambiguation [10,11], software-controlled object-level partitioning [16], and so forth.

The rise of multicore has complicated the characterization of data locality. With cache being shared among multiple cores, accesses to memory by a process are not solely determined by that process itself, but also affected by the other processes running on the same chip. The processes (or threads) that co-run on a chip equipped with shared cache are called the *cache sharers* or *co-runners* of one another.

Many recent studies [14,20,21,12] in the architecture area have started to explore the implications of such architectural changes to the application of hardware-level locality metrics. But we are not aware of any such systematic studies on reuse distance.

In this work, we initiate an exploration in that direction. The exploration reveals that in CMP environments, reuse distance loses some of its appealing properties, and that loss impairs many of its uses. However, for a large class of multithreading programs, the loss is remediable through a probabilistic model that connects co-run locality with the memory behaviors of individual cache sharers.

Specifically, our exploration includes three components. First, we analyze the complexities in extending the traditional reuse distance model to CMP environments (*Section 2*). The analysis is based on a straightforward extension of the concept of reuse distance. In the measurement of a reuse distance, the extended concept counts the number of distinct data elements of *all cache sharers* that are accessed between two consecutive references to the same data element. For clarity, we call such a reuse distance *concurrent reuse distance* and the traditional one *standalone reuse distance*. By comparing these two types of reuse distance, we uncover the loss of hardware-independence by concurrent reuse distance and the special challenges in its measurement. We show that the loss of hardware-independence causes a chicken-egg dilemma for performance prediction. Furthermore, the dilemma is hard to resolve through the standard iterative approach.

Second, by drawing on the observations exposed in a recent study, we find that the hardware dependence of concurrent reuse distance can be relaxed for a class of multithreading applications (*Section 3*). Based on the relaxation, we develop a probabilistic model to capture the statistical connections between concurrent reuse distance and standalone data locality for multithreading applications. The model simplifies the attainment of concurrent reuse distance, laying the foundation for many of its uses.

Finally, we evaluate the accuracy of the probabilistic model on both synthetic and real traces (*Section 4*). The results demonstrate that with the probabilistic model, concurrent reuse distance can be obtained in a reasonable accuracy, suggesting its potential for locality enhancement in CMP environments. We conclude the paper with some discussions on the potential uses and limitations of concurrent reuse distance, some related work, and a short summary.

2 Concept and Properties of Concurrent Reuse Distance

Concurrent reuse distance is a direct extension of the traditional concept of reuse distance (standalone reuse distance). This section discusses the distinctive complexities of concurrent reuse distance and the implications by comparing it with standalone reuse distance. As a preparation, we first review the properties and uses of standalone reuse distance.

2.1 Review of Standalone Reuse Distance and Its Properties

Standalone reuse distance is a widely used locality model on traditional architecture without cache sharing. It is also called LRU stack distance [19], defined as the number of distinct data elements accessed between the current and the previous references to the same element [9]. Its appealing properties include the following.

- *Rigorousness:* Standalone reuse distance is point-to-point, offering a rigorous measurement of locality. In contrast, a cache miss rate is an average value over an interval, and its value depends on the length of the interval.
- *Value:* The value of standalone reuse distance is bounded—no greater than the number of distinct data elements in the program. This property has simplified the search for patterns between standalone reuse distance and program data size [9].
- *Cross-Input Predictability:* A number of studies have shown that the standalone reuse distance histograms of many programs are predictable across program inputs [9,36,24,17]. This property is essential for its uses in program performance prediction.
- *Independence on Hardware:* Standalone reuse distance is a program-level attribute, determined by the program and input data sets, independent to the hardware configurations. But on the other hand, it is strongly related to hardware performance. As shown in Figure 1, from the histogram of reuse distance, it is easy to estimate the cache miss rate of the execution on an arbitrary cache.

Different levels of standalone reuse distance suit different uses. The first class of uses are for cross-architecture prediction of cache miss rates (as illustrated in Figure 1) and program performance [17,33,32]. The used reuse distance histograms are typically on the whole-program level, with the accesses of all data in the execution considered.

The second class of uses is for program refactoring [3], data reorganization [34,36], and software-controlled object-level partitioning [16]. For these uses, the reuse distance is typically on the object level; each reuse distance histogram corresponds to an important data object (e.g., an array) in the program. Such a histogram reflects the match or mismatch of cache and the accesses to the object, offering hints for data transformations or cache partition.

The third class of uses is on the instruction level. From the distance of data store instructions, Fang and others [11] accurately determine on which specific store instruction a load depends, and use that information for memory disambiguation.

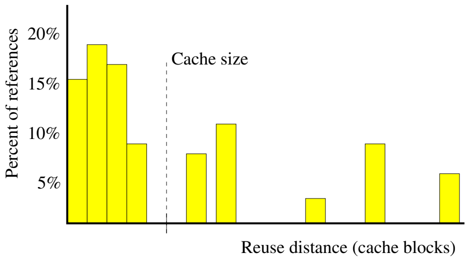


Fig. 1. The histogram of the standalone reuse distance of an execution. Every memory reference on the right side of the cache-size line is considered a cache miss because too many other data have been brought into cache since its previous reference.

2.2 Concurrent Reuse Distance

Concurrent reuse distance is a straightforward extension of standalone reuse distance for programs running on shared cache. It is defined as the number of distinct data elements that *all sharers* of a cache access between the current and the previous references to the same data element. In this section, we consider the general case, where cache sharers can be independent programs, or threads of parallel applications that share the same address space.

Properties. As a straightforward extension, concurrent reuse distance keeps some properties that standalone reuse distance has. It is point-to-point, and its value is bounded, no greater than the sum of the numbers of total distinct data elements of all cache sharers.

However, concurrent reuse distance has a distinctive property:

> Its value depends on the relative execution speeds of cache sharers.

Consider two processes, P_1 and P_2, running on a chip with shared cache. The process P_1 conducts a sequence of memory references as $abcba$ (each letter for one data element) during a time interval T (called a *reuse internal*). Without loss of generality, suppose P_1 and P_2 access different sets of data in that interval. The concurrent reuse distance of the second access to a is $2 + x$, where 2 is the number of distinct data elements (b and c) accessed by P_1 in that time interval, and x is the number of distinct data elements accessed by P_2 in that time interval. The value of x depends on the relative speeds of the two processes, $r = Speed(P_2)/Speed(P_1)$. The larger r is, the more data are likely to be accessed by P_2 in that time interval, and hence the greater x tends to be.

Challenges. This property results in some implications important for the measurement and application of concurrent reuse distance.

Challenges to Measurement. Traditional approaches are insufficient for measuring concurrent reuse distance. A typical way to obtain standalone reuse distance is through program instrumentation, which inserts memory monitoring and other relevant instructions into the program code so that when the program runs, the inserted instructions would collect the memory reference trace and compute the standalone reuse distance.

The instrumented program typically runs hundreds of times slower than the original program does. This slowdown causes inconveniences but no errors to the collection of standalone reuse. However, for concurrent reuse distance, the slowdown would change the relative running speed r among cache sharers, hence causing measurement errors.

To examine the seriousness of this problem, we measure how much the relative speed r changes because of the instrumentation. We use a set of randomly chosen SPEC CPU2000 programs and a dual-core Xeon 7120M with 4MB shared L3 cache. For every pair of the programs, say program i and j, we first run them on two sibling cores and record their respective average IPCs (instructions per cycle), denoted as IPC_i and IPC_j, by reading the hardware performance counters through PAPI [5]. We then use PIN [37] to instrument the programs with the code for the collection of standalone reuse distance. We run the instrumented version on the two sibling cores and record the new IPCs as IPC'_i and IPC'_j. The relative running speeds before and after the instrumentation are $r = IPC_i/IPC_j$, and $r' = IPC'_i/IPC'_j$. We compute the changes of the relative running speed as follows:

$$change\ \ of\ \ relative\ \ speeds = |r - r'|/r. \tag{1}$$

Table 1 reports the results. After instrumentation, the differences of the speeds of those programs become smaller than before. This phenomenon is intuitive considering that the instrumented code dominates the running time of all programs. (The instrumented code is similar for all programs.) The 31–248% changes of the relative speeds caused by the instrumentation suggest the large departure of the measured concurrent distance from the real. This large departure hurts many uses of concurrent reuse distance as most typical uses of reuse distance—such as program refactoring, data reorganization, object-level partitioning, memory disambiguation—have relied on an accurate measurement of the reuse distances.

Deprivation of Hardware-Independence. The reliance on relative execution speeds of cache sharers deprives concurrent reuse distance of hardware-independence. That independence has been a property important for many uses of standalone reuse distance. The deprivation is because the variance in running environments—such as the cache size, the number of cores per chip, the operating systems—often affects the running speeds of different programs in different degrees, and hence changes the relative speeds.

Table 1. Changes of Relative Running Speeds Caused by Program Instrumentation

sharers	1 & 4	2 & 4	3 & 4	1 & 5	2 & 5	3 & 5	1 & 6	2 & 6	3 & 6
r	0.40	0.59	0.25	0.44	0.55	0.43	0.37	0.48	0.30
r’	0.77	0.77	0.87	0.93	0.89	0.93	0.99	1.11	0.88
changes (%)	92.5	30.5	248	111	61.8	116	168	131	193

* programs: 1-ammp; 2-art; 3-mcf; 4-bzip2; 5-gzip; 6-mesa.

Table 2. Changes of Relative Running Speeds Due to Architectural Differences

sharers	1 & 4	2 & 4	3 & 4	1 & 5	2 & 5	3 & 5	1 & 6	2 & 6	3 & 6
r	0.40	0.59	0.25	0.44	0.55	0.43	0.37	0.48	0.30
r'	0.40	0.48	0.3	0.57	0.72	0.54	0.37	0.48	0.31
changes (%)	0	18.6	20.0	29.5	30.9	25.6	0	0	3.3

* r: on Xeon E5310; r': on Xeon 7120M.

* programs: 1-ammp; 2-art; 3-mcf; 4-bzip2; 5-gzip; 6-mesa.

Table 2 shows the changes of the relative speeds when the co-runs happen on a quad-core Intel Xeon E5310 processor with two 4MB L2 cache on each chip, compared to their co-runs on the dual-core Xeon 7120M. Four of the co-runs show negligible changes. Examination shows that the IPC of each of the programs does change considerably in the two architectures. For instance, in the co-runs of *ammp* and *bzip2* (i.e., 1&4) , the IPCs of *ammp* are respectively 0.34 and 0.79 on the two machines. But the IPC of *bzip2* changes proportionally from 0.72 to 1.97. Their relative speeds hence remain the same. However, the relative speeds of the other five co-runs do change substantially, from 19% to 31%, reflecting the hardware-dependence of concurrent reuse distance.

Chicken-Egg Dilemma. As a consequence of the hardware-dependence, one of the main uses of standalone reuse distance—cross-architecture performance prediction [9,36,24,17]—becomes difficult if not impossible for concurrent reuse distance. The difficulty is a chicken-egg dilemma as illustrated in Figure 2. The ultimate target of the performance prediction is the IPC on the new platform, which is supposed to be predicted from the reuse distance collected on a training platform. This prediction is possible for standalone reuse distance because the distance is inherent to the program (and input data sets) and do not change across architecture. However, the prediction becomes difficult for concurrent reuse distance because of its dependence of architecture. To solve this issue, we need first predict the concurrent reuse distance on the new architecture. This prediction however depends on exactly what the concurrent reuse distance is used to predict—the IPC. This inter-dependence forms a chicken-egg dilemma, as showed by the circular flow in Figure 2.

A typical way to solve such kind of dilemmas is through iterative processes. For this particular problem, the process may start with guessing some initial values for the IPCs of co-running jobs. Suppose we have two co-running jobs, I and J and their IPCs are initially guessed to be $IPC_0(I)$ and $IPC_0(J)$. The concurrent reuse distances can then

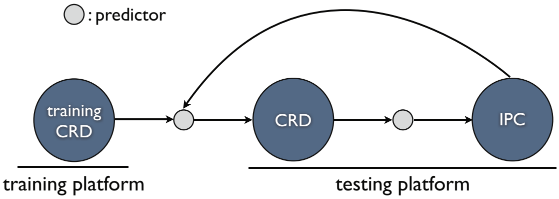


Fig. 2. The difficulty in applying concurrent reuse distance (CRD) to cross-platform performance prediction. The inter-dependence between CRD and IPC forms a chicken-egg dilemma.

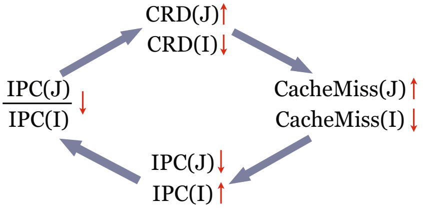


Fig. 3. Analysis showing that the iterative approach cannot solve the chicken-egg dilemma in performance prediction based on concurrent reuse distance

be approximated, denoted as $CRD_0(I)$ and $CRD_0(J)$, from which, the IPCs can be updated accordingly to $IPC_1(I)$ and $IPC_1(J)$. This process continues until reaching a stable point, where the distances or the IPCs remain constant across iterations.

Further analysis however shows that this iterative process does not work for concurrent reuse distance as illustrated in Figure 3. Without loss of generality, assume

$$\frac{IPC_1(J)}{IPC_1(I)} < \frac{IPC_0(J)}{IPC_0(I)}.$$

It means that the relative speed of J (with the speed of I in the respective iteration as the baseline) becomes lower in iteration 1 than in iteration 0. So more data of I would be likely to be accessed in a reuse interval of J in iteration 1 than in iteration 0. The result is that the distances in $CRD_1(J)$ would be larger than those in $CRD_0(J)$. For the opposite reason, the distances in $CRD_1(I)$ would be smaller than those in $CRD_0(I)$. Consequently, the number of cache misses predicted from $CRD_1(J)$ would be higher than from $CRD_0(J)$, leading to a further decrease of $IPC(J)$ and a further increase of $IPC(I)$. Hence, $(IPC_2(J)/IPC_2(I))$ would become even smaller than $(IPC_1(J)/IPC_1(I))$. This decreasing trend would continue for more iterations until the ratio becomes 0 (unless it stops in a local trap).

In summary, this section shows that despite being a direct extension of standalone reuse distance, concurrent reuse distance is both hard to measure and difficult to use in multicore environment. These conclusions are obtained through an analysis of general co-runs. Fortunately, the next section will show that these difficulties can be overcome for a class of important multithreading programs.

3 Concurrent Reuse Distance for Multithreading Programs

The previous section shows the difficulties in applying concurrent reuse distance to *independent* programs co-running on a multicore platform. This section shows that for co-running threads of a *multithreading* program, those obstacles are circumventive.

The solution is based on some recently uncovered features of the execution of multithreading programs on multicore processors. At the kernel of the solution is a probabilistic model that connects concurrent reuse distance with the data locality of each individual thread. We first examine the features of the multithreading programs and then present the probabilistic model.

Table 3. Relative Speeds of Co-Running Threads in Multithreading Applications and the Changes Due to Architecture and Input Variations

input	machine	IPC(thread 0)/IPC(thread 1) for programs						
		blackscholes	bodytrack	canneal	facesim	fluidanimate	streamcluster	swaptions
simlarge	7120M	1.00	0.96	1.00	1.00	1.00	1.00	1.00
	E5310	1.00	1.00	1.00	1.00	0.99	1.00	1.00
native	7120M	1.00	0.92	1.00	1.00	0.99	1.00	1.00
	E5310	1.00	0.99	1.00	1.01	0.99	1.00	1.00
changes by arch. (%)		0	5.9	0	0	0.5	0	0
changes by input (%)		0	2.6	0	0.5	0.5	0	0

3.1 Independence to Architecture and Inputs

A recent study [30] on PARSEC [4], a suite of contemporary multithreading benchmarks, exhibits two phenomena. First, for most of those programs (except pipelining programs), all parallel threads conduct similar computations. Second, the relations among threads, in terms of the amount of shared data and communications, are quite uniform across different thread groups. These phenomena hold across architectures, numbers of threads, assignment of threads to cores, input data sets, and program phases.

The implication to concurrent reuse distance is that contrary to those of the co-runs of independent programs, the relative speeds among threads tend to remain the same across architectures and program inputs. For confirmation, we run all the non-pipelining pthread programs in PARSEC on two types of architectures. One is quad-core Intel Xeon E5310 processors with two 4MB L2 cache on each chip. The other is dual-core Xeon 7120M with 4MB shared L3 cache. We employ two inputs for each program, a small one (*simlarge*) and a large one (*native*). The running times on the two inputs differ by a factor of 43 to 180. In every run, 8 threads are created and are bound to cores such that adjacent threads (e.g., threads 0 and 1) are ensured to run on two sibling cores with cache shared.

Table 3 reports the relative running speeds between two co-running threads (e.g., *IPC(thread 0)/IPC(thread 1)*). The numbers are the average of five repetitive runs (negligible variation appears among the five runs). The bottom two rows are the average changes in the relative speeds— computed in a similar way as Formula 1 in Section 2.2—caused respectively by the variation of architecture and program inputs. The unanimous close-to-1 relative speeds indicate that two co-running threads have virtually the same speeds, no matter on what machine they run or what inputs they use.

We note that the absolute speeds of two co-running threads do change across architectures and inputs. But they change in the same rate so that their relative speed remains the same. As it is the relative speed that matters to the concurrent reuse distance, the results confirm the independence of the concurrent reuse distance of those programs to architecture and input data sets.

3.2 Probabilistic Model for Approximating Concurrent Reuse Distance

The previous section suggests that concurrent reuse distance is potentially useful for a class of multithreading applications. To realize the potential, it is important to explore the connections between concurrent reuse distance and the memory behaviors of individual threads. The rationale is that if concurrent reuse distance can be derived from the locality information of each individual cache sharer, the appealing properties of standalone locality would directly benefit the prediction and application of concurrent reuse distance.

Overview. We propose a probabilistic model to derive concurrent reuse distance histogram from locality information of each individual thread. The model starts with the locality of individual threads, characterized with *time distance histograms*. *Time distance* is defined as the number of memory references in a reuse interval[1]. In the reference sequence "a b b c a", the time distance of the final access is 4 (while the reuse distance is 2.) *Time distance histogram* is similar to the reuse distance histogram shown in Figure 1 except that the X-axis is replaced by time distance.

The probabilistic model includes two parts. The first computes the number of distinct data elements accessed by each cache sharer in an arbitrary time interval. The second handles the effects that data sharing among threads imposes on concurrent reuse distance. The next two sub-sections explain the two parts respectively.

Part I: From Time to Data Accesses. Let $M^{(j)}(\Delta)$ represent the statistical expectation of the number of distinct data accessed by process j in an arbitrary Δ-long time interval. The goal of this part of the model is to compute $M^{(j)}(\Delta)$ from the time distance histogram of the process j.

The computation includes three steps. *Step 1:* From the time distance histogram of each data object, we calculate the probability for a data object, say O_i, of process j to appear in a Δ-long time interval, denoted by $P_i(\Delta)$. *Step 2:* From $P_i(\Delta)$ ($i = 0, 1, \cdots, N-1$; N is the total number of distinct data objects ever accessed by process j in its entire execution), we obtain the probability for that interval to contain k ($k = 0, 1, \cdots, N$) distinct objects of process j, denoted by $P(k, \Delta)$. *Step 3:* From $P(k, \Delta)$, we compute the expected number of distinct objects that process j accesses in the interval, which is the value of $M^{(j)}(\Delta)$. We explain each of the three steps as follows.

