LNCS 7791

ARCoSS

Ranjit Jhala
Koen De Bosschere (Eds.)

Compiler Construction

22nd International Conference, CC 2013
Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013
Rome, Italy, March 2013, Proceedings

ETAPS
EUROPEAN JOINT CONFERENCES ON THEORY & PRACTICE OF SOFTWARE

Springer

Ranjit Jhala Koen De Bosschere (Eds.)

Compiler Construction

22nd International Conference, CC 2013
Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013
Rome, Italy, March 16-24, 2013
Proceedings

Springer

Volume Editors

Ranjit Jhala
University of California, San Diego
3110 Computer Science and Engineering
9500 Gilman Drive, La Jolla, CA 92093-0404, USA
E-mail: jhala@cs.ucsd.edu

Koen De Bosschere
Ghent University
System Software Lab.
Sint Pietersnieuwstraat 41, 9000 Ghent, Belgium
E-mail: koen.debosschere@elis.ugent.be

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-37050-2 e-ISBN 978-3-642-37051-9
DOI 10.1007/978-3-642-37051-9
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013932567

CR Subject Classification (1998): D.3.4, D.2.5-7, D.3.2, D.4.2, F.3.2, B.1.4, I.2.6

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

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

Printed on acid-free paper

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

Foreword

ETAPS 2013 is the sixteenth 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 six sister conferences (CC, ESOP, FASE, FOSSACS, POST, TACAS), 20 satellite workshops (ACCAT, AiSOS, BX, BYTECODE, CerCo, DICE, FESCA, GRAPHITE, GT-VMT, HAS, Hot-Spot, FSS, MBT, MEALS, MLQA, PLACES, QAPL, SR, TERMGRAPH and VSSE), three invited tutorials (*e-education*, by John Mitchell; *cyber-physical systems*, by Martin Fränzle; and *e-voting* by Rolf Küsters) and eight invited lectures (excluding those specific to the satellite events).

The six main conferences received this year 627 submissions (including 18 tool demonstration papers), 153 of which were accepted (6 tool demos), giving an overall acceptance rate just above 24%. (ETAPS 2013 also received 11 submissions to the software competition, and 10 of them resulted in short papers in the TACAS proceedings). 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 to participate in this exciting event, and that you will all continue to submit to ETAPS and contribute to making 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, security and improvement. The languages, methodologies and tools that support these activities are all well within its scope. Different blends of theory and practice are represented, with an inclination towards theory with a practical motivation on the one hand and soundly based practice on the other. Many of the issues involved in software design apply to systems in general, including hardware systems, and the emphasis on software is not intended to be exclusive.

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

ETAPS 2013 was organised by the *Department of Computer Science of ‘Sapienza’ University of Rome*, 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).

The organising team comprised:

General Chair:	*Daniele Gorla;*
Conferences:	*Francesco Parisi Presicce;*
Satellite Events:	*Paolo Bottoni* and *Pietro Cenciarelli;*
Web Master:	*Igor Melatti;*
Publicity:	*Ivano Salvo;*
Treasurers:	*Federico Mari* and *Enrico Tronci.*

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

Vladimiro Sassone (Southampton, chair), Martín Abadi (Santa Cruz), Erika Ábrahám (Aachen), Roberto Amadio (Paris 7), Gilles Barthe (IMDEA-Software), David Basin (Zürich), Saddek Bensalem (Grenoble), Michael O'Boyle (Edinburgh), Giuseppe Castagna (CNRS Paris), Albert Cohen (Paris), Vittorio Cortellessa (L'Aquila), Koen De Bosschere (Gent), Ranjit Jhala (San Diego), Matthias Felleisen (Boston), Philippa Gardner (Imperial College London), Stefania Gnesi (Pisa), Andrew D. Gordon (MSR Cambridge and Edinburgh), Daniele Gorla (Rome), Klaus Havelund (JLP NASA Pasadena), Reiko Heckel (Leicester), Holger Hermanns (Saarbrücken), Joost-Pieter Katoen (Aachen), Paul Klint (Amsterdam), Jens Knoop (Vienna), Steve Kremer (Nancy), Gerald Lüttgen (Bamberg), Tiziana Margaria (Potsdam), Fabio Martinelli (Pisa), John Mitchell (Stanford), Anca Muscholl (Bordeaux), Catuscia Palamidessi (INRIA Paris), Frank Pfenning (Pittsburgh), Nir Piterman (Leicester), Arend Rensink (Twente), Don Sannella (Edinburgh), Zhong Shao (Yale), Scott A. Smolka (Stony Brook), Gabriele Taentzer (Marburg), Tarmo Uustalu (Tallinn), Dániel Varró (Budapest) and Lenore Zuck (Chicago).

The ordinary running of ETAPS is handled by its management group comprising: Vladimiro Sassone (chair), Joost-Pieter Katoen (deputy chair and publicity chair), Gerald Lüttgen (treasurer), Giuseppe Castagna (satellite events chair), Holger Hermanns (liaison with local organiser) and Gilles Barthe (industry liaison).

I would like to express here my sincere gratitude to all the people and organisations that contributed to ETAPS 2013, 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-Verlag for agreeing to publish the ETAPS proceedings in the ARCoSS subline.

Last but not least, I would like to thank the organising chair of ETAPS 2013, Daniele Gorla, and his Organising Committee, for arranging for us to have ETAPS in the most beautiful and historic city of Rome.

My thoughts today are with two special people, profoundly different for style and personality, yet profoundly similar for the love and dedication to our discipline, for the way they shaped their respective research fields, and for the admiration and respect that their work commands. Both are role-model computer scientists for us all.

ETAPS in Rome celebrates *Corrado Böhm.* Corrado turns 90 this year, and we are just so lucky to have the chance to celebrate the event in Rome, where he has worked since 1974 and established a world-renowned school of computer scientists. Corrado has been a pioneer in research on programming languages and their semantics. Back in 1951, years before FORTRAN and LISP, he defined and implemented a *metacircular compiler* for a programming language of his invention. The compiler consisted of just 114 instructions, and anticipated some modern list-processing techniques.

Yet, Corrado's claim to fame is asserted through the breakthroughs expressed by the *Böhm-Jacopini Theorem* (CACM 1966) and by the invention of *Böhm-trees.* The former states that any algorithm can be implemented using only sequencing, conditionals, and while-loops over elementary instructions. Böhm trees arose as a convenient data structure in Corrado's milestone proof of the decidability inside the λ-calculus of the equivalence of terms in β-η-normal form.

Throughout his career, Corrado showed exceptional commitment to his roles of researcher and educator, fascinating his students with his creativity, passion and curiosity in research. Everybody who has worked with him or studied under his supervision agrees that he combines an outstanding technical ability and originality of thought with great personal charm, sweetness and kindness. This is an unusual combination in problem-solvers of such a high calibre, yet another reason why we are ecstatic to celebrate him. *Happy birthday from ETAPS, Corrado!*

ETAPS in Rome also celebrates the life and work of *Kohei Honda.* Kohei passed away suddenly and prematurely on December 4th, 2012, leaving the saddest gap in our community. He was a dedicated, passionate, enthusiastic scientist and –more than that!– his enthusiasm was contagious. Kohei was one of the few theoreticians I met who really succeeded in building bridges between theoreticians and practitioners. He worked with W3C on the standardisation of web services choreography description languages (WS-CDL) and with several companies on *Savara* and *Scribble*, his own language for the description of application-level protocols among communicating systems.

Among Kohei's milestone research, I would like to mention his 1991 epoch-making paper at ECOOP (with M. Tokoro) on the treatment of asynchrony in message passing calculi, which has influenced all process calculi research since. At ETAPS 1998 he introduced (with V. Vasconcelos and M. Kubo) a new concept in type theories for communicating processes: it came to be known as '*session types*,' and has since spawned an entire research area, with practical and multi-disciplinary applications that Kohei was just starting to explore.

Kohei leaves behind him enormous impact, and a lasting legacy. He is irreplaceable, and I for one am proud to have been his colleague and glad for the opportunity to arrange for his commemoration at ETAPS 2013.

My final ETAPS '*Foreword*' seems like a good place for a short reflection on ETAPS, what it has achieved in the past few years, and what the future might have in store for it.

On April 1st, 2011 in Saarbrücken, we took a significant step towards the consolidation of ETAPS: the establishment of *ETAPS e.V.* This is a *non-profit association* founded under German law with the immediate purpose of supporting the conference and the related activities. ETAPS e.V. was required for practical reasons, e.g., the conference needed (to be represented by) a legal body to better support authors, organisers and attendees by, e.g., signing contracts with service providers such as publishers and professional meeting organisers. Our ambition is however to make of '*ETAPS the association*' more than just the organisers of '*ETAPS the conference*'. We are working towards finding a voice and developing a range of activities to support our scientific community, in cooperation with the relevant existing associations, learned societies and interest groups. The process of defining the structure, scope and strategy of ETAPS e.V. is underway, as is its first ever membership campaign. For the time being, ETAPS e.V. has started to support community-driven initiatives such as open access publications (LMCS and EPTCS) and conference management systems (Easychair), and to cooperate with cognate associations (European Forum for ICT).

After two successful runs, we continue to support POST, *Principles of Security and Trust*, as a candidate to become a permanent ETAPS conference. POST was the first addition to our main programme since 1998, when the original five conferences met together in Lisbon for the first ETAPS. POST resulted from several smaller workshops and informal gatherings, supported by IFIP WG 1.7, and combines the practically important subject of security and trust with strong technical connections to traditional ETAPS areas. POST is now attracting interest and support from prominent scientists who have accepted to serve as PC chairs, invited speakers and tutorialists. I am very happy about the decision we made to create and promote POST, and to invite it to be a part of ETAPS.

Considerable attention was recently devoted to our *internal processes* in order to streamline our procedures for appointing Programme Committees, choosing invited speakers, awarding prizes and selecting papers; to strengthen each member conference's own Steering Group, and, at the same time, to strike a balance between these and the ETAPS Steering Committee. A lot was done and a lot remains to be done.

We produced a *handbook* for local organisers and one for PC chairs. The latter sets out a code of conduct that all the people involved in the selection of papers, from PC chairs to referees, are expected to adhere to. From the point of view of the authors, we adopted a *two-phase submission* protocol, with fixed deadlines in the first week of October. We published a *confidentiality policy* to set high standards for the handling of submissions, and a *republication policy* to clarify what kind of material remains eligible for submission to ETAPS after presentation at a workshop. We started an *author rebuttal phase*, adopted by most of the conferences, to improve the author experience. It is important to acknowledge that – regardless of our best intentions and efforts – the quality of reviews is not always what we would like it to be. To remain true to our commitment to the authors who elect to submit to ETAPS, we must endeavour to improve our standards of refereeing. The rebuttal phase is a step in that direction and, according to our experience, it seems to work remarkably well at little cost, provided both authors and PC members use it for what it is. ETAPS has now reached a healthy paper acceptance rate around the 25% mark, essentially uniformly across the six conferences. This seems to me to strike an excellent balance between being selective and being inclusive, and I hope it will be possible to maintain it even if the number of submissions increases.

ETAPS signed a favourable three-year publication contract with Springer for publication in the ARCoSS subline of LNCS. This was the result of lengthy negotiations, and I consider it a good achievement for ETAPS. Yet, publication of its proceedings is possibly the hardest challenge that ETAPS – and indeed most computing conferences – currently face. I was invited to represent ETAPS at a most interesting Dagstuhl Perspective Workshop on the '*Publication Culture in Computing Research*' (seminar 12452). The paper I gave there is available online from the workshop proceedings, and illustrates three of the views I formed also thanks to my experience as chair of ETAPS, respectively on open access, bibliometrics, and the roles and relative merits of conferences versus journal publications. Open access is a key issue for a conference like ETAPS. Yet, in my view it does not follow that we can altogether dispense with publishers – be they commercial, academic, or learned societies – and with their costs. A promising way forward may be based on the '*author-pays*' model, where publications fees are kept low by resorting to learned-societies as publishers. Also, I believe it is ultimately in the interest of our community to de-emphasise the perceived value of conference publications as viable – if not altogether superior – alternatives to journals. A large and ambitious conference like ETAPS ought to be able to rely on quality open-access journals to cover its entire spectrum of interests, even if that means promoting the creation of a new journal.

Due to its size and the complexity of its programme, hosting ETAPS is an increasingly challenging task. Even though excellent candidate *locations* keep being volunteered, in the longer run it seems advisable for ETAPS to provide more support to local organisers, starting e.g., by taking direct control of the organisation of satellite events. Also, after sixteen splendid years, this may be a good time to start thinking about exporting ETAPS to other continents. The US East Coast would appear to be the obvious destination for a first ETAPS outside Europe.

The strength and success of ETAPS comes also from presenting – regardless of the natural internal differences – a homogeneous interface to authors and participants, i.e., to look like one large, coherent, well-integrated conference rather than a mere co-location of events. I therefore feel it is vital for ETAPS to regulate the centrifugal forces that arise naturally in a 'union' like ours, as well as the legitimate aspiration of individual PC chairs to run things their way. In this respect, we have large and solid foundations, alongside a few relevant issues on which ETAPS has not yet found agreement. They include, e.g., submission by PC members, rotation of PC memberships, and the adoption of a rebuttal phase. More work is required on these and similar matters.

January 2013

Vladimiro Sassone
ETAPS SC Chair
ETAPS e.V. President

Preface

This volume contains the proceedings of the 22nd International Conference on Compiler Construction (CC) held during March 21–22, 2013, in Rome, Italy as part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2013).

CC is a forum for the presentation of the latest research on processing *programs* in the most general sense, that is, in analyzing, transforming, or executing any *description* of how a computing system operates, which includes traditional compiler construction as a special case. Papers were solicited on a broad range of topics, including compilation and interpretation techniques, including program representation and analysis, code generation and code optimization; run-time techniques, including memory management and dynamic and just-in-time compilation; programming tools, from refactoring editors to checkers to compilers to virtual machines to debuggers; techniques for specific domains, such as secure, parallel, distributed, embedded or mobile environments; design of novel language constructs and their implementation.

This year, 53 papers were submitted to CC. Each submission was reviewed by three or more Program Committee members. After evaluating the quality and pertinence of each paper, the committee chose to accept 13 papers for presentation at the conference. This year's program also included an invited talk by a distinguished researcher:

– Emery Berger (University of Massachusetts, Amherst) on "Programming with People: Integrating Human-Based and Digital Computation"

The success of the conference is due to the unstinting efforts of several people. First, the authors for carrying out and submitting high-quality research to CC. Second, the Program Committee and subreviewers for their perspicacious and timely reviews, which ensured the high quality of the proceedings. Third, the ETAPS organizers and Steering Committee for hosting ETAPS and for the local co-ordination and organization. Finally, we are grateful to Andrei Voronkov whose EasyChair system eased the processes of submission, paper selection, and proceedings compilation.

January 2013

Koen De Bosschere
Ranjit Jhala

Organization

Program Chairs

Koen De Bosschere	Ghent University, Belgium
Ranjit Jhala	University of California, San Diego, USA

Program Committee

Gogul Balakrishnan	NEC, Princeton, USA
Francois Bodin	CAPS entreprise, France
Albert Cohen	École Normale Supérieure, Paris, France
Koen De Bosschere	Ghent University, Belgium
Bjöorn Franke	University of Edinburgh, UK
Grigori Fursin	INRIA, France
Mary Hall	University of Utah, USA
Ben Hardekopf	University of California, Santa Barbara, USA
Ranjit Jhala	University of California, San Diego, USA
Christian Lengauer	University of Passau, Germany
Matt Might	University of Utah, USA
Alan Mycroft	Cambridge University, UK
George Necula	University of California, Berkeley, USA
Keshav Pingali	University of Texas, Austin, USA
Jurgen Vinju	CWI Amsterdam, The Netherlands
Eelco Visser	TU Delft, The Netherlands
Greta Yorsh	ARM, UK
Ben Zorn	Microsoft Research, USA

Additional Reviewers

Aldous, Peter
Baranga, Silviu
Basten, Bas
Basu, Protonu
Berdine, Josh
Calvert, Peter
Chen, Juan
Dolan, Stephen
Fahndrich, Manuel
Gilray, Thomas
Gould, Miles
Groesslinger, Armin
Hills, Mark
Khoo, Wei Ming
Klint, Paul
Koestler, Harald

Lyde, Steven
Ma, Kin-Keung
Maleki, Saeed
Margiolas, Christos
McPeak, Scott
Mytkowicz, Todd
Nguyen, Donald
Orchard, Dominic
Pasteur, Cédric
Petricek, Tomas
Petter, Michael
Powell, Daniel
Proust, Raphael
Reames, Philip
Regis-Gianas, Yann
Renggli, Lukas
Sergey, Ilya
Simbuerger, Andreas
van der Storm, Tijs
Venkat, Anand
Voigt, Janina
Wei, Tao
Westbrook, Edwin
Yi, Qing
Zhang, Yingzhou

Programming with People: Integrating Human-Based and Digital Computation (Invited Talk)

Emery Berger

University of Massachusetts, Amherst (USA)
emery@cs.umass.edu

Humans can perform many tasks with ease that remain difficult or impossible for computers. Crowdsourcing platforms like Amazon's Mechanical Turk make it possible to harness human-based computational power on an unprecedented scale. However, their utility as a general-purpose computational platform remains limited. The lack of complete automation makes it difficult to orchestrate complex or interrelated tasks. Scheduling human workers to reduce latency costs real money, and jobs must be monitored and rescheduled when workers fail to complete their tasks. Furthermore, it is often difficult to predict the length of time and payment that should be budgeted for a given task. Finally, the results of human-based computations are not necessarily reliable, both because human skills and accuracy vary widely, and because workers have a financial incentive to minimize their effort.

This talk presents AutoMan, the first fully automatic crowd programming system. AutoMan integrates human-based computations into a standard programming language as ordinary function calls, which can be intermixed freely with traditional functions. This abstraction allows AutoMan programmers to focus on their programming logic. An AutoMan program specifies a confidence level for the overall computation and a budget. The AutoMan runtime system then transparently manages all details necessary for scheduling, pricing, and quality control. AutoMan automatically schedules human tasks for each computation until it achieves the desired confidence level; monitors, reprices, and restarts human tasks as necessary; and maximizes parallelism across human workers while staying under budget.

AutoMan is available for download at `www.automan-lang.org`

Table of Contents

Optimal Register Allocation in Polynomial Time

Philipp Klaus Krause

Goethe-Universität, Frankfurt am Main

Abstract. A graph-coloring register allocator that optimally allocates registers for structured programs in polynomial time is presented. It can handle register aliasing. The assignment of registers is optimal with respect to spill and rematerialization costs, register preferences and coalescing. The register allocator is not restricted to programs in SSA form or chordal interference graphs. It assumes the number of registers is to be fixed and requires the input program to be structured, which is automatically true for many programming languages and for others, such as C, is equivalent to a bound on the number of goto labels per function. Non-structured programs can be handled at the cost of either a loss of optimality or an increase in runtime. This is the first optimal approach that has polynomial runtime and works for such a huge class of programs.

An implementation is already the default register allocator in most backends of a mainstream cross-compiler for embedded systems.

Keywords: register allocation, tree-decomposition, structured program.

1 Introduction

Compilers map variables to physical storage space in a computer. The problem of deciding which variables to store into which registers or into memory is called register allocation. Register allocation is one of the most important stages in a compiler. Due to the ever-widening gap in speed between registers and memory the minimization of spill costs is of utmost importance. For CISC architectures, such as ubiquitous x86, register aliasing (i. e. multiple register names mapping to the same physical hardware and thus not being able to be used at the same time) and register preferences (e. g. due to certain instructions taking a different amount of time depending on which registers the operands reside in) have to be handled to generate good code. Coalescing (eliminating moves by assigning variables to the same registers, if they do not interfere, but are related by a copy instruction) is another aspect, where register allocation can have a significant impact on code size and speed.

Our approach is based on graph coloring and assumes the number of registers to be fixed. It can handle arbitrarily complex register layouts, including all kinds of register aliasing. Register preferences, coalescing and spilling are handled using a cost function. Different optimization goals, such as code size, speed, energy consumption, or some aggregate of them can be handled by choice of the cost function. The approach is particularly well-suited for embedded systems, which often have a small number of registers, and where optimization is of utmost importance due to constraints on energy consumption, monetary cost, etc. We have implemented a prototype of our approach, and it has become the default register allocator in most backends of sdcc [12], a mainstream C cross-compiler for architectures commonly found in embedded systems. Virtually all programs are structured, and for these the register allocator has polynomial runtime. This is the first optimal approach that has polynomial runtime and works for such a huge class of programs.

Chaitin's classic approach to register allocation [7] uses graph coloring. The approach assumes r identical registers, identical spill cost for all variables, and does not handle register preferences or coalescing. Solving this problem optimally is equivalent to finding a maximal r-colorable induced subgraph in the interference graph of the variables and coloring it. In general this is NP-hard [8]. Even when it is known that a graph is r-colorable it is NP-hard to find a r-coloring compatible with a fraction of $1 - \frac{1}{33r}$ of the edges [18]. Thus Chaitin's approach uses heuristics instead of optimally solving the problem. It has been generalized to more complex architectures [31]. The maximum r-colorable induced subgraph problem for fixed r can be solved optimally in polynomial time for chordal interference graphs [27,35], which can be obtained when the input programs are in static single assignment (SSA) form [20]. Recent approaches have modeled register allocation as an integer linear programming (ILP) problem, resulting in optimal register allocation for all programs [17,15]. However ILP is NP-hard, and the ILP-based approaches tend to have far worse runtime compared to graph coloring. There are also approaches modeling register allocation as a partitioned boolean quadratic programming (PBQP) problem [30,22]. They can handle some irregularities in the architecture in a more natural way than older graph-coloring approaches, but do not handle coalescing and other interactions that can arise out of irregularities in the instruction set. PBQP is NP-hard, but heuristic solvers seem to perform well for many, but not all practical cases. Linear scan register allocation [28] has become popular for just in time compilation [13]; it is typically faster than approaches based on graph coloring, but the assignment is further away from optimality. Kannan and Proebsting [23] were able to approximate a simplified version of the register allocation problem within a factor of 2 for programs that have series-parallel control-flow graphs (a subclass of 2-structured programs). Thorup [32] uses the bounded tree-width of structured programs to approximate an optimal coloring of the intersection graph by a constant factor. Bodlaender et alii [4] present an algorithm that decides in linear time if it is possible to allocate registers for a structured program without spilling.

Section 2 introduces the basic concepts, including structured programs. Section 3 presents the register allocator in its generality and shows its polynomial runtime. Section 4 discusses further aspects of the allocator, including ways to reduce the practical runtime and how to handle non-structured programs. Section 5 discusses the complexity of register allocation and why certain NP-hardness results do not apply in our setting. Section 6 presents the prototype implementation, followed by the experimental results in Section 7. Section 8 concludes and proposes possible directions for future work.

2 Structured Programs

Compilers transform their input into an intermediate representation, on which they do many optimizations. At the time when register allocation is done, we deal with such an intermediate representation. We also have the *control-flow graph* (CFG) (Π, K), with node set Π and edge set $K \subseteq \Pi^2$ which represents the control flow between the instructions in the intermediate representation. For the code written in the C programming language from Figure 1, the sdcc compiler [12] generates the CFG in Figure 2(a). In this figure, the nodes of the the CFG are numbered, and annotated with the set of variables alive there, and the intermediate representation. As can be seen, sdcc introduced two temporary variables, which make up the whole set of variables $V = \{a, b\}$ to be handled by the register allocator for this code.

```
#include <stdint.h>
#include <stdbool.h>

bool get_pixel(uint_fast8_t x, uint_fast8_t y);
void set_pixel(uint_fast8_t x, uint_fast8_t y);

void fill_line_left(uint_fast8_t x, const uint_fast8_t y)
{
    for(;; x--)
    {
        if(get_pixel(x, y))
            return;
        set_pixel(x, y);
    }
}
```

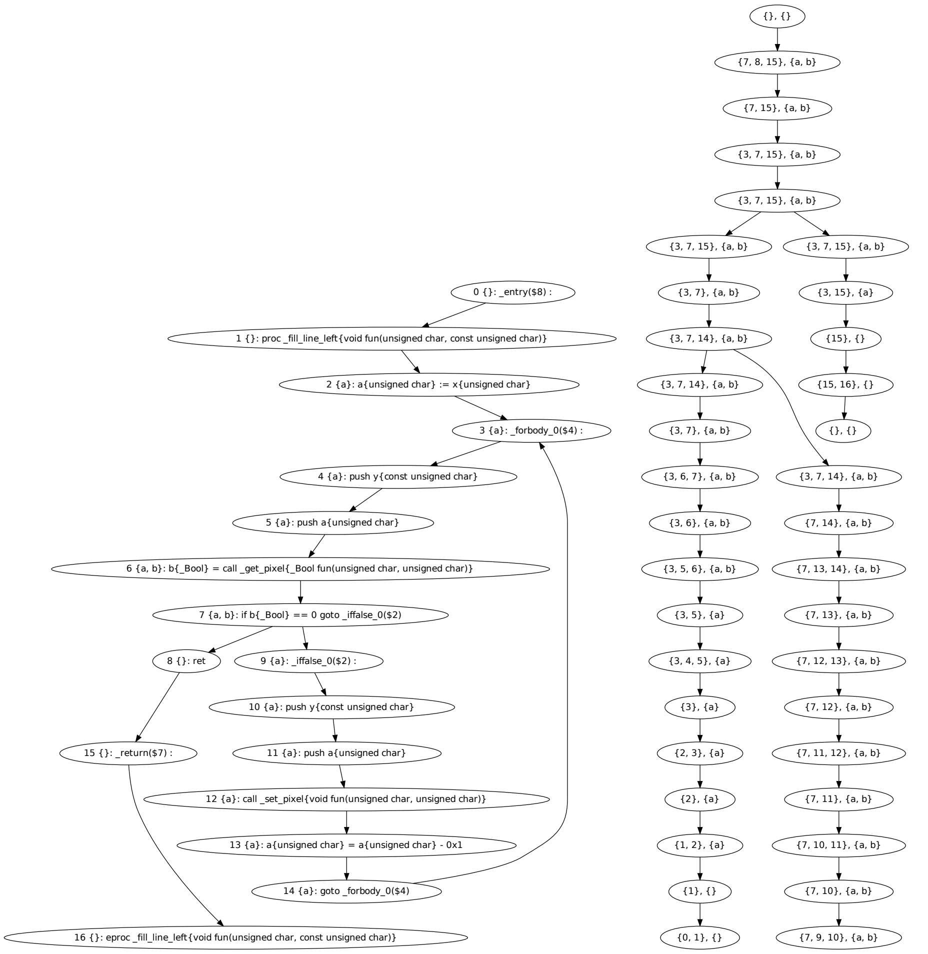
Fig. 1. C code example

Let r be the number of registers. Let $[r] := \{0, \ldots, r-1\}$ be the the set of registers.

Definition 1. *Let V be a set of variables. An* assignment *of variables V to registers $[r]$ is a function $f\colon U \to [r], U \subseteq V$. The assignment is* valid *if it is possible to generate correct code for it, which implies that no conflicting variables are assigned to the same register.*

Variables in $V \setminus U$ are to be placed in memory (spilt) or removed and their value recalculated as needed (rematerialized).

Definition 2 (Register Allocation). *Let the number of available registers be fixed. Given an input program containing variables and their live-ranges and a cost function, that gives costs for register assignments, the problem of* register allocation *is to find an assignment of variables to the registers that minimizes the total cost.*



(a) Control-flow graph (b) Nice tree-decomposition

Fig. 2. CFG and decomposition example

Our approach is based on tree decompositions. Tree decompositions [21] have commonly been used to find polynomial algorithms on restricted graph classes for many problems that are hard on general graphs, since their rediscovery by Robertson and Seymour [29]. This includes well known problems such as graph coloring and vertex cover.

Definition 3 (Tree Decomposition). *Given a graph* $G = (\Pi, K)$ *a* tree decomposition *of* G *is a pair* $(T, \mathcal{X})$ *of a tree* T *and a family* $\mathcal{X} = \{X_i \mid i \text{ node of } T\}$ *of subsets of* Π *with the following properties:*

- $\bigcup_{i \text{ node of } T} X_i = \Pi$,
- *For each edge* $\{x, y\} \in K$, *there is an* i *node of* T, *such that* $x, y \in X_i$,
- *For each node* $x \in \Pi$ *the subgraph of* T *induced by* $\{i \text{ node of } T \mid x \in X_i\}$ *is connected.*

The width *of a tree decomposition* $(T, \mathcal{X})$ *is* $\max\{|X_i| \mid i \text{ node of } T\} - 1$. *The* tree-width $tw(G)$ *of a graph* G *is the minimum width of all tree decompositions of* G.