Compute $P_i(\Delta)$

For the object O_i to be accessed in a Δ-long interval, it can be either accessed in the first Δ-1 time points, or, not until the end of the interval. With $q_i(\Delta)$ representing the probability for the data to be not accessed until the end of the interval, $P_i(\Delta)$ can be expressed as

$$P_i(\Delta) = P_i(\Delta - 1) + q_i(\Delta).$$

Hence the following equations:

$$\begin{aligned}
P_i(\Delta - 1) &= P_i(\Delta - 2) + q_i(\Delta - 1);\\
P_i(\Delta - 2) &= P_i(\Delta - 3) + q_i(\Delta - 2);\\
\cdots &\quad \cdots\\
P_i(1) &= P_i(0) + q_i(1).
\end{aligned}$$

[1] We use logical time—that is, the number of data references—for the length of an interval.

Apparently $P_i(0)$ is 0 (no objects can be accessed in a 0-long interval.) Deduction from these equations produces the following formula:

$$P_i(\varDelta) = \sum_{\tau=1}^{\varDelta} q_i(\tau). \tag{2}$$

Notice that $q_i(\tau)$ equals the probability for O_i to *1)* be the final data reference in an interval of length τ, and meanwhile, *2)* have a time distance larger than τ at that data reference (otherwise, it would be also accessed at other points in that interval.) With $p_i^{(1)}$ and $p_i^{(2)}$ respectively denoting the probabilities for the two conditions to hold, $q_i(\tau)$ can be computed as $q_i(\tau) = p_i^{(1)} p_i^{(2)}$.

The probability $p_i^{(2)}$ comes directly from the time distance histogram (denoted as H_i) of object O_i as $\sum_{\delta=\tau+1}^{T} H_i(\delta)$. With $p_i^{(1)} = n_i/T$ (n_i is the total references to O_i in all the T data references in the execution), $q_i(\tau)$ can be computed as

$$q_i(\tau) = \frac{n_i}{T} \sum_{\delta=\tau+1}^{T} H_i(\delta). \tag{3}$$

Together, Equations 2 and 3 lead to the following computation of $P_i(\varDelta)$ from the time distance histogram:

$$P_i(\varDelta) = \frac{n_i}{T} \sum_{\tau=1}^{\varDelta} \sum_{\delta=\tau+1}^{T} H_i(\delta). \tag{4}$$

Compute $P(k,\varDelta)$ and $M^{(j)}(\varDelta)$

With $P_i(\varDelta)$ ($i = 0, 1, \cdots, N-1$), we can compute the probability for an interval to contain k distinct data, denoted as $P(k,\varDelta)$ as follows:

> $P(k,\varDelta) = \sum_S$ (the probability for the interval to contain and only contain all the members of S),

where, S is a k-member subset of $A = \{O_1, O_2, \cdots, O_{N-1}\}$. Using $P_i(\varDelta)$, $P(k,\varDelta)$ can be computed as follows[2]:

$$P(k,\varDelta) = \sum_{S:|S|=k; S\subseteq A} \left(\left(\prod_{i\in S} P_i(\varDelta)\right) \left(\prod_{j\in A-S} (1 - P_j(\varDelta))\right) \right). \tag{5}$$

[2] This computation, as most trace-based locality analyses (e.g., [8,25,23]), assumes data distribute independently from one another. Results of those previous studies have shown minor influence of the assumption on locality characterization when the program contains a large number of data.

Recall that $M^{(j)}(\Delta)$ is the statistical expectation of the number of distinct data accessed by process j in an arbitrary time interval of length Δ. According to the definition of statistical expectation, we can compute $M^{(j)}(\Delta)$ from $P(k, \Delta)$ as follows:

$$M^{(j)}(\Delta) = \sum_{k=0}^{min(\Delta-1,N)} k \cdot P(k, \Delta) \tag{6}$$

Discussion. When there are no data sharing among cache sharers, a combination of their $M^{(j)}(\delta)$s ($j = 1, 2, \ldots$,# of sharers) is enough to approximate their concurrent reuse distance histograms. Let d be the time distance of a data reuse by process j. Suppose d_i is the number of memory references by one of its cache sharers, process i, during the same (physical) time period. The concurrent reuse distance of process j can be computed as $M^{(j)}(d) + \sum_{i \in j's\ co-runners} M^{(i)}(d_i)$. (Note, the values of d and d_is may be different, depending on the relative speeds of cache sharers.)

This combination, however, is not sufficient for co-running threads in multithreading applications because of the effects of inter-thread data sharing.

Part II: Handling Data Sharing. In this section, we use the following example for explanation. There are two co-running threads T_1 and T_2. Suppose in a certain time period, the memory reference sequence is

$$\text{a b } \underline{\text{X}}\ \underline{\text{X}} \text{ b } \underline{\text{X}} \text{ c d } \underline{\text{X}} \text{ a}$$

where, an $\underline{\text{X}}$ represents some reference conducted by T_2, and the other letters represent the references by T_1. Clearly, this time period corresponds to a reuse interval of reference to “a” in the standalone execution of T_1 with standalone reuse distance of 3 (for accesses to b, c, and d). We now examine its corresponding concurrent reuse distance for element “a” in three scenarios.

- Scenario 1: All $\underline{\text{X}}$s are something different from the data accessed by T_1. Let the four $\underline{\text{X}}$s be “p q p q”. Apparently, the concurrent reuse distance of the reuse interval is just the sum of the numbers of distinct data in each of the two standalone reference sequences: $3 + 2 = 5$.
- Scenario 2: The four $\underline{\text{X}}$s are “p a p q”. This scenario illustrates the first effect of data sharing. The reference to “a” breaks the reuse interval into two: “a b $\underline{\text{p}}$ $\underline{\text{a}}$” and “$\underline{\text{a}}$ b $\underline{\text{p}}$ c d $\underline{\text{q}}$ a”. The consequence is that the original reuse interval becomes meaningless. The approximation of the ultimate concurrent reuse distances of T_1 has to include a reuse distance of 2 (for “a b $\underline{\text{p}}$ $\underline{\text{a}}$”) and a reuse distance of 5 (for “$\underline{\text{a}}$ b $\underline{\text{p}}$ c d $\underline{\text{q}}$ a”).
- Scenario 3: The four $\underline{\text{X}}$s are “p c p c”. This scenario illustrates the second effect of data sharing. Because “c” is referenced by T_1 in that interval, the references to it by T_2 should not be counted in the concurrent reuse distance. So the resulting concurrent reuse distance is $3 + 1 = 4$ (rather than 5 as in Scenario 1).

The last two scenarios show the two effects of data sharing on concurrent reuse distance approximation.

To approximate the concurrent reuse distance of co-running threads, we first assume no data shared across the threads, and apply the model described in Part I to compute a concurrent reuse distance histogram, R' for each thread. We then revise R' by considering the two effects of data sharing. The revision tries to find the statistical expectation of the correct concurrent reuse distance for each reuse interval contained in R'.

To explain the revision step, we first introduce some notations. For simplicity, we assume there are only two co-running threads. Let N_1 and N_2 represent the total numbers of distinct data accessed by thread 1 and thread 2 (in their entire execution), S represent the set of data shared by the two threads. Suppose that there is a reuse interval V with ending elements as e accessed by thread 1 and its reuse distance in R' is d' (which needs to be revised in this revision process). Let n_1 and n_2 be the numbers of distinct data among the data accessed respectively by the two threads in V; both can be computed by Equation 6.

Treating the First Effect. The revision step first treats the interval-breaking effect that data sharing may impose to the concurrent reuse distance (the second effect is temporarily ignored). It computes the probability for the reuse interval V to be broken. That event happens only when the following two events both occur. The first is that e is a shared data element; clearly the probability is $|S|/N_1$. The second is that e ever appears in the references by thread 2 in the interval V; as any of the n_2 data elements could be e, the probability is n_2/N_2. So the probability for the reuse interval to be broken is $(|S|/N_1) * (n_2/N_2)$. Because e may appear anywhere in V, assume the broken effect distributes to all sub-intervals of V uniformly. The probability for the resulting reuse intervals to have reuse distance of α ($\alpha = 0, 1, \ldots, d'$) is the same, $(|S|/N_1) * (n_2/N_2)/(d' + 1)$. Hence the number of reuse intervals of distance α in R' should increase by $(|S|/N_1) * (n_2/N_2)/(d' + 1)$. Meanwhile, because the original reuse interval is broken, the number of reuse intervals of distance d' in R' should decrease by $(|S|/N_1) * (n_2/N_2)$. We use R'' to denote the resulting histogram after this treatment.

Treating the Second Effect. In the treatment to the second effect of data sharing on concurrent reuse distance, each interval is not breakable as the interval-breaking effect has already been considered. For a reuse interval V in R'', let S_1 denote the set of distinct data among all references conducted by thread 1 in that interval, and S_2 for thread 2. In R'', the reuse distance of that interval would be $n_1 + n_2$. In this step, we want to correct this distance value by considering that there may be some overlap between S_1 and S_2. Let C represent the overlap set. Apparently, $C \subseteq S$. The probability for $|C| = c$ is

$$\frac{1}{\binom{N_1}{n_1} * \binom{N_2}{n_2}} \sum_{d=c}^{|S|} \binom{|S|}{d} \binom{N_1 - |S|}{n_1 - d} \binom{d}{c} \binom{N_2 - d}{n_2 - c},$$

where, $\binom{N_1}{n_1} * \binom{N_2}{n_2}$ is the possible ways to have a reuse interval like V, $\binom{|S|}{d}\binom{N_1-|S|}{n_1-d}$ is the number of ways for d shared data to appear in S_1, and $\binom{d}{c}\binom{N_2-d}{n_2-c}$ is the number of ways for thread 2 to access c data in the d shared data accessed by thread 1.

Those probabilities are enough to compute the statistical expectation of the concurrent reuse distance for every reuse distance in R'. Although our explanation uses two threads as the example, the model supports an arbitrary number of co-running threads.

4 Evaluation

This section reports the accuracy of the concurrent reuse distance produced by the probabilistic model. We use both the traces from real programs and some synthetic traces for the evaluation. The synthetic traces allow us to test memory reference patterns that are not covered by the selected programs.

4.1 Synthetic Traces

In order to test the model on traces with various data reuse patterns, we develop a trace generator that produces data reference traces according to users' specifications. The parameters that control the generated trace include the following:

- $n_1, n_2, \ldots, n_k$: the number of unique data blocks (in the unit of cache lines) in the co-running programs.
- s: the data sharing rate. It is the total number of shared data blocks divided by n_1.
- *distribution*: the distribution of standalone reuse distances. We test the following typical distributions: the random, the exponential ($\lambda = -0.97$), the Normal ($mean = 100$, $std. = 33$). Choosing these distributions is because they have been widely used as the primitive distributions in statistical mixture models [13]; the reuse patterns in many real traces can be regarded as the combination of those distributions [23].

The underlying scheme of the trace generator is a stochastic process similar to the one used in standalone reuse distance studies [22].

Table 4 presents the accuracies on a set of traces. The bottom three groups above the average row are the results when there are four co-runners, among which, the first pair both have n_1 unique data items, and the second pair both have n_2.

Following previous work [9], we define accuracy as $(1 - E/2)$, where E is the sum of the absolute differences between the predicted and the real reuse histograms at every reuse distance. Division by 2 normalizes the accuracy to $[0, 100\%]$. To completely expose prediction errors, we use the finest granularity: The width of each bar in all the histograms used in this experiment is 1.

The overall average accuracy is 87.9%. For larger-grained histograms (e.g., 1K-wide bars in many real uses), the accuracy would be higher as errors inside a bar would be smoothed out. The results also show that the effectiveness of the prediction approach is not significantly sensitive to reuse patterns, indicated by the similar accuracy across distributions. The presence of data sharing reduces the prediction accuracy by 5–7%, reflecting the extra complications caused by the sharing to concurrent reuse distance approximation. For most cases, the prediction accuracy is above 80%, verifying the existence of the statistical connections between concurrent reuse distance and the memory behaviors of individual threads, and demonstrating the capability of the probabilistic model in capturing such connections.

4.2 Traces from Real Programs

Because instrumentation changes the relative speeds of cache sharers, the real memory traces of co-running threads are difficult to collect on real machines. For our evaluation purpose, we employ a simulator to record the traces. The simulator is constructed based on SIMICS [38] with GEMS [18], a cycle-accurate multiprocessor simulator. The simulated system is a dual-core UltraSPARC architecture with 1MB shared L2 cache.

We simulate three representative PARSEC programs [4]. For each program, we use the fast mode of the simulator to move into the region of interest (the labels to those regions come with the original benchmarks) and then collect memory references in one-million-cycle-long detailed simulation.

Program *swaptions* is an Intel RMS workload which uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions. The program uses few (23) locks. There are 27% data that are shared between two threads in the collected memory reference trace. The prediction accuracy by the probabilistic model is 74%. The accuracy is relatively lower than those on synthetic traces. The reason is that this program accesses distinct data elements more frequently than the synthetic traces. The reuse distance tends to span a broader range.

Program *vips* is based on the VASARI Image Processing System (VIPS). It includes fundamental image operations such as an affine transformation and a convolution. The program uses locks intensively. There are totally over 33,000 locks. But there are negligible portion of data that are shared between threads. The probabilistic model is able to predict the concurrent reuse distance by 76% accuracy.

The last program is *streamcluster*. It is an RMS kernel developed by Princeton University that solves the online clustering problem. It is a data-level parallel program. This program uses modest number of locks, but many barriers (129,600). There are 3% data

Table 4. Accuracy of the Prediction of Concurrent Reuse Distance Histograms

distr.	s=0		s=10%		s=20%		average
	n_1=200	n_1=200	n_1=200	n_1=200	n_1=200	n_1=200	
	n_2=100	n_2=200	n_2=100	n_2=200	n_2=100	n_2=200	
random	94.9	93.3	91.3	90.0	89.7	79.8	89.8
expon.	93.2	92.3	91.1	92.2	93.4	90.1	92.1
normal	95.9	94.6	94.4	80.8	93.4	91.6	91.8
random+ expon.	94.0	93.3	88.5	87.2	84.0	79.0	87.7
random+ normal	93.9	93.5	87.4	90.9	91.6	89.1	91.1
expon.+ normal	93.6	94.2	92.5	79.9	92.2	89.9	90.4
2random+ expon.+ normal	88.2	88.5	83.3	82.0	82.5	81.6	84.4
random+ 2expon.+ normal	89.0	84.8	70.1	72.8	85.3	83.5	80.9
random+ expon.+ 2normal	85.0	85.9	84.1	80.0	81.2	81.2	82.9
average	92.0	91.2	87.0	84.0	88.1	85.1	87.9

s: the sharing ratio. n_1, n_2: the number of distinct data of the co-running programs.

shared between two threads in the generated memory reference trace. The approximated concurrent reuse distance histogram has the highest error, 28%. It is mainly due to its irregular data references.

4.3 Discussions

The significance of the model is that it shows the possibility of deriving concurrent reuse distance from the memory behaviors of individual threads, opening the door to many potential uses of concurrent reuse distance. Some of these uses are similar to how standalone reuse distance is applied to sequential programs running on uni-core processors. Examples include cross-architecture performance prediction [32,33,17], software refactoring [3], locality enhancement [34,36,10,11,16]. With the statistical model and the discoveries in Section 3, all these uses become possible for multithreading applications running on CMP.

Some other potential uses of concurrent reuse distance are specific to multithreading applications. An example is thread scheduling [28,30]. It is well known that using hyperthreads may both increase and decrease the performance of applications [15]. From the predicted concurrent reuse distance histograms, one can estimate the cache miss rates of a variety of numbers of threads co-running on a chip. On a CMP processor with hyperthreads enabled (such as Intel Nehalem), that prediction will help determine whether to use hyperthreads or not and how many threads to spawn would yield the best performance.

In our experiments, the longest run of the model takes about 20 seconds. There are many ways to reduce the overhead, such as memory reference sampling [35], employment of coarse-grained histograms, and use of mathematical approximation formulas [23]. Recall that the goal of this work is to reveal the inherent properties of concurrent reuse distance, including its connections with standalone reuse distance—what the probabilistic model captures. Creating a lightweight tool for concurrent reuse distance approximation is orthogonal to the main goal of this work. So sophisticated overhead reduction remains our future work.

5 Related Work

Since the early days in computing [8,19], decades of efforts have contributed a solid foundation for understanding the behavior of dedicated cache systems. Standalone reuse distance has been one of the most influential locality metric [19,2,17,11,36].

However, reuse distance has not been systematically studied in the environment of multicore with cache sharing. The studies close to this work include the following several explorations in predicting miss rates on shared cache.

Ding and Chilimbi [31] have proposed an approach to all-window profiling for concurrent executions, through which, they found that memory accesses by multiple threads of a server application typically show non-uniform interleaving patterns. Chandra et al. [6] have developed three statistical models to predict cache miss rates of co-running processes from the circular stack distance histograms of individual process. Chen and Aamodt [7] extend the models to predict cache contention on Simultaneous Multithreading architecture. Our work differs from these studies in three aspects. First, their models predict cache miss rates rather than concurrent reuse distance. As a program-level locality characterization, reuse distance has a variety of uses besides performance prediction, such as software refactoring [3], guiding data transformations [34,36], memory disambiguation [10,11]. It is not clear how the previous models apply to these uses. Second, the previous models are for independent jobs, while our model allows data sharing among cache sharers. Finally, because of the use of circular stack distance histograms, the previous models have certain but limited cross-architecture predictive capability. They require that the number of cache sets must remain the same, and the cache associativity of the new machine must be smaller than the cache associativity of the training architecture.

Berg et al. [1] propose a statistical model to estimate the miss rate of shared cache for multithreading programs. Unlike the previously mentioned two studies, their model starts directly from concurrent reuse distance. They assume that the concurrent reuse distance of the interleaved memory reference traces is already available somehow (they obtain it through a simulator), and use it as one of the inputs to their statistical model. The authors collect the traces using simulator for their experiment. It is not clear how such traces can be obtained on real machines. Our model concentrates on the attainment of concurrent reuse distance.

In addition, there has been some work on analyzing the interactions among different threads on dedicated cache in a time-sharing environment [29,27]. These studies mainly focus on predicting the footprint size of a thread as the interactions on cache mainly occur at context switch time; while with shared cache, the interactions happen at almost every cache access—footprint size prediction becomes insufficient.

There have been a wealth of research trying to optimize shared cache performance through either hardware extensions[14,20,21], or operating system scheduling [12,26,28]. They mainly rely on hardware-level locality information collected by hardware performance monitors or special hardware extensions. As mentioned in Section 1, reuse distance differs from hardware-level metrics. It is hardware-independent and captures program-level locality characterizations, important for a variety of locality analysis and program optimizations.

6 Conclusions

The explorations described in this paper lead to the following conclusions. First, despite the wide applicability of reuse distance on traditional architectures, applying it to CMP environments is challenging. The obstacles stem from the reliance of concurrent reuse distance on the relative running speeds among cache sharers. The reliance makes the measurement of reuse distance difficult as instrumentation would change the relative speeds. It also deprives reuse distance of its hardware-independence, impairing many of its uses. Second, experimental evidences show that the relative speeds of many non-pipelining multithreading applications remain unchanged across architectures and inputs because of the uniformity among threads. That observation grants reuse distance the potential applicability for multithreading applications running on CMP environments. Finally, a probabilistic model shows the promise of facilitating the realization of such potential by offering a mechanism to derive concurrent reuse distance histograms from the memory behaviors of individual threads.

Despite the findings and revealed potential, there is no doubt that much further studies are needed before concurrent reuse distance can be practically applied. This work hopefully can help stimulate such studies to systematically extend the commonly used locality model, reuse distance, to modern CMP environments.

Acknowledgments

We owe the anonymous reviewers our gratitude for their helpful comments on the paper. The discussions with Chen Ding's group at University of Rochester helped the refinement of the final version of this paper. This material is based upon work supported by the National Science Foundation under Grant No. 0720499 and 0811791 and IBM CAS Fellowship. Any opinions, findings, and conclusions or recommendations ex- pressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or IBM.

References

1. Berg, E., Hagersten, E.: Fast data-locality profiling of native execution. ACM SIGMETRICS Performance Review 33, 169–180 (2005)
2. Beyls, K., D'Hollander, E.H.: Reuse Distance as a Metric for Cache Behavior. In: Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems (2001)
3. Beyls, K., D'Hollander, E.: Discovery of locality-improving refactoring by reuse path analysis. In: Gerndt, M., Kranzlmüller, D. (eds.) HPCC 2006. LNCS, vol. 4208, pp. 220–229. Springer, Heidelberg (2006)
4. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: characterization and architectural implications. In: Proceedings of International Conference on Parallel Architectures and Compilation Techniques, Toronto, pp. 72–81 (2008)
5. Browne, S., Deane, C., Ho, G., Mucci, P.: PAPI: A portable interface to hardware performance counters. In: Proceedings of Department of Defense HPCMP Users Group Conference (1999)
6. Chandra, D., Guo, F., Kim, S., Solihin, Y.: Predicting inter-thread cache contention on a chip multi-processor architecture. In: Proceedings of the International Symposium on High Performance Computer Architecture (2005)
7. Chen, X.E., Aamodt, T.M.: A First-Order Fine-Grained Multithreaded Throughput Model. In: Proceedings of the International Symposium on High-Performance Computer Architecture, Raleigh, pp. 329–340 (2009)
8. Denning, P.: Thrashing: Its causes and prevention. In: Proceedings of the AFIPS 1968 Fall Joint Computer Conference (1968)
9. Ding, C., Zhong, Y.: Predicting Whole-Program Locality with Reuse Distance Analysis. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, San Diego, pp. 245–257 (2003)
10. Fang, C., Carr, S., Onder, S., Wang, Z.: Instruction Based Memory Distance Analysis and its Application to Optimization. In: Proceedings of International Conference on Parallel Architectures and Compilation Techniques, pp. 27–37 (2005)
11. Fang, C., Carr, S., Onder, S., Wang, Z.: Feedback-directed Memory Disambiguation Through Store Distance Analysis. In: Proceedings of the 20th ACM International Conference on Supercomputing, Cairns, Queensland, Australia, pp. 278–287 (2006)

12. Fedorova, A., Seltzer, M., Smith, M.D.: Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, pp. 25–38 (2007)
13. Hastie, T., Tibshirani, R., Friedman, J.: The elements of statistical learning. Springer, Heidelberg (2001)
14. Hsu, L.R., Reinhardt, S.K., Lyer, R., Makineni, S.: Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, Seattle, pp. 13–22 (2006)
15. Liao, C., Liu, Z., Huang, L., Chapman, B.: Evaluating OpenMP on Chip Multithreading Platforms. In: Proceedings of International Workshop on OpenMP (2005)
16. Lu, Q., Lin, J., Ding, X., Zhang, Z., Zhang, X., Sadayappan, P.: Soft-OLP: improving hardware cache performance through software-controlled object-level partitioning. In: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, pp. 246–257 (2009)
17. Marin, G., Mellor-Crummey, J.: Cross architecture performance predictions for scientific applications using parameterized models. In: Proceedings of Joint International Conference on Measurement and Modeling of Computer Systems, New York, pp. 2–13 (2004)
18. Martin, M., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D., Wood, D.A.: Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. Computer Architecture News, 92–99 (2005)
19. Mattson, R.L., Gecsei, J., Slutz, D., Traiger, I.L.: Evaluation techniques for storage hierarchies. IBM System Journal 9(2), 78–117 (1970)
20. Rafique, N., Lim, W., Thottethodi, M.: Architectural support for operating system-driven CMP cache management. In: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, pp. 2–12 (2006)
21. Settle, A., Kihm, J.L., Janiszewski, A., Connors, D.A.: Architectural Support for Enhanced SMT job scheduling. In: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, pp. 63–73 (2004)
22. Shen, X., Shaw, J.: Scalable Implementation of Efficient Locality Approximation. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 202–216. Springer, Heidelberg (2008)
23. Shen, X., Shaw, J., Meeker, B., Ding, C.: Locality approximation using time. In: Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages (2007)
24. Shen, X., Zhong, Y., Ding, C.: Regression-based multi-model prediction of data reuse signature. In: Proceedings of the 4th Annual Symposium of the Las Alamos Computer Science Institute, Sante Fe, New Mexico (2003)
25. Smith, A.J.: On the Effectiveness of Set Associative Page Mapping and Its Applications in Main Memory Management. In: Proceedings of the 2nd International Conference on Software Engineering, pp. 286–292 (1976)
26. Snavely, A., Tullsen, D.M.: Symbiotic jobscheduling for a simultaneous multithreading processor. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 66–76 (2000)
27. Suh, G.E., Devadas, S., Rudolph, L.: Analytical Cache Models with Applications to Cache Partitioning. In: Proceedings of the 15th international conference on Supercomputing, Sorrento, Italy, pp. 1–12 (2001)
28. Tam, D., Azimi, R., Stumm, M.: Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. SIGOPS Oper. Syst. Rev. 41(3), 47–58 (2007)
29. Thiebaut, D., Stone, H.S.: Footprints in the Cache. ACM Transactions on Computer Systems 5(4) (1987)
30. Zhang, E.Z., Jiang, Y., Shen, X.: Does Cache Sharing on Modern CMP Matter to the Performance of Contemporary Multithreaded Programs? In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2010)

31. Ding, C., Chilimbi, T.: All-Window Profiling of Concurrent Executions. In: Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 265–266 (2008)
32. Zhong, Y., Dropsho, S.G., Ding, C.: Miss Rate Prediction Across All Program Inputs. In: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (2003)
33. Zhong, Y., Dropsho, S.G., Shen, X., Studer, A., Ding, C.: Miss rate prediction across program inputs and cache configurations. IEEE Transactions on Computers 56(3), 328–343 (2007)
34. Zhong, Y., Orlovich, M., Shen, X., Ding, C.: Array Regrouping and Structure Splitting using Whole-Program Reference Affinity. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 255–266 (2004)
35. Zhong, Y., Chang, W.: Sampling-based Program Locality Approximation. In: Proceedings of the International Symposium on Memory Management (2008)
36. Zhong, Y., Shen, X., Ding, C.: Program Locality Analysis Using Reuse Distance. ACM Transactions on Programming Languages and Systems 31(6) (2009)
37. Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (2005)
38. Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hållberg, G., Högberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A Full System Simulation Platform. Computer, 50–58 (2002)

The Polyhedral Model Is More Widely Applicable Than You Think

Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet,
Albert Cohen, and Cédric Bastoul

ALCHEMY Group, INRIA Saclay Île-de-France and University of Paris-Sud 11
{firstname.lastname}@inria.fr

Abstract. The polyhedral model is a powerful framework for automatic optimization and parallelization. It is based on an algebraic representation of programs, allowing to construct and search for complex sequences of optimizations. This model is now mature and reaches production compilers. The main limitation of the polyhedral model is known to be its restriction to statically predictable, loop-based program parts. This paper removes this limitation, allowing to operate on general data-dependent control-flow. We embed control and exit predicates as first-class citizens of the algebraic representation, from program analysis to code generation. Complementing previous (partial) attempts in this direction, our work concentrates on extending the code generation step and does not compromise the expressiveness of the model. We present experimental evidence that our extension is relevant for program optimization and parallelization, showing performance improvements on benchmarks that were thought to be out of reach of the polyhedral model.

1 Introduction

The ability to perform complex loop nest restructuring is required for optimizing and parallelizing tools, to cope with the complexity of modern architectures. The widespread adoption of multicore processors and massively parallel hardware accelerators (GPUs) urge production compilers to provide such capability. The polyhedral model has demonstrated its potential to achieve portability of performance over a variety of targets. So far, these successes have been limited to static-control, regular loop nests. Time has come to address these challenges on a much wider class of programs.

Since the very first compilers, the internal representation of programs has been in direct correspondance with their operational semantics. In such abstract syntaxes, each statement appears only once even if it is executed many times. This representation has severe limitations. First of all, it may limit the accuracy of program analysis. For instance, if a statement in a loop has some data dependence relation with another statement, it will consider both of them as single entities while the dependence relation may involve only very few of the dynamic iterations of these statements. This is particularly common in loop-based programs accessing arrays. Next, it may limit program transformation

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 283–303, 2010.

applicability. For instance, loop transformations operate on *individual statement iterations*. Lastly, it limits the expressiveness of program transformations: the most impactful loop nest transformations cannot be expressed as structural, incremental updates of the loop tree structure [19].

The polyhedral model is a semantical, algebraic representation which combines analysis power, transformation expressiveness and flexibility to design sophisticated optimization heuristics. It was born with the seminal work of Karp, Miller and Winograd on systems of uniform recurrence equations [23]. The polyhedral model is closer to the program execution than operational/syntactic representations because it operates on individual statement iterations, or *statement instances*. It has been the basis for major advances in automatic optimization and parallelization of programs [15,6,26,20,5]. After decades of research, production compilers are getting closer to making effective use of the polyhedral model to compile for multicore architectures, including GCC 4.4 and IBM XL.

Compilers based on the Polyhedral model — including recent research tools like PoCC [29] or CHiLL [8] — target code parts that exactly fit the affine constraints of the model. Only loop nests with affine bounds and conditional expressions can be translated to a polyhedral representation. The reason behind this limitation is *not* that exact dependence analysis is required to make use of the polyhedral model, but rather that there is no general scheme to support dynamic control flow in the *program transformation* and *code generation* algorithms. To fight a common misunderstanding, *the power of the polyhedral model is not to achieve exact data dependence analysis, but to implement compositions of complex transformations as a single algebraic operation, and to model these transformations in a convex optimization space* [15,26,19,5,28].

In this paper, we expand the application domain of the polyhedral model. We present slight extensions to the representation itself, based on the notions of *exit* and *control* predicates that allow to consider general `while` loops and `if` conditions. We revisit the whole framework, from input code analysis to output code generation, while taking care of preserving expressiveness and flexibility. We present experimental evidence that this extended framework offers new optimization opportunities for *existing* optimization algorithms, and opens the door to novel techniques targetting full functions.

The paper is organized as follows. Section 2 introduces the classical polyhedral representation of programs and extensions to support irregular control flow. Section 3 revisits the polyhedral model to target full functions, from analysis to code generation. Section 4 discusses control overhead and some solutions. Section 5 presents experimental results in the extended framework. Section 6 discusses related work, before the conclusion in Section 7.

2 Polyhedral Representation of Programs

Static Control Parts (SCoP) are a subclass of general loops nests that can be represented in the polyhedral model.

```
for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++)
    if (i <= n + 2 - j)
      S(i,j);
```

$$\mathcal{D}_S = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \middle| \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2, \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \\ -1 \\ n \\ n+2 \end{pmatrix} \geq \mathbf{0} \right\}$$

(a) Surrounding Control of S

(b) Iteration Domain of S

Fig. 1. Static control and iteration domain

2.1 Static Control Parts

A SCoP is defined as a maximal set of consecutive statements, where loop bounds and conditionals are affine functions of the surrounding loop iterators and the parameters (constants whose values are unknown at compilation time). The iteration domain of these loops can always be specified thanks to a set of linear inequalities defining a polyhedron. The term polyhedron will be used to denote a set of points in a $\mathbb{Z}^n$ vector space bounded by affine inequalities:

$$\mathcal{D} = \{\boldsymbol{x} \mid \boldsymbol{x} \in \mathbb{Z}^n, A\boldsymbol{x} + \boldsymbol{a} \geq \mathbf{0}\}$$

where $\boldsymbol{x}$ is the iteration vector (the vector of the loop counter values), A is a constant matrix and $\boldsymbol{a}$ is a constant vector, possibly parametric. The iteration domain is a subset of the full possible iteration space: $\mathcal{D} \subseteq \mathbb{Z}^n$. Figure 1 illustrates the matching between surrounding control and polyhedral domain: the iteration domain in Figure 1(b) can be defined using affine inequalities that are extracted directly from the program in Figure 1(a) (e.g, the first row $i - 1 \geq 0$ corresponds to the lower bound of the first loop). To the best of our knowledge, all previous works using the polyhedral model used a similar representation. However, because of its strong mathematical constraints, any irregularity in the code splits the program into several smaller SCoPs. Furthermore, irregularities inside a loop nest will result in SCoPs with lower dimensionality (only the inner regular loops may be considered) [19]. For instance, let us consider the Outer Product Kernel shown in Figure 4(a): because of the irregular conditional, existing polyhedral frameworks can only consider the two *innermost* loops *separately*, or, to the contrary, consider the whole `if-else` statements as an atomic block, hence with a significantly reduced potential impact.

2.2 Relaxing the Constraints

The program model we target in this paper is general functions where the only control statements are `for` loops, `while` loops and `if` conditionals. This means function calls have to be inlined and `goto`, `continue` and `break` statements have been removed thanks to some preprocessing. To move from static control parts to such general control flow we need to address two issues: (1) modeling loop structures with arbitrary bounds (typically `while` loops); and (2) modeling arbitrary conditionals (typically data-dependent ones). In both cases, it implies to not be anymore able to exactly characterize statically the iteration domain of statements, which remains the privilege of Static Control Parts.

First, we demonstrate that it is possible to express safe over-approximations of the iteration domains to allow the construction of a polyhedral representation in the case of arbitrary control-flow.

Modeling Arbitrary Loop Structure. Any arbitrarily iterative structure such as `for` loops with non-affine bounds or `while` loops is actually amenable to polyhedral representation. As explained in Section 2.1 the iteration domain of a statement is a subset of $\mathbb{Z}^n$. The convex hull of all executed instances of any statement, even with a non-polyhedral iteration domain, is a subset of $\mathbb{Z}^n$. Thus, an over-approximation that fits the polyhedral model for the iteration domain of any statement enclosed in a non-static loop is $\mathbb{Z}^n$ itself. We actually choose to over-approximate it as $\mathbb{N}^n$ to match the standard loop normalization scheme, represented by the non-negative half-space polyhedron. This translates to over-approximate any non-static loop with a static loop iterating from 0 to infinity. Such over-estimate have been used in the same way by Griebl and Collard for `while` loop parallelization [21].

To guarantee that the program semantics will be preserved, we introduce an **exit predication** statement which bears the loop bound check. This statement is executed at the beginning of any iteration of the infinite loop, and exits the loop thanks to a `break` instruction if the loop conditional is no longer satisfied. This is summarized in Figure 2: we consider the original code in Figure 2(a) as the equivalent code in Figure 2(b) with the exit predicate `ep`. In the case of arbitrary `for` loops, initialization statements are inserted just before the loop and at the end of the loop body for the increment. Note that all statements in the body of the loop depends on the exit predication statement. Each statement S has a set of exit predicates, $\mathcal{E}_S$. The exit predicate is attached to the iteration domain of the predicated statements as illustrated in the example in Figure 2(c).

```
while (condition)
    S;
```

(a) Original Code

```
for (i=0;; i++)
    ep = condition;
    if (ep)
        S(i);
    else
        break;
```

(b) Equivalent Code

$$\mathcal{D}_S = \{ (\,i\,) \,|\, (\,i\,) \in \mathbb{Z}, ep \in \mathcal{E}_S, [\,1\,]\,(\,i\,) + (\,0\,) \geq \mathbf{0} \wedge ep \}$$

(c) Iteration Domain of S

Fig. 2. Exit predication

```
for (i=0; i<N; i++)
  if (condition(i))
    S(i);
```

(a) Original Code

```
for (i=0; i<N; i++)
  cp(i) = condition(i);
  if (cp(i))
    S(i);
```

(b) Equivalent Code

$$\mathcal{D}_S = \left\{ (\,i\,) \,\middle|\, (\,i\,) \in \mathbb{Z}, cp(i) \in \mathcal{C}_S, \begin{bmatrix} 1 \\ -1 \end{bmatrix} (\,i\,) + \begin{pmatrix} 0 \\ N-1 \end{pmatrix} \geq \mathbf{0} \wedge cp(i) \right\}$$

(c) Iteration Domain of S

Fig. 3. Control predication

Modeling Arbitrary Conditionals. We apply a similar reasoning to represent non-affine conditionals. To model such a conditionally executed statement in the polyhedral representation we decouple the regular part of the iteration domain and the irregular conditional. Again, the polyhedral iteration domain is over-approximated and we need to ensure the semantics is preserved. To do so we introduce a **control predication** which consists in predicating individually each statement dominated by the non-static conditional by its condition (similar to if-conversion). Each statement S has a set of control predicates, $\mathcal{C}_S$. This is summarized in Figure 3: we consider the code in Figure 3(a) as the equivalent code in Figure 3(b) with the control predicate `cp(i)`. This predicate is attached to the iteration domain of the predicated statements as shown in Figure 3(c).

Being able to safely describe (from the iteration domain point of view) the convex hull of the dynamic control flow is only the first step towards supporting full functions. The following section presents necessary and sufficient modifications of the framework that allow to *transform* general codes with polyhedral techniques. Our goal is to show that, provided a suitable dependence analysis (static, dynamic or both), only the code generation step needs to be altered to enable any polyhedral optimization technique on full functions.

3 Revisiting the Polyhedral Framework

Restructuring programs using the polyhedral model is a three steps framework. First, the Program Analysis phase aims at translating high level codes to their polyhedral representation and to provide data dependence analysis based on this representation. Second, some optimizing or parallelizing algorithm use the analysis to restructure the programs in the polyhedral model. This is the Program Transformation step. Lastly, the Code Generation step returns back from the polyhedral representation to a high level program. Targeting full functions requires revisiting the whole framework, from analysis to code generation.

3.1 Program Analysis

Once a function has been translated to the polyhedral model with the predicate extensions described in Section 2.2, data dependence analysis must be performed. Two statements are said to be in dependence if they access the same memory reference, and at least one of these accesses is a write. When restricting the study to SCoPs and to array references with affine subscripts — we talk about *static references* — it is possible to compute on which instance (iteration) of a given statement any other instance depends [13,14].

As we broaden the set of handled programs, we have to deal with dynamic behavior (e.g., `while` loops) and structural complexity (e.g. subscript of subscript, as in `A[B[i]]`). As a result, an exact analysis is no more possible statically. Instead, we rely on a *conservative* policy, over-estimating data dependences, preventing some optimizations when semantics safety is unsure.