Intuitively, tree-width indicates how tree-like a graph is. Nontrivial trees have tree-width 1. Cliques on $n \geq 1$ nodes have tree-width $n - 1$. Series-parallel graphs have tree-width at most 2. Tree-decompositions are usually defined for undirected graphs, and our definition of a tree-decomposition for a directed graph is equivalent to the tree-decomposition of the graph interpreted as undirected.

Definition 4 (Structured Program). *Let* $\mathfrak{k} \in \mathbb{N}$ *be fixed. A program is called* $\mathfrak{k}$-structured, *iff its control-flow graph has tree-width at most* $\mathfrak{k}$.

Programs written in Algol or Pascal are 2-structured, Modula-2 programs are 5-structured [32]. Programs written in C are $(7 + \mathfrak{g})$-structured if the number of labels targeted by gotos per function does not exceed $\mathfrak{g}$. Similarly, Java programs are $(6 + \mathfrak{g})$-structured if the number of loops targeted by labeled breaks and labeled continues per function does not exceed $\mathfrak{g}$ [19]. Ada programs are $(6 + \mathfrak{g})$-structured if the number of labels targeted by gotos and labeled loops per function does not exceed $\mathfrak{g}$ [6]. Coding standards tend to place further restrictions, resulting e. g. in C programs being 5-structured when adhering to the widely adopted MISRA-C:2004 [2] standard. A survey of 12522 Java methods from applications and the standard library found tree-width above 3 to be very rare. With one exception of tree-width 5, all methods had tree-width 4 or lower [19].

Often proofs and algorithms on tree-decompositions are easier to describe, understand and implement when using nice tree-decompositions:

Definition 5 (Nice Tree Decomposition). *A tree decomposition* $(T, \mathcal{X})$ *of a graph* G *is called* nice, *iff*

- T *is oriented, with root* t, $X_t = \emptyset$.
- *Each node* i *of* T *is of one of the following types:*
  - *Leaf node, no children*
  - *Introduce node, has one child* $j, X_j \subsetneq X_i$
  - *Forget node, has one child* $j, X_j \supsetneq X_i$
  - *Join node, has two children* $j_1, j_2, X_i = X_{j_1} = X_{j_2}$

Given a tree-decomposition, a nice tree-decomposition of the same width can be found easily. Figure 2(b) shows a nice tree-decomposition of width 2 for the CFG in Figure 2(a). At each node i in the figure, the left set is X_i.

From now on let $G = (\Pi, K)$ be the control flow graph of the program, let $I = (V, E)$ be the corresponding conflict graph of the variables of the program

(i. e. the intersection graph of the variables' live-ranges). The live-ranges are connected subgraphs of G. Let $(T, \mathcal{X})$ be a nice tree decomposition of minimum width of G with root t. For $\pi \in \Pi$ let V_π be the set of all variables $v \in V$, that are alive at π (in the example CFG in figure 2(a) this is the set directly after the node number). Let $\mathcal{V} := \max_{\pi \in \Pi}\{V_\pi\}$ be the maximum number of variables alive at the same node. For each $i \in T$ let $V_i := \bigcup_{\pi \in X_i} V_\pi$ be the set of variables alive at any of the corresponding nodes from the CFG (in the nice tree-decomposition in figure 2(b) this is the right one of the sets at each node).

3 Optimal Polynomial Time Register Allocation

The goal in register allocation is to minimize costs, including spill and rematerialization costs, costs from not respecting register preferences, costs from not coalescing, etc. These costs are modeled by a cost function that gives costs for an instruction π under register assignment f:

$$c\colon \{(\pi, f) \mid f\colon U \to [r], U \subseteq V_\pi, \pi \in \Pi\} \to [0, \infty]$$

Different optimization goals, such as speed or code size can be implemented by choosing c. E. g. when optimizing for code size c could give the code size for π under assignment f, or when optimizing for speed c could give the number of cycles π needs to execute multiplied by an execution probability obtained from a profiler. We assume that c can be evaluated in constant time. The goal is thus finding an f for which $\sum_{\pi \in \Pi} c(\pi, f|V_\pi)$ is minimal.

Let S be the function that gives the minimum possible costs for instructions in the subtree rooted at $i \in T$, excluding instructions in X_i when assigning variables alive in the subtree rooted at $i \in T$ when choosing $f\colon U \to [r], U \subseteq V$ as the assignment of variables alive at instructions $i \subseteq \Pi$ to registers, i. e.

$$S\colon \{(i, f) \mid i \in T, f\colon U \to [r], U \subseteq V_i\} \to [0, \infty].$$

$$S(i, f) := \min_{g|_{V_i} = f|_{V_i}} \left\{ \sum_{\pi \in T_i} c(\pi, g|_{V_\pi}) \right\}.$$

Where T_i is the set of instructions in the subtree of T rooted at $i \in T$, excluding instructions in X_i. This function at the root $t \in T$, and the corresponding assignment that results in the minimum is what we want:

$$S(t, f) = \min_{g|_{V_t} = f|_{V_t}} \left\{ \sum_{\pi \in T_t} c(\pi, g|_{V_\pi}) \right\} =$$

$$= \min_{g|_\emptyset = f|_\emptyset} \left\{ \sum_{\pi \in \Pi} c(\pi, g|_{V_\pi}) \right\} = \min_g \left\{ \sum_{\pi \in \Pi} c(\pi, g|_{V_\pi}) \right\}.$$

To get S, we first define a function s, and then show that $S = s$ and that s can be calculated in polynomial time. We define s inductively, and depending on the type of i:

- Leaf: $s(i, f) := 0$
- Introduce with child j: $s(i, f) := s(j, f|_{V_j})$
- Forget with child j: $s(i, f) := \min\{\sum_{\pi \in X_j \setminus X_i} c(\pi, g|_{V_\pi}) + s(j, g) \mid g|_{V_i} = f\}$
- Join with children j_1 and j_2: $s(i, f) := s(j_1, f) + s(j_2, f)$

By calculating all the $s(i, f)$ and recording which g gave the minimum we can obtain an optimal assignment. We will show that s correctly gives the minimum possible cost and that it can be calculated in polynomial time.

Lemma 1. *For each $i \in T, f \colon U \to [r], U \subseteq V_i$ the value $s(i, f)$ is the minimum possible cost for instructions in the subtree rooted at $i \in T$, excluding instructions in X_i when assigning variables alive in the subtree rooted at $i \in T$ when choosing f as the assignment of variables alive at instructions $i \subseteq \Pi$ to registers, i. e. $s = S$. Using standard bookkeeping techniques we obtain the corresponding assignments for the subtree.*

Proof. By induction we can assume that the lemma is true for all children of i. Let T_i be the set of instructions in the subtree rooted at $i \in T$, excluding instructions in X_i.

Case 1: i is a leaf. There are no instructions in $T_i = X_i \setminus X_i = \emptyset$, thus the cost is zero: $s(i, f) = 0 = S(i, f)$.

Case 2: i is an introduce node with child j. $T_i = T_j$, since $X_i \supseteq X_j$, thus the cost remains the same: $s(i, f) = s(j, f) = S(j, f) = S(i, f)$

Case 3: i is a forget node with child j. $T_i = T_j \cup (X_j \setminus X_i)$, the union is disjoint. Thus we get the correct result by adding the costs for the instructions in $X_j \setminus X_i$:

$$s(i, f) = \min_{g|_{V_i} = f} \left\{ \sum_{\pi \in X_j \setminus X_i} c(\pi, f|_{V_\pi}) + s(j, g) \right\} =$$

$$\min_{g|_{V_i} = f} \left\{ \sum_{\pi \in X_j \setminus X_i} c(\pi, f|_{V_\pi}) + S(j, g) \right\} =$$

$$\min_{g|_{V_i} = f|_{V_i}} \left\{ \sum_{\pi \in X_j \setminus X_i} c(\pi, f|_{V_\pi}) + \sum_{\pi \in T_j} c(\pi, g|_{V_\pi}) \right\} =$$

$$\min_{g|_{V_i} = f|_{V_i}} \left\{ \sum_{\pi \in T_i} c(\pi, g|_{V_\pi}) \right\} = S(i, f).$$

Case 4: i is a join node with children j_1 and j_2. $T_i = T_{j_1} \cup T_{j_2}$, since $X_i = X_{j_1} = X_{j_2}$. The union is disjoint. Thus we get the correct result by adding the costs from both subtrees: $s(i, f) = s(j_1, f) + s(j_2, f) = S(j_1, f) + S(j_2, f) = S(i, f)$.

Lemma 2. *Given the tree-decomposition of minimum width, s can be calculated in polynomial time.*

Proof. Each $V_\pi, \pi \in \Pi$ is the union of two cliques, each of size at most $\mathcal{V}$: The variables alive at the start of the instruction form the clique, and so do the variables alive at the end of the instruction. Thus $V_i, i \in T$ is the union of at most $2(tw(G)+1)$ cliques. From each clique at most r variables can be placed in registers.

At each node i of the tree decomposition time $O(\mathcal{V}^{2(tw(G)+1)r})$ is sufficient:

Case 1: i is a leaf. There are at most $O(\mathcal{V}^{2|X_i|r}) \subseteq (\mathcal{V}^{2(tw(G)+1)r})$ possible f, and for each one we do a constant number of calculations.

Case 2: i is an introduce node with child j. The reasoning from case 1 holds.

Case 3: i is a forget node. There are at most $O(\mathcal{V}^{2|X_i|r})$ possible f. For each one we need to consider at most $O(\mathcal{V}^{2|X_j \setminus X_i|r})$ different g. Thus time $O(\mathcal{V}^{2|X_i|r}) \cdot (\mathcal{V}^{2|X_j \setminus X_i|r}) \subseteq O(\mathcal{V}^{2|X_i|r+2|X_j \setminus X_i|r}) = O(\mathcal{V}^{2|X_j|r}) \subseteq O(\mathcal{V}^{2(tw(G)+1)r})$ is sufficient.

Case 4: i is a join node with children j_1 and j_2. The reasoning from case 1 holds.

The tree decomposition has at most $|T|$ nodes, thus the total time is in $O(|T|\mathcal{V}^{2(tw(G)+1)r}) = O(|T|\mathcal{V}^{\mathfrak{c}})$ for a constant $\mathfrak{c}$ and thus polynomial.

Theorem 1. *The register allocation problem can be solved in polynomial time for structured programs.*

Proof. Given an input program of bounded tree-width we can calculate a tree-decomposition of minimum width in linear time [3]. We can then transform this tree-decomposition into a nice one of the same width. The linear time for these steps implies that $|T|$ is linear in $|G|$. Using this nice tree decomposition s is calculated in polynomial time as above. The total runtime is thus in $O(|G|\mathcal{V}^{2(tw(G)+1)r}) = O(|G|\mathcal{V}^{\mathfrak{c}})$ for a constant $\mathfrak{c}$.

4 Remarks

Remark 1. The runtime bound is reduced by a factor of $\mathcal{V}^{tw(G)r}$, if the intermediate representation is three-address code.

Proof. In that case there is at most one variable alive at the end of an instruction that was not alive at the start of the instruction, so in the proof of Lemma 2, we can replace $O(\mathcal{V}^{2(tw(G)+1)r})$ by $O(\mathcal{V}^{(tw(G)+2)r})$.

Remark 2. Bodlaender's algorithm [3] used in the proof above is not a practical option. However there are other, more practical alternatives, including a linear-time algorithm that is not guaranteed to give decompositions of minimal width, but will do so for many programming languages [32,10].

Remark 3. Implementations of the algorithm can be massively parallel, resulting in linear runtime.

Proof. At each $i \in T$ the individual $s(i, f)$ do not depend on each other. They can be calculated in parallel. By requiring that $|X_j| = |X_i| + 1$ at forget nodes, we can assume that the number of different g to consider is at most $\mathcal{V}^{2r}$, resulting in time $O(r)$ for calculating the minimum over the $s(j, g)$. Thus given enough processing elements the runtime of the algorithm can be reduced to $O(|G|r)$.

Remark 4. Doing live-range splitting as a preprocessing step is cheap.

The runtime bound proved above only depends on $\mathcal{V}$, not $|V|$. Thus splitting of non-connected live-ranges before doing register allocation doesn't affect the bound. When the splitting is done to allow more fine-grained control over spilling, then the additional cost is small (even the extreme case of inserting permutation instructions between any two original nodes in the CFG, and splitting all live-ranges there would only double Π and $\mathcal{V}$).

Remark 5. Non-structured programs can be handled at the cost of either a loss of optimality or an increase in runtime.

Programs of high tree-width are extremely uncommon (none have been found so far, with the exception of artificially constructed examples). Nevertheless they should be handled correctly by compilers. One approach would be to handle these programs like the others. Since $tw(G)$ is no longer constant, the algorithm is no longer guaranteed to have polynomial runtime. Where polynomial runtime is essential, a preprocessing step can be used. This preprocessing stage would spill some variables (or allocate them using one of the existing heuristic approaches). Edges of G, at which no variables are alive, can be removed. Once enough edges have been removed, $tw(G) \leq \mathfrak{k}$ and our approach can be applied to allocate the remaining variables. Another option is the heuristic limit used in our prototype as mentioned in Section 6.

Remark 6. The runtime of the polynomial time algorithm can be reduced by a factor of more than $(2(tw(G) + 1)r)!$, if there is no register aliasing and registers are interchangeable within each class. Furthermore r can then be chosen as the maximum number of registers that can be used at the same time instead of the total number of registers, which gives a further runtime reduction in case of register aliasing.

Proof. Instead of using $f : U \to [r]$ we can directly use U.

- Leaf: $s(i, U) := 0$
- Introduce with child j: $s(i, U) := s(j, U \cup V_j)$
- Forget with child j: $s(i, U) := \sum_{\pi \in X_j \setminus X_i} c(\pi, U) + \min\{s(j, W) \mid W \cap V_i = U\}$
- Join with children j_1 and j_2: $s(i, U) := s(j_1, U) + s(j_2, U)$

Most of the proofs of the lemmata are still valid. However instead of the number of possible f we now look at the number of possible U, which is at most

$$\binom{\mathcal{V}}{2(tw(G) + 1)r}.$$

Remark 7. Using a suitable cost function and $r = 1$ we get a polynomial time algorithm for *maximum independent set* on intersection graphs of connected subgraphs of graphs of bounded tree-width.

Remark 8. The allocator is easy to re-target, since the cost function is the only architecture-specific part.

5 Complexity of Register Allocation

The complexity of register allocation in different variations has been studied for a long time and there are many NP-hardness results.

Publication	Difference to our setting
Register allocation via coloring [8]	$tw(G)$ unbounded
On the Complexity of Register Coalescing [5]	$tw(G)$ unbounded
The complexity of coloring circular arcs and chords [16]	r is part of input
Aliased register allocation for straight line programs is NP-complete [26]	r is part of input
On Local Register Allocation [14]	r is part of input

Given a graph I a program can be written, such that the program has conflict graph I [8]. Since 3-colorability is NP-hard [24], this proves the NP-hardness of register allocation, as a decision problem for $r = 3$. However the result does not hold for structured programs. Coalescing is NP-hard even for programs in SSA-form [5]. Again this result does not hold for structured programs. Register allocation, as a decision problem, is NP-hard, even for series-parallel control flow graphs, i. e. for $tw(G) \leq 2$ and thus for structured programs, when the number of registers is part of the input [16]. Register allocation, as a decision problem, is NP-hard when register aliasing is possible, even for straight-line programs, i. e. $tw(G) = 1$ and thus for structured programs, when the number of registers is part of the input [26]. Minimizing spill costs is NP-hard, even for straight-line programs, i. e. $tw(G) = 1$ and thus for structured programs, when the number of registers is part of the input [14].

It is thus fundamental to our polynomial time optimal approach, which handles register aliasing, register preferences, coalescing and spilling, that the input program is structured and the number of registers is fixed.

The runtime bound of our approach proven above is exponential in the number of registers r. However, even a substantially simplified version of the register allocation problem is W[SAT]- and co-W[SAT]-hard when parametrized by the number of registers even for $tw(G) = 2$ [25]. Thus doing optimal register allocation in time faster than $\mathcal{V}^{O(r)}$ would imply a collapse the parametrized complexity hierarchy. Such a collapse is considered highly unlikely in parametrized complexity theory. This means that not only we cannot get rid of the r in the exponent, but we can't even separate it from the $\mathcal{V}$ either.

6 Prototype Implementation

We have implemented a prototype of the allocator in C++ for the HC08, S08, Z80, Z180, Rabbit 2000/3000, Rabbit 3000A and LR35902 ports of sdcc [12], a C compiler for embedded systems. It is the default register allocator for these architectures as of the sdcc 3.2.0 release in mid-2012 and can be found in the public source code repository of the sdcc project.

S08 is the architecture of the current main line of Freescale microcontrollers, a role previously filled by the HC08 architecture. Both architectures have three 8-bit registers, which are assigned by the allocator. The Z80 architecture is a classic architecture designed by Zilog, which was once common in general-purpose computers. It currently is mostly used in embedded systems. The Z180, Rabbit 2000/3000 and Rabbit 3000A are newer architectures derived from the Z80, which are also mostly used in embedded systems. The differences are in the instruction set, not in the register set. The Z80 architecture is simple enough to be easily understood, yet has many of the typical features of complex CISC architectures. Nine 8-bit registers are assigned by the allocator (A, B, C, D, E, H, L, IYL, IYH). IYL and IYH can only be used together as 16-bit register IY; there are instructions that treat BC, DE or HL as 16-bit registers; many 8-bit instructions can only use A as the left operand, while many 16-bit instructions can only use HL as the left operand. There are some complex instructions, like djnz, a decrement-and-jump-if-not-zero instruction that always uses B as its operand, or ldir, which essentially implements memcpy() with the pointer to the destination in DE, source pointer in HL and number of bytes to copy in BC. All these architectural quirks are captured by the cost function. The LR35902 is the CPU used in the Game Boy video game system. It is inspired by the Z80 architecture, but has a more restricted instruction set and fewer registers. Five 8-bit registers are assigned by the allocator.

The prototype still has some limitations, e. g. current code generation does not allow the A or IY registers to hold parts of a bigger variable in the Z80 port. Code size was used as the cost function, due to its importance in embedded systems and relative ease of implementation (optimal speed or energy optimization would require profiler-guided optimization). We obtain the tree decomposition using Thorup's method [32], and then transform it into a nice tree decomposition. The implementation of the allocator essentially follows Section 3, and is neither very optimized for speed nor parallelized. However a configurable limit on the number of assignments considered at each node of the tree decomposition has been introduced. When this limit is reached, some assignments are discarded heuristically. The heuristic mostly relies on the $s(i, f)$ to discard those assignments that have the highest cost so far first, but takes other aspects into account to increase the chance that compatible assignments will exist at join nodes. When the limit is reached, and the heuristic applied, the assignment is no longer provably optimal. This limit essentially provides a trade-off between runtime and quality of the assignment.

The prototype was compared to the current version of the old sdcc register allocator, which has been improved over years of use in sdcc. The old allocator is basically an improved linear scan [28,13] algorithm extended to take the architecture into account, e.g. preferring to use registers HL and A, since accesses to them typically are faster than those to other registers and taking coalescing, register aliasing and some other preferences into account. This comparison was done using the Z80 architecture, which has been around for a long time, so there is a large number of programs available for it.

Furthermore we did a comparison between the different architectures, which shows the impact of the number of registers on the performance of the allocator.

7 Experimental Results

Six benchmarks considered representative of typical applications for embedded systems have been used to evaluate the register allocator, by compiling them with sdcc 3.2.1 #8085:

- The Dhrystone benchmark [33], version 2 [34]. An ANSI-C version was used, since sdcc does not support K&R C.
- A set of source files taken from real-world applications and used by the sdcc project to track code size changes over sdcc revisions and to compare sdcc to other compilers.
- The Coremark benchmark [1], version 1.0.
- The FatFS implementation of the FAT filesystem [9], version R0.09.
- Source code from two games for the ColecoVision video game console. All C source code has been included, while assembler source files and C source files that only contain data have been omitted.
- The Contiki operating system [11], version 2.5.

We first discuss the results of compiling the benchmarks for the Z80 architecture. Figure 3 shows the code size with the peephole optimizer (a post code-generation optimization stage not taken into account by the cost function) enabled, Figure 4 with the peephole optimizer disabled. Furthermore, Figure 3 shows the compilation time, and Figure 4 shows the fraction of provably optimally allocated functions (i. e. those functions for which the heuristic never was applied); the former is little affected by enabling the peephole optimizer and the latter not at all.

The dhrystone benchmark is rather small. At 10^8 assignments per node we find a provably optimal assignment for 83.3% of the functions. This also results in a moderate reduction in code size of 6.0% before and 4.9% after the peephole optimizer when compared to the old allocator. The sdcc benchmark, even though small, contains more complex functions; at 10^8 assignments per node we find a provably optimal assignment for 93.9% of the functions. However code size seems to be stable from 6×10^7 onwards. We get a code size reduction of 16.9% before and 17.3% after the peephole optimizer. For Coremark, we find an optimal assignment for 77% of the functions at 10^8 assignments per node. We get a code size reduction of 7.8% before and 6.9% after the peephole optimizer.

FatFS is the benchmark which is the most problematic for our allocator; it contains large functions with complex control flow, some containing nearly a kilobyte of local variables. Even at 4.5×10^7 assignments per node (we did not run compilations at higher values due to lack of time) only 45% of the functions are provably optimally allocated. We get a reduction in code size of 9.8% before and 11.4% after the peephole optimizer. Due to the low fraction of provably optimally allocated functions the code size reduction and compilation time are likely to be much higher for a higher number of assignments per node.

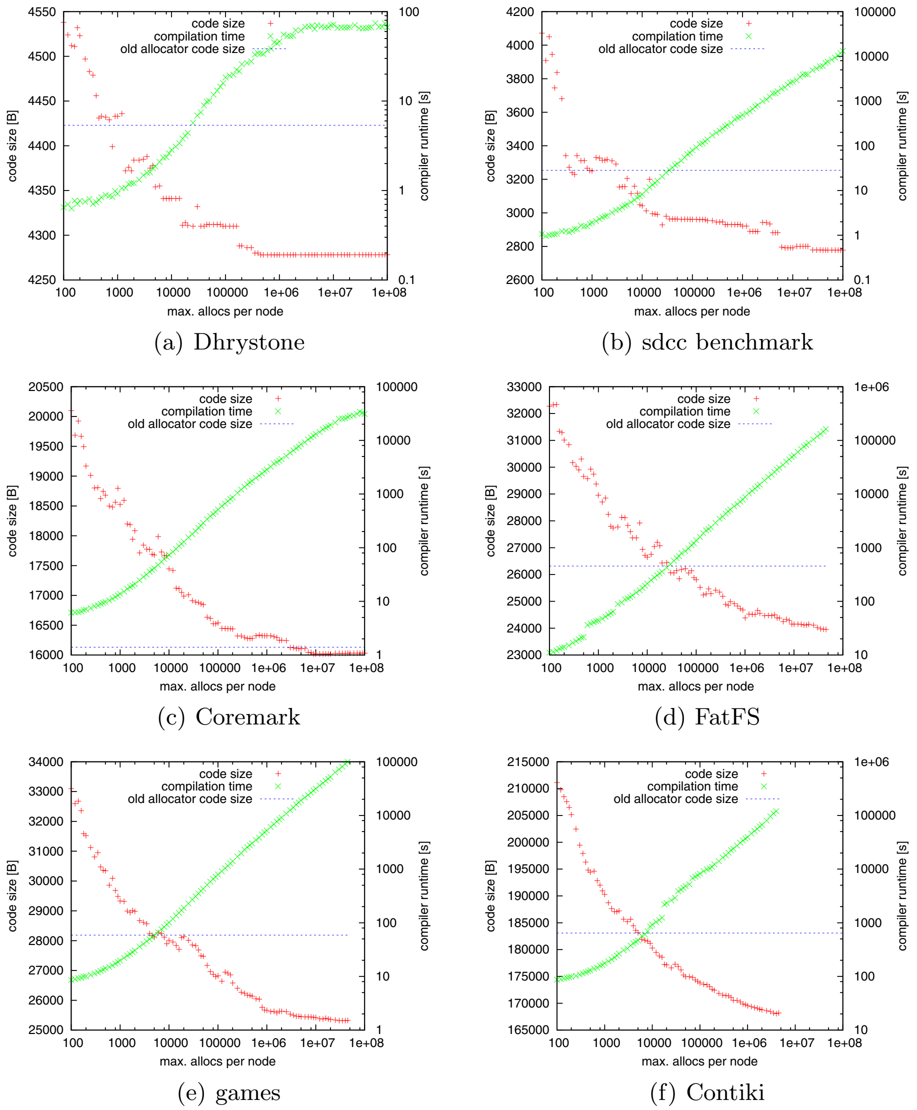
In the games benchmark, about 73% of the functions are provably optimally allocated at 4.5×10^7 assignments per node; at that value the code size is reduced by 11.2% before and 12.3% after the peephole optimizer. This result is consistent with the previous two: The source code contains both complex and simple functions (and some data, since only source files containing data only were excluded, while those that contain both code and data were included).

For Contiki, about 76% of the functions are provably optimally allocated at 4.5×10^6 assignments per node (we did not run compilations at higher values due to lack of time); at that value the code size is reduced by 9.1% before and 8.2% after the peephole optimizer. Contiki contains some complex control flow, but it tends to use global instead of local variables; where there are local variables they are often 32-bit variables, of which neither the optimal nor the old allocator can place more than one in registers at a given time (due to the restriction in code generation that allows the use of IY for 16-bit variables only).

We also did a comparison of the different architectures (except for the Rabbit 3000A, since it is very similar to the Rabbit2000/3000). Figure 5 shows the code size with the peephole optimizer enabled, Figure 6 with the peephole optimizer disabled. Furthermore, Figure 5 shows the compilation time, and Figure 6 shows the fraction of provably optimally allocated functions.

The results clearly show that for the runtime of the register allocator and the fraction of provably optimally allocated function the number of registers is much more important than the architecture: For the architectures with 3 registers, code size is stable from 3.8×10^3 (for HC08) and 4.0×10^3 (for S08) assignments, and all functions are provably optimally allocated from 2.5×10^4 assignments onwards. The effect of the register allocator on compiler runtime is mostly lost in noise. For the architecture with 5 registers (LR35902), code size is stable from 9.0×10^3 assignments onwards, and all functions are provably optimally allocated from 1.4×10^5 assignments onwards. For the architectures with 9 registers, there are still functions for which a provably optimal assignment is not found at 1.0×10^8 assignments. Architectural differences other than the number of registers have a substantial impact on code size, but only a negligible one on the performance of the register allocator.