Conservative policies are widely used in compilation to achieve an approximate analysis of programs without slowing down the compiler. GCD-test [2] or I-test [25] are popular examples of such analysis for array references: they can state thanks to a fast GCD computation that two references do not depend on each other, then safely consider a dependence relation exists otherwise (for instance, GCC 4.4 relies on a multi-dimensional GCD-test for production and on a more costly but exact Omega-test [30] for testing). When dedicated preprocessing techniques fail to simplify complex array references (typically subscript of subscript or linearized subscripts) it is usual to consider the reference as an access to a single variable, i.e., to suppose that the whole array is read or written. In the same way, when *array recovery* fails to translate pointer-based accesses to explicit array references [16], it is usual to consider a dependence between the pointer access and every previously accessed references. Overall, it is possible to handle any kind of data access in a conservative way.

A conservative approach for irregular data dependence analysis is adding new statements or new statement iterations because the only effect is adding extra data dependences. Hence, as long as the additional statements do not modify directly the control flow (as `break`, `continue` or `goto` statements), we can add them with regard to the analysis. Therefore for data dependence analysis, it is safe to consider irregular conditions (from `while` loops as well as `if` conditionals) are always true. A convenient data dependence analysis for our purpose is described by Feautrier [14,15]. This approach does not generalize to all analyses because considering predicates are always true may not be conservative. For instance, it is not convenient for dead code analysis: in the example in Figure 4(a), if both branches are considered to be executed, the first branch would be considered dead (data are totally over-written by the second branch). In the same way, Feautrier's data-flow analysis [13] that relies on last writer computation is not directly suitable for our conservative approach.

In this paper, we translate the program control structures in such a way we only have to deal with regular `for` loops, regular `if` conditionals and infinite `for` loops. Irregularity has been spread thanks to control and exit predicates to the iteration domains of irregular-control-surrounded statements. One can achieve a naive but simple conservative analysis by considering an altered representation of the input irregular program called *abstract program*. We build this representation from the original program in this way:

1. Introduce control and exit predicates as described in Section 2.2.
2. Predicate evaluations are considered as statements that write the predicate, and read the necessary data to compute the predicate.
3. Irregular data accesses are modeled conservatively (an array with a complex subscript is considered as a single variable).
4. Predicated statements are considered to read their predicates.

Writing and reading predicates ensure the semantics is preserved when a statement modifies an element necessary for the predicate evaluation. Ultimately we may perform on this representation usual data dependence elimination techniques like array privatization [1] then exact data dependence analysis [14].

We illustrate the construction of the abstract program for conservative data dependence analysis in Figure 4. The considered program in Figure 4(a) is an optimized version of the Outer Product Kernel in the case one vector contains some zeros. The conditional introduces irregular control flow that usually prevents considering such kernel in the polyhedral model. The first step is to introduce a control predicate and to attach it to the predicated statements. The predicate evaluation is considered as a new statement as shown in Figure 4(b). Lastly, we consider the value of the predicate is read by each predicated statement and that the predicate is always true for conservative data dependence analysis as shown in Figure 4(c). Figure 4(c) presents the information sent to the data dependence algorithm (everything is regular): for each statement, its iteration domain and the sets of written and read references. We may use well known techniques to remove some dependences. In this example we can privatize `p` to remove loop-carried dependence and parallelize the code or even interchange the loops using existing polyhedral techniques.

Discussion. Many previous works aim at providing less naive and conservative solutions to avoid, as much as possible, to consider additional dependences. Griebl and Collard proposed a solution in the context of `while` loops parallelization, focusing on control flow [21]. Collard et al. extended this approach to support complex data references [11]. Other techniques aim at removing some dependences as, e.g., Value-based Array Data Dependence Analysis [31], Array Region Analysis [12] Array SSA [24] or Maximal Static Expansion [3]. These techniques would expose their full potential in the context of manipulating full functions in the polyhedral model to minimize the unavoidable conservative aspects. Combining these static analyses with dynamic dependence tests [34,37,36,35] into hybrid polyhedral/dynamic analyses remains to be investigated.

3.2 Program Transformation

A (sequence of) program transformation(s) in the polyhedral model is represented by a set of affine functions, one for each statement, called scheduling,

```
for (i = 0; i < N; i++)
  if (x[i] == 0)
    for (j=0; j < M; j++)
      A[i][j] = 0;
  else
    for (j=0; j < M; j++)
      A[i][j] = x[i] * y[j];
```

(a) Outer product kernel

```
for (i = 0; i < N; i++)
  p = (x[i] == 0);
  for (j=0; j < M; j++)
    if (p)
      A[i][j] = 0;
  for (j=0; j < M; j++)
    if (!p)
      A[i][j] = x[i] * y[j];
```

(b) Using a control predicate

```
for (i = 0; i < N; i++)
  S0: Written = p, Read = x[i]
  for (j=0; j < M; j++)
    S1: Written = A[i][j], Read = p
  for (j=0; j < M; j++)
    S2: Written = A[i][j], Read = x[i],y[j],p
```

(c) Abstract program for conservative data dependence analysis

Fig. 4. Abstract program representation for the irregular outer product

allocation, chunking, etc. depending on the technique. In this paper we will use the generic term *scattering functions*. Scattering functions depend on the counters of the loops surrounding their corresponding statement; they map each run-time statement instance to a logical execution date. The literature is full of algorithms to find such functions dedicated to parallelization, data locality or global performance improvement [15,26,20,5]. Our approach allows to reuse most existing techniques based on the polyhedral model and multi-dimensional scattering *directly*.

However, managing `while` loops, that are translated into unbounded `for` loops requires a slight adaptation to preserve the expressiveness of affine scattering functions. This is particularly important in the context of one-dimensional affine functions, where it is necessary to know the upper bounds of the loops to be able to reorder them. For instance let us consider the pseudo-code in Figure 5(a) composed of two loops enclosing two statements, `S1` and `S2`. To implement a transformation such that the loop enclosing `S2` will be executed before the loop enclosing `S1`, we need the logical dates of the instances of `S1` to be *higher* than those of the instances of `S2`. Such transformation may be implemented by the scattering functions $\theta_{S1}(i) = i + \mathit{Up2}$ and $\theta_{S2}(i) = i$. In these functions, the i part ensures the instances of a given statement are executed in the same order as in the original code, and the upper bound *Up2* of the second loop is used to ensure the loop of `S1` *starts* after the end of the loop of `S2`. The target code is shown in Figure 5(b), where variable t represents logical time.

In this work, we may consider `for` loops with no upper bounds. It is not possible in this way to reorder those loops respectively to other loops (bounded or

```
for (i = 0; i < Up1; i++)
  S1;
for (i = 0; i < Up2; i++)
  S2;
```

(a) Original program

```
for (t = 0; t < Up2; t++)
  i = t;
  S2;
for (t = Up2; i < Up2 + Up1; i++)
  i = t - Up2;
  S1;
```

(b) Loop reordering with scattering $\theta_{S1}(i) = i + Up2$ and $\theta_{S2}(i) = i$

Fig. 5. Loop reordering using one-dimensional scattering

unbounded) using one-dimensional schedules only.[1] We thus introduce a virtual parametric upper bound `w`, the same for all unbounded `for` loops with the constraint that `w` is strictly greater than all upper bounds of bounded `for` loops. The `w`-parameter will be considered during the program transformation and code generation steps. It will be removed during a dedicated stage of code generation as detailed in Section 3.3. This parameter has to be chosen strictly greater than other loop bounds to ensure a fusion between a bounded and an unbounded loop will always be partial (hence the code generation step will always be able to recreate the unbounded part). A single `w`-parameter for multiple unbounded loops is enough to be able to reorder them relatively to each other by using coefficients of this parameter (e.g., to reorder three unbounded loops, we can use scattering functions like $\theta_{S1}(i) = i$, $\theta_{S2}(i) = i + w$ and $\theta_{S3}(i) = i + 2w$). The `w`-parameter allows to reuse any of the existing algorithms supporting parameters to compute scattering functions in our irregular context.

3.3 Code Generation

Once a transformation (i.e., a scattering function) has been computed by an optimization heuristic, applying it in the polyhedral model is straightforward and leads to a new coordinate system for each iteration domain [4]. The last step consists in translating the transformed program from its polyhedral representation back to a syntactic representation. This phase amounts to finding a set of nested loops visiting each integral point of each polyhedron once and only once. This is a critical step in the polyhedral framework since the final program effectiveness highly depends on the target code quality. In particular, we must ensure that a bad control management does not spoil performance, for instance by producing redundant conditions, complex loop bounds or under-used iterations. On the other hand, we have to avoid code explosion typically because a large code may pollute the instruction cache.

Among existing methods to scan polyhedra and generate code, the extended Quilleré et al. algorithm is considered now as the most efficient algorithm [32,4].

[1] It is easy to remove the limitation using more dimensions, but several algorithms to compute scattering functions are based on one-dimensional scattering only, and some others rely on the full expressiveness of each dimension.

CodeGeneration: build a polyhedron scanning code AST without redundant control.

Input: a polyhedron list, a context C, the current dimension d.
Output: the AST of the code scanning the input polyhedra.

1. Intersect each polyhedron in the list with the context C;
2. Project the polyhedra onto the outermost d dimensions;
3. Separate these projections into disjoint polyhedra (this generates loops for dimension d and new lists for dimension $d + 1$);
4. Sort the loops to respect the lexicographic order;
5. Recursively generate loop nests that scan each new list with dimension $d+1$, under the context of the dimension d;
6. Return the AST for dimension d.

Fig. 6. Quilleré et al. algorithm

This algorithm is not able in its original form to generate semantically correct code for our extended polyhedral representation, as special care is needed to handle properly predicates and their impact on the generated control-flow. Nevertheless, it is possible to extend this algorithm to scan and generate regular codes corresponding to the over-estimates of the iteration domains then to post-process its output to guarantee semantically correct code generation.

We first provide a short description of the Quilleré et al. algorithm, then we present a new extension to this algorithm to support irregular code generation.

Quilleré et al. Algorithm. Quilleré, Rajopadhye and Wilde proposed the first code generation algorithm to directly eliminate redundant control in the target code, in contrast of other approaches starting from a naive code and trying to improve it [32]. The main part of the algorithm is a recursive generation of the scanning code, maintaining a list of polyhedra from the outermost to the innermost loops. Figure 6 describes briefly this algorithm. Its input is a list of polyhedra that need to be scanned in lexicographical order of their points (iterations), the context (constraints on the global parameters), and the first dimension to scan.

Extension for Irregular Programs. Previous approaches to model irregular codes (see Section 6) were based on complex representations that did not allow any easy modification of the extended Quilleré et al. algorithm to generate the code. Instead they rely on ad-hoc, mostly syntactic, code generation schemes. By relaxing the static constraints thanks to exit and control predication, we make possible, and even natural, the adjustment of the Quilleré et al. code generation algorithm. This adaptation takes into account the additional data dependences on control predicates. The price to pay is displacing the problem of modeling data dependent non-affine conditions into legality constraints. There is no alteration of the core Quilleré et al. algorithm: we apply it on the polyhedral over-estimated iteration domains, leaving predicates attached to each statement. Then we post-process the result to handle the predicates. There are two tasks to perform: (1) to achieve a semantically-correct generation of control predicates and exit predicates, and (2) to reconstruct while loops in the generated code.

Generation of Arbitrary Conditionals. Generating arbitrary conditionals is straightforward: the control predicate is available as a statement information, attached to the polyhedral iteration domain. The only task is to generate the `if` instruction containing the predicate around the convenient statement.

Generation of `while` Loop Structure. The task of generating `while` loops starts by identifying loops with the `w` parameter introduced in Section 3.2 as an upper bound. Next, we have to identify exit predicates corresponding to each `while` loop. Again, this information can be easily extracted because it is attached to the polyhedral iteration domain of each statement that belongs to a `while` loop in the original program.

However, due to the separation step of the extended Quilleré et al. algorithm, several statements with different exit predicates could be found in the same iteration domain without corresponding to the same `while` loop. So we need to separate these statements and generate the appropriate `while` loops. we distinguish three main cases of separation that involve exit predicates:

1. If all statements of the loop have the same exit predicate, no case distinction is needed during the separation phase. The predicate is therefore considered as the exit predicate of the generated `while` loop. Figure 7(a) is an example of such a case.
2. If statements or block of statements have different exit predicates, this means (1) they belong to different `while` loops; and (2) these statements can be executed in any order (the semantics of `while` loops transformations is particular, as discussed in Section 3.3). For this second case, we can proceed to a separation quite similar to the separation of polyhedra in the regular case. More exactly, it consists in scanning the domain where both predicates are true at the same time, thanks to the intersection of two polyhedra, i.e., the space of common points. Then, we scan domains where only one of the two predicates is true, thanks to the differences between polyhedra. Figure 7(b) shows separation of while loops based on exit predicates attached to statements `s1` and `s2`.
3. If some statements have exit predicates while some others do not have any, this means a regular `for` loop has been fused with a part of a `while` loop. In such a case, we find a statement with an exit predicate attached to it without identifying the `while` loop (by identifying the `w` parameter). The exit predicate is transformed here into a control predicate plus an exit Boolean (false at the start of the program). Figure 7(c) illustrates this case.

Re-injecting irregular control inside the generated code is likely to bring high control overhead as it is inserted close to the statement, at the innermost level.

Discussion. The semantics of transformations involving `while` loops is particular: fusion of such loops should be performed only if the loops can be executed

```
for (i=0; i<w; i++)        while(ep1)
  s1;{ep1}            ⇒      s1;
  s2;{ep1}                   s2;
```

(a) Same Exit Predicates

```
                          while(ep1 && ep2)
                            s1;
for (i=0; i<w; i++)         s2;
  s1;{ep1}            ⇒   while(ep1)
  s2;{ep2}                  s1;
                          while(ep2)
                            s2;
```

(b) Different Exit Predicates

```
                            for (i=0; i<N; i++)
                              s1;
for (i=0; i<N; i++)           if (cp1 && !exit)
  s1;                   ⇒       s2;
  s2;{ep1}                    else
                                exit = true
```

(c) An Exit Predicate Inside A Regular Loop

Fig. 7. Separation of `while` loops

in any order (in Figure 7(b), the order of the last two `while` loops is arbitrary) and `while` loop reversal is clearly not supported by our extended framework. Also, when the transformation states the loop may be run in parallel (e.g., no scattering functions means all loops are parallel) it means that, except what is necessary for the predicate evaluation, iterations of the loop may be run in parallel (this allows basic parallelization, e.g., a process devoted to the predicate computation that spread bundles of full iterations to different processors).

4 Reducing Control Overhead

The underlying principle of converting programs to the extended polyhedral representation is to conditionally execute statements depending on the value of a given predicate, which is not necessarily statically computable. To put the program into the model, we extensively predicate statements regardless of the control overhead we introduce. We rely on post-pass optimizations to limit this overhead for the generation of efficient code.

We discuss two main optimizations, namely the *computation of the predicate value* and the *placement of control predicates*. A preliminary for those optimizations to be performed is the gathering of the set of *read* and *written* variables, for each statement and each predicate. Obviously, the optimality of our optimization processes is constrained by the accuracy of this analysis.

4.1 Computing the Value of Predicates

The main overhead induced by predication is the re-computation of the p predicate when its value has not been modified. To address this problem we decouple the computation stages of the predicate from its evaluation. We first define the set of variables used to compute the predicate value. Let p be a predicate used to guard a statement, $\mathcal{V}_p$ is the set of variables used to compute p. For instance, if we consider the predicate `p = x + 2 * y + b[i]` (where `i` is the *generated* iterator name), then $\mathcal{V}_p = \{x, y, b, i\}$.

The algorithm operates on the generated abstract syntax tree (AST), in a two-step process. The first step consists in identifying the statements in the AST which compute the value of p, for each predicated statement. To guarantee the optimality of the predicate computation placement, we ensure it is not possible to execute `p` less frequently while preserving the program semantics. This is done by putting the statement `p` at the highest tree level such that no statement dominated by `p` modifies any of the variables in $\mathcal{V}_p$. The second step consists in eliminating duplicated predicate computations when a given predicate is used from multiple calling sites. We proceed by inspecting the AST for all `p` statements (involving the same predicate p), and checking if any of the variables in $\mathcal{V}_p$ is ever assigned in any execution path between two occurrences of `p`. If not, then the second occurrence can be safely removed.

As a result of this optimization, the computation of the value of each predicate is minimized in terms of number of executions — again given the accuracy of $\mathcal{V}_p$ computation. The check of the predicate value before each executed instance of a predicated statement is reduced to a simple test instruction over a scalar, as shown in Figure 3(b).

4.2 Predicate Placement

The second critical optimization is to reduce the number of executed checks on the value of a predicate. To do so, we hoist the conditional `if (p)` to the highest possible level in the AST, provided the location of the computation of `p`. A typical example is the case of all reachable instances of a given loop being predicated by the same p, which is never modified during the loop execution. The instruction `if (p)` can then be hoisted outside the loop, dramatically reducing the control overhead. We proceed by merging under a common conditional all consecutive statements (under the same loop) which involve the same predicate, such that none of the statements modify the predicate value. Then, if all statements inside a loop are under the same conditional and this conditional does not depend on neither the loop iterator nor any of the statements under it, then the conditional can be safely moved around the loop instead. This optimization is reminiscent of classical if-hoisting compiler techniques, and it is efficiently performed as a code generation optimization pass. We extended the code generation tool CLooG [4] to support these extensions.

5 Experimental Results

The extension and the associated algorithms presented in this paper have been implemented in the *Polyhedral Compiler Collection* framework PoCC. It is a complete source-to-source polyhedral compiler based on available free software such as Clan (polyhedral representation extraction), Candl (data dependence analysis), LetSee and PLuTo (optimization, parallelization) CLooG (code generation), PIPLib (parametric integer programming) and PolyLib (polyhedral operations).[2]

Specifically, the implementation consisted in upgrading two modules: the extension of the polyhedral model has been implemented in irClan — an extended version of the Clan tool to extract the polyhedral representation — and in irCLooG built on the code generator CLooG[3].

To show the impact of our approach, we illustrate it with two of the state-of-the-art polyhedral optimizers.

- LetSee[4] is a complete platform for iterative compilation in the polyhedral model [28]. It leverages the algebraic properties of the polyhedral model to build an expressive search space of affine schedules, encompassing only *legal and distinct* program versions. It uses multiple heuristics to prune and search for a best program version within this space. Its optimization goal is fine-grain parallelism for vectorization and locality enhancement.
- Pluto[5] is an automatic parallelization tool based on the polyhedral model [5]. It optimizes for coarse-grain parallelism and locality simultaneously, looking for complex affine transformations based on rectangular time-tiling [20] and fusion. OpenMP parallel code can be automatically generated from sequential C, together with finer grain register tiling and transformations to enable automatic vectorization.

Our goal is to experiment these existing optimization tools *without any modification*, demonstrating the effectiveness of our extended approach on a set of irregular benchmarks. We also compare the performance improvements considering only the regular parts of these programs, when applicable. Notice that because it is out of the scope of this paper, we did not implement a sophisticated analysis of the predicates themselves or a dynamic parallelization scheme; this may significantly recuce conservativeness and allow to find better transformations. Hence, we consider the following results as a *lower bound* of the extended framework's potential.

Our experimental setup is a 2-socket Intel Quad-Core E5430 at 2.66 GHz with 16 GB of RAM, running Linux. We used the ICC compiler version 11.0, the best performing compiler on the benchmarks considered. All programs were compiled with `icc -fast -parallel -openmp` (i.e., the baseline includes automatic parallelization in ICC).

We studied typical kernels solving real computational problems that are not (partially or totally) amenable to standard polyhedral representation because of control flow irregularities. `2strings` is a program counting the occurrences of two different strings in another string. It features a very data-dependent `while`-loop

[2] `http://pocc.sourceforge.net`

[3] `http://www.cloog.org`

[4] `http://letsee.sourceforge.net`

[5] `http://pluto.sourceforge.net`

	#loops	#refs	Max Depth	SCoP Depth	Data Size
2strings	4	15	2	0	1M
Sat-add	6	27	2	2	1920x1080
QR	6	29	3	2	1024x1024
ShortPath	3	6	3	0	1000 nodes
TransClos	3	3	3	0	1000 nodes
Givens	5	64	3	1	1024x1024
Dither	2	12	2	0	1024x1024
Svdvar	4	10	3	3	1024x1024
Svdksb	5	10	2	2	1024x1024
Gauss-J	4	14	2	1	1024x1024
PtIncluded	3	19	3	1	350 vars, 15000 csts

Fig. 8. Kernel description

typical of search and pattern-matching programs. sat-add is a saturated addition of two images deblurred thanks to two stencil-based filters. It represents an example of saturated arithmetic, a very common source of irregularity in numerical or image processing programs. QR is a QR decomposition computed by Householder reflections on real data, featuring dynamic control flow in outer loops like the outer product example in Figure 4. Other forms of outer loop irregularity are exhibited in two additional benchmarks: ShortPath and TransClos, respectively a shortest-path and a transitive closure kernel based on adjacency matrices. We also provide larger loop nests to exercise search space construction and code generation scalability: the Givens benchmark computes the R matrix of the QR decomposition using Givens rotations on complex numbers; Dither is a kernel for error-distribution dithering; Svdvar computes a covariance matrix; Svbksb solves $A\boldsymbol{x} = B$ for a vector $\boldsymbol{x}$ where A is on a singular value decomposition; Gauss-J is a Gauss-Jordan elimination finding a maximum pivot, pivoting being a relevant source of data-dependent control flow; and PtIncluded checks if an integer point is included in a polyhedron, involving a linked list traversal, another usual source of control-flow irregularity.

Figure 8 lists the main properties of these programs: their number of loops, their number of array references, the maximum loop depth, the maximum loop depth of strictly affine SCoPs in the program (to quantify the extra expressiveness offered by our extension), and the data-set size.

Our results are summarized in Figure 9. For each kernel, we provide the speedup achieved by LetSee and Pluto[6] when considering only the regular parts of the program, then when using the extended representation. We also provide the compilation time penalty when considering the extended representation. N/A means that the benchmark cannot be handled in the specific context.

The results show that for the programs we considered — spanning representative sources of irregularity in loop-based computations — we are able to significantly improve performance.

[6] With or without tiling, whatever performs best.

	Speedup regular		Speedup extended		Compilation time penalty	
	LetSee	Pluto	LetSee	Pluto	LetSee	Pluto
2strings	N/A	N/A	1.18×	1×	N/A	N/A
Sat-add	1×	1.08×	1.51×	1.61×	1.22×	1.35×
QR	1.04×	1.09×	1.04×	8.66×	9.56×	2.10×
ShortPath	N/A	N/A	1.53×	5.88×	N/A	N/A
TransClos	N/A	N/A	1.43×	2.27×	N/A	N/A
Givens	1×	1×	1.03×	7.02×	21.23×	15.39×
Dither	N/A	N/A	1×	5.42×	N/A	N/A
Svdvar	1×	3.54×	1×	3.82×	1.93×	1.33×
Svbksb	1×	1×	1×	1.96×	2×	1.66×
Gauss-J	1×	1.46×	1×	1.77×	2.51×	1.22×
PtIncluded	1×	1×	1×	1.44×	10.12×	1.44×

Fig. 9. Performance and compilation time

On our target platform, applying existing polyhedral optimizers with the help of the proposed extension allows to achieve up to a 1.53× speedup for ShortPath when applying LetSee (single-threaded), and up to a 8.66× speedup for QR when applying Pluto (multithreaded, on 8 cores). We were also able to significantly improve performance for codes that were already partially regular.[7] For those programs, we obtained speedup reaching 1.51× using LetSee and from 1.09× to 8.66× using Pluto.

Typically, the performance achieved using the LetSee algorithm comes from a better locality of the memory accesses (with carefully crafted loop fusions) and compiler optimizations that have been *enabled* (e.g. vectorization). On the other hand, our approach also exposes parallelization opportunities which are exploited by Pluto (with efficient tiling and coarse-grain parallelization), which combines both parallelization and locality improvement.

We summarize our findings with more detailed insight about the transformations obtained by LetSee and Pluto for our benchmark suite:

- 2Strings is composed of two distinct non-dependent `while` loops. Using our approach, LetSee is able to fuse them leading to performance improvements. Pluto did not manage to parallelize the benchmark.
- Sat-add could be divided into two parts, a static control part and a non-static control part. Both these parts are parallel. Without our approach, Pluto is able to detect parallelism in the static control part only, yielding a performance improvement of 1.08× compared to the original code. Note that this parallelism was already found by ICC. However, through our extension, Pluto can handle and parallelize the whole code, with a speedup of 1.61×.
- QR is a code where most of the inner loops are guarded by non-affine `if` conditionals. All these loops are regular, hence LetSee and Pluto are

[7] We call a program partially regular if it contains a SCoP depth of at least 1, i.e., if it has at least one purely static loop.

able to optimize and parallelize some of them, leading to 1.04× and 1.09× speedup respectively. Nevertheless, the best performance is achieved when relying on our extension, as Pluto may now parallelize and to tile the full code. The super-linear speedup is a consequence of SIMDization that has been enabled by the transformation.

- ShortPath is composed by a perfectly nested loop of depth 3 without any SCoP, dealing with 2-dimensional matrices. Using our approach, Pluto is able to parallelize the outer loop, hence a significant 5.88× speedup; LetSee applies a loop interchange transformation on the original code. These two optimizations were performed as well on TransClos providing 1.43× and 2.27× speedup on LetSee and Pluto respectively.
- Givens features at depth 2 a sequence of data-dependent conditions to separate different cases of complex sine/cosine computations for Givens rotations. These conditions may prevent optimization. Using the extensions discussed in this paper, Pluto is able to parallelize the code. We show in the appendix the result of the optimization achieved by Pluto with the help of our extended framework. The result may be understood as a sequence of basic transformations such as skewing, tiling or index-set-splitting to extract coarse grain parallelism and to improve data locality [5]. The parallelism has been made explicit through OpenMP pragmas. The target code shows a 7.02× speedup over the original code.
- Dither is a code composed of a perfectly nested loop of depth 2 and with all the statements guarded with various non-affine `if` conditionals. Relying on the extension, Pluto is able to identify parallelism and to tile the loops, achieving a 5.42× speedup over the original code.
- On Svbksb, on the extended framework, Pluto is able to parallelize the outermost loop, leading to a speedup of 1.85× over the original code.
- Svdvar is a code composed of two perfectly nested loops, one of them is a SCoP. With the regular framework, Pluto is able to parallelize this SCoP only. Using the extended framework, Pluto performs a parallelization on both loops. Nevertheless, the same performance is achieved. This is due to the amount of calculations the SCoP carries out in this code. Gauss-J is another code where parallelization and tiling of non SCoP part become possible on Pluto with our approach, but where the SCoP part holds most of the computation time.

These results were achieved without modifying either LetSee or Pluto and using a conservative dependence analysis. They demonstrate the power of this approach, finding new or better opportunities for deep optimizations in the polyhedral model.

The price to pay for these improvements is a longer compilation time as we consider larger kernels, up to a factor 20 for LetSee due to its iterative nature. This remains practical in our experiments as the compilation time is at worse a matter of seconds. As the applicability of the polyhedral grows with our extended framework, so is the problem size for the optimizations. Our extended model raises the question of designing novel, highly scalable polyhedral optimization algorithms, while its answer is out of the scope of the paper.

6 Related Work

Much work aims at optimizing irregular codes, but only few of them are based on the polyhedral model. Most irregular polyhedral techniques were developed in the context of `while` loop parallelization. Collard explored a speculative approach to parallelize loops nests with `while` loops [10,9]. The idea is to allow a speculative execution of iterations which are not in the iteration domain of the original program. This method leads to more potential parallelism than with traditional polyhedral methods, at the expense of an invalid space-time mapping which is fixed thanks to a backtracking policy. In contrast to the speculative approach, Griebl et al. explore a conservative one. They try to enumerate a superset of the target execution space, and propose solutions to eliminate iterations that are not in the target execution space and to take care of the termination condition. For the first problem, they define what they call *execution determination* where they introduce a predicate to determine if a point in the iteration space can be executed or not. For the second point, they define and compute *termination detection.* Griebl and Lengauer [22] propose another solution using a communication scheme in a distributed-memory model to determine the upper bounds of the target loops, but this solution increases the execution time of the scanning. For the same problem but on shared-memory models, Griebl and Collard [21] describe a so-called counter scheme. Griebl et al. [17,18] present another one called maximum scheme.

Other authors concentrated on extending the expressiveness of the polyhedral model in special cases; these efforts are complementary to our conservative yet general approach. Palkovič wrote the most comprehensive monograph on the topic [27]. In contrast, our approach handles any function body and *transparently inherits all existing optimization and parallelization techniques* based on the polyhedral model.

In addition, our extended model opens the door to important loop transformations targeted to data-dependent control flow. For example, Decoupled Software Pipelining (DSWP) [33] extracts and exploits pipeline parallelism from irregular codes involving complex control flow and data structures. Full automation of DSWP remains a challenge, due to the intricacy of the transformations involved and their interplay with other optimizations. Another example is Deep Jam [7], a generalization of loop fusion and unroll-and-jam to dynamic control flow, targeted at instruction-level and vector parallelism. Deep Jam is at least as complex as DSWP to automate.

7 Conclusion

This paper completely and definitely overcomes the control-flow limitations of the polyhedral model in an intraprocedural setting. The solution comes from a sleek and natural modeling of control-flow predicates at all stages of a polyhedral compilation framework. This extension goes far beyond the state-of-the-art which only addresses special non-affine cases. The main difficulty resides in the design of an extended code generation algorithm supporting those extensions while limiting control-flow overhead. Several subtle difficulties also trickle down to the extraction of the polyheral representation and the storage mapping of control predicates (privatization). We experimentally validated our approach, demonstrating new optimization opportunities for irregular programs as well as improving previous results on partially-regular applications.

The static control limitations of the polyhedral model are now history. Research may now concentrate on accurate static/dynamic analysis, and complementing speculative optimization and parallelization techniques with aggressive program transformations. The only important limitation left is the high complexity of the algorithms supporting polyhedral operations — typically exponential in the number of statements and/or the number of array references and/or the loop nesting depth. Enlarging its application domain stresses the scalability of these algorithms even further. In this context, we are working on macro-block and region formation heuristics, as well as novel polyhedral optimizations that scale to full functions.

References

1. Allen, J., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann Publishers, San Francisco (2002)
2. Banerjee, U.: Data dependence in ordinary programs. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign (November 1976)
3. Barthou, D., Cohen, A., Collard, J.-F.: Maximal static expansion. In: ACM Symp. on Principles of Programming Languages (POPL 1998), San Diego, California (1998)
4. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT 2004), Juan-les-Pins, September 2004, pp. 7–16 (2004)
5. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: Proc. of the 2008 ACM Conf. on Programming language design and implementation (PLDI 2008), Tucson, AZ, USA (June 2008)
6. Boulet, P., Darte, A., Silber, G.-A., Vivien, F.: Loop parallelization algorithms: From parallelism extraction to code generation. Parallel Computing (1998)
7. Carribault, P., Cohen, A., Jalby, W.: Deep Jam: Conversion of coarse-grain parallelism to instruction-level and vector parallelism for irregular applications. In: Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT 2005), pp. 291–300 (2005)
8. Chen, C., Chame, J., Hall, M.: A framework for composing high-level loop transformations. Technical Report 08-897, USC Computer Science (June 2008)
9. Collard, J.-F.: Space-time transformation of while-loops using speculative execution. In: Proc. of the 1994 Scalable High Performance Computing Conf. (1994)
10. Collard, J.-F.: Automatic parallelization of while-loops using speculative execution. Int. J. Parallel Program. 23(2), 191–219 (1995)
11. Collard, J.-F., Barthou, D., Feautrier, P.: Fuzzy array dataflow analysis. In: ACM Symp. on Principles and practice of parallel programming (PPOPP 1995), Santa Barbara, California, pp. 92–101 (1995)

12. Creusillet, B., Irigoin, F.: Exact versus approximate array region analyses. In: Sehr, D., Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) LCPC 1996. LNCS, vol. 1239, pp. 86–100. Springer, Heidelberg (1997)
13. Feautrier, P.: Dataflow analysis of scalar and array references. Intl. Journal of Parallel Programming 20(1), 23–53 (1991)
14. Feautrier, P.: Some efficient solutions to the affine scheduling problem, part I: one dimensional time. Intl. J. of Parallel Programming 21(5), 313–348 (1992)
15. Feautrier, P.: Some efficient solutions to the affine scheduling problem, part II: multidimensional time. Intl. J. of Parallel Programming 21(6), 389–420 (1992)
16. Franke, B., O'Boyle, M.: Array recovery and high level transformations for dsp applications. In: CPC 10 Intl. Workshop on Compilers for Parallel Computers, Amsterdam, January 2003, pp. 29–38 (2003)
17. Geigl, M., Griebl, M., Lengauer, C.: A scheme for detecting the termination of a parallel loop nest. In: Proc. GI/ITG FG PARS 1998 (1998)
18. Geigl, M., Griebl, M., Lengauer, C.: Termination detection in parallel loop nests with while loops. Parallel Comput. 25(12), 1489–1510 (1999)
19. Girbal, S., Vasilache, N., Bastoul, C., Cohen, A., Parello, D., Sigler, M., Temam, O.: Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. Intl. J. of Parallel Programming 34(3) (2006)
20. Griebl, M.: Automatic parallelization of loop programs for distributed memory architectures. Habilitation thesis. FMI, universität Passau (2004)
21. Griebl, M., Collard, J.-F.: Generation of synchronous code for automatic parallelization of while loops. In: Haridi, S., Ali, K., Magnusson, P. (eds.) Euro-Par 1995. LNCS, vol. 966, pp. 315–326. Springer, Heidelberg (1995)
22. Griebl, M., Lengauer, C.: On scanning space-time mapped while loops. In: Buchberger, B., Volkert, J. (eds.) CONPAR 1994 and VAPP 1994. LNCS, vol. 854, pp. 677–688. Springer, Heidelberg (1994)
23. Karp, R., Miller, R., Winograd, S.: The organization of computations for uniform recurrence equations. J. ACM 14(3), 563–590 (1967)
24. Knobe, K., Sarkar, V.: Array ssa form and its use in parallelization. In: ACM Symp. on Principles of Programming Languages (POPL 1998), California (1998)
25. Kong, X., Klappholz, D., Psarris, K.: The i test: A new test for subscript data dependence. In: ICPP 1990 Intl. Conf. on Parallel Processing (August 1990)
26. Lim, A.: Improving Parallelism and Data Locality with Affine Partitioning. PhD thesis, Stanford University (2001)
27. Palkovič, M.: Enhanced Applicability of Loop Transformations. PhD thesis, T. U. Eindhoven, The Netherlands (September 2007)
28. Pouchet, L.-N., Bastoul, C., Cohen, A., Cavazos, S.: Iterative optimization in the polyhedral model: Part II, multidimensional time. In: ACM Conf. on Programming Language Design and Implementation (PLDI 2008), Tucson, Arizona (June 2008)
29. Pouchet, L.-N., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., Sadayappan, P.: Hybrid iterative and model-driven optimization in the polyhedral model. Technical Report 6962, INRIA Research Report (June 2009)
30. Pugh, W.: The omega test: a fast and practical integer programming algorithm for dependence analysis. In: Proc. of the ACM/IEEE Conf. on Supercomputing (SC 1991), pp. 4–13 (1991)
31. Pugh, W., Wonnacott, D.: An exact method for analysis of value-based array data dependences. In: Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) LCPC 1993. LNCS, vol. 768, pp. 546–566. Springer, Heidelberg (1994)
32. Quilleré, F., Rajopadhye, S., Wilde, D.: Generation of efficient nested loops from polyhedra. Intl. Journal of Parallel Programming 28(5), 469–498 (2000)

33. Rangan, R., Vachharajani, N., Vachharajani, M., August, D.I.: Decoupled software pipelining with the synchronization array. In: Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT 2004) (September 2004)
34. Rauchwerger, L., Padua, D.A.: The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In: ACM Conf. on Programming Language Design and Implementation (PLDI 1995) (June 1995)
35. Rus, S., Pennings, M., Rauchwerger, L.: Sensitivity analysis for automatic parallelization on multi-cores. In: ACM Intl. Conf. Supercomputing, ICS 2007 (2007)
36. Rus, S., Rauchwerger, L.: Hybrid dependence analysis for automatic parallelization. Technical report, Parasol Laboratory, Texas A&M University (2003)
37. Rus, S., Rauchwerger, L., Hoeflinger, J.: Hybrid analysis: Static & dynamic memory reference analysis. Intl. J. of Parallel Programming 31(4) (2003)

The Hot Path SSA Form: Extending the Static Single Assignment Form for Speculative Optimizations

Subhajit Roy and Y.N. Srikant

Computer Science and Automation Department,
Indian Institute of Science
{subhajit,srikant}@csa.iisc.ernet.in

Abstract. The Static Single Assignment (SSA) form has been an eminent contribution towards analyzing programs for compiler optimizations. It has been affable to the design of simpler algorithms for existing optimizations, and has facilitated the development of new ones. However, speculative optimizations — optimizations targeted towards speeding-up the "common cases" of a program — have not been fortunate enough to savor an SSA-like intermediate form. We extend the SSA form for speculative analyses and optimizations by allowing only *hot reaching definitions* — definitions along frequent acyclic paths in the program profile — to reach its respective uses; we call this representation the *Hot Path SSA form*. We propose an algorithm for constructing such a form, and demonstrate its effectiveness by designing the analysis phase of a novel optimization — Speculative Sparse Conditional Constant Propagation: an almost obvious extension of Wegman and Zadeck's Sparse Conditional Constant Propagation algorithm. Our experiments on some SPEC2000 programs proves the potency of such an optimization.

1 Introduction

Program analyses and optimizations have benefited immensely from the SSA form as an intermediate representation. An extremely simple idea — allow only a single definition of a variable to reach the statements using it — prunes out false dependencies, and factors long use-def chains into a web of short, simple ones. A multitude of optimizations were either made possible, or were heavily empowered by the SSA form — sparse conditional constant propagation, global value numbering, and strength reduction to name a few.

However, speculative optimizations — optimizations biased towards frequently executed paths — have not been fortunate enough to enjoy an SSA-like intermediate representation. These optimizations have recently attracted a lot of attention, and are now recognised as a major vehicle towards improving program performance.