We also see that the improvement in code size compared to the old allocator was the most substantial for architectures that have just three registers: For the HC08 17.6% before and 18% after the peephole optimizer, for the S08 20.1% before and 21.1% after the peephole optimizer. This has substantially reduced, but not completely eliminated the gap in generated code size between sdcc and

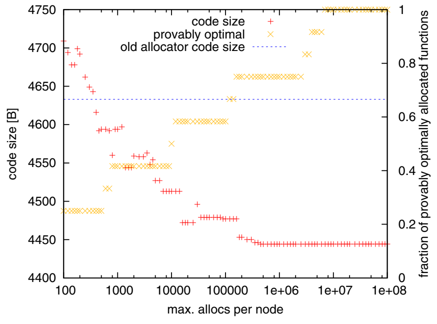


(a) Dhrystone (b) sdcc benchmark

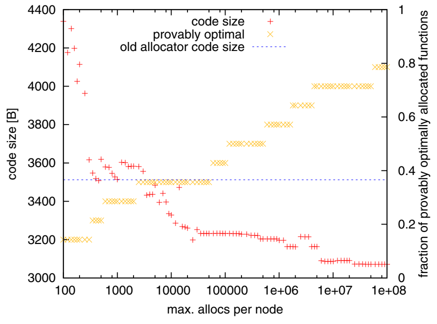
(c) Coremark (d) FatFS

(e) games (f) Contiki

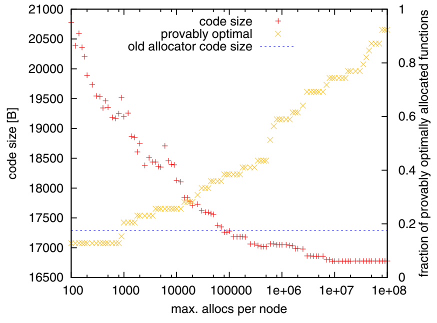
Fig. 3. Experimental Results (Z80, with peephole optimizer)



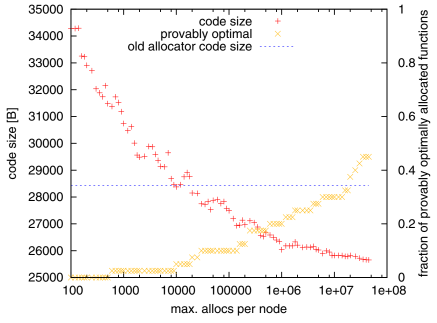
(a) Dhrystone



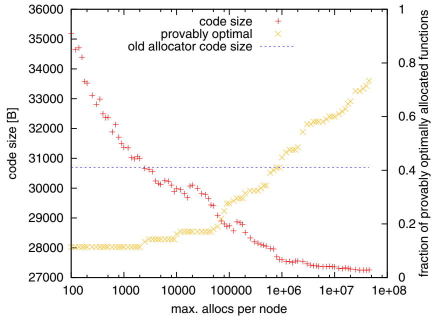
(b) sdcc benchmark



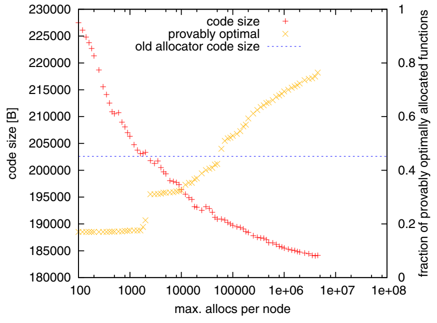
(c) Coremark



(d) FatFS

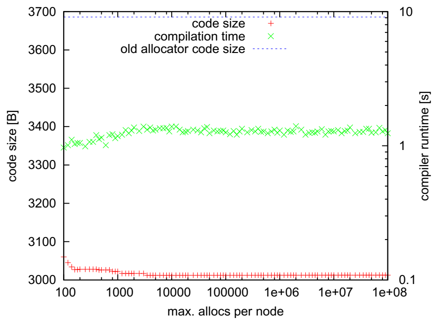


(e) games

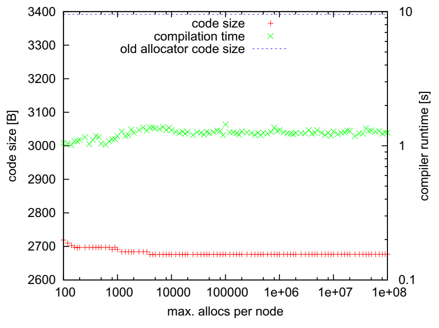


(f) Contiki

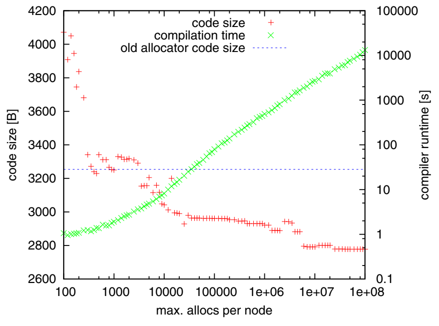
Fig. 4. Experimental Results (Z80, without peephole optimizer)



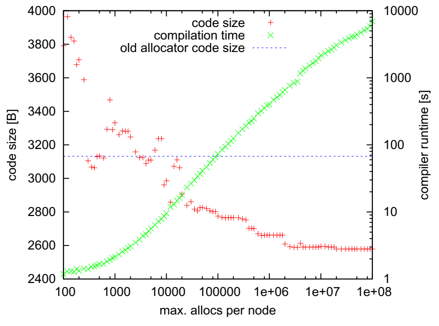
(a) HC08



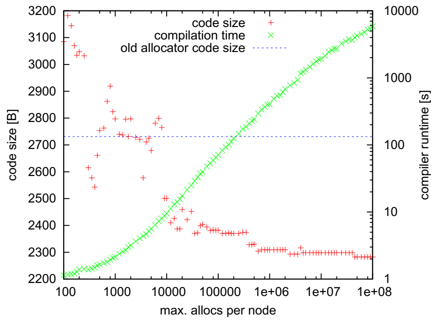
(b) S08



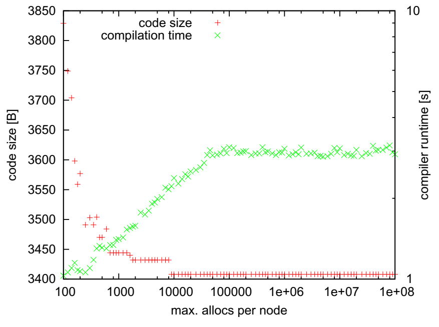
(c) Z80



(d) Z180

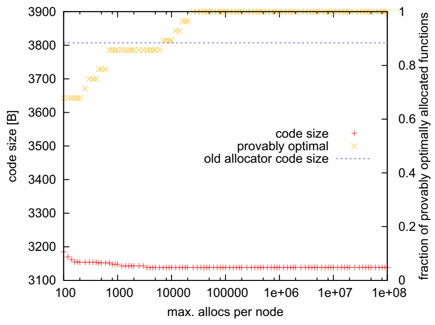


(e) Rabbit 2000/3000

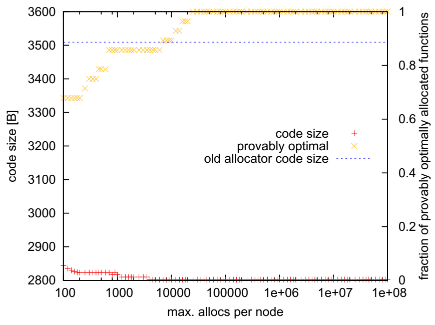


(f) LR35902

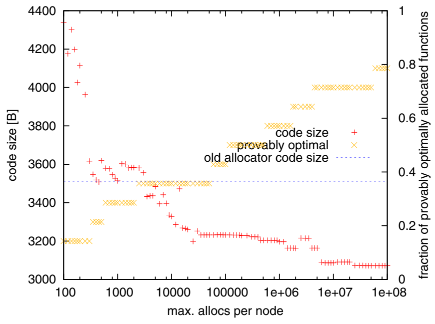
Fig. 5. Experimental Results (sdcc benchmark, with peephole optimizer)



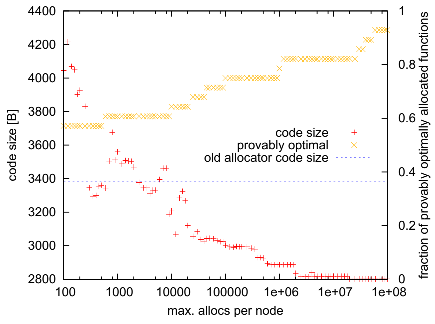
(a) HC08



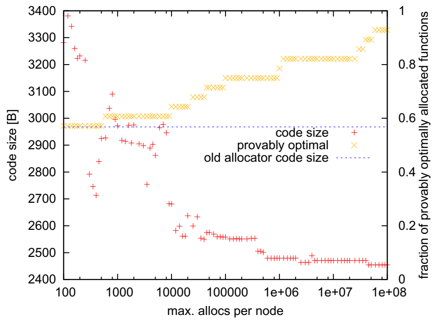
(b) S08



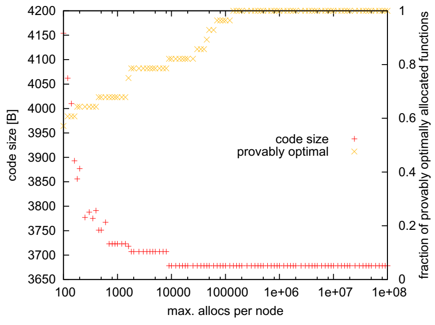
(c) Z80



(d) Z180



(e) Rabbit 2000/3000



(f) LR35902

Fig. 6. Experimental Results (sdcc benchmark, without peephole optimizer)

the competing Code Warrior and Cosmic C compilers. The Z180 and Rabbit 2000/3000 behave similar to the Z80, which was already discussed above. Our register allocator makes sdcc substantially better in generated code size than the competing z88dk, HITECH-C and CROSS-C compilers for these architectures. The LR35902 backend had been unmaintained in sdcc for some time, and was brought back to life after the 3.1.0 release, at which time it was not considered worth the effort to make the old register allocator work with it. There is no other current compiler for the LR35902.

8 Conclusion

We presented an optimal register allocator, that has polynomial runtime. Register allocation is one of the most important stages of a compiler. Thus the allocator is a major step towards improving compilers. The allocator can handle a variety of spill and rematerialization costs, register preferences and coalescing.

A prototype implementation shows the feasibility of the approach, and is already in use in a major cross-compiler targeting architectures found in embedded systems. Experiments show that it performs excellently for architectures with a small number of registers, as common in embedded systems.

Future research could go towards improving the runtime further, completing the prototype and creating a massive parallel implementation. This should make the approach feasible for a broader range of architectures.

References

1. Coremark, http://www.coremark.org
2. Guidelines for the Use of the C Language in Critical Systems (MISRA-C:2004, 2nd Edition). Technical report, MISRA (2008)
3. Bodlaender, H.L.: A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM Journal of Computation 25(6), 1305–1317 (1996)
4. Bodlaender, H.L., Gustedt, J., Telle, J.A.: Linear-Time Register Allocation for a Fixed Number of Registers. In: SODA 1998: Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 574–583. Society for Industrial and Applied Mathematics (1998)
5. Bouchez, F., Darte, A., Rastello, F.: On the Complexity of Register Coalescing. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO 2007, pp. 102–114. IEEE Computer Society (2007)
6. Burgstaller, B., Blieberger, J., Scholz, B.: On the Tree Width of Ada Programs. In: Llamosí, A., Strohmeier, A. (eds.) Ada-Europe 2004. LNCS, vol. 3063, pp. 78–90. Springer, Heidelberg (2004)
7. Chaitin, G.J.: Register allocation & spilling via graph coloring. In: SIGPLAN 1982: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, pp. 98–105. Association for Computing Machinery (1982)
8. 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)
9. ChaN. Fatfs, http://elm-chan.org/fsw/ff/00index_e.html

10. Dendris, N.D., Kirousis, L.M., Thilikos, D.M.: Fugitive-search games on graphs and related parameters. Theoretical Computer Science 172(1-2), 233–254 (1997)
11. Dunkels, A., Grönvall, B., Voigt, T.: Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In: Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I) (November 2004)
12. Dutta, S.: Anatomy of a Compiler. Circuit Cellar 121, 30–35 (2000)
13. Evlogimenos, A.: Improvements to Linear Scan register allocation, Technical report, University of Illinois, Urbana-Champaign (2004)
14. Farach, M., Liberatore, V.: On local register allocation. In: Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1998, pp. 564–573. Society for Industrial and Applied Mathematics (1998)
15. Fu, C., Wilken, K.: A Faster Optimal Register Allocator. In: MICRO 35: Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture, pp. 245–256. IEEE Computer Society Press (2002)
16. Garey, M.R., Johnson, D.S., Miller, G.L., Papadimitriou, C.H.: The Complexity of Coloring Circular Arcs and Chords. SIAM Journal on Algebraic Discrete Methods 1(2), 216–227 (1980)
17. Goodwin, D.W., Wilken, K.D.: Optimal and near-optimal global register allocations using 0–1 integer programming. Software Practice & Experience 26(8), 929–965 (1996)
18. Guruswami, V., Sinop, A.K.: Improved Inapproximability Results for Maximum k-Colorable Subgraph. In: Dinur, I., Jansen, K., Naor, J., Rolim, J. (eds.) APPROX and RANDOM 2009. LNCS, vol. 5687, pp. 163–176. Springer, Heidelberg (2009)
19. Gustedt, J., Mæhle, O.A., Telle, J.A.: The Treewidth of Java Programs. In: Mount, D.M., Stein, C. (eds.) ALENEX 2002. LNCS, vol. 2409, pp. 86–97. Springer, Heidelberg (2002)
20. 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)
21. Halin, R.: Zur Klassifikation der endlichen Graphen nach H. Hadwiger und K. Wagner. Mathematische Annalen 172(1), 46–78 (1967)
22. Hames, L., Scholz, B.: Nearly Optimal Register Allocation with PBQP. In: Lightfoot, D.E., Ren, X.-M. (eds.) JMLC 2006. LNCS, vol. 4228, pp. 346–361. Springer, Heidelberg (2006)
23. Kannan, S., Proebsting, T.: Register Allocation in Structured Programs. In: Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1995, pp. 360–368. Society for Industrial and Applied Mathematics (1995)
24. Karp, R.M.: On the Computational Complexity of Combinatorial Problems. Networks 5, 45–68 (1975)
25. Krause, P.K.: The Complexity of Register Allocation. To appear in the GROW 2011 special issue of Discrete Applied Mathematics (2011)
26. Lee, J.K., Palsberg, J., Pereira, F.M.Q.: Aliased register allocation for straight-line programs is NP-complete. Theoretical Computer Science 407(1-3), 258–273 (2008)
27. Pereira, F.M.Q.: Register Allocation Via Coloring of Chordal Graphs. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 315–329. Springer, Heidelberg (2005)
28. Poletto, M., Sarkar, V.: Linear scan register allocation. ACM Transactions on Programming Languages and Systems (TOPLAS) 21(5), 895–913 (1999)
29. Robertson, N., Seymour, P.D.: Graph Minors. I. Excluding a Forest. Journal of Combinatorial Theory, Series B 35(1), 39–61 (1983)
30. Scholz, B., Eckstein, E.: Register Allocation for Irregular Architectures. SIGPLAN Notices 37(7), 139–148 (2002)

31. Smith, M.D., Ramsey, N., Holloway, G.: A Generalized Algorithm for Graph-Coloring Register Allocation. In: PLDI 2004: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, pp. 277–288. Association for Computing Machinery (2004)
32. Thorup, M.: All Structured Programs Have Small Tree Width and Good Register Allocation. Information and Computation 142(2), 159–181 (1998)
33. Weicker, R.P.: Dhrystone: a synthetic systems programming benchmark. Communications of the ACM 27, 1013–1030 (1984)
34. Weicker, R.P.: Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules. SIGPLAN Notices 23, 49–62 (1988)
35. Yannakakis, M., Gavril, F.: The maximum k-colorable subgraph problem for chordal graphs. Information Processing Letters 24(2), 133–137 (1987)

Optimal and Heuristic Global Code Motion for Minimal Spilling

Gergö Barany and Andreas Krall[*]

Vienna University of Technology
{gergo,andi}@complang.tuwien.ac.at

Abstract. The interaction of register allocation and instruction scheduling is a well-studied problem: Certain ways of arranging instructions within basic blocks reduce overlaps of live ranges, leading to the insertion of less costly spill code. However, there is little previous research on the extension of this problem to global code motion, i .e., the motion of instructions between blocks. We present an algorithm that models global code motion as an optimization problem with the goal of minimizing overlaps between live ranges in order to minimize spill code.

Our approach analyzes the program to identify the live range overlaps for all possible placements of instructions in basic blocks and all orderings of instructions within blocks. Using this information, we formulate an optimization problem to determine code motions and partial local schedules that minimize the overall cost of live range overlaps. We evaluate solutions of this optimization problem using integer linear programming, where feasible, and a simple greedy heuristic.

We conclude that global code motion with the sole goal of avoiding spills rarely leads to performance improvements because code is placed too conservatively. On the other hand, purely local optimal instruction scheduling for minimal spilling is effective at improving performance when compared to a heuristic scheduler for minimal register use.

1 Introduction

In an optimizing compiler's backend, various code generation passes apply code transformations with different, conflicting goals in mind. The register allocator attempts to assign the values used by the program to CPU registers, which are usually scarce. Where there are not enough registers, *spilling* must be performed: Excess values must be stored to memory and reloaded before they can be used. Memory accesses are slower than most other instructions, so avoiding spill code is usually beneficial on modern architectures [GYA+03]. This paper investigates the applicability of this result to global code motion directed at minimizing live range overlaps.

[*] This work is supported by the Austrian Science Fund (FWF) under contract P21842, *Optimal Code Generation for Explicitly Parallel Processors*, http://www.complang.tuwien.ac.at/epicopt/

```
start:
    i0 := 0
    a := read()

loop:
    i1 := ϕ(i0, i2)
    b := a + 1
    c := f(a)
    i2 := i1 + b
    d := i2 × 2
    compare i2 < c
    blt loop
end:
    return d
```

(a) Original function

```
start:
    i0 := 0
    a := read()
    b := a + 1
loop:
    i1 := ϕ(i0, i2)
    c := f(a)
    i2 := i1 + b
    compare i2 < c
    blt loop


end:
    d := i2 × 2
    return d
```

(b) After GCM

```
start:
    i0 := 0
    a := read()

loop:
    i1 := ϕ(i0, i2)
    b := a + 1
    i2 := i1 + b
    c := f(a)
    compare i2 < c
    blt loop

end:
    d := i2 × 2
    return d
```

(c) GCMS for 3 registers

Fig. 1. Optimization using GCM and GCMS for a three-register processor

Register allocation and spilling conflict with transformations that lengthen a value's *live range*, the set of all program points between the value's definition and a subsequent use: *Instruction scheduling* arranges instructions within basic blocks. To maximize pipeline utilization, definitions and uses of values can be moved apart by scheduling other instructions between them, but this lengthens live ranges and can lead to more overlaps between them, which can in turn lead to excessive register demands and insertion of spill code. Similarly, *code motion* techniques that move instructions between basic blocks can increase performance, but may also lead to live range overlaps. In particular, loop-invariant code motion moves instructions out of loops if their values do not need to be computed repeatedly. However, for a value defined outside a loop but used inside the loop, this extends its live range across the entire loop, leading to an overlap with all of the live ranges inside the loop.

This paper introduces our GCMS (global code motion with spilling) algorithm for integrating the code motion, instruction scheduling, and spilling problems in one formalism. The algorithm is 'global' in the sense that it considers the entire function at once and allows code motion into and out of loops. We describe both heuristic GCMS and an integer linear programming formulation for an optimal solution of the problem.

Our algorithm is based on Click's aggressive Global Code Motion (GCM) algorithm [Cli95]. We use an example to illustrate the differences between GCM and GCMS. Figure 1(a) shows a small program in SSA form adapted from the original paper on GCM. It is easy to see that the computation of variable `b` is loop-invariant and can be hoisted out of the loop; further, the computation for `d` is not needed until after the loop. Since the value of its operand `i2` is available at the end of the loop, we can sink this multiplication to the `end` block. Figure 1(b) illustrates both of these code motions, which are automatically performed by GCM. The resulting program contains less code in the loop, which means it can be expected to run faster than the original.

This expectation fails, however, if there are not enough registers available in the target processor. Since after GCM variable `b` is live through the loop, it conflicts with `a`, `c`, and all of $\{\texttt{i0}, \texttt{i1}, \texttt{i2}\}$. Both `a` and `c` conflict with each other and with at least one of the `i` variables, so after GCM we need four CPU registers for a spill-free allocation. If the target only has three registers available for allocation of this program fragment, costly spill code must be inserted into the loop. As memory accesses are considerably more expensive than simple arithmetic, GCM would trade off a small gain through loop invariant code motion against a larger loss due to spilling.

Compare this to Figure 1(c), which shows the result of applying our GCMS algorithm for a three-register CPU. To avoid the overlap of `b` with all the variables in the loop, GCMS leaves its definition inside the loop. It also applies another change to the original program: The overlap between the live ranges of `b` and `c` is avoided by changing the instruction schedule such that `c`'s definition is after `b`'s last use. This ensures that a register limit of 3 can be met. However, GCMS is not fully conservative: Sinking `d` out of the loop can be done without adversely affecting the register needs, so this code motion is performed by GCMS. Note, however, that this result is specific to the limit of three registers: If four or more registers were available, GCMS would detect that unrestricted code motion is possible, and it would produce the same results as GCM in Figure 1(b).

The idea of GCMS is thus to perform GCM in a way that is more sensitive to the needs of the spiller. As illustrated in the example, code motion is restricted by the spilling choices of the register allocator, but only where this is necessary. In functions (or parts of functions) where there are enough registers available, GCMS performs unrestricted GCM. Where there is higher register need, GCMS serializes live ranges to avoid overlaps and spill fewer values. In contrast to most other work in this area, GCMS does not attempt to estimate the register needs of the program before or during scheduling. Instead, it computes a set of promising code motions that *could* reduce register needs if necessary. An appropriately encoded register allocation problem ensures that the spiller chooses which code motions are actually performed. Code motions that are not needed to avoid spilling are not performed.

The rest of this paper is organized as follows. Section 2 discusses related work in the areas of optimal instruction scheduling, optimal spilling, and integrated scheduling and register allocation techniques. Section 3 gives an overview of our GCMS algorithm. Section 4 describes our analysis for identifying all possible live range overlaps, and how to apply code motion and scheduling to avoid them. Section 5 discusses the problem of selecting a subset of possible overlaps to avoid and shows our integer linear programming (ILP) model for computing an optimal solution. Section 6 evaluates our implementation and compares the effects of heuristic and optimal GCMS to simpler existing heuristics implemented in the LLVM compiler suite. Section 7 concludes.

2 Related Work

Instruction scheduling for pipelined architectures was an early target for optimal approaches [EK91, WLH00]. The goal of such models was usually to optimize for minimal total schedule length only. However, both the increasing complexity of modern hardware and the increasing gap between processor and memory speeds made it necessary to consider register needs as well. An optimal integrated formulation was given by Chang et al.[CCK97]. More recently, Govindarajan et al. [GYA+03] concluded that on modern out-of-order superscalar processors, scheduling to minimize spilling appears to be the most profitable instruction scheduling target. Various optimal spillers for a given arrangement of instructions have also been proposed. Colombet et al. [CBD11] summarize and generalize this work.

A large body of early work in integrated instruction scheduling and register allocation [GH88, NP93, Pin93, AEBK94] aimed at balancing scheduling for instruction level parallelism against register allocation, with some spilling typically allowed. As with optimal schedulers, this trend also shifted towards work that attempted to avoid live range overlaps entirely, typically by adding sequencing arcs to basic block dependence graphs, as our GCMS algorithm also does [Tou01, XT07, Bar11].

All of the work mentioned above considers only local instruction scheduling within basic blocks, but no global code motion. At an intermediate level between local and global approaches, software pipelining [Lam88] schedules small loop kernels for optimal execution on explicitly parallel processors. Here, too, careful integration of register allocation has proved important over time [CSG01, EK12].

RASER [NP95b] performs register allocation sensitive region scheduling by first using rematerialization (duplication of computations) to reduce register pressure where needed, and then only applying global code motion operations (such as loop-invariant code motion) that do not increase register pressure beyond the number of available registers. No attempt is made to reduce register pressure by serializing registers as in our approach. RASER gives impressive improvements on machines with artificially few registers; later results on a machine with 16 registers look similar to ours [NP95a].

Johnson and Mycroft [JM03] describe an elegant combined global code motion and register allocation method based on the Value State Dependence Graph (VSDG). The VSDG is similar to our acyclic global dependence graph, but it represents control flow by using special nodes for conditionals and reducible loops (their approach does not handle irreducible loops) rather than our lists of legal blocks for each instruction. The graph is traversed bottom-up in a greedy manner, measuring ‘liveness width’, the number of registers needed at each level. Excessive register pressure is reduced by adding dependence arcs, by spilling values, or by duplicating computations. Unfortunately, we are not aware of any data on the performance of this allocator, nor the quality of the generated code.

The concept of liveness width is similar to Touati’s ‘register saturation’, which is only formulated for basic blocks and pipelined loops. It is natural to try to adapt this concept to general control flow graphs, but this is difficult to do if instructions may move between blocks and into and out of loops. It appears that to compute saturation, we would need to build a detailed model of where each value may be live, and this might quickly lead to combinatorial explosion. Our method is simpler because it tries to minimize spills without having to take a concrete number of available registers into account.

Many authors have worked on what they usually refer to as global instruction scheduling problems, but their solutions are almost invariably confined to acyclic program regions, i. e., they do not perform loop invariant code motion [BR91, ZJC03]. The notable exception is work by Winkel [Win07] on 'real' global scheduling including moving code into and out of loops, as our algorithm does. Crucially, Winkel's optimal scheduler runs in two phases, the second of which has the explicit goal of limiting code motion to avoid lengthening live ranges too much. Besides considerable improvements in schedule length, Winkel reports reducing spills by 75 % relative to a heuristic global scheduler. In contrast to our work, Winkel compiled for the explicitly parallel Itanium processor, so his reported speedups of 10 % cannot be meaningfully compared to our results on our out-of-order target architecture (ARM Cortex-A9).

3 Global Code Motion for Minimal Spilling

As an extension of GCM, our GCMS algorithm is also based on a program representation in SSA form and a global dependence graph. In SSA form [CFR+91], every value in the program has exactly one definition. Definitions from different program paths, such as in loops, are merged using special ϕ pseudo-instructions that are later replaced by register copies if necessary.

Like GCM, we build a global dependence graph with instructions as nodes and data dependences as arcs. We also add arcs to capture any ordering dependences due to side effects on memory, such as store instructions and function calls, and any instructions that explicitly access processor registers, such as moves to and from argument registers around calls. Because we will use this dependence graph to find a linear arrangement of instructions within basic blocks, we must impose an important restriction: The graph must always be acyclic to ensure that a topological ordering exists. Conveniently, in SSA form the only cyclic data dependences arise from values that travel along loop backedges to ϕ instructions. We can safely ignore these dependences as long as we ensure that the definitions of such values are never sunk out of their loops.

Our dependence graph is also used for code motion. We associate each instruction with a set of legal basic blocks as follows: Function calls, any other instructions that access memory or explicit CPU registers, and ϕ instructions are not movable, their only legal block is the block in which they originally appear. For all other blocks, we employ the concept of dominance: A block a *dominates* a block b iff any path from the function's unique entry block to b must pass through a. Following Click [Cli95], we say that an instruction is legally placed in a block b if all of its predecessors in the dependence graph are based in blocks that dominate b, and all of its successors are in blocks dominated by b. The set of legal blocks for every instruction is computed in a forward and a backward pass over the dependence graph.

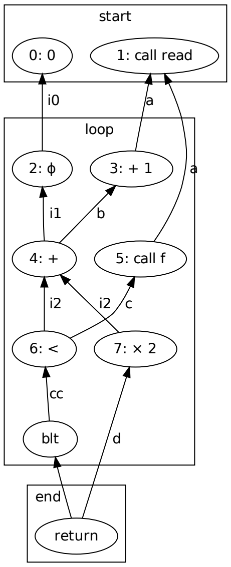


Fig. 2. Global dependence graph for example program

Figure 2 shows the global dependence graph for the example program from Figure 1. Data dependences are annotated with the name of the corresponding value. Note that the cyclic dependence of the ϕ instruction on `i2` is not shown; it is implicit, and our analysis keeps track of the fact that `i2` is live out of the `loop` block and live across the loop's backedge.

Recall that function calls, ϕ instructions, and branches are not movable in our model. Due to dependence arcs, other instructions become unmovable, too: Instructions 4 and 6 are 'stuck' between the unmovable ϕ and the branch. This leaves only instructions 3 and 7 movable into and out of the loop, but as we have seen before, this is enough to illustrate interesting interactions between global code motion and spilling.

Given the dependence graph and legal blocks for each instruction, GCMS proceeds in the following steps:

Overlap analysis determines for every pair of values whether their live ranges might overlap. The goal of this analysis is similar to traditional liveness analysis for register allocation, but with the crucial difference that in GCMS, instructions may move. Our overlap analysis must therefore take every legal placement and every legal ordering of instructions within blocks into account. For every pair, the analysis determines whether the ranges definitely overlap in all schedules, never overlap in any schedule, or whether they might overlap for some arrangements of instructions. In the latter case, GCMS computes a set of code placement restrictions and extra arcs that can be added to the global dependence graph. Such restrictions ensure that the live ranges do not overlap in any schedule of the new graph, i. e., they enable *reuse* of the same processor register for both values.

Candidate selection chooses a subset of the avoidable overlaps identified in the previous phase. Not all avoidable overlaps identified by the analysis are avoidable *at the same time*: If avoiding overlaps for two register pairs leads to conflicting code motion restrictions, such as moving an instruction to two different blocks, or adding arcs that would cause a cycle in the dependence graph, at least one of the pairs cannot be chosen for reuse. GCMS must therefore choose a promising set of *candidates* among all avoidable overlaps. Only these candidate pairs will be considered for actual overlap avoidance by code motion and instruction scheduling.

Since our goal is to avoid expensive spilling as far as possible, we try to find a candidate set that maximizes the sum of the spill costs of every pair of values selected for reuse.

Spilling and code motion use the results of the candidate selection phase by building a register allocation problem in which the live ranges of reuse candidates are treated as non-conflicting. The solution computed by the register allocator is then used to guide code motion: For any selected candidate whose live ranges were allocated to the same CPU register, we apply its code motion restrictions to the dependence graph. The result of this phase is a restricted graph on which we can perform standard GCM, with the guarantee that code motion will not introduce excessive overlaps between live ranges.

Each of these phases is discussed in more depth in the following sections.

4 Overlap Analysis

An optimal solution to GCMS requires us to consider all possible ways in which a pair of values might overlap. That is, we must consider all possible placements and orderings of all of the instructions defining or using either value. To keep this code simple, we implemented this part of the analysis in Prolog. This allows us to give simple declarative specifications of when values overlap, and Prolog's built-in backtracking takes care of actually enumerating all configurations.

The core of the overlap analysis, simplified from our actual implementation, is sketched in Figure 3. The Figure shows the three most important cases in the analysis: The first clause deals with the case where values ('virtual registers') A and B might overlap because A's use is in the same block as B's definition, but there is no dependence ensuring that A's live range ends before B's definition. The second clause applies when A is defined and used in different blocks, and B's definition might be placed in an intervening block between A's definition and use. The third clause handles the case where A and B are live at the end of the same block because they are both defined there and used elsewhere.

The code uses three important auxiliary predicates for its checks:

cfg_forward_path(A, B) succeeds if there is a path in the control flow graph from block A to block B using only forward edges, i. e., not taking loop backedges into account.

```
overlapping_virtreg_pair(virtreg(A), virtreg(B)) :-
    % B is defined by instruction BDef in BDefBlock, A has a use in
    % the same block.
    virtreg_def_in(B, BDef, BDefBlock),
    virtreg_use(A, AUse, BDefBlock),
    % A's use is not identical to B's def, and there is no existing
    % dependence from B's def to A's use. That is, B's def might be
    % between A's def and use.
    AUse \= BDef,
    no_dependence(BDef, AUse),
    % There is an overlap that might be avoided if B's def were
    % scheduled after A's use by adding an arc.
    Placement = [AUse-BDefBlock, BDef-BDefBlock],
    record_blame(A, B, blame(placement(Placement), no_arc([BDef-AUse]))).

overlapping_virtreg_pair(virtreg(A), virtreg(B)) :-
    % A and B have defs ADef and BDef in blocks ADefBlock and
    % BDefBlock, respectively.
    virtreg_def_in(A, ADef, ADefBlock),
    virtreg_def_in(B, BDef, BDefBlock),
    % A has a use in a block different from its def.
    virtreg_use(A, AUse, AUseBlock),
    ADefBlock \= AUseBlock,
    % There is a non-empty path from A's def to B's def...
    ADefBlock \= BDefBlock,
    cfg_forward_path(ADefBlock, BDefBlock),
    % ... and a path from B's def to A's use that does not pass
    % through a redefinition of A. That is, B is on a path from A's
    % def to its use.
    cfg_loopypath_notvia(BDefBlock, AUseBlock, ADefBlock),
    % There is an overlap that might be avoided if at least one of
    % these instructions were in a different block.
    Placement = [ADef-ADefBlock, BDef-BDefBlock, AUse-AUseBlock],
    record_blame(A, B, blame(placement(Placement))).

overlapping_virtreg_pair(virtreg(A), virtreg(B)) :-
    % A and B are defined in the same block.
    virtreg_def_in(A, ADef, DefBlock),
    virtreg_def_in(B, BDef, DefBlock),
    % A has a use in a different block, so it is live out of
    % its defining block.
    virtreg_use(virtreg(A), AUse, AUseBlock),
    AUseBlock \= DefBlock,
    % B is also live out.
    virtreg_use(virtreg(B), BUse, BUseBlock),
    BUseBlock \= DefBlock,
    % There is an overlap that might be avoided if at least
    % one of these instructions were in a different block.
    Placement = [ADef-DefBlock, BDef-DefBlock,
                 AUse-AUseBlock, BUse-BUseBlock],
    record_blame(A, B, blame(placement(Placement))).
```

Fig. 3. Overlap analysis for virtual registers A and B

cfg_loopypath_notvia(A, B, C) succeeds if there is a path, possibly including loops, from A to B, but not including C. We use this to check for paths lacking a redefinition of values.

no_dependence(A, B) succeeds if there is no existing arc in the dependence graph from instruction A to B, but it could be added without causing a cycle in the graph.

If all of the conditions in the clause bodies are satisfied, a possible overlap between the values is recorded. Such overlaps are associated with 'blame terms', data structures that capture the reason for the overlap. For any given pair of values, there might be several different causes for overlap, each associated with its own blame. An overlap can be avoided if all of the circumstances captured by the blame terms can be avoided.

There are two kinds of blame. First, there are those blames that record arcs missing from the dependence graph, computed as in the first clause in Figure 3. If this arc can be added to the dependence graph, B's definition will be after A's use, avoiding this overlap. Alternatively, if these two instructions are *not* placed in the same block, the overlap is also avoided. The second kind of blame concerns only the placement of instructions in basic blocks, as in the second and third clauses in Figure 3. If all of the instructions are placed in the blocks listed in the blame term, there is an overlap between the live ranges. If at least one of them is placed in another block, there is no overlap—at least, not due to *this* placement.

As mentioned before, we use Prolog's backtracking to enumerate all invalid placements and missing dependence arcs. We collect the associated blame terms and check them for validity: If any of the collected arcs to put a value v before w can not be added to the dependence graph because it would introduce a cycle, then the other arcs for putting v before w are useless, so all of these blames are deleted. Blames for the reversed ordering, scheduling w before v, are retained because they might still be valid.

Even after this cleanup we might end up with an overlap that cannot be avoided. For example, for the pair `a` and `b` in the example program, the analysis computes that instruction 3 defining `b` may not be placed in the start block because it would then be live out of that block and overlap with `a`'s live-out definition; but neither may instruction 3 be placed in the loop block because it would be on a path from `a`'s definition to its repeated use in the loop. As these two blocks are the only ones where instruction 3 may be placed, the analysis of all blames for this pair determines that an overlap between `a` and `b` cannot be avoided. Table 1 shows the blame terms computed for the example program after these checks and some cleanup (removal of unmovable instructions from placement blames). Each blame is accompanied by a brief explanation of why the live ranges would overlap if placed and arranged as stated. Pairs not listed here are found to be either non-overlapping or definitely overlapping.

Table 1. Blame terms computed for the example program, listing instruction placements and missing dependence arcs that may cause overlaps

Pair	Invalid placements	Missing arcs	Explanation
`a`, `d`	7 in `loop`		`a` live through `loop`
`b`, `d`	3 in `start`, 7 in `loop`		`b` live through `loop` if defined in `start`
`b`, `i0`	3 in `start`		`b` live out of `start` if defined in `start`
`b`, `i2`	3 in `start`		`b` live through `loop` if defined in `start`
`c`, `d`	7 in `loop`	$7 \to 6$	order `d`'s definition after `c`'s last use
`c`, `i1`		$5 \to 4$	order `c`'s definition after `i1`'s last use
`d`, `i2`	7 in `loop`		`i2` live out of `loop` (across backedge)

The analysis discussed so far only considers pairs of values in SSA form. However, we must also consider overlaps between SSA values and explicitly named CPU registers, such as argument registers referenced in copy instructions before function calls. As such copies can occur in several places in a function, these physical registers are not in SSA form. We assume a representation in which all uses of such registers are in the same block as their definition; this allows us to treat each of these short live ranges separately, and the analysis becomes similar to the case for virtual registers.

5 Reuse Candidate Selection

After performing overlap analysis and computing all blames, a subset of reuse candidates must be selected for reuse.

5.1 Integer Linear Programming Formulation

The optimization problem we must solve is finding a nonconflicting set of reuse candidates with maximal weight, where the weight is the sum of the spill costs of the two values. That is, of all possible overlaps, we want to avoid those that would lead to the largest total spill costs. We model this as an integer linear program and use an off-the-shelf solver (CPLEX) to compute an optimum.

Variables. The variables in the problem are:

- a binary variable $select_c$ for each reuse candidate c; this is 1 iff the candidate is selected
- a binary variable $place_{i,b}$ for each legal block b for any instruction i occurring in a placement constraint in any blame; this is 1 iff it is legal to place i in b in the optimal solution
- a binary variable $arc_{i,j}$ for any dependence arc $i \to j$ occurring in any blame; this is 1 iff the arc must be present in the optimal solution
- a variable $instr_i$ for each instruction in the program, constrained to the range $0 \leq instr_i < N$ where N is the total number of instructions; these are used to ensure that the dependence graph for the optimal solution does not contain cycles

Objective Function. We want to maximize the weight of the selected candidates; as a secondary optimization goal, we want to preserve as much freedom of code motion as possible for a given candidate selection. The objective function is therefore

$$\text{maximize} \sum_c w_c select_c + \sum_i \sum_b place_{i,b}$$

where the first sum ranges over all candidates c, w_c is the weight of candidate c, and the second sum ranges over all placement variables for instructions i and their legal blocks b. In our problem instances, there are typically considerably more candidate selection variables than placement variables, and the candidate weights are larger than 1. Thus the first sum dominates the second, and this objective function really treats freedom of code motion as secondary to the avoidance of overlaps. However, adding weights to the second sum would easily enable us to investigate trade-offs between avoiding spills and more aggressive code motion. We intend to investigate this trade-off in future work.

Constraints. The constraints in equations (1)–(7) ensure a valid selection. First, we give the constraints that model the structure of the existing dependence graph. We need this to detect possible cycles that would arise from selecting an invalid set of arcs. Therefore, we give a partial ordering of instructions that corresponds to dependences in the graph. For each instruction i with a direct predecessor p, the following must hold:

$$instr_i > instr_p \tag{1}$$

Next, we require that all instructions must be placed in some legal block. For each such instruction i:

$$\sum place_{i,b} \geq 1 \tag{2}$$

where the sum ranges over all valid blocks b for instruction i.

We can now proceed to give the constraints related to selecting a reuse candidate. For a candidate c and each of the arcs $i \to j$ associated with it, require

$$select_c + place_{i,b} + place_{j,b} \leq 2 + arc_{i,j} \tag{3}$$

to model that if c is selected and both i and j are placed in some common block b, the arc must be selected as well. For each invalid placement constraint ($instr_{i_1}$ in $b_1, \ldots, instr_{i_n}$ in b_n), require:

$$select_c + \sum place_{i,b} \leq n \tag{4}$$

This ensures that if c is selected, at least one of these placements is *not* selected.

If an arc is to be selected due to one of the candidates that requires it, ensure that it can be added to the dependence graph without causing a cycle. That is, we want to formulate the condition $arc_{i,j} \Rightarrow instr_i > instr_j$. If N is the total number of instructions, this constraint can be written as:

$$instr_i - instr_j > N \cdot arc_{i,j} - N \tag{5}$$

If $arc_{i,j}$ is selected, this reduces to $instr_i - instr_j > 0$, i. e., $instr_i > instr_j$. Otherwise, it is $instr_i - instr_j > -N$, which is always true for $0 \leq instr_i, instr_j < N$. These constraints ensure that the instructions along every path in the dependence graph are always topologically ordered, i. e., there is no cycle in the graph.

Finally, we must take interactions between dependence arcs and instruction placement into account. An arc $instr_i \rightarrow instr_j$ means that $instr_j$ may not be executed after $instr_i$ along a program path, so it is not valid to place $instr_j$ into a later block than $instr_i$. Therefore, for all arcs $instr_i \rightarrow instr_j$ in the original dependence graph where $instr_i$ may be placed in some block b_i, $instr_j$ may be placed in block b_j, and there is a non-empty forward path from b_j to b_i, require

$$place_{i,b_i} + place_{j,b_j} \leq 1 \tag{6}$$

to ensure that such a placement is not selected.

Similarly, for every selectable arc $arc_{i,j}$ and an analogous invalid path:

$$arc_{i,j} + place_{i,b_i} + place_{j,b_j} \leq 2 \tag{7}$$

That is, selecting an arc means that we also ensure that the placement of instructions respects the intended ordering.

5.2 Greedy Heuristic Solution

Solving integer linear programming problems is NP-hard. While very powerful solvers exist, many problems cannot be solved optimally within reasonable time limits. We therefore complement our optimal solver with a greedy heuristic solver. This solver inspects reuse candidates one by one and commits to any candidate that it determines to be an avoidable overlap. Committing to a candidate means immediately applying its instruction placement constraints and dependence arcs; this ensures that the candidate will definitely remain avoidable, but it restricts freedom of code motion for subsequent candidates.

Due to this greedy behavior, it is important to process candidates in an order that maximizes the chance to pick useful candidates early on. Since a live range's spill weight is a measure of how beneficial it is to keep the live range in a register, we want to avoid as many overlaps between live ranges with large weights as possible. We therefore order our candidates by decreasing weight before applying the greedy solver. As a small exception, we have found it useful to identify very short live ranges with a single use in the same block as the definition in the original program. We order these after all the other ranges because we have found that committing to such a very short range too early often destroys profitable code motion possibilities.

5.3 Spilling and Code Motion

Regardless of whether candidate selection was performed optimally or heuristically, GCMS finally moves on to the actual spilling phase. We use a spiller based on the PBQP formalism [SE02, HS06]. For the purposes of this paper, PBQP is simply a generalization of graph coloring register allocators [Cha82]. The difference is that nodes and edges in the conflict graph are annotated with weights modeling spill costs and register use constraints such as register pairing or aliasing.

We build a conflict graph with conflict edges both for register pairs whose live ranges definitely overlap as well as register pairs that were *not* selected by the candidate selection process. Conversely, any pair that was selected for reuse can be treated as non-conflicting, so we need not insert conflict edges for these. In fact, we want to ensure that any overlap that *can* be avoided actually *is* avoided by the register allocator. Using PBQP's edge cost matrices, we can ensure this by adding an edge with costs of some very small ϵ value between any pair of registers that we want to be allocated to different registers. The PBQP problem solver then tries to find an allocation respecting all of these constraints.

If the solver does not find a valid allocation, it returns some values to spill or rematerialize; we perform these actions and then rerun the entire algorithm on the modified function. When an allocation is found, we perform our actual code motion: In the final schedule, live ranges selected for reuse may not overlap. We therefore inspect all selected candidate pairs to see if they were allocated to the same CPU register. If so, we must restrict code motion and add ordering arcs to the dependence graph as specified by the pair's blame term. For selected candidate pairs that were *not* allocated to the same register, we do not need to do anything. Thus, GCMS balances the needs of the spiller with aggressive global code motion: The freedom to move code is only restricted if this is really needed to avoid spills, but not otherwise.

After applying all the needed constraints, we simply perform unmodified GCM on the resulting restricted dependence graph: Instructions are placed in their latest possible blocks in the shallowest loop nest. This keeps instructions out of loops as far as possible, but prefers to shift them into conditionally executed blocks.

6 Experimental Evaluation

We have implemented optimal and heuristic GCMS in the LLVM compiler framework's backend. Since LLVM's native frontend, Clang, only handles C and C++, we use GCC as our frontend and the Dragonegg GCC plugin to generate LLVM intermediate code from GCC's internal representation. This allows us to apply our optimization to Fortran programs from the SPEC CPU 2000 benchmark suite as well. Unfortunately, our version of Dragonegg miscompiles six of the SPEC benchmarks, but this still leaves us with 20 benchmarks to evaluate. We generate code for the ARM Cortex-A9 architecture with VFP3 hardware floating point support and use the `-O3` optimization flag to apply aggressive optimizations both at the intermediate code level and in the backend.

The overlap analysis was implemented using SWI-Prolog, and we use CPLEX as our ILP solver. Whenever CPLEX times out on a problem, we inspect the best solution it has found up to that point; if its overlap weight is lower than the weight in the prepass schedule, we use this approximation, and otherwise fall back to the prepass schedule.

The greedy heuristic solver could in principle be based on the Prolog analysis as well, but we simply continue using our old C++ implementation. However, comparing it to the Prolog implementation of essentially the same analysis helped us considerably in finding bugs in both.

6.1 Solver Time

We ran our compiler on the SPEC CPU 2000 benchmark suite, applying the optimal GCMS algorithm on all functions of 1000 or fewer instructions (this includes more than 97 % of all functions in the suite). CPLEX was set to run with a time limit of 60 seconds of wall clock time per problem. CPLEX automatically runs as many parallel threads as is appropriate for the hardware platform and can almost fully utilize the 8 cores on our Xeon CPU, so the timeout corresponds to typically about 6 to 8 minutes of CPU time. The entire build of our 20 benchmarks with optimal GCMS takes 18 hours of wall clock time.

Figure 4 shows a scatterplot of CPLEX solver times relative to the number of instructions. Marks at or very near the 60 second line are cases where the solver reached its time limit and did not return a provably optimal result. We can see that the majority of problems is solved very quickly. It is difficult to pinpoint a general trend, although obviously solving the optimization problem for larger functions tends to take longer. Note, however, that there are some quite small functions, some even with fewer than 100 instructions, where CPLEX does not terminate within the time limit. Inspection of such cases shows that this typically happens in functions containing relatively large basic blocks with much scheduling freedom. In the ILP problems for such functions, there are many *arc* variables to consider. These affect the values of the *instr* variables, which have large domains, and we believe that this may be one of the reasons CPLEX is having difficulties in exploring the search space efficiently. Nevertheless, we are able to solve 5287 of 5506 instances (96 %) optimally, and 4472 of these (81 % overall) even within a single second.

We do not report times for the Prolog overlap analysis separately, but the overall distribution is very similar to the one in Figure 4. All blames for almost all of the functions are computed within a few seconds. Functions with a large number of basic blocks and many paths in the CFG may take longer due to the combinatorial explosion of taking all instruction placements into account. We run the overlap analysis with a 60 second time limit and fall back to the C++ heuristics if the limit is exceeded, but this only happens rarely.

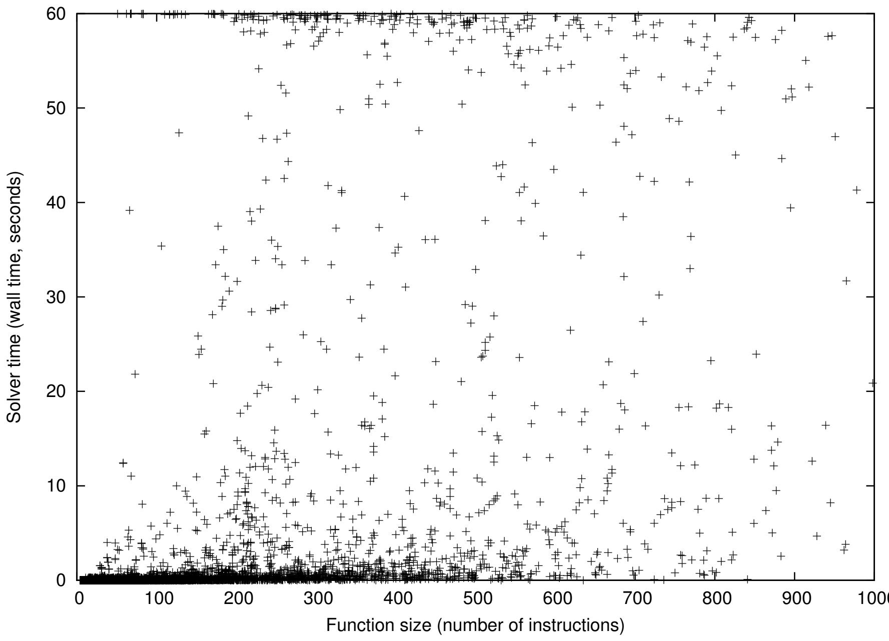


Fig. 4. Scatterplot of CPLEX solver times relative to function size

6.2 Execution Time Statistics

Table 2 shows a comparison of the execution times of our benchmark programs, compiled using five different code generation methods. Baseline is the configuration using LLVM's code motion pass which attempts to perform loop invariant code motion without exceeding a heuristically determined register usage limit. Within blocks, instructions are scheduled using a list scheduler that attempts to minimize live range lengths. The Baseline configuration is thus a good representative of modern optimizing compilers.

Heuristic GCMS is our GCMS algorithm, always using the greedy heuristic solver described in Section 5.2. Optimal GCMS uses the optimal ILP formulation for functions of up to 1000 instructions with a solver time timit of 60 seconds, as discussed above. All three configurations use the same spiller based on a PBQP formulation and using LLVM's near-optimal PBQP solver.

The 'Local' variants are also GCMS, but restricted by treating every instruction as unmovable. Thus the Local algorithm only performs instruction scheduling within the blocks LLVM chose for each instruction. We evaluate two Local variants, one with the greedy heuristic scheduler and one with the optimal ILP formulation of the candidate selection problem, as before. Each time shown in the table is CPU time, obtained as the minimum of timing five runs of each benchmark in each configuration. Additionally, the GCMS and Local variants are shown normalized to Baseline.

Table 2. Execution time statistics for SPEC CPU 2000 benchmark programs, in seconds and relative to Baseline

Benchmark	Baseline	GCMS				Local			
		Heuristic		Optimal		Heuristic		Optimal	
164.gzip	62.82	60.25	0.96	60.31	0.96	60.66	0.97	60.68	0.97
168.wupwise	60.25	59.96	1.00	60.18	1.00	60.30	1.00	60.03	1.00
171.swim	31.84	31.89	1.00	31.85	1.00	31.49	0.99	31.76	1.00
172.mgrid	50.87	52.85	1.04	52.25	1.03	53.08	1.04	51.58	1.01
173.applu	31.00	31.31	1.01	31.24	1.01	31.35	1.01	31.21	1.01
175.vpr	40.72	40.80	1.00	40.71	1.00	40.71	1.00	40.47	0.99
177.mesa	58.89	59.26	1.01	58.15	0.99	58.78	1.00	58.74	1.00
178.galgel	54.07	54.21	1.00	54.33	1.00	53.62	0.99	53.49	0.99
179.art	9.42	9.59	1.02	9.59	1.02	9.69	1.03	9.18	0.97
181.mcf	56.82	57.58	1.01	57.54	1.01	58.29	1.03	57.05	1.00
183.equake	40.81	41.42	1.01	41.14	1.01	42.42	1.04	40.72	1.00
187.facerec	80.42	81.99	1.02	85.91	1.07	82.71	1.03	80.80	1.00
189.lucas	79.48	79.35	1.00	79.24	1.00	79.14	1.00	78.88	0.99
197.parser	13.50	13.46	1.00	13.43	0.99	13.55	1.00	13.50	1.00
252.eon	13.63	13.34	0.98	13.64	1.00	13.80	1.01	13.47	0.99
253.perlbmk	25.12	24.42	0.97	24.27	0.97	26.28	1.05	24.64	0.98
255.vortex	15.32	15.42	1.01	15.68	1.02	15.35	1.00	15.45	1.01
256.bzip2	56.20	56.56	1.01	56.59	1.01	56.53	1.01	56.63	1.01
300.twolf	25.09	25.60	1.02	25.35	1.01	26.19	1.04	24.79	0.99
301.apsi	20.46	20.36	1.00	20.41	1.00	20.42	1.00	20.35	0.99
geometric mean			1.003		1.004		1.011		0.995

The results show interesting effects due to the comparison of global code motion and local scheduling. Our original research goal was to investigate whether the effect observed by Govindarajan et al. [GYA+03], that scheduling for minimal register use improves performance, also holds true for global code motion for minimial spilling. While we see performance improvements due to reduced spilling and more aggressive code motion in a few cases, in other cases performance degrades. These regressions are due to code motions that do reduce spills but at the same time move instructions to unfavorable basic blocks. We conclude that although GCMS is careful to restrict the schedule only where this is absolutely necessary to avoid spilling, such restrictions can still have an unfavorable impact on overall performance in a number of cases. On average, both heuristic and optimal GCMS produce code with comparable performance to LLVM's simpler heuristics.

This is different for purely local scheduling for minimal spilling: Here, our optimal variant is often better than LLVM's scheduling heuristic, which also has the goal of reducing spills by shortening live ranges. On average, we achieve an improvement of about 0.5 %. This local result shows that while LLVM is already close to optimal, there is still potential to improve the code produced by its state-of-the-art heuristics, and we believe that more careful global code motion operations can be even more beneficial. We noted in Section 5 that our optimal

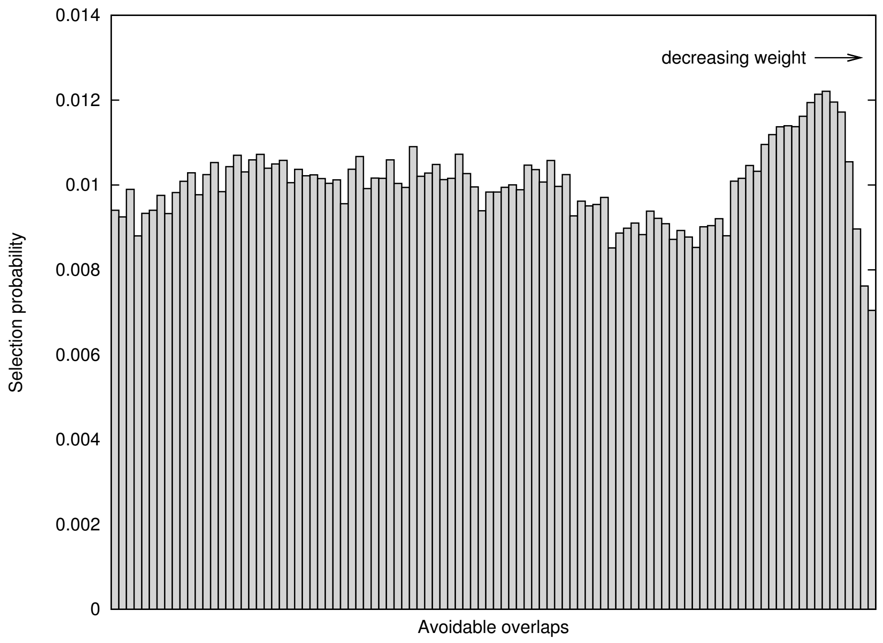


Fig. 5. Distribution of selected candidates by weight

algorithm's objective function can easily be extended to balance avoidance of spills against freedom of code motion. As future work, we intend to evaluate this design space to find a better tradeoff between global code motion and scheduling to optimize program performance.

6.3 Comparison of Optimal Selection vs. Greedy Heuristics

The quality of our greedy heuristic solution depends on the ordering of register pairs. If we managed to select an ordering in which all the reuse candidates in the optimal solution come first, applying the greedy algorithm would produce the optimal solution. The idea behind our ordering by decreasing weight is to select the most expensive pairs, which make the biggest difference in spilling quality, as early as possible.

To gain insight into whether this is a good idea, we look at the distribution of candidates actually selected by the optimal solver. If this distribution is skewed towards candidates of larger weight, this could be a hint that our ordering by decreasing weight is a sound approach; otherwise, analyzing the distribution might suggest better alternatives.

Figure 5 shows a histogram representing the distribution of reuse candidates in the optimal solutions we found. We obtained this figure by producing a 0-1 'selection vector' for each set of candidates ordered by decreasing weight, where a 0 entry represents 'not selected', and a 1 represents 'selected'. We divided each selection vector into 100 buckets of equal size and computed the population count (number of 1s) normalized by bucket size for each bucket. The histogram is the sum of all of these normalized vectors.

Apart from the small peak towards the lower end of the weight scale, the distribution of the selected candidates is quite even. Thus this analysis does not suggest a good weight-based ordering for the greedy analysis, although starting with a certain subset of lower-weight pairs might result in a slightly better overall selection than with our current ordering by decreasing weight.