Modern compilation systems, acknowledging the importance of such unconventional optimizations, have started providing support for speculative analysis and transformation. However, in most of the intermediate representations,

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 304–323, 2010.

the profiling information is not integrated into the static program representation. This makes implementing speculative optimizations cumbersome, having to handle too many data-structures. Additionally, the absence of an SSA-like sparse representation has hindered the development of efficient algorithms for speculative optimizations.

We propose to extend the power of the SSA form to speculative optimizations by separating the *hot* use-def chains from the cold ones, thus allowing a speculative optimizer to “see” only the most-likely dataflow facts. However, the “non-speculative” SSA form is not lost: a traditional optimizer can still choose to constrain itself to the non-speculative form by ignoring the speculative information. The SSA form is not erased — just suitably extended with speculative information — obviating the necessity of constructing and maintaining the non-speculative SSA form separately; at the same time, this SSA-like intermediate form is much more amenable to speculative analyses and optimizations.

We call this extension to the SSA form as the “Hot Path SSA (HPSSA) form”. *As the HPSSA form honours the constraint imposed by the SSA form (that of a single reaching definition for every use), many of the SSA-based algorithms for traditional optimizations developed over the last couple of decades (almost) immediately become available to speculative optimizers.*

Following are our contributions in this paper:

- We propose a novel program representation — the Hot Path SSA (HPSSA) form — that allows a use to witness only the “more-likely” reaching definitions (section 4);
- We present an algorithm for constructing the HPSSA form (section 5);
- We demonstrate the potency of the HPSSA form by designing the analysis phase of a novel speculative optimization — Speculative Sparse Conditional Constant Propagation (SSCP) — that identifies both “safe” (expressions that are sure to be constants) and “speculative” (expressions that are more-likely to be constants) constants in a given program. An almost trivial extension of Wegman and Zadeck’s SCP algorithm [21], SSCP exhibits the possibilities of developing new speculative optimizations using the HPSSA form by tailoring of existing SSA-based traditional optimizations (section 6).

2 Background

2.1 The Static Single Assignment Form

A program is said to be in Static Single Assignment (SSA) form if each use of a variable has *exactly* one reaching definition. A special operator, the ϕ-function, *merges* multiple definitions from different paths into a single definition, forcing any subsequent *use* to see exactly one definition.

Figure 1 shows the SSA form of a program. Notice how the definitions of x at b_1, d_1 and e_1 are “merged” into a single definition at the statement f_1, thus making x_9 the only definition reaching the uses g_3, h_3 and i_1. Understandably, the use-def structure of a program in SSA form is extremely simple — allowing the design of cleaner and faster algorithms.

2.2 Acyclic Path Profiling

Ball and Larus [2] proposed an efficient algorithm for profiling acyclic paths — paths that terminate either at loop-backedges or at procedure exits. Essentially, an acyclic path profiler "chops-off" paths at a backedge, erasing the sequence in which the acyclic paths in the loop were actually executed. The Ball-Larus algorithm is widely used for speculative analyses and optimizations. We use acyclic path profiles to expose frequent use-def chains in the HPSSA form.

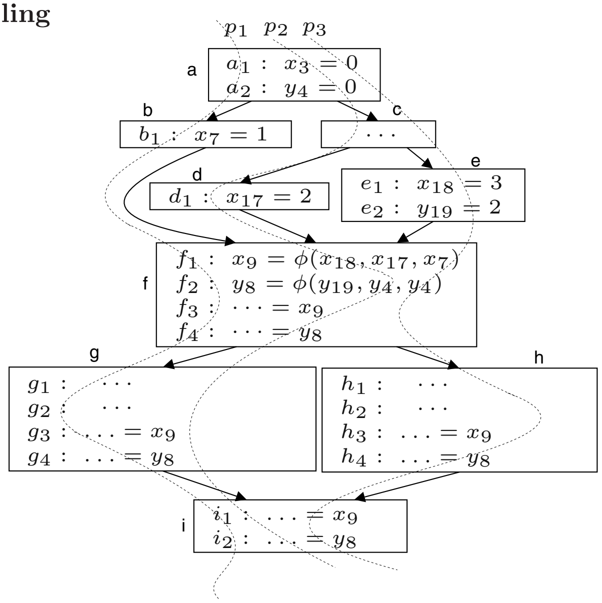


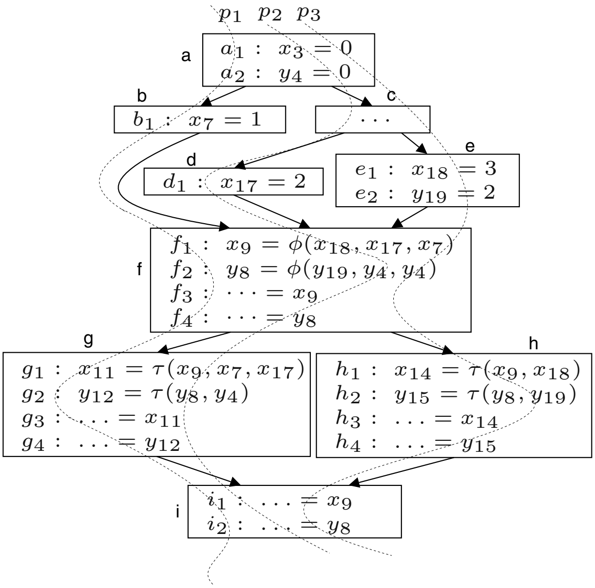
Fig. 1. A program in the SSA form. (***Hot acyclic paths***: p_1:`abfgi`; p_2:`acdfgi`; p_3:`acefhi`).

2.3 A Peek at the Hot Path SSA Form

In this paper, we propose to tie the run-time behaviour of a program — as indicated by the frequently executed acyclic paths — directly to its static program representation, thus providing a convenient data-structure for the speculative optimizers. In the proposed representation, which we call the Hot Path SSA (HPSSA) form, an additional construct — the τ-function — is introduced to capture information relevant for speculative analyses and optimizations. The τ-functions act as "filters", separating the more-likely use-def chains from the lesser-likely ones. The first argument of the τ-function is the traditional meet-over-all-paths reaching definition; the rest of the arguments are the "hot" reaching definitions: definitions that are more-likely to reach the respective program point.

Figure 2 shows the HPSSA form of the program in Figure 1. Consider the basic-block g: the τ-function at g_1 indicates that x_9 is the "safe" meet-of-all-paths reaching definition, though the definitions of x_7 and x_{17} are more likely to reach this program point (via the ϕ-statement at f_1). Similarly, for g_4, h_3 and h_4, the hot reaching definitions are from definitions of y_4, x_{18} and x_{19} respectively — all of which are definitions to constants. Hence, the HPSSA form exposes the fact that the variables y_{12}, x_{14} and y_{15} are *more likely* to be constants with values 0, 3 and 2 respectively — enabling a speculative optimizer to *speculatively* "predict" the value of these variables.

Though the HPSSA form uses acyclic path profiles, it is still adept at propagating hot reaching definitions across loop-boundaries. Figure 3 shows the



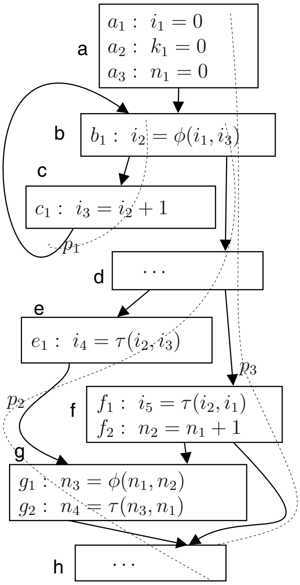

a
$a_1 :\ i_1 = 0$
$a_2 :\ k_1 = 0$
$a_3 :\ n_1 = 0$
b
$b_1 :\ i_2 = \phi(i_1, i_3)$
c
$c_1 :\ i_3 = i_2 + 1$
p_1
d
$\cdots$
e
$e_1 :\ i_4 = \tau(i_2, i_3)$
p_2
p_3
f
$f_1 :\ i_5 = \tau(i_2, i_1)$
$f_2 :\ n_2 = n_1 + 1$
g
$g_1 :\ n_3 = \phi(n_1, n_2)$
$g_2 :\ n_4 = \tau(n_3, n_1)$
h
$\cdots$

Fig. 2. The program in Figure 1 translated to the Hot Path SSA (HPSSA) form (***Hot paths***: p_1:`abfgi`; p_2:`acdfgi`; p_3:`acefhi`)

Fig. 3. HPSSA form for a program with loops (***Hot paths***: p_1:`bc`; p_2:`bdegh`; p_3:`abdfh`)

HPSSA form of a program with a loop. Notice how the variable i_3 becomes the hot reaching definition at the basic block e, even though i_3 reaches the node e along a path that contains a backedge (as `c-b` is a backedge, `c-b-d-e` is not a segment of any acyclic path).

In this paper, we only assume reducible flow-graphs; we also assume the existence of a loop-preheader node (leading to the loop-header) for each loop in the program.

3 Thermal Properties of a Program

In this section, we establish a few terms and notations that we use in the rest of the paper.

3.1 Thermal States of Program Entities

Definition 1. *Hot/Cold Paths: A program path $p : n_1 \rightsquigarrow n_2$ is said to be* hot (cold) *if the sequence of edges from node n_1 to n_2 appears (*does not appear*) in any* profiled path *that occurs* frequently *in the program profile.*

The above definition has been intentionally left slightly ambiguous to make it general enough to encompass various profiling and hot path selection schemes. The phrase "profiled path" implies any sequence of basic-blocks that is collected by a control-flow profiler; for instance, the “profiled path” is an edge for an edge profiler, an acyclic path for a Ball-Larus path profiler, and a path spanning multiple loop iterations for a k-iteration profiler [17,20]. In this paper (and our implementation), a “profiled path” refers to intraprocedural acyclic paths, profiled using a Ball-Larus profiler. The qualifier “frequently” in the above definition depends on the hot path selection scheme: we may select hot paths by a threshold frequency, or pick a finite number of the most commonly executed paths from each procedure.

Definition 2. *Temperature (θ) of a node (edge) is defined as:*

- *hot: if the node (edge) is present on a hot path;*
- *cold: if the node (edge) is not present on any hot path.*

A backedge b in a flow-graph is marked hot if, either of the dummy edges, δ_{start} to a loop-header h or δ_{end} from a loop-tail t, is hot[1]; this is understandable, as any control-flow through a dummy edge reported by the Ball-Larus profiler indicates a control-flow through the corresponding backedge in the program flow-graph.

We will use the notation $\theta(n)$ to denote the temperature (hot/cold) of a program entity (nodes, edges or paths). The predicates $\theta_h(n)$ /$\theta_c(n)$ denote that the entity n is hot/cold.

For example, in Figure 1, all the nodes and edges are hot; the path $c \rightarrow d \rightarrow f \rightarrow g$ is hot (through the path p_2) while the path $e \rightarrow f \rightarrow g$ is cold.

Definition 3. *Hot/Cold Reaching Definitions and Definition Chains*
A definition δ at a basic-block n_1 is said to reach a respective use at a basic-block n_2 hot if there exists a hot path from n_1 to n_2, and δ is not killed along that path.
A definition δ at a basic-block n_1 is said to reach a respective use at a basic-block n_2 cold if there does not exist a hot path from n_1 to n_2, and δ is not killed at least along one cold path from n_1 to n_2.

Consider Figure 1: treating a ϕ-function not as a definition, but as a label to the set of definitions in its argument set, we can see that though the meet-over-all-paths reaching definition set at g_3 is $\{x_{18}, x_{17}, x_7\}$, the definition x_{18} does not reach it via any hot path. So, x_{18} is a cold reaching definition at g_3, while x_7 and x_{17} are the hot reaching definitions (reaching the node g via the paths p_1 and p_2). In the SSA form, the ϕ-functions can be seen as creating a *definition chain*, that is broken only by a non-ϕ definition: $x_7 \rightarrow x_9$ and $x_{17} \rightarrow x_9$ are the *hot reaching definition chains* at g_3 , while $x_{18} \rightarrow x_9$ is a *cold reaching definition chain.* In the HPSSA form, the τ-functions “kill” the cold definition chains: for example, in Figure 2, $x_{18} \rightarrow x_9$ no longer reaches g_3 as it is killed by g_1.

[1] The Ball-Larus profiler converts a flow-graph with cycles into a directed acyclic graph (DAG) by adding dummy edges, $\delta_{start}/\delta_{end}$, to and from the backedge source/target (respectively) for each loop in the program [2].

3.2 The Structure of Profiled Acyclic Paths

The set of acyclic paths can be grouped by the node they initiate from — the program entry or a loop header; we refer to this node as the *incubation node* for the acyclic paths originating from it. In Figure 3, node a is the incubation node for p_3, while b is the incubation node for p_1 and p_2.

A set of profiled acyclic paths $\{p_1, p_2, \ldots p_n\}$ entering a node u are said to be *buddies* at u if the paths $p_1, p_2, \ldots, p_n$ have seen *exactly the same sequence of edges from their incubation node*; the group of all buddies are said to form the *BuddySet* at a node. Consider Figure 1 with the following set of hot paths:

p_1 : a-b-**f**-g-i; p_2: a-c-d-**f**-g-i; p_3: a-c-e-**f**-h-i; p_4: a-c-e-**f**-g-i; p_5: a-b-**f**-h-i.

$BuddySet_a(f) = \{\{p_1, p_5\}, \{p_2\}, \{p_3, p_4\}\}$; i.e. p_1 and p_5 are buddies, so are p_3 and p_4, while p_2 has no buddy at f.

Notations. Let us define a few notations to ease the following discussion:

- $Paths(u)$: The set of all profiled "hot" acyclic paths reaching the node u.
- $Paths_s(u)$: The set of all profiled "hot" acyclic paths reaching the node u that initiate from the incubation node s.
- $Paths_s(u \rightarrow v)$: The set of all profiled "hot" acyclic paths reaching the node u that initiate from the incubation node s and progress along the edge $u \rightarrow v$ from u; without the subscript s, it denotes paths from all incubation nodes that progress along $u \rightarrow v$.
- $S(u)$: Set of all incubation nodes in the set of all profiled "hot" acyclic paths reaching node u.
- $N(\alpha)/E(\alpha)$: Set of all nodes/edges in the path α.

4 The Hot Path SSA (HPSSA) Form

A speculative optimizer needs to identify "highly likely facts" — facts propagated along frequently executed paths — to perform optimizations that, though not legal on all static paths, "mostly" benefits the program. The HPSSA form uses a novel construct — the τ-function — to "filter" definitions along cold paths, thus allowing only *hot* definitions to propagate further. The form of a τ-statement is shown below:

$$x_{out} = \tau(x_0, x_1, \ldots, x_n)$$

The τ-function argument list contains two types of arguments:

- Safe (or non-speculative) argument: The first argument, x_0, is the safe argument. It carries the variable version that needs to be assigned to x_{out} to perform safe analyses and optimizations over the program.
- Speculative arguments: The rest of the arguments, $x_1 \ldots x_n$, are the speculative arguments, carrying the variable versions that reach the current node along the frequently executed paths; a speculative optimizer can treat the definition of x_{out} as the union of these speculative arguments to perform speculative analyses and optimizations over the heavily executed paths.

The τ-function can be seen as a conditional ϕ-function:

$$\tau(x_0, x_1, \ldots, x_n) = \begin{cases} \phi(x_0) & \textit{safe interpretation} \\ \phi(x_1, \ldots, x_n) & \textit{speculative interpretation} \end{cases}$$

If a program is in the Hot Path SSA form, then,

- each use of a variable is reachable by a single definition;
- if the *safe* interpretation of the τ-function is used, each use of a variable is reachable by the meet-over-all-paths reaching definition chains;
- if the *speculative* interpretation of the τ-function is used, each use of a variable in a *hot* basic-block is reachable ***only*** by the ***meet-over-hot-paths*** reaching definition chains (or the meet-over-all-paths reaching definition chains, if the use is not reachable from any meet-over-hot-paths reaching definition chain).

With the *speculative* interpretation, the set of reaching definition chains at even a cold basic-block might be smaller than that corresponding to the meet-over-all-paths, as some of the definition chains may be "killed" by τ-functions on their way to the cold node.

Each speculative argument x_i in a τ-function is mapped to the set of hot profile paths along which the definition corresponding to x_i is reached. In Figure 2, for the variable x in g_1, the τ-function allocates the parameter x_7 corresponding to the path p_1, and the parameter x_{17} for the path p_2. However, for the variable y at g_2, it allocates only one parameter, y_4, corresponding to both p_1 and p_2 as the same definition (from statement a_2) reaches it along both the paths.

The HPSSA form honours the constraint imposed by the SSA form: each use is reachable by a single definition — encouraging the development of speculative extensions of existing SSA-based algorithms on the HPSSA form.

Exiting the HPSSA form

Exiting the HPSSA form is extremely simple — a τ-statement is replaced by a copy statement from the safe-argument to the defined variable:

$$x_{out} = \tau(x_0, x_1, \ldots, x_n) \quad \rightarrowtail \quad x_{out} = x_0$$

This puts the program in the SSA form; one can then use a standard out-of-SSA algorithm to exit the SSA form.

5 Constructing the HPSSA Form

In this this section, we discuss the construction of the HPSSA form. The original program (not in SSA form) is transformed into HPSSA form in four steps:

- Insert ϕ-statements: The classic algorithm for construction of the minimal SSA form [8] places ϕ-statements at the iterated dominance frontier of each definition in the program. A node v is said to be in the dominance frontier of another node u iff u does not dominate v while a predecessor of v is dominated by u.

- Insert τ-statements: For each variable x, we identify program points that necessitate a τ-function, and, at all such points, insert a definition of the form $x = \tau(x)$ (discussed in detail in section 5.1).
- Variable renaming: The definitive variable renaming algorithm [8] uses a variable stack to propagate reaching definitions by traversing the basic-blocks over the dominator tree. The correctness of our algorithm requires a depth-first traversal over the dominator tree. Note that this phase also renames the sole argument in the inserted τ-functions to the variable version corresponding to the meet-over-all-paths "safe" reaching definition.
- Allocation of the τ-function arguments: Finally, we allocate the speculative arguments to the τ-functions in correspondence to the hot reaching definition chains (discussed in detail in section 5.2).

Note that after step 3, the program is in SSA form, and after step 4, it is in HPSSA form. We have intentionally kept the phases for building the SSA form (steps 1 and 3) clearly distinct from the steps required for constructing the HPSSA form (steps 2 and 4) to apprise the essentials of the HPSSA construction algorithm. It will be apparent that the phases need not be separate — some of them can be combined in an efficient implementation.

5.1 Thermal Frontiers: Placing τ-functions

We call definitions due to ϕ and τ- functions as *pseudo* definitions, differentiating them from other *concrete* definitions; the corresponding statements are called pseudo/concrete statements. We define the set of *visible* definitions in the basic-block u as the last definition of each variable in the block: these definitions are the only ones that are "seen" by the basic-blocks reachable from u. In the following discussion, a reaching definition would refer to only concrete definitions; pseudo reaching definitions can be seen as the set of concrete definitions that were "merged" due to a ϕ- or a τ-function.

Each definition $x := \ldots$ in the program can potentially lead to the insertion of a τ-statement for variable x. In a basic-block, a τ-statement is inserted after all the ϕ-statements (if any), before any of the *concrete* statements.