7 Summary and Conclusions

In this paper we presented GCMS, a global code motion algorithm for minimal spilling. In GCMS, we consider all possible overlaps between live ranges in a function, taking all possible placements of instructions and schedules within basic blocks into account. From all overlaps that can be avoided, we select a profitable candidate set to avoid. These candidates can be removed from the register allocation problem's conflict set; if a candidate is chosen for reuse of a processor register, we apply the associated changes to the dependence graph that models all code motion and scheduling possibilities. However, if enough registers are available for an allocation without spilling, we do not restrict scheduling or code motion and can aggressively move code out of loops.

We evaluate both optimal and greedy heuristic solutions of the candidate selection problem. Our evaluation shows that global code motion can reduce spilling and occasionally improve program performance, but it often performs code motions that can lead to an overall performance regression. On the other hand, restricting our optimal algorithm to perform only local instruction scheduling leads to consistent improvements in performance over a state-of-the art heuristic scheduler. Finding a restricted form of GCMS that more carefully balances the needs of the spiller against aggressive code motion is future work.

References

[AEBK94] Ambrosch, W., Ertl, M.A., Beer, F., Krall, A.: Dependence-conscious Global Register Allocation. In: Gutknecht, J. (ed.) Programming Languages and System Architectures. LNCS, vol. 782, pp. 125–136. Springer, Heidelberg (1994)

[Bar11] Barany, G.: Register reuse scheduling. In: 9th Workshop on Optimizations for DSP and Embedded Systems (ODES-9), Chamonix, France, http://www.imec.be/odes/ (April 2011)

[BR91] Bernstein, D., Rodeh, M.: Global instruction scheduling for superscalar machines. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, PLDI 1991, pp. 241–255. ACM, New York (1991)

[CBD11] Colombet, Q., Brandner, F., Darte, A.: Studying optimal spilling in the light of ssa. In: Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2011, pp. 25–34. ACM, New York (2011)

[CCK97] Chang, C.-M., Chen, C.-M., King, C.-T.: Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. In: Computers and Mathematics with Applications (1997)

[CFR$^+$91] 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)

[Cha82] Chaitin, G.J.: Register allocation & spilling via graph coloring. In: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, SIGPLAN 1982, pp. 98–105. ACM, New York (1982)

[Cli95] Click, C.: Global code motion/global value numbering. In: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, PLDI 1995, pp. 246–257 (1995)

[CSG01] Codina, J.M., Sánchez, J., González, A.: A unified modulo scheduling and register allocation technique for clustered processors. In: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques, PACT 2001, pp. 175–184. IEEE Computer Society, Washington, DC (2001)

[EK91] Ertl, M.A., Krall, A.: Optimal Instruction Scheduling using Constraint Logic Programming. In: Małuszyński, J., Wirsing, M. (eds.) PLILP 1991. LNCS, vol. 528, Springer, Heidelberg (1991)

[EK12] Eriksson, M., Kessler, C.: Integrated code generation for loops. ACM Trans. Embed. Comput. Syst. 11S(1), 19:1–19:24 (2012)

[GH88] Goodman, J.R., Hsu, W.-C.: Code scheduling and register allocation in large basic blocks. In: ICS 1988: Proceedings of the 2nd International Conference on Supercomputing, pp. 442–452. ACM, New York (1988)

[GYA$^+$03] Govindarajan, R., Yang, H., Amaral, J.N., Zhang, C., Gao, G.R.: Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures. IEEE Transactions on Computers 52(1), 4–20 (2003)

[HS06] Hames, L., Scholz, B.: Nearly Optimal Register Allocation with PBQP. In: Lightfoot, D.E., Ren, X.-M. (eds.) JMLC 2006. LNCS, vol. 4228, pp. 346–361. Springer, Heidelberg (2006)

[JM03] Johnson, N., Mycroft, A.: Combined Code Motion and Register Allocation Using the Value State Dependence Graph. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 1–16. Springer, Heidelberg (2003)

[Lam88] Lam, M.: Software pipelining: an effective scheduling technique for VLIW machines. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, PLDI 1988, pp. 318–328. ACM, New York (1988)

[NP93] Norris, C., Pollock, L.L.: A scheduler-sensitive global register allocator. In: Supercomputing 1993: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing, pp. 804–813 (1993)

[NP95a] Norris, C., Pollock, L.L.: An experimental study of several cooperative register allocation and instruction scheduling strategies. In: Proceedings of the 28th Annual International Symposium on Microarchitecture, MICRO 28, pp. 169–179. IEEE Computer Society Press, Los Alamitos (1995)

[NP95b] Norris, C., Pollock, L.L.: Register allocation sensitive region scheduling. In: Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques, PaCT 1995, pp. 1–10. IFIP Working Group on Algol, Manchester (1995)

[Pin93] Pinter, S.S.: Register allocation with instruction scheduling. In: PLDI 1993: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, pp. 248–257. ACM, New York (1993)

[SE02] Scholz, B., Eckstein, E.: Register allocation for irregular architectures. In: Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems, LCTES/SCOPES 2002, pp. 139–148. ACM, New York (2002)

[Tou01] Touati, S.A.A.: Register Saturation in Superscalar and VLIW Codes. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 213–228. Springer, Heidelberg (2001)

[Win07] Winkel, S.: Optimal versus heuristic global code scheduling. In: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 40, pp. 43–55. IEEE Computer Society, Washington, DC (2007)

[WLH00] Wilken, K., Liu, J., Heffernan, M.: Optimal instruction scheduling using integer programming. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI 2000, pp. 121–133. ACM, New York (2000)

[XT07] Xu, W., Tessier, R.: Tetris: a new register pressure control technique for VLIW processors. In: LCTES 007: Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 113–122. ACM, New York (2007)

[ZJC03] Zhou, H., Jennings, M.D., Conte, T.M.: Tree Traversal Scheduling: A Global Instruction Scheduling Technique for VLIW/EPIC Processors. In: Dietz, H.G. (ed.) LCPC 2001. LNCS, vol. 2624, pp. 223–238. Springer, Heidelberg (2003)

Efficient and Effective Handling of Exceptions in Java Points-to Analysis

George Kastrinis and Yannis Smaragdakis

Dept. of Informatics, University of Athens, Greece
{gkastrinis,smaragd}@di.uoa.gr

Abstract. A joint points-to and exception analysis has been shown to yield benefits in both precision and performance. Treating exceptions as regular objects, however, incurs significant and rather unexpected overhead. We show that in a typical joint analysis most of the objects computed to flow in and out of a method are due to exceptional control-flow and not normal call-return control-flow. For instance, a context-insensitive analysis of the Antlr benchmark from the DaCapo suite computes 4-5 times more objects going in or out of a method due to exceptional control-flow than due to normal control-flow. As a consequence, the analysis spends a large amount of its time considering exceptions.

We show that the problem can be addressed both effectively and elegantly by coarsening the representation of exception objects. An interesting find is that, instead of recording each distinct exception object, we can collapse all exceptions of the same type, and use one representative object per type, to yield nearly identical precision (loss of less than 0.1%) but with a boost in performance of at least 50% for most analyses and benchmarks and large space savings (usually 40% or more).

1 Introduction

Points-to analysis is a fundamental static program analysis. It consists of computing a static abstraction of all the data that a pointer variable (and, by extension, any pointer expression) can point to during program execution. Points-to analysis is often the basis of most other higher-level client analyses (e.g., may-happen-in-parallel, static cast elimination, escape analysis, and more). It is also inter-related with call-graph construction, since the values of a pointer determine the target of dynamically resolved calls, such as object-oriented dynamically dispatched method calls or functional lambda applications.

An important question regarding points-to analysis (as well as client analyses based on it) concerns the handling of exceptions, in languages that support exception-based control flow. The emphasis of our work is on Java—a common target of points-to analysis work—but similar ideas are likely to apply to other languages, such as C# and C++. This is an important topic because exceptional control flow cannot be ignored for several client analysis (e.g., information leak or other security analyses) and if handled crudely it can destroy the precision of the base points-to analysis.

In the past, most practical points-to analysis algorithms have relied on conservative approximations of exception handling [19, 20]. The well-known points-to analysis libraries Spark [19] and Paddle [18] both model exception throwing as an assignment to a single global variable for all exceptions thrown in a program. The variable is then read at the site of an exception catch. This approach is sound but highly imprecise because it ignores the information about what exceptions can propagate to a catch site. For clients that care about exception objects specifically (e.g., computing which `throw` statement can reach which `catch` clause), precise exception handling has been added on top of a base points-to analysis [7–9]. Fu and Ryder's "exception-chain analysis" [8] is representative. It works on top of Spark, with its conservative modeling of exceptions, but then performs a very precise analysis of the flow of exception objects. However, this approach has a high computational overhead. Furthermore, the approach does not recover the precision lost for the base points-to results for objects that do not represent exceptions.

Based on the above, the Doop framework [2] (which is also the context of our work) has introduced a joint points-to and exception analysis [1]. Doop expresses exception-analysis logic in modular rules, mutually recursive with the points-to analysis logic: Exception handling can cause variables to point to objects (of an exception type), can make code reachable, etc. Points-to results are, in turn, used to compute what objects are thrown at a `throw` statement. The exception analysis logic on its own is "as precise as can be" as it fully models the Java semantics for exceptions. Approximation is only introduced due to the static abstractions used for contexts and objects in the points-to analysis. Thus, exception analysis is specified in a form that applies to points-to analyses of varying precision, and the exception analysis transparently inherits the points-to analysis precision. The result is an analysis that achieves very high precision and performance for points-to results, while also matching the precision of techniques specifically for exception-related queries, as in the Fu and Ryder exception-chain analysis.

The motivation for our work is that, despite the benefits of the Doop approach, there is significant room for improvement. The joint points-to and exception analysis performs heavy work for exception objects alone. An indicative metric is the following: Consider the number of objects pointed to by method parameters or its return value vs. the objects thrown and not caught by the current method or by methods called by it. The former number represents the objects that flow into or out of each method due to normal control-flow, while the latter shows the objects that flow out of the method due to exceptions. Our experiments show that the latter number is often several times larger than the former. (We present full results later.) This is counterintuitive and suggests that the analysis performs unexpectedly much work on exceptions alone.

To address this issue we observe that most client analyses do not care about exception objects specifically. They do, however, care about the impact of exceptions to the rest of the points-to and call-graph facts. For instance, the effectiveness of a client analysis such as static cast elimination is not impacted in practice by the few optimization opportunities that lie inside exception handlers or that involve objects of an exception type. But the analysis *is* impacted by code possibly executed only because of exception handling, or variables that point to extra objects as a result of an exception handler's execution. In other words, we would like *precise handling of exceptions only to the extent that they impact the precision of the base points-to analysis, even if the information over exception objects themselves is less precise*. (Note that this is very different from the Spark or Paddle handling of all exceptions through a single global variable: That approach does adversely impact the precision and performance of the base analysis—e.g., it more than doubles the number of edges of a context-sensitive call-graph [1, Fig.12]).

Therefore, our approach consists of coarsening the representation of exception objects in two ways. First, we treat exception objects context-insensitively, even for an otherwise context-sensitive analysis.[1] Second, we merge exception objects and represent them as a single object per-dynamic-type. The per-type treatment is important for maintaining precision, since the main purpose of an exception object is to trigger appropriate exception handling code (i.e., a `catch` clause keyed on the type of the object caught).

We find that this approach is both easy to specify and implement, as well as highly effective. For instance, for a 1-object-sensitive analysis we obtain a 60% average speedup for the "antlr" benchmark and a 225% average speedup for the "eclipse" benchmark of the DaCapo suite (with similar speedups for other benchmarks) just by employing the "merge exception objects per-type" idea. This speedup is accompanied by significant space savings in the analysis. Crucially, the performance increase does not entail any loss of precision for results unrelated to exception objects. All precision metrics of the analysis remain virtually identical. Namely, the numbers of call-graph nodes and edges, methods that can be successfully devirtualized, and casts that can be statically eliminated remain the same up to at least three significant digits.

In summary, the contributions of our work are as follows:

- We give a concise and general model of flow-insensitive, context- and field-sensitive points-to analyses and call-graph construction for a language with exceptions. Although a joint exception and points-to analysis has been formulated before [1], it was expressed by-example. In contrast, we give a small, closed set of rules and definitions of input domains. That is, we present all the relevant detail of the analysis in a closed form, assuming a simplified intermediate language as input.
- We present measurements demonstrating that the impact of exceptions on points-to analysis performance metrics is significant. A points-to analysis that tries to model exceptions precisely ends up spending much of its time and space computing results for exception-based control-flow.
- We define on top of our model two simple ways to coarsen the representation of exception objects without affecting any other aspect of the points-to or exception logic.
- We show that our approach is very effective in practice, yielding both significant speedup and space savings. Our technique is the default in the upcoming version of the Doop framework as it gains performance without adversely impacting precision.

In the following sections we define an abstraction of context-sensitive points-to analysis and enhance it with exception handling logic (Section 2), present our technique in this abstract model (Section 3), detail its performance in a series of experiments (Section 4), and discuss related work in more detail (Section 5).

[1] *Context-sensitivity* is a general approach that achieves tractable and usefully high precision in points-to analyis. It consists of qualifying local program variables, and possibly (heap) object abstractions, with context information: the analysis collapses information (e.g., "what objects this method argument can point to") over all possible executions that result in the same context, while separating all information for different contexts.

2 Background: Model of Points-to Analysis

We next present a model of context-sensitive, flow-insensitive points-to analysis algorithms, as well as the enhancement of the model for computing exception information in mutual recursion with the analysis. Interestingly, the logical formalism that we use in our model is quite close to the actual implementation of the analysis in the Doop framework, under simplifications and omissions that we describe.

2.1 Base Points-to Analysis

We model a wide range of flow-insensitive points-to analyses together with the associated call-graph computation as a set of customizable Datalog rules, i.e., monotonic logical inferences that repeatedly apply to infer more facts until fixpoint. Our rules do not use negation in a recursive cycle, or other non-monotonic logic constructs, resulting in a declarative specification: the order of evaluation of rules or examination of clauses cannot affect the final result. The same abstract model applies to a wealth of analyses. We use it to model a context-insensitive Andersen-style analysis, as well as several context-sensitive analyses, both call-site-sensitive [25,26] and object-sensitive [23].

The input language is a simplified intermediate language with a) a "new" instruction for allocating an object; b) a "move" instruction for copying between local variables; c) "store" and "load" instructions for writing to the heap (i.e., to object fields); d) a "virtual method call" instruction that calls the method of the appropriate signature that is defined in the dynamic class of the receiver object. This language models well the Java bytecode representation, but also other high-level intermediate languages. (It does not, however, model languages such as C or C++ that can create pointers through an address-of operator. The techniques used in that space are fairly different—e.g., [12, 13].) The specification of our points-to analysis as well as the input language are in line with those in the work of others [10,21], although we also integrate elements such as on-the-fly call-graph construction and field-sensitivity.

Specifying the analysis logically as Datalog rules has the advantage that the specification is close to the actual implementation. Datalog has been the basis of several implementations of program analyses, both low-level [2,17,24,29,30] and high-level [5,11]. Indeed, the analysis we show is a faithful model of the implementation in the Doop framework [2]. Our specification of the analysis (Figures 1-2) is an abstraction of the actual implementation in the following ways:

- The implementation has many more rules. It covers the full complexity of the language, including rules for handling reflection, native methods, static calls and fields, string constants, implicit initialization, threads, and a lot more. The Doop implementation currently contains over 600 rules in the common core of all analyses, as opposed to the dozen-or-so rules we examine here. (Note, however, that these dozen rules are the most crucial for points-to analysis. They also correspond fairly closely to the algorithms specified in other formalizations of points-to analyses in the literature [22,28].)
- The implementation also reflects considerations for efficient execution. The most important is that of defining indexes for the key relations of the evaluation. Furthermore, it designates some relations as functions, defines storage models for relations

V is a set of variables	H is a set of heap abstractions
M is a set of methods	S is a set of method signatures (including name)
F is a set of fields	I is a set of instructions (e.g., invocation sites)
T is a set of class types	$\mathbb{N}$ is the set of natural numbers
HC is a set of heap contexts	C is a set of contexts

Alloc (*var* : *V*, *heap* : *H*, *meth* : *M*)	FormalArg (*meth* : *M*, *i* : $\mathbb{N}$, *arg* : *V*)
Move (*to* : *V*, *from* : *V*)	ActualArg (*invo* : *I*, *i* : $\mathbb{N}$, *arg* : *V*)
Load (*to* : *V*, *base* : *V*, *fld* : *F*)	FormalReturn (*meth* : *M*, *ret* : *V*)
Store (*base* : *V*, *fld* : *F*, *from* : *V*)	ActualReturn (*invo* : *I*, *var* : *V*)
VCall (*base* : *V*, *sig* : *S*, *invo* : *I*)	ThisVar (*meth* : *M*, *this* : *V*)
HeapType (*heap* : *H*, *type* : *T*)	LookUp (*type* : *T*, *sig* : *S*, *meth* : *M*)
InMethod (*instr* : *I*, *meth* : *M*)	Subtype (*type* : *T*, *superT* : *T*)

VarPointsTo (*var* : *V*, *ctx* : *C*, *heap* : *H*, *hctx* : *HC*)
CallGraph (*invo* : *I*, *callerCtx* : *C*, *meth* : *M*, *calleeCtx* : *C*)
FldPointsTo (*baseH* : *H*, *baseHCtx* : *HC*, *fld* : *F*, *heap* : *H*, *hctx* : *HC*)
InterProcAssign (*to* : *V*, *toCtx* : *C*, *from* : *V*, *fromCtx* : *C*)
Reachable (*meth* : *M*, *ctx* : *C*)

Record (*heap* : *H*, *ctx* : *C*) = ***newHCtx* : *HC***
Merge (*heap* : *H*, *hctx* : *HC*, *invo* : *I*, *ctx* : *C*) = ***newCtx* : *C***

Fig. 1. Our domain, input relations, output relations, and constructors of contexts

(e.g., how many bits each variable uses), designates intermediate relations as "materialized views" or not, etc.

Figure 1 shows the domain of our analysis (i.e., the different sets that comprise the space of our computation), its input relations, the intermediate and output relations, as well as two constructor functions, responsible for producing new objects that represent contexts. We explain some of these components below:

- The input relations are standard and correspond to the intermediate language for our analysis. For instance, the Alloc relation represents every instruction that allocates a new heap object, *heap*, and assigns it to local variable *var* inside method *meth*. (Note that every local variable is defined in a unique method, hence the *meth* argument is also implied by *var* but included for conciseness of later rules.) There are input relations for each instruction type (Move, Load, Store and VCall) as well as input relations encoding the type system and symbol table information. For instance, LookUp matches a method signature to the actual method definition inside a type.
- The main output relations of our points-to analysis and call-graph computation are VarPointsTo and CallGraph. The VarPointsTo relation links a variable (*var*) to a heap object (*heap*). (A heap object is identified by its allocation site.) Both the variable and the heap object are qualified by "context" elements in our analysis: a plain context for the variable and a heap context for the heap object. Similarly, the CallGraph relation qualifies both its source (an invocation site) and its target (a method) with contexts. Other intermediate relations (FldPointsTo, InterProcAssign, Reachable) correspond to standard concepts and are introduced for conciseness and readability.
- The base rules are not concerned with what kind of context-sensitivity is used. The same rules can be used for a context-insensitive analysis (by only ever creating a single context object), for a call-site-sensitive analysis, or for an object-sensitive analysis, for

any context depth. These aspects are completely hidden behind constructor functions RECORD and MERGE, following the usage and naming convention of earlier work [28]. RECORD takes all available information at the allocation site of an object and combines it to produce a new heap context, while MERGE takes all available information at the call site of a method and combines it to create a new context. (Hence, the name "MERGE" refers to merging contexts and is unrelated to the idea of merging exception objects per-type, which we discuss later in this paper.) These functions are sufficient for modeling a very large variety of context-sensitive analyses.[2] Note that the use of such constructors is not part of regular Datalog and can result in infinite structures (e.g., one can express unbounded call-site sensitivity) if care is not taken.

```
InterProcAssign (to, calleeCtx, from, callerCtx) ←
    CallGraph (invo, callerCtx, meth, calleeCtx),
    FormalArg (meth, i, to), ActualArg (invo, i, from).
InterProcAssign (to, callerCtx, from, calleeCtx) ←
    CallGraph (invo, callerCtx, meth, calleeCtx),
    FormalReturn (meth, from), ActualReturn (invo, to).

Record (heap, ctx) = hctx,
VarPointsTo (var, ctx, heap, hctx) ←
    Reachable (meth, ctx), Alloc (var, heap, meth).
VarPointsTo (to, ctx, heap, hctx) ←
    Move (to, from), VarPointsTo (from, ctx, heap, hctx).
VarPointsTo (to, toCtx, heap, hctx) ←
    InterProcAssign (to, toCtx, from, fromCtx),
    VarPointsTo (from, fromCtx, heap, hctx).
VarPointsTo (to, ctx, heap, hctx) ←
    Load (to, base, fld), VarPointsTo (base, ctx, baseH, baseHCtx),
    FldPointsTo (baseH, baseHCtx, fld, heap, hctx).
FldPointsTo (baseH, baseHCtx, fld, heap, hctx) ←
    Store (base, fld, from), VarPointsTo (base, ctx, baseH, baseHCtx),
    VarPointsTo (from, ctx, heap, hctx).

Merge (heap, hctx, invo, callerCtx) = calleeCtx,
Reachable (toMeth, calleeCtx),
VarPointsTo (this, calleeCtx, heap, hctx),
CallGraph (invo, callerCtx, toMeth, calleeCtx) ←
    VCall (base, sig, invo),
    VarPointsTo (base, callerCtx, heap, hctx), HeapType (heap, heapT),
    Lookup (heapT, sig, toMeth), ThisVar (toMeth, this).
```

Fig. 2. Datalog rules for the points-to analysis and call-graph construction

[2] Explaining the different kinds of context-sensitivity produced by varying RECORD and MERGE is beyond the scope of this paper but is fully covered in past literature [28]. To give a single example, however, a 1-call-site-sensitive analysis with a context-sensitive heap has $C = HC = I$ (i.e., both the context and the heap context are a single instruction), **RECORD** (*heap, ctx*) = ***ctx*** and **MERGE** (*heap, hctx, invo, callerCtx*) = ***invo***. That is, when an object is allocated, its (heap) context is that of the allocating method, and when a method is called, its context is its call-site.

Figure 2 shows the points-to analysis and call-graph computation. The rule syntax is simple: the left arrow symbol (←) separates the inferred fact (i.e., the *head* of the rule) from the previously established facts (i.e., the *body* of the rule). For instance, the very last rule says that if the original program has an instruction making a virtual method call over local variable *base* (this is an input fact), and the computation so far has established that *base* can point to heap object *heap*, then the called method is looked up inside the type of *heap* and several further facts are inferred: that the looked up method is reachable, that it has an edge in the call-graph from the current invocation site, and that its *this* variable can point to *heap*. Additionally, the MERGE function is used to possibly create (or look up) the right context for the current invocation.

2.2 Adding Exceptions

We can now easily add exception handling to our input language and express a precise exception analysis via rules that are mutually recursive with the base points-to analysis rules. The algorithm is essentially that of [1] but stated more concisely: we hide exception handler lookup details by assuming a more sophisticated input relation CATCH.

THROW (*instr : I, e : V*)	CATCH (*heapT : T, instr : I, arg : V*)
THROWPOINTSTO (*meth : M, ctx : C, heap : H, hctx : HC*)	

Fig. 3. Datalog input and output relations for the exception analysis

```
THROWPOINTSTO (meth, ctx, heap, hctx) ←
    THROW (instr, e), VARPOINTSTO (e, ctx, heap, hctx),
    HEAPTYPE (heap, heapT), ¬CATCH (heapT, instr, _), INMETHOD (instr, meth).
THROWPOINTSTO (meth, callerCtx, heap, hctx) ←
    CALLGRAPH (invo, callerCtx, toMeth, calleeCtx),
    THROWPOINTSTO (toMeth, calleeCtx, heap, hctx),
    HEAPTYPE (heap, heapT), ¬CATCH (heapT, invo, _), INMETHOD (invo, meth).
VARPOINTSTO (arg, ctx, heap, hctx) ←
    THROW (instr, e), VARPOINTSTO (e, ctx, heap, hctx),
    HEAPTYPE (heap, heapT), CATCH (heapT, instr, arg).
VARPOINTSTO (arg, callerCtx, heap, hctx) ←
    CALLGRAPH (invo, callerCtx, toMeth, calleeCtx),
    THROWPOINTSTO (toMeth, calleeCtx, heap, hctx),
    HEAPTYPE (heap, heapT), CATCH (heapT, invo, arg).
```

Fig. 4. Datalog rules for the Exception analysis

Figure 3 presents the input and output relations for our analysis. The input relations enhance the language-under-analysis with `catch` and `throw` instructions, with Java-like semantics. The THROW (*i,e*) relation captures throwing at instruction *i* an expression object that is referenced by local variable *e*. The CATCH (*t,i,a*) relation connects an instruction *i* that throws an exception of dynamic type *t* with the local variable *a* that will be assigned the exception object at the appropriate catch-site. Although CATCH does not directly map to intermediate language instructions, one can compute it easily from such low-level input. Furthermore, hiding the definition of CATCH allows modeling of exception handlers at different degrees of precision—e.g., a definition of CATCH may or may not consider exception handlers in-order.

Figure 4 shows the exception computation, in mutual recursion with the points-to analysis. Two syntactic constructs we have not seen before are "_", meaning "any value", and "¬", signifying negation. The relation we want to compute is THROWPOINTSTO, which captures what exception objects a method may throw at its callers. As can be seen, VARPOINTSTO is used in the definition of THROWPOINTSTO and vice versa.

3 Coarsening the Representation of Exceptions

Although a precise joint points-to and exception analysis algorithm offers significant benefits [1], we next show that there is large room for improvement. The analysis ends up spending much of its time and space computing exception flow. We propose ideas for coarsening the representation of exception objects to address this issue and yield more efficient analyses, without sacrificing precision.

3.1 Motivation

Consider the size of the THROWPOINTSTO relation for an analysis. This represents the total flow of objects out of methods due to exceptions. For a context-sensitive analysis, this number is a good metric of the work performed by the analysis internally for reasoning about exceptions. It is interesting to compare this number with a similar metric over the VARPOINTSTO relation, namely the subset of VARPOINTSTO facts that concern variables that are either method arguments or return values. This represents the total flow of objects in and out of methods due to normal call and return sequences.

Table 1 shows the results of comparing these two measures for several different analyses: insensitive, call-site sensitive, object-sensitive, and type-sensitive [28], with a context-sensitive heap. The results are over five of the benchmarks in the DaCapo benchmark suite, analyzed with Oracle JDK 1.6. (A full description of our experimental setting can be found in the next section.) Entries with a dash instead of a number did not terminate within the time allotted (90mins).

For the context-insensitive analysis (first results column), the ratio can be understood in intuitive terms: the antlr ratio of 0.22, for instance, means that, on average, 4.5 times more objects are possibly thrown out of a method than passed into it or returned through regular call and return sequences. This is a counterintuitive result. Human reasoning about how a method interacts with its callers is certainly not dominated by exception objects. Therefore, we see that the joint points-to and exception analysis perhaps pays a disproportionate amount of attention (and expends much effort) on exceptions.

3.2 Coarse Exceptions

To reduce the cost of reasoning about exception objects, we propose two simple approaches for coarsening the representation of exception objects. The first is to represent exception objects context-insensitively. This is a rather straightforward idea—even in

Table 1. Objects on method boundaries compared to exception objects thrown by a method (measured in thousands)

		insens	1obj+H	2obj+H	1type+H	1call+H
antlr	objs passed	697	-	10,440	3,955	17,486
	objs thrown	3,123	-	164,392	20,783	44,118
	ratio	**.22**	-	**.06**	**.19**	**.40**
bloat	objs passed	829	-	-	5,681	46,952
	objs thrown	4,112	-	-	32,905	78,593
	ratio	**.20**	-	-	**.17**	**.60**
eclipse	objs passed	637	15,750	-	6,570	18,690
	objs thrown	4,064	138,361	-	37,634	42,140
	ratio	**.16**	**.11**	-	**.17**	**.44**
luindex	objs passed	383	8,473	-	3,328	8,413
	objs thrown	2,544	60,897	-	18,928	25,297
	ratio	**.15**	**.14**	-	**.17**	**.33**
xalan	objs passed	668	-	-	7,480	18,895
	objs thrown	3,876	-	-	41,351	43,376
	ratio	**.17**	-	-	**.18**	**.44**

context-sensitive analyses, several different kinds of objects (e.g., string constants) are more profitably represented context-insensitively. Even before our current work, the Doop framework had the ability to represent exceptions context-insensitively with the right choice of flags. The second approach consists of not just omitting context for exception objects, but also merging the objects themselves, remembering only a single representative per (dynamic) type. That is, all points-to information concerning exception objects is merged "at the source"—all objects of the same type become one.

This is a fitting approach for exception objects because it relies upon intuition on how exception objects are used in practice. Specifically, the intuition is that exception objects have mostly control-flow significance (i.e., they are used as type labels determining what exception handler is to be executed) and little data-flow impact (i.e., the data stored in exception objects' fields do not affect the precision of an overall points-to analysis). In other words, an exception object's dynamic type alone is an excellent approximation of the object itself. Our measurements of the next section show that this is the case.

Figure 5 shows the changes to earlier rules required to implement the two approaches. The original logic of allocating an object is removed and replaced with two cases: if the allocated object is not an instance of an exception type, then the original allocation logic applies. If it is, then the object is allocated context-insensitively (by using a constant context instead of calling the Record function to create a new one). Furthermore, in the case of merging exception objects, the object itself is replaced by a representative object of its type (arbitrarily chosen to be the object of the same type with the minimum internal identifier). Note that the definition of an exception type consists of merely looking up all types used in `catch` clauses—the definition could also be replaced by the weaker condition of whether a type is a subtype of `Throwable`.

There are some desirable properties of replacing objects with per-type representatives at their creation site. Most importantly, this approach leaves the rest of the analysis unchanged and can maintain all its precision features. Compared to past approaches to

Commmon core of coarsening logic: object allocation rule is replaced by refined version

CHContext (*"ConstantHeapCtx"*) ← True.
ExceptionType (*t*) ← Catch (*superT*, _, _), Subtype (*t*, *superT*).

~~**Record** (*heap*, *ctx*) = ***hctx***,~~
~~VarPointsTo (*var*, *ctx*, *heap*, *hctx*) ←~~
~~Reachable (*meth*, *ctx*), Alloc (*var*, *heap*, *meth*).~~

Record (*heap*, *ctx*) = ***hctx***,
VarPointsTo (*var*, *ctx*, *heap*, *hctx*) ←
  Reachable (*meth*, *ctx*), Alloc (*var*, *heap*, *meth*),
  HeapType (*heap*, *heapT*), ¬ExceptionType (*heapT*).

Additional Rule (over common core) for Context-insensitive treatment

VarPointsTo (*var*, *ctx*, *heap*, *hctx*) ←
  Reachable (*meth*, *ctx*), Alloc (*var*, *heap*, *meth*), HeapType (*heap*, *heapT*),
  ExceptionType (*heapT*), CHContext (*hctx*).

Additional Rules (over common core) for merging exceptions by use of representative objects

Representative (*heap*, *reprH*) ←
  HeapType (*heap*, *heapT*), *reprHeap* = **min**<HeapType (***?***, *heapT*)>.

VarPointsTo (*var*, *ctx*, *reprH*, *hctx*) ←
  Reachable (*meth*, *ctx*), Alloc (*var*, *heap*, *meth*), HeapType (*heap*, *heapT*),
  ExceptionType (*heapT*), CHContext (*hctx*), Representative (*heap*, *reprH*).

Fig. 5. Changes over the rules of Figures 2 and 4 for the two treatments that coarsen the representation of exception objects. The object allocation rule is shown striken out to indicate that it is replaced by a new, conditional version, immediately below. The rules introduce a constant heap context, CHContext, as well as auxiliary relations ExceptionType (*t* : *T*) and Representative (*heap* : *H*, *reprH* : *H*).

merging exceptions (e.g., the single-global-variable assignment of Spark or Paddle) we can maintain all the precision resulting from considering exception handlers in order, filtering caught exceptions, and taking into account the specific instructions under the scope of an exception handler. These have been shown to be important features for the precision and performance of the underlying points-to analysis. Ignoring the order of exception handlers, for instance, results in a much less precise context-sensitive call-graph, with 50% more edges [1, Fig.13].

4 Experiments

We next present the results of our experiments with the two ideas for coarsening the representation of exception objects. As we will see, our approach yields substantial performance improvements without sacrificing virtually *any* precision. This is a rather surprising result. Given how crucial the handling of exceptions has been for the precision of the joint points-to and exception analysis, one would expect that representing exception objects crudely (by merging them per-type) would have serious precision implications. For comparison, Bravenboer and Smaragdakis attempted a different approximation: they represented the ThrowPointsTo relation context-insensitively (i.e., by dropping the *ctx* argument) and found this to significantly hurt the precision of the points-to analysis [1, Sec.5.2], e.g., increasing points-to sets by 10%.[3]

Our implementation is in the Doop framework and was run on the LogicBlox Datalog engine, v.3.9.0. We use a 64-bit machine with a quad-core Xeon E5530 2.4GHz CPU (only one thread was active at a time) and 24GB of RAM. We analyzed the DaCapo benchmark programs, v.2006-10-MR2 with JDK 1.6.0_30. The choice of JDK is highly significant for Java static analysis. Earlier published Doop results [1, 2, 28] were for JDK 1.4. We chose to present JDK 1.6 results since it is recent, much larger, and more representative of actual use. However, results for JDK 1.4 can also be found in the first author's M.Sc. thesis, available at `http://cgi.di.uoa.gr/~gkast/MSc_Thesis.pdf`.

There are three primary questions we would like to answer with our experiments:

1. Can we reduce the cost of points-to and exception analysis by coarsening the representation of exception objects, without sacrificing precision?
2. Is the "simple" coarsening approach of treating exceptions context-insensitively sufficient or do we get significant extra benefit from merging exception objects per-type?
3. Do our techniques address the motivation of Table 1, i.e., produce results that roughly match human expectations when reasoning about objects that flow in and out of methods due to exceptions vs. normal call-returns?

Tables 2 and 3 show the time and space savings of the coarsening techniques over a large set of analyses, ranging from context-insensitive to a highly-precise 2-object-sensitive with a 2-context-sensitive heap (2obj+2H). The analysis variety includes a mix of call-site-, type-, and object-sensitive analyses. Entries with a dash instead of a number are due to analyses that did not terminate within 90 mins. Entries with neither a number nor a dash mean that we did not run the corresponding experiment. (This only happened for experiments on the full-sensitive treatment of exceptions, which we omitted because the main trends were already clear from a smaller subset of our measurements—those for benchmarks and analyses also shown earlier in Table 1.)

As can be seen, the results demonstrate a substantial benefit in the "bottom-line" performance of the joint analysis from representing exception objects coarsely. Furthermore, the simple approach of dealing with exceptions context-insensitively is clearly insufficient. The advantage of merging objects over merely eliding context can be as high as a 3.4x boost in performance, and rarely falls below a 50% speedup. Space savings tell a similar story, to a lesser but still large extent.

The major question we are addressing next is whether these significant performance improvements entail sacrifices in precision. This requires us to first state the question

[3] We repeated several experiments from [1] in our setting for validation but do not report them here since the results are effectively the same as in that publication.

Table 2. Execution time (seconds) for a variety of analyses on various benchmarks

		insens	1obj	1obj+H	2obj+H	2obj+2H	1type+H	2type+H	1call	1call+H
antlr	sens	120	338	-	3207	-	543	400	245	975
	insens	111	319	1089	574	2785	334	209	238	543
	merge	75	199	899	249	2313	217	121	128	420
	sen/ins	**1.08**	**1.06**	-	**5.58**	-	**1.62**	**1.91**	**1.02**	**1.79**
	ins/mer	**1.48**	**1.60**	**1.21**	**2.30**	**1.20**	**1.53**	**1.72**	**1.85**	**1.29**
bloat	sens	120	1065	-	-	-	826	1921	426	3403
	insens	120	1057	2337	-	-	483	553	429	1795
	merge	68	432	1727	-	-	292	162	208	1496
	sen/ins	**1.00**	**1.00**	-	-	-	**1.71**	**3.47**	**1.00**	**1.79**
	ins/mer	**1.76**	**2.44**	**1.35**	-	-	**1.65**	**3.41**	**2.06**	**1.19**
chart	sens									
	insens	240	2932	-	-	-	1597	699	591	1334
	merge	138	1434	-	999	-	1253	256	319	1115
	sen/ins									
	ins/mer	**1.73**	**2.04**	-	-	-	**1.27**	**2.73**	**1.85**	**1.19**
eclipse	sens	91	314	2243	-	-	892	800	230	1099
	insens	90	315	800	1059	-	508	348	231	642
	merge	52	140	634	623	-	269	187	90	500
	sen/ins	**1.00**	**1.00**	**2.80**	-	-	**1.75**	**2.30**	**1.00**	**1.71**
	ins/mer	**1.73**	**2.25**	**1.26**	**1.69**	-	**1.88**	**1.86**	**2.56**	**1.28**
jython	sens									
	insens	96	636	2023	-	-	-	-	237	613
	merge	60	129	944	-	-	258	768	98	429
	sen/ins									
	ins/mer	**1.60**	**4.93**	**2.14**	-	-	-	-	**2.42**	**1.43**
luindex	sens	71		838			411			524
	insens	67	168	395	391	2247	239	168	133	288
	merge	45	86	281	153	1982	142	86	67	195
	sen/ins	**1.06**		**2.12**						**1.82**
	ins/mer	**1.48**	**1.95**	**1.40**	**2.55**	**1.13**	**1.68**	**1.95**	**1.98**	**1.47**
lusearch	sens									
	insens	69	185	428	459	3024	262	169	145	302
	merge	45	97	299	214	2723	158	87	71	210
	sen/ins									
	ins/mer	**1.53**	**1.90**	**1.43**	**2.14**	**1.11**	**1.66**	**1.94**	**2.04**	**1.43**
pmd	sens									
	insens	105	274	567	427	2330	327	213	181	392
	merge	66	153	379	188	2076	207	129	100	280
	sen/ins									
	ins/mer	**1.59**	**1.79**	**1.49**	**2.27**	**1.12**	**1.57**	**1.65**	**1.81**	**1.40**
xalan	sens	116		-	-	-	1091			1214
	insens	118	463	1139	-	-	579	349	249	672
	merge	70	219	836	-	-	470	200	126	521
	sen/ins	**1.00**		-			**1.88**	-	-	**1.80**
	ins/mer	**1.68**	**2.11**	**1.36**	-	-	**1.23**	**1.74**	**1.97**	**1.29**

Table 3. Disk footprint (MB) for a variety of analyses on various benchmarks

		insens	1obj	1obj+H	2obj+H	2obj+2H	1type+H	2type+H	1call	1call+H
antlr	sens	649	996	-	4608	-	1433	1228	945	3174
	insens	649	996	2560	1536	3686	963	735	945	1740
	merge	544	683	2048	978	3072	675	518	661	1433
	sen/ins	**1.00**	**1.00**	-	**3.00**	-	**1.48**	**1.67**	**1.00**	**1.82**
	ins/mer	**1.19**	**1.45**	**1.25**	**1.57**	**1.19**	**1.42**	**1.41**	**1.42**	**1.21**
bloat	sens	460	1126	-	-	-	1433	2150	1228	5120
	insens	461	1126	2457	-	-	923	992	1228	3379
	merge	363	663	1843	-	-	586	505	773	2969
	sen/ins	**1.00**	**1.00**	-	-	-	**1.55**	**2.16**	**1.00**	**1.51**
	ins/mer	**1.26**	**1.69**	**1.33**	-	-	**1.57**	**1.96**	**1.58**	**1.13**
chart	sens									
	insens	968	3072	-	-	-	2764	1536	1945	3276
	merge	653	1740	-	-	-	1945	811	1331	2560
	sen/ins									
	ins/mer	**1.48**	**1.76**	-	-	-	**1.42**	**1.89**	**1.46**	**1.27**
eclipse	sens	428	748	4505	-	-	2048	1945	694	2867
	insens	429	748	2048	2252	-	1433	1012	694	1638
	merge	300	405	1433	1536	-	679	602	444	1433
	sen/ins	**1.00**	**1.00**	**2.20**	-	-	**1.43**	**1.92**	**1.00**	**1.75**
	ins/mer	**1.43**	**1.84**	**1.42**	**1.46**	-	**2.11**	**1.68**	**1.56**	**1.14**
jython	sens									
	insens	604	1228	3584	-	-	-	-	980	1740
	merge	472	609	2355	-	-	798	1740	601	1331
	sen/ins									
	ins/mer	**1.27**	**2.01**	**1.52**	-	-	-	-	**1.63**	**1.30**
luindex	sens	346		2252	-	-	929			1638
	insens	346	496	1126	1126	2764	645	549	508	874
	merge	265	317	716	565	2150	399	327	349	684
	sen/ins	**1.00**		**2.00**	-	-				**1.87**
	ins/mer	**1.30**	**1.56**	**1.57**	**1.99**	**1.28**	**1.62**	**1.67**	**1.45**	**1.27**
lusearch	sens									
	insens	367	517	1228	1228	3379	681	550	548	926
	merge	277	334	723	690	2867	424	327	370	722
	sen/ins									
	ins/mer	**1.32**	**1.54**	**1.69**	**1.77**	**1.17**	**1.60**	**1.68**	**1.48**	**1.28**
pmd	sens									
	insens	602	817	1536	1331	2969	948	785	814	1331
	merge	505	585	1017	839	2457	647	560	631	1126
	sen/ins									
	ins/mer	**1.19**	**1.39**	**1.51**	**1.58**	**1.20**	**1.46**	**1.40**	**1.29**	**1.18**
xalan	sens	614		-	-	-	2457			2969
	insens	614	1331	3174	-	-	1638	1126	940	1843
	merge	487	710	2252	-	-	823	684	659	1536
	sen/ins	**1.00**		-	-	-	**1.50**			**1.61**
	ins/mer	**1.26**	**1.87**	**1.41**	-	-	**2.00**	**1.64**	**1.42**	**1.20**

Table 4. Metrics concerning precision for a variety of analyses. Jython omitted for space.

			edges	meths	v-calls	poly v-calls	casts	fail casts
antlr	1obj+H	sens	-	-	-	-	-	-
		insens	59075	8886	33467	1924	1767	985
		merge	59075	8886	33467	1924	1767	985
	2obj+H	sens	55445	8714	32976	1712	1709	611
		insens	55445	8714	32976	1712	1709	611
		merge	55445	8714	32976	1712	1709	611
	1type+H	sens	59738	8916	33507	1948	1770	1070
		insens	59738	8916	33507	1948	1770	1070
		merge	59738	8916	33507	1948	1770	1070
	1call+H	sens	60797	8961	33631	1985	1778	1037
		insens	60797	8961	33631	1985	1778	1037
		merge	60797	8961	33631	1985	1778	1037
bloat	1obj+H	sens	-	-	-	-	-	-
		insens	65672	10116	31049	2067	2815	1911
		merge	65672	10116	31049	2067	2815	1911
	2obj+H	sens	-	-	-	-	-	-
		insens	-	-	-	-	-	-
		merge	-	-	-	-	-	-
	1type+H	sens	66697	10150	31089	2137	2818	2045
		insens	66697	10150	31089	2137	2818	2045
		merge	66697	10150	31089	2137	2818	2045
	1call+H	sens	70340	10200	31214	2129	2829	2007
		insens	70340	10200	31214	2129	2829	2007
		merge	70340	10200	31214	2129	2829	2007
chart	1obj+H	sens						
		insens	-	-	-	-	-	-
		merge	-	-	-	-	-	-
	2obj+H	sens						
		insens	-	-	-	-	-	-
		merge	59027	12510	31111	1610	2765	1055
	1type+H	sens						
		insens	79871	16044	39462	2725	3858	2445
		merge	79896	16044	39462	2730	3858	2450
	1call+H	sens						
		insens	81865	16134	39724	2887	3887	2480
		merge	81890	16134	39724	2892	3887	2485
eclipse	1obj+H	sens	49279	9408	23505	1386	1984	1092
		insens	49279	9408	23505	1386	1984	1092
		merge	49282	9408	23505	1386	1984	1092
	2obj+H	sens	-	-	-	-	-	-
		insens	44792	9188	22852	1168	1912	729
		merge	44795	9188	22852	1168	1912	729
	1type+H	sens	51161	9452	23634	1438	1987	1198
		insens	51161	9452	23634	1438	1987	1198
		merge	51162	9452	23634	1438	1987	1198
	1call+H	sens	52800	9511	23716	1507	2000	1154
		insens	52800	9511	23716	1507	2000	1154
		merge	52800	9511	23716	1507	2000	1154

Table 5. Metrics concerning precision for a variety of analyses (cont'd from Table 4)

			edges	meths	v-calls	poly v-calls	casts	fail casts
luindex	1obj+H	sens	40004	7876	18263	1110	1521	796
		insens	40004	7876	18263	1110	1521	796
		merge	40004	7876	18263	1110	1521	796
	2obj+H	sens	-	-	-	-	-	-
		insens	36477	7702	17748	899	1463	496
		merge	36477	7702	17748	899	1463	496
	1type+H	sens	40646	7906	18303	1138	1524	889
		insens	40646	7906	18303	1138	1524	889
		merge	40646	7906	18303	1138	1524	896
	1call+H	sens	41790	7953	18492	1171	1532	837
		insens	41790	7953	18492	1171	1532	837
		merge	41790	7953	18492	1171	1532	837
lusearch	1obj+H	sens						
		insens	42977	8526	19556	1289	1622	812
		merge	42977	8526	19556	1289	1622	812
	2obj+H	sens						
		insens	39352	8344	19048	1071	1564	508
		merge	39352	8344	19048	1071	1564	508
	1type+H	sens						
		insens	43676	8558	19620	1319	1625	927
		merge	43676	8558	19620	1319	1625	934
	1call+H	sens						
		insens	45071	8626	19857	1352	1643	938
		merge	45071	8626	19857	1352	1643	938
pmd	1obj+H	sens						
		insens	46826	9277	21591	1168	1990	1210
		merge	46826	9277	21591	1168	1990	1210
	2obj+H	sens						
		insens	42988	9090	21004	942	1,931	846
		merge	42988	9090	21004	942	1,931	846
	1type+H	sens						
		insens	47539	9311	21632	1192	1993	1311
		merge	47540	9311	21632	1192	1993	1317
	1call+H	sens						
		insens	48895	9371	21843	1240	2003	1273
		merge	48895	9371	21843	1240	2003	1273
xalan	1obj+H	sens	-	-	-	-	-	-
		insens	54033	10511	25683	1857	2042	1055
		merge	54038	10511	25683	1858	2042	1055
	2obj+H	sens	-	-	-	-	-	-
		insens	-	-	-	-	-	-
		merge	-	-	-	-	-	-
	1type+H	sens	54792	10561	25760	1887	2049	1235
		insens	54792	10561	25760	1887	2049	1235
		merge	54796	10561	25760	1888	2049	1235
	1call+H	sens	56658	10613	25891	1966	2059	1203
		insens	56658	10613	25891	1966	2059	1203
		merge	56666	10613	25891	1967	2059	1203

Table 6. Objects on method boundaries compared to exception objects thrown by a method (measured in thousands)

		insens		1obj+H		2obj+H		1type+H		1call+H	
		insens	merge	insens	merge	insens	merge	insens	merge	insens	merge
antlr	objs passed	697	651	26,867	25,271	6,983	6,079	4,702	4,528	17,236	17,155
	objs thrown	3,123	204	14,197	965	23,856	1,730	7,084	428	12,461	822
	ratio	**.22**	**3.18**	**1.89**	**26.16**	**.29**	**3.51**	**.66**	**10.56**	**1.38**	**20.86**
bloat	objs passed	829	781	22,633	22,325	-	-	5,338	5,167	46,650	46,563
	objs thrown	4,112	257	18,697	1,189	-	-	10,140	589	20,048	1,251
	ratio	**.20**	**3.02**	**1.21**	**18.76**	-	-	**.52**	**8.76**	**2.32**	**37.22**
chart	objs passed	2,315	1,723	-	-	-	16,739	-	18,721	46,158	41,909
	objs thrown	8,331	414	-	-	-	7,231	-	1,357	19,747	1,371
	ratio	**.27**	**4.15**	**-**	**-**	**-**	**2.31**	**-**	**13.79**	**2.33**	**30.56**
eclipse	objs passed	637	539	14,045	16,875	15,234	14,357	6,025	6,471	18,311	18,186
	objs thrown	4,064	248	15,983	1,047	36,699	2,612	10,516	593	11,829	777
	ratio	**.15**	**2.17**	**.87**	**16.10**	**.41**	**5.49**	**.57**	**10.90**	**1.54**	**23.40**
jython	objs passed	801	479	43,068	35,106	-	-	-	-	19,307	-
	objs thrown	3,452	215	20,449	1,025	-	-	-	-	11,288	-
	ratio	**.23**	**2.23**	**2.10**	**34.21**	**-**	**-**	**-**	**-**	**1.71**	**-**
luindex	objs passed	383	339	7,664	7,414	3,525	3,037	3,031	2,881	8,186	8,109
	objs thrown	2,544	166	8,953	606	21,401	1,521	6,055	363	7,218	486
	ratio	**.15**	**2.03**	**.85**	**12.21**	**.16**	**1.99**	**.50**	**7.92**	**1.13**	**16.67**
lusearch	objs passed	429	385	8,085	7,822	4,519	4,010	3,319	3,165	9,005	8,928
	objs thrown	2,764	180	9,222	626	19,432	1,435	6,341	379	7,880	528
	ratio	**.15**	**2.13**	**.87**	**12.49**	**.23**	**2.79**	**.52**	**8.33**	**1.14**	**16.90**
pmd	objs passed	461	414	9,273	8,949	4,388	3,864	3,500	3,337	10,948	10,867
	objs thrown	3,008	198	10,409	695	19,961	1,465	7,100	431	8,529	577
	ratio	**.15**	**2.09**	**.89**	**12.87**	**.21**	**2.63**	**.49**	**7.73**	**1.28**	**18.81**
xalan	objs passed	668	589	24,618	24,155	-	-	6,350	5,821	18,448	18,301
	objs thrown	3,876	251	20,765	1,372	-	-	11,278	641	12,198	810
	ratio	**.17**	**2.34**	**1.18**	**17.60**	-	-	**.56**	**9.08**	**1.51**	**22.59**

appropriately. Since all exception objects of the same type are merged, it makes little sense to query points-to information for variables holding such objects (e.g., local variables inside exception handlers). Every such variable will appear to spuriously point to any object of the same dynamic type. Instead, what we want to do is examine the impact of merging exception objects on *the rest* of the points-to analysis. That is, a client analysis that cares about exception objects themselves should not employ our techniques. The question, however, is whether an analysis that applies over the whole program will be affected by the coarse exception representations.

Tables 4 and 5 show precision metrics for our benchmarks and a subset (for space reasons) of our analyses. We show the size of the computed call-graph, in terms of both nodes (i.e., methods) and edges, as well as the number of virtual calls that cannot be statically resolved and casts that cannot be statically be proven safe. (The total number of reachable virtual calls and casts are given for reference.) From our past experience, the call-graph metrics are generally excellent proxies for the overall precision of an analysis, and even tiny changes are reflected on them.

As can be seen, the precision of the program analysis remains virtually unaffected by the coarse representations of exceptions. This confirms that our merged exception objects still carry the essence of the information that the rest of the program needs from them.

The final question from our experiments is whether these two techniques address the motivating measurements of Section 3.1. Table 6 shows the same metrics of (context-sensitive) objects passed to/from methods vs. thrown for the analysis using our coarse representations of exceptions. As can be seen, the handling of exceptions context-insensitively does not suffice to bring the relative ratio of the metrics close to expected values, but merging exception objects per-type does. Specifically, for all values of the "merge" column, the total number of objects passed via calls and returns is several times higher than the number of objects potentially thrown. Thus, the analysis is allocating its reasoning effort in a way that closely matches what one would expect intuitively, possibly indicating that large further improvements are unlikely.

5 Related Work

We next discuss in more detail some of the past work in points-to analysis combined with exceptions.

As mentioned earlier, points-to analysis frameworks Spark [19] and Paddle [18,20] both use imprecise exception analysis via assignment of thrown exceptions to a single global variable. Even if this were to change to distinct per-type variables, it would still have significant precision shortcomings compared to our approach since the order and scope of exception handlers would be ignored. The Soot framework also has a separate exception analysis [16] that is not based on a pointer analysis.

The IBM Research Wala [6] static analysis library supports several different pointer analysis configurations. The points-to analyses of Wala support computing which exceptions a method can throw (analogously to our ThrowPointsTo relation), but no results of Wala's accuracy or speed have been reported in the literature. Wala allows an exception object to be represented by-type (analogously to our coarsening) but it is unclear how the underlying analysis compares to our joint exception and points-to analysis. It will be interesting to compare our analyses to Wala in future work.

Type-based approaches to dealing with exception objects have also been explored before [14,15], in the context of a separate exception analysis (i.e., not jointly with a precise points-to analysis and not in comparison to an object-based exception representation).

bddbddb is a Datalog and BDD-based database that has been employed for points-to analysis [29,30]. The publications do not discuss exception analysis, yet the bddbddb distribution examples do propagate exceptions over the control-flow graph. One of the differences between Doop and bddbddb is that Doop expresses the entire analysis in Datalog and only relies on basic input facts. In contrast, the points-to analyses of bddbddb largely rely on pre-computed input facts, such as a call-graph, reducing the Datalog analysis to just a few lines of code for propagating points-to data. For exception analysis, bddbddb ignores the order of exception handlers and also disables filtering of caught exceptions. Both of these features are crucial for precision.

Chatterjee et al. analyze the worst-case complexity of fully-precise pointer analysis with exceptions [3]. This is a theoretical analysis with no current application to practical points-to algorithms.

Sinha et al. discuss how to represent exception flow in the control-flow graph [27]. One of the topics is handling `finally` clauses. We analyze Java bytecode, hence the complex control-flow of `finally` clauses is already handled by the Java compiler.

Choi et al. suggested a compact intraprocedural control-flow representation that collapses the large number of edges to exceptions handlers [4]. Our analyses are interprocedural and flow-insensitive, so not directly comparable to that work.

6 Conclusions

When analyzing an object-oriented program, exceptions pose an interesting challenge. If completely ignored, valuable properties of the program are lost and large amounts of code appear unexercised. If handled in isolation (either before or after points-to analysis) the result is imprecise and the analysis suffers from inefficiency. A joint points-to and exception analysis offers the answer but has significant time and space cost due to the precise representation of exception objects. We showed that we can profitably coarsen the representation of exception objects in such a joint analysis. Precision remains unaffected, for the parts of the analysis not directly pertaining to exception objects, i.e., for most common analysis clients, such as cast elimination, devirtualization, and call-graph construction. At the same time, performance is significantly enhanced. Thus the approach is a clear win and is now the default policy for exception handling in the Doop framework.

There are interesting avenues for further work along the directions of the paper. Our approach is based on standard patterns of use of exception objects. These patterns can perhaps be generalized to other kinds of objects used as "message carriers". Furthermore, the question arises of how such patterns translate across languages. Is there an analogous concept in functional languages that can be exploited to gain scalability? Also, it remains to be seen whether similar approaches can apply to alias analysis in C++ in the presence of exceptions.

Acknowledgments. We gratefully acknowledge funding by the European Union under a Marie Curie International Reintegration Grant and a European Research Council Starting/Consolidator grant. We also gratefully acknowledge the support of the Greek Secretariat for Research and Technology under an Excellence (Aristeia) award. The work in this paper was significantly aided by pre-existing mechanisms that Martin Bravenboer has implemented in the Doop framework.