The ϕ-functions act as *definition mergers* — "merging" multiple definitions into a single one. Comparably, the τ-functions act as *definition filters* — separating hot definitions from cold ones, which were merged by previously occurring ϕ-functions. *Hence, a node n will need a τ-function for a variable v if, and only if, both a hot and a cold reaching definition for the variable v arrive at n.*

The minimal SSA construction algorithm uses an exquisite structure — the Dominance Frontier — to insert the ϕ-statements. To build the HPSSA form, we identified a similar structure to place the τ-statements: the Thermal Frontier.

Definition 4. *Thermal Frontier: A node v is said to be in the Thermal Frontier (TF) of a reaching definition d, where d is defined at a node u, ($v \in TF(u, d)$), iff the node v is also exposed to a reaching definition d', defined at a node w (w not dominated by u), such that $\theta(u \rightsquigarrow v) \neq \theta(w \rightsquigarrow v)$. Also, v must be the first node in the paths $u \rightsquigarrow v$ and $w \rightsquigarrow v$ that satisfies the above properties.*

Stated informally, a node v is in the thermal frontier of a hot/cold reaching definition d (defined at u), if v is also reachable by a different cold/hot (respectively) definition d' (defined at w), while being the first node along $u \rightsquigarrow v$ and $w \rightsquigarrow v$ to satisfy the conditions.

Unlike Dominance Frontiers, Thermal Frontiers need not be join nodes. For example, in Figure 2, node $g \in TF(b, x_7)$ as x_7 is a hot reaching definition (along p_1) and g is also reachable by the cold reaching definition x_{18}.

It is apparent that τ-functions for a definition d at a node u will be needed at the iterated TF(u,d). We define the Iterated Thermal Frontier in exactly the same way as iterated join and iterated dominance frontier were defined by Cytron et al.[8].

Definition 5. *Let $\gamma_x(u)$ return the visible definition of the variable x in the basic-block u; then, for a set of nodes κ, the Iterated Thermal Frontier (ITF) is the limit of the increasing sequence of sets of basic-blocks:*
$TF^x(\kappa) = \bigcup_{u \in \kappa} TF(u, \gamma_x(u))$
$TF_1^x = TF^x(\kappa)$
$TF_{i+1}^x = TF^x(\kappa \cup TF_i^x)$
$ITF^x = TF_\infty^x$, *where* TF_∞^x refers to the fixpoint, i.e. when $TF_i^x = TF_{i+1}^x$

However, as the ϕ-statements are inserted by a prior phase, placing the τ-functions does not require fixpoint computation: a simple topological traversal over the CFG nodes suffices. Fixpoint computation is generally required if dataflow information can change after propagating through a backedge. While placing the τ-functions, if a τ-statement for a variable x is inserted in the header h of a loop due to a definition in the loop body (the only case that requires fixpoint computation), then, the loop-header h is sure to contain a ϕ-statement (as no node in the loop-body can dominate h). Hence, if the CFG nodes are processed in the topologial order, insertion of τ-functions at the required nodes due to the definition of the variable x at h would have already happened.

Theorem 1. For a set of visible definitions of a variable x at a set of nodes κ, τ-statements would be required at the Iterated Thermal Frontier ITF^x for variable x.

The following lemma states the necessary condition for computing the set of Thermal Frontiers.

Lemma 1. *A node $n \in TF(u, d^x)$ for a definition d^x (of a variable x) if*

- Condition I: n is the junction of a hot and a cold path, i.e., paths at different temperatures meet at this node;
- Condition II: n is reachable by at least two different definitions of the variable x.

Proof. If condition I fails, a τ-function is unnecessary as n can then be reachable by only hot or only cold definitions of x. If condition II fails, a τ-function is again unnecessary as the node is then *dominated* by a definition of x.

However, note that the above lemma is not a sufficient condition: a node $v \notin TF(u, d^x)$ if the same definition d^x reaches v via both a hot and cold path (satisfying condition I), while v is also reachable by a different hot definition (of x), d', along a *separate* hot path (satisfying condition II). Hence, the above lemma may identify spurious Thermal Frontiers: our HPSSA algorithm inserts τ-function templates at all points identified by the lemma, leaving the task of weeding out unnecessary τ-statements to the τ-argument allocation phase (section 5.2). In the rest of the discussion, we denote the set of Thermal Frontiers computed according to Lemma 1 as $TF(u, d)$, and denote the ideal set of Thermal Frontiers (as defined in Definition 4) as $TF_{ideal}(u, d)$.

Let us now sketch an algorithm for computing the Thermal Frontier of a node: we first identify certain nodes that are "junctions" of hot and cold paths (we call them Caloric Connectors), and thus, satisfy the first condition of Lemma 1; we then identify a scheme for satisfying the second condition.

Caloric Connector

Definition 6. *Caloric Connector (CC): A node $n_{cc} \in CC$ if, for distinct nodes n and n' ($n \neq n'$), there exist paths $n \rightsquigarrow n_{cc}$, $n' \rightsquigarrow n_{cc}$ such that $\theta(n \rightsquigarrow n_{cc}) \neq \theta(n' \rightsquigarrow n_{cc})$, and for all nodes $n'' \in (N(n \rightsquigarrow n_{cc}) \cap N(n' \rightsquigarrow n_{cc})) - \{n_{cc}\}$, $n'' \notin CC$.*

In other words, a node n_{cc} is a Caloric Connector in a given graph (for a given set of *hot* paths) if there exist distinct nodes n and n', such that n and n' can reach n_{cc} through paths having different temperatures, and n_{cc} is the first common node in $n \rightsquigarrow n_{cc}$ and $n' \rightsquigarrow n_{cc}$ satisfying these properties.

Consider Figure 1: the node g is a Caloric Connector as the path $d \rightarrow f \rightarrow g$ is hot while $e \rightarrow f \rightarrow g$ is cold, while both the "predecessor" paths ($d \rightarrow f$ and $e \rightarrow f$) are hot.

Lemma 2. *A hot acyclic path $t \rightsquigarrow u$ extended by a forward edge $u \rightarrow v$ forms a cold path $t \rightsquigarrow u \rightarrow v$ if, for some incubation node s, there exists a set of buddy paths $B \in BuddySet_s(u)$ among the paths at u, such that none of the buddies $\sigma \in B$ traverse the edge $u \rightarrow v$.*

Lemma 3. *If an acyclic path $t \rightsquigarrow u \rightarrow v$ is cold, then, either*

- *$t \rightsquigarrow u$ is cold, or*
- *$s \rightsquigarrow t \rightsquigarrow u$ is hot, and $\exists B \in BuddySet_s(u)$, such that none of the buddies $\sigma \in B$ traverses $u \rightarrow v$ (where s is the incubation node for $s \rightsquigarrow t \rightsquigarrow u$).*

The intuition for the above lemmas is as follows: Each set of buddies at u, $B_i \in BuddySet_s(u)$, correspond to a unique sequence of edges $(s \rightsquigarrow u)_i$ from s to u, distinct from that of any other buddy set $B_j \in BuddySet_s(u)$, $B_i \neq B_j$. If no hot path $p \in B_i$ selects the edge $u \rightarrow v$, that particular sequence of edges $(s \rightsquigarrow u)_i \rightarrow v$ is surely missing among the hot paths reaching v. This implies that the path $(s \rightsquigarrow u)_i \rightarrow v$ is cold. We omit the formal proofs for want of space.

Algorithm 1. Computing the set of Caloric Connectors

```
Traverse each node $v$ in the graph (in the topological order) in the following manner:
 1. Initialize $hasAColdPath$ and $hasAHotPath$ to $false$.
 2. For all edges $e : u \rightarrow v$,
    – if $\theta_c(u \rightarrow v)$, set $hasAColdPath = true$;
    – if $\theta_h(u \rightarrow v)$,
       (a) Set $hasAHotPath = true$;
       (b) If $e$ is not a backedge, and if, $\exists B \in BuddySet_s(u)$ (for some incubation node $s$) such that $B$ does not intersect $Paths(u \rightarrow v)$, set $hasAColdPath = true$.
 3. If both $hasAColdPath$ and $hasAHotPath$ are $true$, add $v$ to the set of Caloric Connectors.
```

The algorithm for computing the set of Caloric Connectors (Algorithm 1) is targeted at identifying if both a hot and a cold path can reach a node. Iterating through all nodes in the CFG in topological order, for each node u, the algorithm examines the temperature of each outgoing edge $u \rightarrow v$. It decides on the existence of a hot and/or a cold path at v in accordance to Lemma 2 and 3, and sets the flags $hasHotPath$ and $hasColdPath$ accordingly. A node v is marked as a Caloric Connector if it has both a hot and a cold path reaching it.

Computing Thermal Frontiers. For a concrete definition d and a basic-block $v \in TF(u, d)$, the second condition of Lemma 1 is satisfied if v is in the dominance frontier of u (the node v is then also exposed to a different definition d' at a node w that is not dominated by u).

The case for pseudo definitions is slightly different: We ideate a ϕ-statement $x_3 = \phi(x_1, x_2)$ not as a single definition, but as a set of concrete definitions $\{x_3 = x_1, x_3 = x_2\}$ being propagated to all the outgoing paths from the definition-site; we also envision the τ-statements similarly, but with only the speculative arguments[2]. As all paths from a pseudo definition d, defined at a node u, are now ideated as carrying this *set* of definitions (instead of just d), the first Caloric Connector (n_{cc}) on each outgoing path from u, called the Closest Caloric Connectors of u (CCC(u)), satisfies Lemma 1 — provided the pseudo-definition d actually reaches n_{cc}. Figure 4 illustrates this case when d does not reach n_{cc}: Let $w \in CCC(u)$; however, $w \notin TF(u, d_1)$ as the pseudo-definition d_1 is "killed" by the concrete definition d_2 at v, making d_2 the dominating definition for w — violating condition II of Lemma 1.

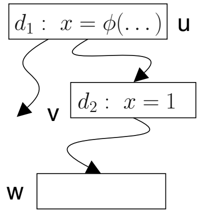


Fig. 4. Violation of condition II of Lemma 1

Algorithm 2 outlines our algorithm for inserting τ-nodes.

[2] In the HPSSA construction algorithm, the hot definitions are "percolated" through the ϕ and τ statements as the percolated definitions may appear as arguments to future τ-statements.

Algorithm 2. Inserting τ-statements

```
Process each control-flow graph node $v$ in the topological order as follows:
  1. For all visible definitions "$d:\ x = \ldots$" in the basic-block $v$,
       – if $d$ is a pseudo definition: if the pseudo definition $d$ is a reaching definition at
         $v$ ($d$ is not killed by concrete definitions along some path to $v$), add the set of
         the Closest Caloric Connectors for $v$ to $TF(v, d)$;
       – if $d$ is a concrete definition: $TF(v, d) = DF(v) \cap CC$.
  2. For all $u \in TF(v, d)$, for all visible definitions "$d:\ x = \ldots$" in the basic-block $v$:
     if $u$ does not already have a $\tau$-function for $x$, insert a $\tau$-statement: $x = \tau(x)$ just
     after all $\phi$-statements (if any) at $u$, before any concrete statement.
```

5.2 Allocating τ-function Arguments

Before delving into the details of the algorithm, we take a slight digression into a deeper understanding of the ϕ and τ statements. We view a pseudo definition — not as a new definition — but as a label to an existing set of definitions, namely, the definitions corresponding to its argument set. So, when we talk of reaching definitions in this section, we would refer to all definitions (pseudo and concrete) that are not killed by a concrete definition; we do not allow pseudo definitions to kill an existing set of definitions. For example, in Figure 2, we would say that the definitions for x_9, x_{17}, and x_7 are the set of hot definitions that reach g; we call this set as the set of *active definitions* at g. In the SSA form, as each definition corresponds to a unique version of the variable, we use the terms *definition* and *variable version* interchangeably.

The algorithm, in essence, computes the path-sensitive *active* reaching definitions at each node u containing a τ-function. The hot reaching definitions (variable versions) stand as arguments in the τ-functions at u, each definition mapped to the set of hot paths along which it reaches u. A definition x_i that reaches u along the set of hot-paths ξ_i can be used as a parameter for a τ-function only if the following conditions are satisfied:

- if x_i is a concrete reaching definition: x_i can only be used as a parameter if $\xi_i \neq \emptyset$, i.e., it does reach u along a hot path;
- if x_i is a pseudo reaching definition: As discussed above, pseudo definitions are just labels to a set of concrete definitions. Even if $\xi_i \neq \emptyset$, not all concrete definitions *contained*[3] in x_i may be reaching u: In Figure 2, the pseudo-definition x_9 reaches g_1 along the hot paths $\xi_i = \{p_1, p_2\}$, i.e. $\xi_i \neq \emptyset$. However, if x_9 is used as parameter for the τ-function at g_1, it would invariably mean the inclusion of the definition x_{18}, which is not a hot reaching definition at g. Hence, a pseudo-definition can be used as an argument for some set of hot paths ξ if, and only if, *all the concrete reaching definitions that it merges reaches* u along ξ. This condition can be ensured by checking if all

[3] A definition for x_i is *contained* in a ϕ-definition if the ϕ-function argument-list either includes x_i, or includes a variable-version x_j such that x_i is *contained* in the definition for x_j; for the τ-functions, we only consider the speculative argument-list.

the *contained* concrete definitions for x_i are available as active definitions at u for the set of paths ξ.

Allowing definitions corresponding to pseudo-definitions in the τ-function argument list requires tracking of both pseudo and concrete definitions (which might appear along intersecting set of paths), while ensuring that a pseudo definition never kills a concrete definition, even along the same path. For the sake of simplicity, we abandon any further discussion on the same: in the following discussion, we ignore all pseudo definitions and maintain only the concrete definitions as active definitions (except if a pseudo-definition occurs as the only available reaching definition, or if a pseudo-definition is propagated along a backedge). As pseudo-definition "labels" to a set of merged definitions can no longer appear in the τ-function argument lists, the implication of ignoring the pseudo definitions is a larger argument list for the τ-functions.

Instead of performing an expensive classical path-sensitive dataflow analysis, we designed an algorithm very similar to the variable renaming phase of SSA construction [8] — using a variable stack to maintain the active definitions (or renamed variables) reaching each node. Our algorithm is defined as a recursive procedure running over the dominator tree of the control-flow graph. The variable stack maintains the set of active reaching definitions (x_i), along with the set of hot paths (ξ_i) that carry the definitions to the current node [4]. Our algorithm is more efficient than context-tupled classical path-sensitive dataflow analysis as it does not require storing of path-sensitive dataflow information at each basic-block.

Let P be the set of profiled acyclic path identifiers, and $DefPaths$ be the set of P. A *frame* in the variable stack is a map $[DefPaths \mapsto Version]$, where $Version$ is the renamed version of a variable; a frame can be seen as a set containing pairs $\{[\xi_1, x_1], [\xi_2, x_2], \ldots, [\xi_n, x_n]\}$, where $\xi_i \in DefPaths$. A variable stack $VarStack_x$ is a stack of frames for the base variable x.

$VarStack$ supports the following operations: `push(`ξ_i`:DefPaths,` x_i`:Version,` u:Basic-block) pushes a new frame with the association $[\xi_i, x_i]$ on $VarStack_x$; `pop(u:Basic-block)` pops off all frames that were pushed in the basic-block u; and `top()` returns the topmost frame on the stack.

A $Frame$ in $VarStack$ supports the following operations: `get(`ξ`:DefPaths)` returns the version associated with ξ in the map; `accumulate(`ξ_i`:DefPaths,` x_i`:Version)` accumulates definitions: if a pair $[\xi_j, x_i] \in Frame$, replace $[\xi_j, x_i]$ by $[\xi_j \cup \xi_i, x_i]$, else add a new association $[\xi_i, x_i]$ to the frame.

The top of the variable stack contains the set of active definitions — definitions that can be used to allocate arguments to the τ-functions in the current basic-block. The algorithm traverses the control-flow graph recursively in a depth-first order over the dominator tree (as does the variable renaming phase for SSA construction); the set of dominatees[5] are traversed in the topological order of

[4] The updates to ξ_i is done lazily; so a certain points, they may contain more paths than the actual set of hot reaching paths.

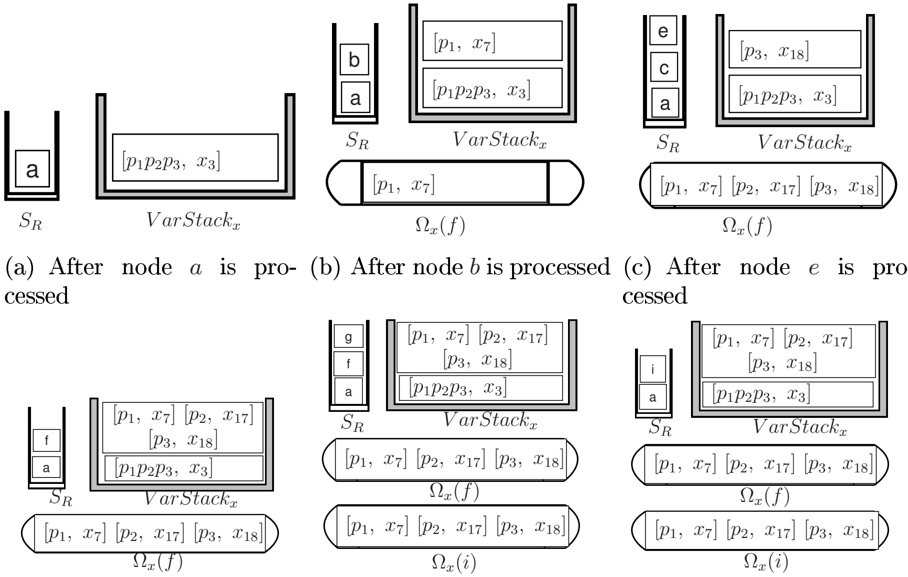
[5] The children of a node n in the dominator tree are the dominatees of n.

the nodes in the control-flow graph: the order is important to ensure that when a basic-block is processed, the definitions from all its incoming paths reach it. The active definitions are propagated via $VarStack$ from a parent node to its children in the dominator tree; for a join node u, the active definitions are accumulated (by a similar operation as `accumulate(`ξ_i`:DefPaths,` x_i`:Version)` for a frame) in a *Definition Accumulator* $\Omega_x(u)$ from its predecessors in the CFG — it is loaded up on $VarStack$ when the node u is processed.

The τ-allocation algorithm is sketched in Algorithm 3. Let us describe the algorithm via an example (Figure 5) for the flow-graph in Figure 2:

Let the basic-blocks be processed in the order a, b, c, d, e, f, g, h, i.