References

1. Bravenboer, M., Smaragdakis, Y.: Exception analysis and points-to analysis: Better together. In: Dillon, L. (ed.) ISSTA 2009: Proceedings of the 2009 International Symposium on Software Testing and Analysis, New York, NY, USA (July 2009)

2. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: OOPSLA 2009: 24th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications. ACM, New York (2009)
3. Chatterjee, R., Ryder, B.G., Landi, W.A.: Complexity of points-to analysis of Java in the presence of exceptions. IEEE Trans. Softw. Eng. 27(6), 481–512 (2001)
4. Choi, J.D., Grove, D., Hind, M., Sarkar, V.: Efficient and precise modeling of exceptions for the analysis of Java programs. SIGSOFT Softw. Eng. Notes 24(5), 21–31 (1999)
5. Eichberg, M., Kloppenburg, S., Klose, K., Mezini, M.: Defining and continuous checking of structural program dependencies. In: ICSE 2008: Proc. of the 30th Int. Conf. on Software Engineering, pp. 391–400. ACM, New York (2008)
6. Fink, S.J., et al.: T.J. Watson libraries for analysis (WALA), http://wala.sourceforge.net
7. Fu, C., Milanova, A., Ryder, B.G., Wonnacott, D.G.: Robustness testing of Java server applications. IEEE Trans. Softw. Eng. 31(4), 292–311 (2005)
8. Fu, C., Ryder, B.G.: Exception-chain analysis: Revealing exception handling architecture in Java server applications. In: ICSE 2007: Proceedings of the 29th International Conference on Software Engineering, pp. 230–239. IEEE Computer Society, Washington, DC (2007)
9. Fu, C., Ryder, B.G., Milanova, A., Wonnacott, D.: Testing of Java web services for robustness. In: ISSTA 2004: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 23–34. ACM, New York (2004)
10. Guarnieri, S., Livshits, B.: GateKeeper: mostly static enforcement of security and reliability policies for Javascript code. In: Proceedings of the 18th USENIX Security Symposium, SSYM 2009, pp. 151–168. USENIX Association, Berkeley (2009), http://dl.acm.org/citation.cfm?id=1855768.1855778
11. Hajiyev, E., Verbaere, M., de Moor, O.: *codeQuest:* Scalable Source Code Queries with Datalog. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 2–27. Springer, Heidelberg (2006)
12. Hardekopf, B., Lin, C.: The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In: PLDI 2007: Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 290–299. ACM, New York (2007)
13. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. In: POPL 2009: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 226–238. ACM, New York (2009)
14. Jo, J.-W., Chang, B.-M.: Constructing Control Flow Graph for Java by Decoupling Exception Flow from Normal Flow. In: Laganá, A., Gavrilova, M.L., Kumar, V., Mun, Y., Tan, C.J.K., Gervasi, O. (eds.) ICCSA 2004. LNCS, vol. 3043, pp. 106–113. Springer, Heidelberg (2004)
15. Jo, J.W., Chang, B.M., Yi, K., Choe, K.M.: An uncaught exception analysis for Java. Journal of Systems and Software 72(1), 59–69 (2004)
16. Jorgensen, J.: Improving the precision and correctness of exception analysis in Soot. Tech. Rep. 2003-3, McGill University (September 2004)
17. Lam, M.S., Whaley, J., Livshits, V.B., Martin, M.C., Avots, D., Carbin, M., Unkel, C.: Context-sensitive program analysis as database queries. In: PODS 2005: Proc. of the Twenty-Fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 1–12. ACM, New York (2005)
18. Lhoták, O.: Program Analysis using Binary Decision Diagrams. Ph.D. thesis, McGill University (January 2006)
19. Lhoták, O., Hendren, L.: Scaling Java Points-to Analysis Using SPARK. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 153–169. Springer, Heidelberg (2003)
20. Lhoták, O., Hendren, L.: Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. ACM Trans. Softw. Eng. Methodol. 18(1), 1–53 (2008)

21. Madsen, M., Livshits, B., Fanning, M.: Practical static analysis of Javascript applications in the presence of frameworks and libraries. Tech. Rep. MSR-TR-2012-66, Microsoft Research (Jully 2012)
22. Might, M., Smaragdakis, Y., Van Horn, D.: Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In: Conf. on Programming Language Design and Implementation (PLDI), pp. 305–315. ACM (June 2010)
23. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM Trans. Softw. Eng. Methodol. 14(1), 1–41 (2005)
24. Reps, T.: Demand interprocedural program analysis using logic databases. In: Ramakrishnan, R. (ed.) Applications of Logic Databases, pp. 163–196. Kluwer Academic Publishers (1994)
25. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Muchnick, S.S., Jones, N.D. (eds.) Program Flow Analysis, pp. 189–233. Prentice-Hall, Inc., Englewood Cliffs (1981)
26. Shivers, O.: Control-Flow Analysis of Higher-Order Languages. Ph.D. thesis, Carnegie Mellon University (May 1991)
27. Sinha, S., Harrold, M.J.: Analysis and testing of programs with exception handling constructs. IEEE Trans. Softw. Eng. 26(9), 849–871 (2000)
28. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: Understanding object-sensitivity (the making of a precise and scalable pointer analysis). In: ACM Symposium on Principles of Programming Languages (POPL), pp. 17–30. ACM Press (January 2011)
29. Whaley, J., Avots, D., Carbin, M., Lam, M.S.: Using Datalog with Binary Decision Diagrams for Program Analysis. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 97–118. Springer, Heidelberg (2005)
30. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PLDI 2004: Proc. of the ACM SIGPLAN 2004 Conf. on Programming Language Design and Implementation, pp. 131–144. ACM, New York (2004)

An Incremental Points-to Analysis with CFL-Reachability

Yi Lu[1], Lei Shang[1], Xinwei Xie[1,2], and Jingling Xue[1]

[1] Programming Languages and Compilers Group
School of Computer Science and Engineering
University of New South Wales, Sydney, NSW 2052, Australia
{ylu,shangl,xinweix,jingling}@cse.unsw.edu.au
[2] School of Computer Science
National University of Defence Technology, Changsha 410073, China

Abstract. Developing scalable and precise points-to analyses is increasingly important for analysing and optimising object-oriented programs where pointers are used pervasively. An incremental analysis for a program updates the existing analysis information after program changes to avoid reanalysing it from scratch. This can be efficiently deployed in software development environments where code changes are often small and frequent. This paper presents an incremental approach for demand-driven context-sensitive points-to analyses based on Context-Free Language (CFL) reachability. By tracing the CFL-reachable paths traversed in computing points-to sets, we can precisely identify and recompute on demand only the points-to sets affected by the program changes made. Combined with a flexible policy for controlling the granularity of traces, our analysis achieves significant speedups with little space overhead over reanalysis from scratch when evaluated with a null dereferencing client using 14 Java benchmarks.

1 Introduction

Points-to analysis is a static program analysis technique to approximate the set of memory locations that may be pointed or referenced by program variables, which is crucial to software testing, debugging, program understanding and optimisation. But performing precise points-to analysis is an expensive activity, even for small programs. Developing scalable and precise points-to analyses is increasingly important for analysis and optimisation of object-oriented programs where pointers are used pervasively.

Points-to analysis has been studied extensively in order to improve its scalability, precision or tradeoffs [15,17,33,34], and continues to attract significant attention [10,9,26,25,27,35,30,37,39]. The majority of the previous solutions perform a whole-program points-to analysis to exhaustively compute points-to information for all variables in the program, which is often too resource-intensive in practice, especially for large programs. Some recent efforts have focused on demand-driven points-to analysis [11,26,27,37], which mostly rely on *Context-Free Language (CFL) reachability* [22] to perform only the necessary work for a set of variables queried by a client rather than a whole-program analysis to find the points-to information for all its variables.

Incremental static analysis seeks to efficiently update existing analysis information about an evolving software system without recomputing from scratch [4], allowing the previously computed information to be reused. Incremental analysis is especially important for large projects in a software development environment where it is necessary to maintain a global analysis in the presence of small and frequent edits. Several solutions have been proposed by using incremental elimination [3,5], restarting iteration [20], a combination of these two techniques [18], timestamp-based backtracing [13], and logic program evaluation [24]. In this paper, we introduce an incremental approach for points-to analyses based on CFL-reachability.

Many program analysis problems can be solved by transforming them into graph reachability problems [23]. In particular, CFL-reachability is an extension of graph reachability. To perform points-to analysis with CFL-reachability, a program is represented by a *Pointer Assignment Graph (PAG)*, a directed graph that records pointer flow in a program. An object is in the points-to set of a variable if there is a reachable path between them in the PAG, which must be labelled with a string in a specified CFL. Such points-to analysis is typically field-sensitive (by matching load/store edges on the same field), context-sensitive (by matching entry/exit edges for the same call site) and heap-sensitive (by distinguishing the same abstract object from different call paths).

Pointer analyses derived from a CFL-reachability formulation achieve very high precision and are efficient for a small number of queries raised in small programs, but they do not scale well to answer many queries for large programs. Existing solutions address the performance issue from several directions, by using refinement [27,28], (whole-program) pre-analysis [36], ad hoc caching [41], and procedural summarisation [26,25,37]. In this paper, we tackle this issue from a different angle. Our goal is to develop an incremental technique for boosting the performance of points-to analysis by reusing previously computed points-to sets.

In this paper, we combine incremental analysis with graph reachability to obtain naturally a trace-based incremental mechanism for points-to analysis, which is effective and simple to implement. The key to incremental analysis lies in approximating dependency information for analysis results. By observing that each points-to query in a CFL-reachability-based analysis is answered by finding the CFL-reachable paths in the PAG from the queried variable to objects, we trace the set of nodes in the traversed paths that the query depends on. Upon code changes, we can precisely identify and recompute on demand only those queries whose traces may overlap with the changes made. Such trace-based falsification minimises the impact of changes on previously computed points-to sets, avoiding unnecessarily falsifying unaffected queries to make them reusable after code changes. Our approach can support efficiently multiple changes with overlapping traces, since multiple changes usually exhibit locality [40].

Precise tracing is costly in space, because it potentially involves thousands of nodes in a PAG for each query, which may render the whole incremental approach impractical, especially for answering many queries in large programs. It is therefore useful to allow tradeoffs between time and space to be made in an incremental analysis. Based on the observation that a large portion of the analysis is performed on Java library code, which is less likely to be changed, we introduce a flexible policy to control the granularities of traces by approximating the variable nodes with their scopes (e.g., methods, classes, etc.). Such trace policies describe different granularity levels used for different parts of the program; they may be specified by programmers as an input to our analysis, or inferred adaptively based on the frequency of code changes. Typically a finer (e.g., variable-level) granularity may be used for the code that is more likely to be changed frequently (e.g., for the classes being developed) to minimise falsification and recomputation required after code changes, while a coarser (e.g., package-level) granularity may be used for the code that is less frequently edited (e.g., for the classes in libraries) to minimise the space required for storing the traces as required. In our experiments, we find that only a small part of code needs to use finer granularities. By using the appropriate granularities for different parts of the programs, we are able to maintain sufficient dependency information for precise falsification with little memory overhead.

In summary, this paper makes the following contributions:

- We propose a trace-based incremental approach for points-to analysis by exploiting graph reachability. To our knowledge, this is the first points-to analysis with CFL-reachability that allows previously computed points-to sets to be reused.
- We introduce a flexible trace policy to approximate traces. Programmers may take advantage of domain knowledge to control their granularities for different parts of the program. We also describe an adaptive technique to automatically infer the policy based on the frequency of changes. Trace policies can significantly reduce the size of traces without unnecessarily increasing the chances of falsification.
- We have implemented our incremental analysis in *Soot-2.5.0*, a Java optimisation and analysis framework, and compared it with a state-of-the-art from-scratch analysis, REFINEPTS, introduced in [27] using a null dereferencing client in the presence of small code changes. For a single deletion of a class/method/statement, our incremental analysis achieves an average speedup of 78.3X/60.1X/3195.4X.

The rest of the paper is organised as follows. We introduce the background information on CFL-reachability and PAGs in Section 2. Section 3 introduces reachability traces by example. Section 4 presents our trace-based incremental analysis, including trace policies. Experimental results are presented and analysed in Section 5 with related work discussed in Section 6, followed by a brief conclusion in Section 7.

2 CFL-Reachability

We introduce the state-of-the-art points-to analysis for Java formulated in terms of CFL-reachability [26,27,28,36] which uses Spark's PAGs [17] as the program representation. In Section 2.1, we consider the syntax of PAGs and how to represent a Java example as a PAG. In Section 2.2, we describe the CFL-reachability formulation and show how to answer points-to queries by finding reachable paths in the PAG of our example.

2.1 Program Representation

Points-to analysis for Java is typically flow-insensitive, field-sensitive and context-sensitive (for both method calls and heap abstraction) to balance the precision and efficiency for demand queries. When an analysis is flow-insensitive, control-flow statements are irrelevant and thus ignored.

Local variables	x, y	Allocation sites	o
Global variables	g	Call sites	i
Variables	$v ::= x \mid g$	Instance fields	f
Nodes	$n ::= o \mid v$		

Edges
$$\begin{aligned} e ::= {} & x \xleftarrow{\textsf{new}} o \mid x \xleftarrow{\textsf{assign}} y \mid v \xleftarrow{\textsf{global}} g \mid g \xleftarrow{\textsf{global}} v \\ & \mid x \xleftarrow{\textsf{ld}(f)} y \mid x \xleftarrow{\textsf{st}(f)} y \mid x \xleftarrow{\textsf{entry}_i} y \mid x \xleftarrow{\textsf{exit}_i} y \end{aligned}$$

Fig. 1. Syntax of PAG

In its canonical form, a Java program is represented by a directed graph, known as a Pointer Assignment Graph (PAG), which has threes types of nodes: objects, local variables and global variables. The syntax of PAG is given in Fig. 1.

All edges are oriented in the direction of value flow, representing the statements in the program. For example, $x \xleftarrow{\textsf{new}} o$ indicates the flow of o into x (an assignment from an allocation site o to a local variable x). As a result, x points directly to o. An assign edge represents an assignment between local variables (e.g., $x = y$), so x points to whatever y points to. In a global edge, one or both variables are static variables in a class. A ld edge reads an instance field f (e.g., $x = y.f$) while a st edge writes to an instance field f (e.g., $x.f = y$), where x and y are both local variables. An $\textsf{entry}_i$ edge represents a binding of a (local) actual parameter y to its corresponding formal parameter x for a call at the call site i. Similarly, an $\textsf{exit}_i$ edge represents a call return where the (local) return value in y is bound to the local variable x at the call site i.

Fig. 2 gives an example, extending the original example in [27], which provides an abstraction for the Java container pattern. The `AddrBook` class makes use of two vectors. In lines 42–45, an `AddrBook` object created is assigned to `p` and populated with a pair of objects: one with type `String` and the other with type `Integer`. In lines 46 and 47, calling `getName`/`getNum` results in `v1` = `n` and `v2` = `c`. Note that loads and stores to array elements are modeled by collapsing all elements into a special field arr.

For this example, its PAG is shown in Fig. 3. Some notations are in order: (1) o_i denotes the abstract object o created at the allocation site in line i; (2) for temporary variables (e.g., `ret` and `tmp`), the implicit self variable (`this`) and local variables (declared in different scopes), we subscript them with their method names.

2.2 Points-to Analysis with CFL-Reachability

CFL-reachability [22,38] is an extension of graph reachability that is equivalent to the reachability problem formulated in terms of either recursive state machines [7] or set constraints [14]. Each reachable path in a PAG has a string formed by concatenating in order the labels of edges in the path, where load/store pairs on the same field must be matched (field sensitivity) and entry/exit pairs for the same callsite must be matched (context sensitivity). An object is in the points-to set of a variable if there is a reachable (or *flowsTo*) path from the object to the variable. Two variables may be aliases if there is a reachable path from an object to both of them.

Field Sensitivity. Let us start by considering a PAG G with only new and assign. It suffices to develop a regular language, L_{FT} (FT for flows-to), such that if an object

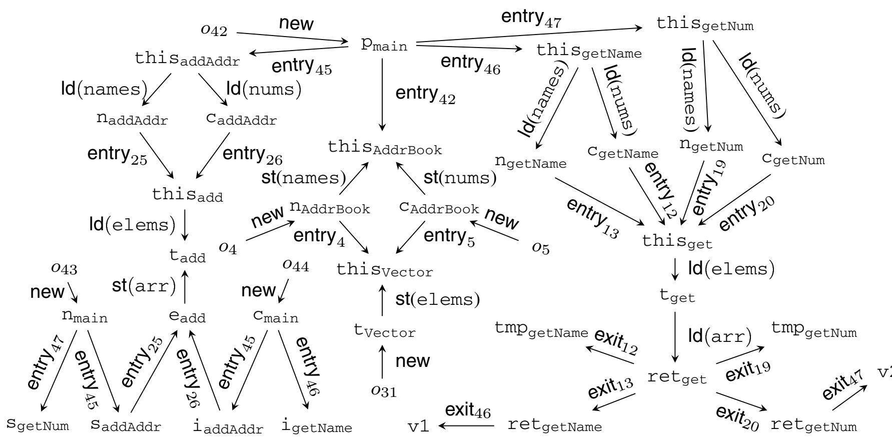
```
1 class AddrBook{
2   private Vector names, nums;
3   AddrBook(){
4     n = new Vector();
5     c = new Vector();
6     this.names = n;
7     this.nums = c; }
8   Object getName(Integer i){
9     n = this.names;
10    c = this.nums;
11    for (int j=0;j<c.count;j++)
12      if (c.get(j)==i)
13        return n.get(j);
14    return null; }
15  Object getNum(String s){
16    n = this.names;
17    c = this.nums;
18    for (int i=0;i<n.count;i++)
19      if (n.get(i)==s)
20        return c.get(i);
21    return null; }
22  void addAddr(Object s, Object i){
23    n = this.names;
24    c = this.nums;
25    n.add(s);
26    c.add(i);
27 }}
28 class Vector{
29   Object[] elems; int count;
30   Vector(){
31     t = new Object[MAXSIZE];
32     this.elems = t; }
33   void add(Object e){
34     t = this.elems;
35     t[count++] = e;    // writes t.arr
36   }
37   Object get(int i){
38     t = this.elems;
39     return t[i];       // reads t.arr
40 }}
41 static void main(String[] args){
42   AddrBook p = new AddrBook();
43   String n =new String("John Smith");
44   Integer c = new Integer(12345);
45   p.addAddr(n,c);
46   String v1 = (String) p.getName(c);
47   Integer v2 = (Integer) p.getNum(n);
48 }
```

(a) original code

```
25    s = new String("Change1"); n.add(s);       // Change 1
26    i = new String("Change2"); c.add(i);       // Change 2
```

(b) code changes

Fig. 2. A Java example



o_{42}
new
$\mathtt{p}_{\mathtt{main}}$
$\mathtt{this}_{\mathtt{addAddr}}$
entry$_{45}$
entry$_{47}$
$\mathtt{this}_{\mathtt{getNum}}$
entry$_{46}$
$\mathtt{this}_{\mathtt{getName}}$
ld(names)
ld(nums)
entry$_{42}$
$\mathtt{n}_{\mathtt{addAddr}}$
$\mathtt{c}_{\mathtt{addAddr}}$
entry$_{25}$
entry$_{26}$
$\mathtt{this}_{\mathtt{AddrBook}}$
$\mathtt{n}_{\mathtt{getName}}$
$\mathtt{c}_{\mathtt{getName}}$
$\mathtt{n}_{\mathtt{getNum}}$
$\mathtt{c}_{\mathtt{getNum}}$
$\mathtt{this}_{\mathtt{add}}$
st(names)
st(nums)
entry$_{12}$
entry$_{19}$
entry$_{20}$
entry$_{13}$
ld(elems)
new
$\mathtt{n}_{\mathtt{AddrBook}}$
$\mathtt{c}_{\mathtt{AddrBook}}$
new
entry$_4$
entry$_5$
$\mathtt{this}_{\mathtt{get}}$
$\mathtt{t}_{\mathtt{add}}$
o_4
o_5
o_{43}
o_{44}
$\mathtt{this}_{\mathtt{Vector}}$
ld(elems)
new
st(arr)
new
st(elems)
$\mathtt{t}_{\mathtt{get}}$
$\mathtt{n}_{\mathtt{main}}$
$\mathtt{e}_{\mathtt{add}}$
$\mathtt{c}_{\mathtt{main}}$
$\mathtt{t}_{\mathtt{Vector}}$
$\mathtt{tmp}_{\mathtt{getName}}$
exit$_{12}$
ld(arr)
$\mathtt{tmp}_{\mathtt{getNum}}$
entry$_{47}$
entry$_{45}$
entry$_{25}$
entry$_{26}$
entry$_{45}$
entry$_{46}$
new
$\mathtt{ret}_{\mathtt{get}}$
exit$_{19}$
exit$_{47}$
$\mathtt{v2}$
o_{31}
exit$_{13}$
exit$_{46}$
exit$_{20}$
$\mathtt{s}_{\mathtt{getNum}}$
$\mathtt{s}_{\mathtt{addAddr}}$
$\mathtt{i}_{\mathtt{addAddr}}$
$\mathtt{i}_{\mathtt{getName}}$
$\mathtt{v1}$
$\mathtt{ret}_{\mathtt{getName}}$
$\mathtt{ret}_{\mathtt{getNum}}$

Fig. 3. Complete PAG for the original code

o can flow to a variable v during the execution of the program, then v will be L_{FT}-reachable from o in G. Then we have the following (regular) grammar for L_{FT}:

$$\mathit{flowsTo} \rightarrow \mathsf{new}\ (\ \mathsf{assign})^*$$

If o *flowsTo* v, then o belongs to the points-to set of v.

For field accesses, precise handling of heap accesses is formulated with the updated L_{FT} being a CFL of *balanced parentheses* [27]. Two variables may be aliases if an object may flow to both of them. Thus, v may point to o flowing into v' if there exists two statements $x.f = v'$ and $v = y.f$, such that the base variables x and y are aliases. So o flows through the two statements with a pair of parentheses (i.e., $\mathsf{st}(f)$ and $\mathsf{ld}(f)$), first into v' and then into v. Therefore, the *flowsTo* production is extended into:

$$\mathit{flowsTo} \rightarrow \mathsf{new}\ (\ \mathsf{assign} \mid \mathsf{st}(f)\ \mathit{alias}\ \mathsf{ld}(f))^*$$

where x *alias* y means that x and y can be aliases. To allow *alias* paths in an *alias* language, $\overline{\mathit{flowsTo}}$ is introduced as the inverse of the *flowsTo* relation. A *flowsTo*-path ρ can be inverted to obtain its corresponding $\overline{\mathit{flowsTo}}$-path $\overline{\rho}$ using inverse edges, and vice versa. For each edge $x \overset{\ell}{\leftarrow} y$ in ρ (where ℓ is the label of the edge), its inverse edge is $y \overset{\overline{\ell}}{\leftarrow} x$ in $\overline{\rho}$. Thus, o *flowsTo* x iff x $\overline{\mathit{flowsTo}}$ o. This means that $\overline{\mathit{flowsTo}}$ actually represents the standard points-to relation. As a result, a $\overline{\mathit{flowsTo}}$-path represents a *points-to path*. (To avoid cluttering, the inverse edges in a PAG, such as the one given in Fig. 3, are not shown explicitly.) As a result, x *alias* y iff x $\overline{\mathit{flowsTo}}$ o *flowsTo* y:

$$\begin{aligned} \mathit{alias} &\rightarrow \overline{\mathit{flowsTo}}\ \mathit{flowsTo} \\ \overline{\mathit{flowsTo}} &\rightarrow (\ \overline{\mathsf{assign}} \mid \overline{\mathsf{ld}(f)}\ \mathit{alias}\ \overline{\mathsf{st}(f)})^*\ \overline{\mathsf{new}} \end{aligned}$$

Context Sensitivity. A context-sensitive analysis requires call entries and exits to be matched, which is solved also as a balanced-parentheses problem [22]. This is done by filtering out *flowsTo*- and $\overline{\mathit{flowsTo}}$-paths corresponding to unrealisable paths. A realisable path may not start and end in the same method. So partially balanced parentheses, i.e., a prefix (suffix) with unbalanced closed (open) parentheses, are allowed.

To compute the points-to set of a variable v, we simply solve a CFL-reachability problem for L_{FT} context-sensitively to find the set of allocation sites o such that v is L-reachable from o. The analysis is *fully* context-sensitive not only for method invocation but for heap abstraction (by distinguishing allocation sites with calling contexts).

Example. We use the PAG of our example in Fig. 3 to show how to resolve some simple points-to relations via CFL-reachability. Let us see how to discover o_4 as a pointer target for $\mathtt{n}_{\mathtt{addAddr}}$. In Fig. 2, o_{42} flows to $\mathtt{p}_{\mathtt{main}}$, which is the actual parameter passed to the formal parameter $\mathtt{this}_{\mathtt{AddrBook}}$ of constructor `AddrBook` and $\mathtt{this}_{\mathtt{addAddr}}$ of `addAddr`. So $\mathtt{this}_{\mathtt{AddrBook}}$ *alias* $\mathtt{this}_{\mathtt{addAddr}}$. This fact is found in L_{FT} because

$$\mathtt{this}_{\mathtt{AddrBook}}\ \overline{\mathsf{entry}_{42}}\ \mathtt{p}_{\mathtt{main}}\ \overline{\mathsf{new}}\ o_{42}\ \mathsf{new}\ \mathtt{p}_{\mathtt{main}}\ \mathsf{entry}_{45}\ \mathtt{this}_{\mathtt{addAddr}}$$

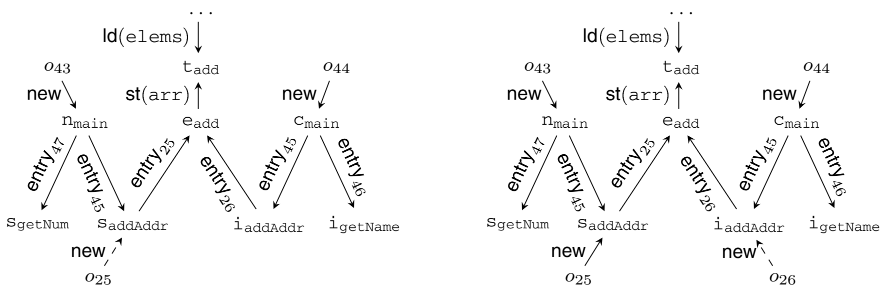
We then know that o_4 *flowsTo* $\mathtt{n}_{\mathtt{addAddr}}$ since L_{FT} has the *flowsTo*-path:

$$o_4\ \mathsf{new}\ \mathtt{n}_{\mathtt{AddrBook}}\ \mathsf{st}(\mathtt{names})\ \mathtt{this}_{\mathtt{AddrBook}}\ \mathit{alias}\ \mathtt{this}_{\mathtt{addAddr}}\ \mathsf{ld}(\mathtt{names})\ \mathtt{n}_{\mathtt{addAddr}}$$

This *flowsTo*-path is a realisable path. So $\mathtt{n}_{\mathtt{addAddr}}$ points to o_4.

3 Tracing CFL-Reachability: An Example

Most points-to analyses only consider fixed programs. We illustrate how we cope with program changes using the example given in Fig. 2. There are two changes made to the



(a) after Change 1 (b) after Change 2

Fig. 4. Partial PAGs after code changes (marked by the dashed edges introduced)

original code in Fig. 2(a), in order in line 25 and line 26 as shown in Fig. 2(b). We show how these changes impact the points-to sets of `v1` and `v2`. Fig. 4 shows the partial PAGs after each change. We have used dashed arrows to indicate newly added edges.

A points-to query is answered by searching for all reachable paths between objects and the queried variable in a PAG. The answer to the points-to query depends on all nodes in the reachable paths traversed. Changes made on these nodes (by either adding or deleting edges connected to them) may falsify the points-to set of the query. The key to incremental analysis lies in tracking such dependent information.