The basic-block a is processed foremost: the algorithm (Step 3(c)) pushes the definition x_3 on $VarStack_x$ (Figure (a)), and then recurses on the children of a in the dominator tree, namely b, c and f (Step 5). At the node b, the algorithm (Step 3(c)) pushes the definition x_7 on the stack; its successor node, f, turns out to be a join node: hence, the algorithm (Step 4) accumulates the definitions in the topmost frame of the stack into the (currently empty) definition accumulator $\Omega_x(f)$ (Figure (b)). As b has no children in the dominator tree, the algorithm (Step 6) retraces the recursive path to node a, popping off the definition pushed by b in the process. The variable stack and the recursion stack (S_R) now again resemble that in Figure (a).


a
S_R
$[p_1p_2p_3,\ x_3]$
$VarStack_x$
b
a
S_R
$[p_1,\ x_7]$
$[p_1p_2p_3,\ x_3]$
$VarStack_x$
$[p_1,\ x_7]$
$\Omega_x(f)$
e
c
a
S_R
$[p_3,\ x_{18}]$
$[p_1p_2p_3,\ x_3]$
$VarStack_x$
$[p_1,\ x_7]\ [p_2,\ x_{17}]\ [p_3,\ x_{18}]$
$\Omega_x(f)$
f
a
S_R
$[p_1,\ x_7]\ [p_2,\ x_{17}]\ [p_3,\ x_{18}]$
$[p_1p_2p_3,\ x_3]$
$VarStack_x$
$[p_1,\ x_7]\ [p_2,\ x_{17}]\ [p_3,\ x_{18}]$
$\Omega_x(f)$
g
f
a
S_R
$[p_1,\ x_7]\ [p_2,\ x_{17}]\ [p_3,\ x_{18}]$
$[p_1p_2p_3,\ x_3]$
$VarStack_x$
$[p_1,\ x_7]\ [p_2,\ x_{17}]\ [p_3,\ x_{18}]$
$\Omega_x(f)$
$[p_1,\ x_7]\ [p_2,\ x_{17}]\ [p_3,\ x_{18}]$
$\Omega_x(i)$
i
a
S_R
$[p_1,\ x_7]\ [p_2,\ x_{17}]\ [p_3,\ x_{18}]$
$[p_1p_2p_3,\ x_3]$
$VarStack_x$
$[p_1,\ x_7]\ [p_2,\ x_{17}]\ [p_3,\ x_{18}]$
$\Omega_x(f)$
$[p_1,\ x_7]\ [p_2,\ x_{17}]\ [p_3,\ x_{18}]$
$\Omega_x(i)$

(a) After node a is processed

(b) After node b is processed

(c) After node e is processed

(d) After node f is processed

(e) After node g is processed

(f) After node i is processed

Fig. 5. Steps in the execution of the τ-argument allocation algorithm

Algorithm 3. A sketch of the τ-function argument allocation algorithm

Process a basic-block u in the following manner:

1. Push the Definition Accumulator $\Omega(u)$ on $VarStack$ (if $\Omega(u)$ exists).
2. If u is the incubation node for a set of hot paths, for all base-variables x which do not have a ϕ-definition appearing in the basic-block u, push a frame $\langle \xi_i, x_i \rangle$, where ξ_i is the set of all paths that incubate from u, and x_i is the meet-over-all paths reaching definition (variable-version) for x at u.
3. Process each statement stm in the basic-block:
   (a) If stm is a ϕ-statement: if u is a loop-header and the dummy profile edge $t \rightarrow \delta_{end}$ is hot (where δ_{end} denotes the dummy-end node for a Ball-Larus profiler, and t is the corresponding loop-tail), accumulate $\langle \xi_i, x_i \rangle$ at the topmost frame of $VarStack_x$, where
      i. ξ_i is the set of all paths that incubate from u, and
      ii. x_i is the ϕ-statement argument corresponding to the backedge $t \rightarrow u$.
   (b) If stm is a τ-statement:
      i. Create a set C of candidate definitions from the definitions in $VarStack.top()$ for each incubation node s: add $\langle \xi_i, x_i \rangle$ to C iff $(Paths_s(u) \cap \xi_i) \neq \emptyset$;
      ii. If there exists at least one $x_i \in C$ such that its variable-version differs from the safe argument x_0, add arguments to the τ-function for each x_i, mapping the respective variable position to ξ_i; otherwise, replace the τ-function with a simple copy statement: $x_{out} = x_0$.
   (c) Update VarStack to include new definitions in the basic-block u:
      – Concrete definition: Push the definition as a new frame associating it with $Paths(u)$;
      – Pseudo definition: Ignore.
4. Save the active definitions in Ω of the (forward) successors (if successor is a join node): for each forward (ignore backedges) successor edge $u \rightarrow v$, if v is a join node, for each $\langle \xi_i, x_i \rangle \in VarStack.top()$ such that $(\xi_i \cap Paths(u \rightarrow v)) \neq \emptyset$, accumulate $\langle \xi_i \cap Paths(u \rightarrow v), x_i \rangle$ in Ω_x.
5. Recurse on the children of u in the dominator tree in accordance to their topological order in the control flow graph.
6. Pop off all frames pushed by u from VarStack.

The nodes c, d, and e are processed similarly; Figure (c) shows the state of the data-structures just after node e is processed. After handling e, the recursion is unwound to node a.

The algorithm then picks the node f: it first pushes the definition accumulator of f, $\Omega_x(f)$, on the variable stack (Step 1); on encountering the ϕ-definition for x_9, it simply ignores the same (Step 3(c)). Finally, it recurses on the immediate dominatees of f, viz. g and h (Step 5).

The node g is processed next: on encountering the τ-definition for x_{11}, the algorithm (Step 3(b)) attempts to allocate arguments for the same: Examining the active definitions (top of the variable stack), the algorithm attempts to assemble the candidate set C — a subset of definitions from the topmost frame of $VarStack_x$ that, together, can map to all the hot paths passing through g.

The set of active definitions at g turn out to be $\{[p_1, x_7], [p_2, x_{17}], [p_3, x_{18}]\}$. To be added to C, the path-component in the definition pairs must intersect with $Paths(g) = \{p_1, p_2\}$; $[p_1, x_7]$ and $[p_2, x_{17}]$ satisfy the condition, while $[p_3, x_{18}]$ does not. Notice how the cold definitions are pruned are from the possible set of definitions to be added as arguments to the τ-function.

As the variable versions in the set C differ from that of the safe argument, we allocate arguments to the τ-function from C.

$$\boxed{x_{11} = \tau(x_9,\ \ x_7 \langle p_1 \rangle\,,\ \ x_{17} \langle p_2 \rangle)}$$

The algorithm then accumulates the active definitions in $\Omega_x(i)$ (Figure (e)). The nodes h, and then i are processed in order in a similar manner.

Note that the set of candidate definitions C for a τ-function at a node v contains the exact set of hot definitions that reach v. Additionally, for each pair $\langle \xi_i, x_i \rangle \in C$, x_i reaches u along the paths in ξ_i, and along no other hot path.

Now consider the control-flow graph with loops (Figure 3): Let us illustrate as to how the the hot reaching definition of i_3 in the block c is identified as a hot reaching definition at the τ-function in the node e even though we use acyclic path-profiles. As the loop-path p_1 is hot, when the node b is processed, the definition-pair $\langle p_1 p_2, i_3 \rangle$ is added to the top of the variable stack (being the parameter to the ϕ-function corresponding to the backedge) by Step 3(a). When the algorithm recurses on the children of d in the dominator tree, the variable stack carries the definition to the basic-block e where it is recognised as an argument for the τ-function along the path p_2. The Step 2 in the algorithm is required to carry the meet-over-all-paths definition n_1 from the node a to the node g, as there does not exist any acyclic hot path from a to g.

6 Speculative Sparse Conditional Constant Propagation

We have implemented the analysis phase of a novel optimization — the Speculative Sparse Conditional Constant Propagation (SSCP) on the HPSSA form. This optimization expands the scope of the SCP [21] algorithm — allowing it to identify speculative constants (expressions that are highly likely to be constants) — along with the conventional "safe" constants (expressions that are guaranteed to be constants).

This section is more than a description of a new analysis — through this novel analysis, we essentially aim to demonstrate how new speculative optimizations can be developed on the HPSSA form by simple extensions of existing "safe" SSA-based optimizations.

The SSCP algorithm operates on a four level lattice (Figure 6 shows the SSCP lattice for integers): the conventional constant propagation lattice is extended by another layer — that of speculative constants (indicated by the constants superscripted with 's'). The speculative constants can be seen as constant values with exactly the same properties as that of ordinary constants — just marked "speculative" — indicating that they are *predicted* values, not guaranteed to hold under all executions.

The transfer functions of all existing operations (including that of the ϕ-function) hold as in SCP, except for the fact that if any operand in an expression turns out to be a speculative constant, the result of the operation, if a constant, would be a *speculative* constant carrying the respective constant value. For example, $2 + 3^s$ would render the speculative constant 5^s.

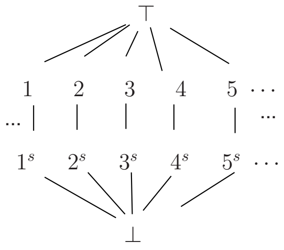


Fig. 6. The SSCP Lattice (the constants superscripted by 's' are the speculative constants)

```
i0=0;
while(...) {
    i1=φ(i0, i3);
    i2=τ(i1, i3);
    i3=i2 + 1;
}
```

Fig. 7. A case that requires meet with its old value in the τ-statement transfer function for SSCP

The transfer function for the τ-functions is defined as follows (where $\sqcap$ is the meet operator): If the meet of all the arguments does not produce $\perp$ (not-constant), the transfer function resembles the transfer function for the ϕ-functions. Even if the meet of all the arguments turns out to be $\perp$, there might still be the chance of the expression being identified as a speculative constant: let $\beta = x_1 \sqcap x_2 \ldots \sqcap x_n$. The transfer function attempts to return β, if β is $\perp$, $\top$ or a speculative constant; if β is a "safe" constant, β moves in the lattice to $(\beta)^s$, the corresponding speculative constant. Formally, the transfer function for $\tau(x_0, x1, \ldots, x_n)$ is given by the following (where each expression refers its abstract value in the lattice, and $\beta = x_1 \sqcap x_2 \ldots \sqcap x_n$):

$$\tau(x_0, x1, \ldots, x_n) \sqcap \begin{cases} x_0 \sqcap \beta & \text{if } x_0 \sqcap \beta \neq \perp \\ \beta & \text{if } x_0 \sqcap \beta = \perp \text{ and } \beta \text{ is not a safe constant} \\ (\beta)^s & \text{otherwise} \end{cases}$$

The meet with the current value of the τ-function is added to ensure termination by ensuring monotonicity; otherwise, code fragments resembling that in Figure 7 will never reach a fixpoint due to i_3 increasing its value in the speculative domain, and the τ-function feeding the same value back to it. We omit detailed discussions on this analysis for want of space.

7 Implementation and Experiments

We implemented our HPSSA construction algorithm, as well as the analysis phase of the SSCP algorithm on the Scale compiler [18]; we were also aided by the CIL [7] tool. We only cast scaler variables whose address has not been taken in the HPSSA form; τ-functions are not introduced for the remaining variables. The SSCP algorithm implementation handles only integer variables; the implementation is interprocedural but context-insensitive; function pointers are ignored (it flags a warning, computing a possibly unsafe solution).

We tested our implementation on some programs from the SPEC2000 benchmark suite. We used a naive hot path selection criteria: all the acyclic paths executed on the *train* input set was considered "hot" for building the HPSSA

Table 1. Speculative Constants discovered by the SSCP algorithm. (’˜’ indicates *almost*; `grp`, `prg`, & `src` refer to inputs `graphic`, `program` & `source` respectively).

		Variable Uses		Expression Uses		Total		
Program	**Inpt**	**Uses**	**HitRt**	**Uses**	**HitRt**	**Hits**	**Misses**	**HitRt**
181.mcf	-	33110	100.00	49665	100.00	82775	0	100.00
175.vpr	-	6938074	100.00	8110837	100.00	15048911	0	100.00
164.gzip	grp	26592	100.00	5	100.00	26597	0	100.00
	prg	17412	100.00	5	100.00	17417	0	100.00
	src	4721	99.98	5	100.00	4725	1	99.98
197.parser	-	165970964	˜100.00	340	97.94	165970861	443	˜100.00
256.bzip2	grp	132106650	˜100.00	938	76.97	132107372	216	˜100.00
	prg	100819492	˜100.00	6576416	15.67	101849942	5545966	94.84
	src	108134316	˜100.00	5256006	17.94	109077366	4312956	96.20

form. Table 1 exhibits our findings for programs run on the *ref* input set. The programs were run with the default parameters, i.e., no parameters were set on the command line, either for training, or for the actual run (on the *ref* set). We collected statistics for *dynamic uses* (use of a variable/expression during the actual run) for variables (*Variable Uses*), and for sub-expressions that could be constant-folded speculatively (*Expression Uses*). The uses are tabulated only for the speculative constants — uses that are likely (but not guaranteed) to be constants. We have not shown the number of “sure” constants as it would be same as that for the original SCP algorithm. We also indicate the *Hit Rate* (*HitRt*): the percentage of uses where the use of variable/expression actually agrees with the “predicted” speculative constant value.

The programs seem to enshroud plenitude opportunities for an optimizer adept at performing speculative program transformations. Most of the programs show a large number of dynamic speculative usages with good hit rates (except *256.bzip2* for the sub-expression uses; still the overall hit rate turns out high, courtesy the variable usages). A more intelligent hot path selection scheme may be able to reap more constants, though it may also have an effect on the hit-rate; we are interested in experimenting with alternative schemes in the future.

8 Related Work

Multitude of interesting extensions and modifications have been proposed on the SSA form. The Hashed SSA (HSSA) form [6] extends the traditional SSA form to accommodate pointer variables by introducing an explicit may modify operator (χ) and may reference operator (μ). The Array SSA [13] form captures element-level data flow information of array variables. The ψ-SSA form [19] simplifies the use of SSA-based optimizations on predicated code. Though we have not addressed aliasing and arrays in this paper, it does not seem difficult to address these issues in the HPSSA form; we may investigate such extensions via concrete implementations in the future.

Lin et al. [14] proposed a speculative SSA form by extending speculative versions of the HSSA operators — speculative update (χ_s) and speculative use (μ_s). The speculative flag, either by use of profiling information and/or a set of heuristic rules, is turned on these operators if it is highly likely that an update or reference will be substantiated at runtime. Lin et al.'s work is orthogonal to our work as we target exposing the hot use-def chains rather than likely alias relations; both these techniques can be seamlessly combined for a more powerful speculative optimization framework.

Towards path-sensitive program optimizations, Ammons and Larus [1] proposed performing flow-analysis on a *hot path graph* that isolates the frequent paths. Das et al. [10] proposed a polynomial-time path-sensitive algorithm for verifying a given temporal safety property, and proved it effective by verifying the file I/O behaviour of a version of the GNU C Compiler.

Researchers have also been interested in inferring *likely* data-flow facts, computed over control-flow profiles. Ramalingam [16] used edge-profiles to infer the probability with which a fact holds true for the class of finite bi-distributive subset problems. Probabilistic pointer analyses [4,9] assign probabilities with which a points-to relation might hold at a program point. Contributions to speculative partial redundancy elimination have been made by [15,22]. Path profile based speculative PRE and PDE have been proposed by [11,12]. Most of these techniques use edge and node profiles which are much weaker than path-profiles used by HPSSA. Also, the HPSSA form provides a common ground for writing efficient optimizations on a sparse program representation; it scores over flow-based speculative optimizations due to the exact reason that the SSA-based algorithms score over the flow-based safe optimizations.

9 Conclusions

We propose a novel extension to the highly successful SSA form, and demonstrate — by an analysis algorithm for Speculative Sparse Conditional Constant Propagation — that novel speculative optimizations can be enabled on the HPSSA form by almost obvious modifications of existing SSA-based traditional optimizations. We are pondering over the design of speculative versions of other existing SSA-based traditional optimizations — Global Value Numbering [3] and Partial Redundancy Elimination [5] being our foremost targets. We are also interested in extending the HPSSA form for richer profiles like the k-iteration [17] profiles.

Acknowledgements. Subhajit Roy was supported by Doctoral Fellowship from Philips Research, India.

References

1. Ammons, G., Larus, J.R.: Improving data-flow analysis with path profiles. SIGPLAN Not. 39(4), 568–582 (2004)
2. Ball, T., Larus, J.R.: Efficient path profiling. In: International Symposium on Microarchitecture (MICRO), pp. 46–57 (1996)

3. Briggs, P., Cooper, K.D., Taylor Simpson, L.: Value Numbering. Software: Practice and Experience (1997)
4. 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)
5. Chow, F., Chan, S., Kennedy, R., Liu, S.-M., Lo, R., Tu, P.: A new algorithm for partial redundancy elimination based on SSA form. In: Programming Language Design and Implementation (PLDI), pp. 273–286 (1997)
6. 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)
7. CIL - Infrastructure for C Program Analysis and Transformation, `http://hal.cs.berkeley.edu/cil/`
8. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Kenneth Zadeck, F.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13(4), 451–490 (1991)
9. Da Silva, J., Gregory Steffan, J.: A probabilistic pointer analysis for speculative optimizations. SIGARCH Comput. Archit. News 34(5), 416–425 (2006)
10. Das, M., Lerner, S., Seigle, M.: ESP: path-sensitive program verification in polynomial time. In: Programming Language Design and Implementation (PLDI), pp. 57–68 (2002)
11. Gupta, R., Berson, D.A., Fang, J.Z.: Path Profile Guided Partial Dead Code Elimination Using Predication. In: Parallel Architectures and Compilation Techniques (PACT), p. 102. IEEE Computer Society, Washington, DC (1997)
12. Gupta, R., Berson, D.A., Fang, J.Z.: Path Profile Guided Partial Redundancy Elimination Using Speculation. In: International Conference on Computer Languages (ICCL), p. 230 (1998)
13. Knobe, K., Sarkar, V.: Array SSA form and its use in parallelization. In: Principles of Programming Languages (POPL), pp. 107–120 (1998)
14. Lin, J., Chen, T., Hsu, W.-C., Yew, P.-C., Ju, R.D.-C., Ngai, T.-F., Chan, S.: A compiler framework for speculative analysis and optimizations. In: Programming Language Design and Implementation (PLDI), pp. 289–299 (2003)
15. Horspool, R.N., Pereira, D.J., Scholz, B.: Fast Profile-Based Partial Redundancy Elimination. In: Lightfoot, D.E., Szyperski, C. (eds.) JMLC 2006. LNCS, vol. 4228, pp. 362–376. Springer, Heidelberg (2006)
16. Ramalingam, G.: Data flow frequency analysis. In: Programming Language Design and Implementation (PLDI), pp. 267–277 (1996)
17. Roy, S., Srikant, Y.N.: Profiling k-Iteration Paths: A Generalization of the Ball-Larus Profiling Algorithm. In: International Symposium on Code Generation and Optimization (CGO), pp. 70–80. IEEE Computer Society, Washington, DC (2009)
18. Scale: A Scalable Compiler for Analytical Experiments, `http://www-ali.cs.umass.edu/Scale/`
19. Stoutchinin, A., de Ferriere, F.: Efficient static single assignment form for predication. In: International Symposium on Microarchitecture (MICRO), pp. 172–181 (2001)
20. Tallam, S., Zhang, X., Gupta, R.: Extending path profiling across loop backedges and procedure boundaries. In: International Symposium on Code Generation and Optimization (CGO), pp. 251–264 (2004)
21. Wegman, M.N., Kenneth Zadeck, F.: Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. 13(2), 181–210 (1991)
22. Xue, J., Cai, Q.: A lifetime optimal algorithm for speculative PRE. ACM Trans. Archit. Code Optim. 3(2), 115–155 (2006)

Author Index