A straightforward way to track precise dependency information is to explicitly maintain a set of variable nodes on which each points-to query depends, as traces. Let us consider the traces for queries on `v1` and `v2` in Fig. 2 and see how they are affected by code changes. By collecting all distinct variable nodes in the reachable path(s) from o_{43} to `v1` in Fig. 3, we get the trace for `v1`: $\{\texttt{v1}, \texttt{ret}_{\texttt{getName}}, \texttt{ret}_{\texttt{get}}, \texttt{t}_{\texttt{get}}, \texttt{this}_{\texttt{get}}, \texttt{n}_{\texttt{getName}}, \texttt{this}_{\texttt{getName}}, \texttt{p}_{\texttt{main}}, \texttt{this}_{\texttt{AddrBook}}, \texttt{n}_{\texttt{AddrBook}}, \texttt{this}_{\texttt{Vector}}, \texttt{t}_{\texttt{Vector}}, \texttt{this}_{\texttt{addAddr}}, \texttt{n}_{\texttt{addAddr}}, \texttt{this}_{\texttt{add}}, \texttt{t}_{\texttt{add}}, \texttt{e}_{\texttt{add}}, \texttt{s}_{\texttt{addAddr}}, \texttt{n}_{\texttt{main}}\}$ and the trace for `v2`: $\{\texttt{v2}, \texttt{ret}_{\texttt{getNum}}, \texttt{ret}_{\texttt{get}}, \texttt{t}_{\texttt{get}}, \texttt{this}_{\texttt{get}}, \texttt{c}_{\texttt{getNum}}, \texttt{this}_{\texttt{getNum}}, \texttt{p}_{\texttt{main}}, \texttt{this}_{\texttt{AddrBook}}, \texttt{c}_{\texttt{AddrBook}}, \texttt{this}_{\texttt{Vector}}, \texttt{t}_{\texttt{Vector}}, \texttt{this}_{\texttt{addAddr}}, \texttt{c}_{\texttt{addAddr}}, \texttt{this}_{\texttt{add}}, \texttt{t}_{\texttt{add}}, \texttt{e}_{\texttt{add}}, \texttt{i}_{\texttt{addAddr}}, \texttt{c}_{\texttt{main}}\}$.

Change 1 adds a new edge to the local variable $\texttt{s}_{\texttt{addAddr}}$. Since the variable is in the trace of `v1`, after the change, the points-to set of `v1` must be falsified and recomputed. However, the trace of `v2` does not contain $\texttt{s}_{\texttt{addAddr}}$. Thus, its points-to set is still valid and reusable. Similarly, change 2 adds a new edge to the local variable $\texttt{i}_{\texttt{addAddr}}$. This falsifies the points-to set of `v2` without affecting `v1` .

Tracing all nodes is costly in space as traces may be large for large programs. Instead of tracking precise dependencies, we approximate the variables in a trace using their scopes, based on a predefined policy. Trace policies control the granularities of traces so that their sizes can be significantly reduced to trade time for space.

Policies are formed by a set of program units (variables or their scopes), which specify what may appear in traces. For example, if the policy for an analysis contains a method name m, when nodes n_1 and n_2 are reached in computing a points-to set such that n_1 and n_2 are contained (defined) in m, m (instead of n_1 and n_2) is tracked in the trace of the points-to set. The default granularity is package-level. For example, if n is in a reachable path but not contained in any program unit in the given policy, then

n's package name is used in the trace. This is particularly useful for specifying an appropriate granularity for libraries. We do not have to explicitly include anything from libraries in the policy. They are by default tracked at the coarsest package-level granularity, because they are the least likely to change. For applications being developed in an interactive programming environment, it is natural to use a finer granularity.

Let us now consider how traces and falsifications are enforced by trace policies. We define a sample trace policy for analysing the Java example in Fig. 2: {main, AddrBook.AddrBook, getName, getNum, addAddr}, which uses method-level granularity for the AddrBook class and the main method (they are considered as application code in contrast to library code). Note that Vector is not explicitly mentioned in the policy, since it is considered as part of library code. As a result, the default package-level granularity is used to track the changes on Vector. In Section 4.2, we introduce some forms of shorthand to simplify the specification of policies.

By applying this policy (assuming that the Vector class is defined in package util), the trace for v1 becomes much smaller: {main, getName, AddrBook.AddrBook, addAddr, util} and the trace for v2 is also smaller: {main, getNum, AddrBook.AddrBook, addAddr, util} . Clearly, either Change 1 or Change 2 may falsify both v1 and v2, because we have used a method-level granularity for the changes made in addAddr. It is possible to define a finer-grained policy for the code being changed, but it may not always be possible to anticipate where changes will be made.

Policies can be inferred automatically based on the frequency of code changes in different parts of a program. Typically finer-grained policies may be inferred for frequently edited code and coarser-grained policies for code that is less likely to be modified. Therefore, the impact of changes on the points-to information related to the frequently changed code may be kept to a minimum.

Let us show how to infer policies adaptively. The initial policy is empty (or supplied by the programmer) and the traces of v1 and v2 are {my_package, util}, assuming that AddrBook and main are defined in the my_package package. After Change 1 at line 25, the policy is adaptively changed to {$\mathtt{this}_{\mathtt{addAddr}}$, $\mathtt{s}_{\mathtt{addAddr}}$, $\mathtt{i}_{\mathtt{addAddr}}$, $\mathtt{n}_{\mathtt{addAddr}}$, $\mathtt{c}_{\mathtt{addAddr}}$, getName, AddrBook.AddrBook, getNum, addAddr, AddrBook, main} by adding finer-grained program units into the policy based on the change made.

Since code changes often exhibit locality, we choose a simple heuristic to reduce the chance of falsification for units that are closely related to a certain change. When a program unit is involved in a change, we add all units directly defined in its enclosing scopes. For example, Change 1 affects variable $\mathtt{s}_{\mathtt{addAddr}}$ and transitively all its enclosing scopes: method addAddr, class AddrBook and package my_package. Therefore, all variables defined in addAddr, all methods defined in AddrBook, and all classes defined in my_package are added into the new policy.

After Change 1 at line 25, both v1 and v2 are conservatively falsified, because the change overlaps with my_package in both traces. After recomputing the points-to sets for v1 and v2 using the new policy, the new trace of v1 is {$\mathtt{s}_{\mathtt{addAddr}}$, $\mathtt{this}_{\mathtt{addAddr}}$, $\mathtt{n}_{\mathtt{addAddr}}$, AddrBook.AddrBook, getName, main, util} , and the new trace of v2 is {AddrBook.AddrBook, getNum, $\mathtt{this}_{\mathtt{addAddr}}$, $\mathtt{c}_{\mathtt{addAddr}}$, $\mathtt{i}_{\mathtt{addAddr}}$, main, util} .

After Change 2 at line 26, an incremental analysis is used again. This time only the points-to set of v2 is falsified, because $\mathtt{i}_{\mathtt{addAddr}}$ does not overlap with the trace of v1. The previous points-to set for v2 before the change was $\{o_{44}\}$. We knew that the typecasting at line 47 was safe because the type of o_{44} was Integer; we could omit a runtime check for this cast. After recomputing the query, the points-set of v2 becomes $\{o_{44}, o_{26}\}$, where o_{26} is introduced by Change 2. Since o_{26} is a String, we know that the typecasting at line 47 may no longer be safe.

4 Incremental Analysis with Reachability Traces

In this section, we describe our incremental analysis formally using inference rules [11,28,24]. Our goal is to incrementalise the points-to analysis based on CFL-reachability.

In Section 4.1, we first introduce a simple form of incremental points-to analysis based on reachability traces, where all nodes in the reachable paths traversed in computing the points-to sets are traced. In Section 4.2, we then introduce a space-efficient analysis by approximating traces using policies. Finally, in Section 4.3, we show how to infer trace policies adaptively with each incremental analysis.

Our incremental analysis proceeds in two phases: initial phase and incremental phase. The initial phase initialises the whole analysis by answering all queries from scratch, and the answers (points-to sets) are cached for reuse. This occurs only when a new program is analysed or after major program changes, where it is necessary to reinstall the whole analysis. Unlike the initial phase, the incremental phase performs falsification in addition to points-to analysis. This occurs after small changes and only recomputes a small number of cached answers that are falsified by the changes made.

4.1 Points-to Analysis with Reachability Traces

We have developed our approach based on insights gained from formulating points-to analysis as a graph reachability problem. Our CFL-reachability-based analysis computes both points-to information and traces. The additional syntax is given in Fig. 5.

Contexts	$k ::= \varnothing \mid k{:}i$
Traces/Changes	$\tau, \Delta ::= \varnothing \mid \{\mu\} \mid \tau \cup \tau$
Program units	$\mu ::= v$
Points-to sets	$\sigma ::= \varnothing \mid \{o\} \mid \sigma \cup \sigma$
Stores	$\Sigma ::= \varnothing \mid \Sigma, v \mapsto (\sigma, \tau)$

Fig. 5. Additional syntax for points-to analysis

Contexts represent how method calls are made. A calling context k is a finite stack of call sites, whose order is significant. Traces track nodes traversed in reachable paths in computing points-to sets. A trace τ is a set of variable nodes (we do not track object nodes), whose order is not significant. A points-to set σ contains a set of objects cached in a store Σ, which maps variables to their points-to sets and traces.

In addition to performing the standard CFL-reachability-based points-to analysis in the PAG of a program, we maintain the traces along the search. Our points-to analysis is described in Fig. 6 by a set of inference rules in the form of:

$$n, k \overset{\tau}{\Longrightarrow} n', k'$$

$$\frac{x \xrightarrow{\overline{\mathsf{new}}} o}{x, k \overset{\{x\}}{\Longrightarrow} o, k} \quad \text{(new)} \qquad \frac{x \xrightarrow{\overline{\mathsf{entry}_i}} y}{x, \varnothing \overset{\{x\}}{\Longrightarrow} y, \varnothing} \quad \text{(entry-}\varnothing\text{)}$$

$$\frac{x \xrightarrow{\overline{\mathsf{assign}}} y}{x, k \overset{\{x\}}{\Longrightarrow} y, k} \quad \text{(assign)} \qquad \frac{x \xrightarrow{\overline{\mathsf{exit}_i}} y}{x, k \overset{\{x\}}{\Longrightarrow} y, k{:}i} \quad \text{(exit)}$$

$$\frac{v \xrightarrow{\overline{\mathsf{global}}} g}{v, k \overset{\{v\}}{\Longrightarrow} g, \varnothing} \quad \text{(global-r)}$$

$$\frac{x \xrightarrow{\overline{\mathsf{ld}(f)}} x' \qquad y' \xrightarrow{\overline{\mathsf{st}(f)}} y \qquad x', k \overset{\tau}{\Longrightarrow} o, k'' \qquad y', k' \overset{\tau'}{\Longleftarrow} o, k''}{x, k \xRightarrow{\{x\} \cup \tau \cup \tau'} y, k'} \quad \text{(field)}$$

$$\frac{g \xrightarrow{\overline{\mathsf{global}}} v}{g, \varnothing \overset{\{g\}}{\Longrightarrow} v, \varnothing} \quad \text{(global-l)}$$

$$\frac{n, k \overset{\tau}{\Longrightarrow} n'', k'' \qquad n'', k'' \overset{\tau'}{\Longrightarrow} n', k'}{n, k \xRightarrow{\tau \cup \tau'} n', k'} \quad \text{(transitivity)}$$

$$\frac{x \xrightarrow{\overline{\mathsf{entry}_i}} y}{x, k{:}i \overset{\{x\}}{\Longrightarrow} y, k} \quad \text{(entry)}$$

Fig. 6. Points-to analysis with traces

which follow the $\overline{\mathit{flowsTo}}$ paths, i.e., the *flowsTo* paths in the opposite direction in a PAG. Each edge in a $\overline{\mathit{flowsTo}}$ path is translated into one or more inference rules. For example, node n in context k can be reached by node n' in context k' via a set of nodes in trace τ. Traces are computed by tracking nodes along the traversal. To save space, object nodes o tracked by (new) do not need to appear in a trace as they can be uniquely identified by their corresponding left-hand side variables x that appear in the trace.

Global variables are context-insensitive (as our analysis is flow-insensitive). Thus, (global-r) and (global-r) skip the sequence of calls and returns between the reads and writes of a global variable. (entry) and (exit) achieve context sensitivity for method invocation by matching call entries and exits. (entry-$\varnothing$) allows for partially balanced parentheses. (field) achieves field sensitivity for field accesses (reads and writes) by matching field loads and stores on field f, only if x' and y' may be aliases (when there is an object o that may be pointed by x' and may flow to y'). In this rule, $\Longleftarrow$ denotes the flows-to analysis which is analogous to its inverse points-to analysis (by making inferences based on traversing the *flowsTo* paths in a PAG) and thus omitted.

Given a points-to query for variable v, we find its point-to set, denoted as $\mathit{Pts}(v)$, by deriving all possible reachable paths ending with some objects:

$$\mathit{Pts}(v) = \{(o, \tau) \mid v, \varnothing \overset{\tau}{\Longrightarrow} o, k\} \qquad \text{(pointsto)}$$

where o is a pointed-to object in context k and τ is the trace for computing it.

Initial Phase. During this phase, all queries are answered by computing points-to sets from scratch. The initial analysis *Initialise* takes a set of queried variables ($v_{1..n}$) as input, and computes and caches each variable's points-to set and trace in the store Σ_n.

$$\frac{\Sigma_0 = \varnothing \qquad \forall i \in 1..n \quad \cdot \quad \mathit{Pts}(v_i) = (o, \tau)_{1..m} \qquad \Sigma_i = \Sigma_{i-1}, v_i \mapsto (\bigcup o_{1..m}, \bigcup \tau_{1..m})}{\mathit{Initialise}(v_{1..n}) = \Sigma_n} \quad \text{(initialisation)}$$

Incremental Phase. Our incremental technique is based on the observation that if a points-to set becomes invalid after a code change, then some part of its trace must be involved in the change. In this phase, the incremental analysis $\mathit{Increment}$ takes the changes Δ (represented by a set of program units that are affected by either additions or deletions of program constructs) and the points-to store Σ_0 from the previous analysis as input, and returns an update-to-date store Σ_n, where only the points-to sets affected by the changes (whose traces τ_i overlap the changes Δ) are recomputed.

$$\frac{\begin{array}{c}\Sigma_0 = (v \mapsto (\sigma, \tau))_{1..n} \qquad \forall i \in 1..n \quad \cdot \\ \mathit{if}\ \ \Delta \perp \tau_i\ \ \mathit{then}\ \begin{cases} \mathit{Pts}(v_i) = (o, \tau')_{1..m} \\ \Sigma_i = \Sigma_{i-1}[v_i \mapsto (\bigcup o_{1..m}, \bigcup \tau'_{1..m})] \end{cases} \\ \mathit{else}\ \ \Sigma_i = \Sigma_{i-1}\end{array}}{\mathit{Increment}(\Delta, \Sigma_0) = \Sigma_n} \quad \text{(increment)}$$

We define the inference rules for determining if the changes overlap with a trace:

$$\frac{\mu \in \tau \qquad \mu' \in \tau' \qquad \{\mu\} \perp \{\mu'\}}{\tau \perp \tau'} \ \text{(overlap-trace)} \qquad\qquad \tau \perp \tau \ \text{(overlap-reflectivity)}$$

Here, traces or changes are only sets of variables, as a program unit μ can only be a variable. In the next section, we will provide a more flexible model to handle different types of program units, such as methods, classes and packages.

4.2 Saving Space with Trace Policies

Trace policies control the granularities of traces in order to trade analysis time for memory usage. In Fig. 7, we introduce method, class and package names into the syntax of program units μ, which form traces τ (and changes Δ). Policies are also formed by a set of program units, which specify what program units may appear in traces.

Method names	m		
Class names	c	Program units	$\mu ::= \cdots \mid m \mid c \mid p$
Package names	p	Policies	$\Gamma ::= \varnothing \mid \mu \mid \Gamma \cup \Gamma$

Fig. 7. Syntax of trace policies

Policies may be defined by programmers. Writing down all program units to be tracked in traces may not be practical. To simplify the specification of policies, we introduce some forms of shorthand, formally defined by the syntactical equivalence:

$$\{\mu : \mathsf{variable}\} \equiv \{v \mid v \trianglelefteq \mu\} \quad \text{(policy-variable)}$$
$$\{\mu : \mathsf{method}\} \equiv \{m \mid m \trianglelefteq \mu\} \quad \text{(policy-method)}$$
$$\{\mu : \mathsf{class}\} \equiv \{c \mid c \trianglelefteq \mu\} \quad \text{(policy-class)}$$

Often programmers may simply specify a single line $\mathit{my\_package}$:method in the policy to indicate that $\mathit{my\_package}$ is the package being developed and request the method-granularity to be used. The shorthand essentially includes all methods contained in $\mathit{my\_package}$. Any other code changes are tracked at package-level, which is the default (avoiding a need for a shorthand).

The containment relation between program units is reflective and transitive. We capture it using the structure of a Java program with a few mappings. P maps a package name to the set of names of all classes defined in the package. C maps a class name to the set of names of all methods and global variables defined in the class. M maps a method name to the set of names of all local variables defined in the method. Given P, C and M, we can easily find containment relations between each pair of program units:

$$\frac{x \in M(m)}{x \trianglelefteq m} \quad \text{(contain-local)} \qquad \frac{c \in P(p)}{c \trianglelefteq p} \quad \text{(contain-class)}$$

$$\frac{g \in C(c)}{g \trianglelefteq c} \quad \text{(contain-global)} \qquad \frac{\mu \trianglelefteq \mu'' \qquad \mu'' \trianglelefteq \mu'}{\mu \trianglelefteq \mu'} \quad \text{(contain-transitivity)}$$

$$\frac{m \in C(c)}{m \trianglelefteq c} \quad \text{(contain-method)} \qquad \mu \trianglelefteq \mu \quad \text{(contain-reflectivity)}$$

Now we define the rules for approximating a trace according to a given policy:

$$\frac{\forall i \in 1..n \quad \cdot \quad \mu_i = \mathit{Approx}(v_i, \Gamma)}{\mathit{Approx}(v_{1..n}, \Gamma) = \bigcup \mu_{1..n}} \quad \text{(approx-trace)}$$

$$\frac{v \trianglelefteq p \qquad \mathit{if}\ v \trianglelefteq \mu\ \mathit{then}\ \mu \notin \Gamma}{\mathit{Approx}(v, \Gamma) = p} \quad \text{(approx-default)}$$

$$\frac{\begin{array}{c} v \trianglelefteq \mu \qquad \mu \in \Gamma \\ \forall \mu' \in \Gamma \quad \cdot \quad \mathit{if}\ v \trianglelefteq \mu'\ \mathit{then}\ \mu \trianglelefteq \mu' \end{array}}{\mathit{Approx}(v, \Gamma) = \mu} \quad \text{(approx-contain)}$$

In (approx-default), if no enclosing scope of v is defined in the policy, then its package is tracked by default. (approx-contain) only finds and tracks the smallest enclosing scope in the policy. For example, if we find that both the method name and class name of v are in the policy, we will then use the method name as its granularity.

Initial Phase. The initial analysis is slightly modified to approximate the traces before they are stored, according to the given policy as an input:

$$\frac{\begin{array}{c} \Sigma_0 = \varnothing \\ \forall i \in 1..n \quad \cdot \quad \mathit{Pts}(v_i) = (o, \tau)_{1..m} \\ \Sigma_i = \Sigma_{i-1}, v_i \mapsto (\bigcup o_{1..m}, \mathit{Approx}(\bigcup \tau_{1..m}, \Gamma)) \end{array}}{\mathit{Initialise2}(v_{1..n}, \Gamma) = \Sigma_n} \quad \text{(initialisation-2)}$$

Incremental Phase. The incremental analysis is also slightly changed to approximate the traces when recomputing points-to sets:

$$\frac{\begin{array}{c} \Sigma_0 = (v \mapsto (\sigma, \tau))_{1..n} \qquad \forall i \in 1..n \quad \cdot \\ \mathit{if}\ \Delta \perp \tau_i\ \mathit{then} \begin{cases} \mathit{Pts}(v_i) = (o, \tau')_{1..m} \\ \Sigma_i = \Sigma_{i-1}[v_i \mapsto (\bigcup o_{1..m}, \mathit{Approx}(\bigcup \tau'_{1..m}, \Gamma))] \end{cases} \\ \mathit{else}\ \Sigma_i = \Sigma_{i-1} \end{array}}{\mathit{Increment2}(\Delta, \Sigma_0, \Gamma) = \Sigma_n} \quad \text{(increment-2)}$$

We have just extended the syntax of program units in the traces and changes so that we can now directly represent additions/deletions of not only statements but also, for example, methods or classes. We now need to extend the rules for checking overlap among traces/changes. An overlap relation is reflective and symmetric:

$$\frac{\mu \trianglelefteq \mu'}{\{\mu\} \perp \{\mu'\}} \quad \text{(overlap-contain)} \qquad\qquad \frac{\tau' \perp \tau}{\tau \perp \tau'} \quad \text{(overlap-symmetry)}$$

4.3 Adaptive Inference of Trace Policies

In order to specify a trace policy, we need to anticipate where changes will be made, which may not always be possible. We describe how to gradually refine trace policies from each incremental analysis, allowing policies to be inferred automatically based on the frequency of changes in different parts of a program. Therefore, the impact of changes on the existing points-to information related to the frequently changed code is minimised.

Initial Phase. The trace policy for the initial analysis is either empty or supplied by the programmer, which can be set up by reusing *Initialise2* from Section 4.2.

Incremental Phase. *Increment3* refines the trace policy by adding finer-grained program units into it. This incremental analysis reuses *Increment2* after adapting the policy to the changes, and returns the refined policy as output:

$$\frac{\mathit{Adapt}(\Delta) = \Gamma' \qquad \mathit{Increment2}(\Delta, \Sigma, \Gamma \cup \Gamma') = \Sigma'}{\mathit{Increment3}(\Delta, \Sigma, \Gamma) = \Sigma', \Gamma \cup \Gamma'} \quad \text{(increment-3)}$$

The following adaption rules compute finer-grained program units to be added into the policy, based on the type of changes made:

$$\mathit{Adapt}(\varnothing) = \varnothing \quad \text{(adapt-}\varnothing\text{)}$$

$$\mathit{Adapt}(\{\mu\} \cup \Delta) = \mathit{Adapt}(\{\mu\}) \cup \mathit{Adapt}(\Delta) \quad \text{(adapt-changes)}$$

$$\frac{x \trianglelefteq m}{\mathit{Adapt}(\{x\}) = \{y \mid y \trianglelefteq m\} \cup \mathit{Adapt}(\{m\})} \quad \text{(adapt-local)}$$

$$\frac{g \trianglelefteq c}{\mathit{Adapt}(\{g\}) = \{g' \mid g' \trianglelefteq c\} \cup \mathit{Adapt}(\{c\})} \quad \text{(adapt-global)}$$

$$\frac{m \trianglelefteq c}{\mathit{Adapt}(\{m\}) = \{m' \mid m' \trianglelefteq c\} \cup \mathit{Adapt}(\{c\})} \quad \text{(adapt-method)}$$

$$\frac{c \trianglelefteq p}{\mathit{Adapt}(\{c\}) = \{c' \mid c' \trianglelefteq p\} \cup \mathit{Adapt}(\{p\})} \quad \text{(adapt-class)}$$

$$\mathit{Adapt}(\{p\}) = \varnothing \quad \text{(adapt-package)}$$

If a local variable x is changed in (adapt-local), we add all local variables in its method into the policy, and then adapt the policy to the method changed. In general, we add all programs units that are directly defined in the enclosing scopes of x. The last rule (adapt-package) adapts nothing as package-level is the default granularity.

Table 1. Benchmark statistics. "Whole Program" includes the reachable parts of the Java libraries and "Application Code" does not. The last column gives the number of queries issued.

Benchmark	Whole Program			Application Code			#Queries
	#Classes	#Methods	#Statements	#Classes	#Methods	#Statements	
compress	5262	50667	372268	23	175	2989	443
jess	5402	51318	382460	161	798	13099	2064
db	5254	50667	372327	15	175	3035	239
javac	5422	51803	395661	183	1300	26238	5844
mpegaudio	5302	50944	384133	63	410	14869	7644
mtrt	5275	50799	374981	36	304	5714	911
jack	5307	50948	381756	68	443	12486	3296
avrora	2858	24412	197754	549	3194	42946	1413
batik	6827	60013	507723	1114	7356	125770	3574
fop	8441	74894	538179	978	7055	147677	10739
lusearch	2457	23113	190279	220	1979	32124	4053
sunflow	5508	52238	410396	170	1469	35267	1552
tradebeans	9272	83384	533529	909	6787	106480	4353
xalan	3053	28183	258840	618	6253	103348	2093

5 Experimental Evaluation

We evaluate our incremental analysis using a null dereferencing client, `NullDeref`. We compare our analysis with a state-of-the-art from-scratch analysis, REFINEPTS, from [27] using 14 Java programs, selected from the Dacapo and SPECjvm98 benchmark suites, given in Table 1. In the presence of small code changes targeted by this work, our incremental analysis is significantly faster (by at least one order of magnitude) than REFINEPTS when tracing application code at different granularity levels.

5.1 Implementation

We have implemented our incremental analysis and `NullDeref` in the Soot 2.5.0 [32] and Spark [17] framework, and conducted our experiments using the Sun JDK 1.6.0_26 libraries. REFINEPTS is publicly available in the same framework. Unmodeled native methods and reflection calls [19] are handled conservatively using Tamiflex [2]. The on-the-fly call graph of the program is constructed so that a *context-sensitive* call graph is always maintained for a program during the CFL-reachability computation.

5.2 Methodology

We have conducted our experiments on a machine consisting of Intel Xeon 2.27GHz processors (4 cores) with 24 GB memory, running Ubuntu Linux operating system (kernel version 2.6.38). Although the system has multi-cores, each analysis algorithm is single-threaded. Table 1 gives some statistics about the benchmarks used. Columns 2–4 show the number of classes, methods and statements in each program. Columns 5–7 are similar except the Java libraries are excluded. It can be seen that the application code is usually a small part of a Java program, making it suitable to be analysed with different trace policies depending on the nature of program changes made.

`NullDeref` detects null pointer violations, demanding high precision from points-to analysis. Since this client issues a large number of queries, it is suitable to show the affected and unaffected queries after a program change. The last column in Table 1 gives the number of queries issued by the client in a program.

In this paper, we consider changes to the program in terms of node additions and deletions to its program representation (i.e. PAG). To evaluate our incremental analysis, we have selected three different levels of code changes: class, method and statement. Our experiments are conducted by randomly deleting a class/method/statement in the program being analysed, as in [40]. We handle a class-level code change as a set of multiple method-level changes except that we must also handle the changes related to the fields in a changed class. When a field is deleted from a class, all edges related to the field are removed. When a field is added to the class (without statement additions), the PAG needs not to be updated. We have adopted this approach because it is reasonably simple to implement, which enables us to collect data on many potential changes across many programs. We find, in practice, that many code changes do not cause changes to the points-to information; however such code changes are excluded in our experiments.

Traditional points-to analyses like REFINEPTS, which are not designed to accommodate program changes, must recompute points-to information upon a code change. We compare the incremental analysis time, which includes the times on falsification and query processing, with the from-scratch analysis time, which includes the times on PAG construction and query processing. We repeated each experiment 20 times using randomly generated changes and reported the average of the 20 runs. Below we describe and analyse two sets of experiments depending on the granularities used for tracing the application code of a program. In both cases, the library code of a program is traced at package-level since it is unlikely to be modified.

Optimising for Analysis Time. We show the best speedups of our analysis over a from-scratch analysis by tracing application code at variable-level. Our analysis is significantly faster than REFINEPTS and remains so even under a stress test.

Trading Time for Space. We show that our analysis remains to be at least one order of magnitude faster even if we trace application code at method-level or class-level.

At this stage, we do not have results for the scenario when our analysis uses trace policies adaptively, because, unfortunately, we do not have enough change history data to obtain statistically significant results. However, its performance is expected to lie between the two scenarios studied here.

5.3 Optimising for Analysis Time

We consider code changes comprising a single deletion of a class or method or statement. The situation for adding a class or method or statement is similar.

We have compared the analysis times in Table 2 for REFINEPTS (Columns 2–4) and our incremental analysis (Columns 5–7). The execution times are all in seconds. For each program, there are three level of changes: deleting a class (denoted as "del c"), deleting a method (denoted as "del m") and deleting a statement (denoted as "del s"). For REFINEPTS, "PAG" is the time elapsed on constructing the PAG and "QT" denotes the time spent on recomputing all the issued queries. For our incremental analysis, "Falsification" is the time spent on the falsification process and "QT2" is a fraction of "QT" spent on recomputing the affected queries.

Table 2. Analysis times of `NullDeref` in seconds for deleting a class, method or statement

		REFINEPTS			Incremental Analysis		
		PAG	QT	Total	Falsification	QT2	Total
`compress`	del c	118.8	11.0	129.8	0.011	0.3	0.3
	del m	119.4	6.9	126.3	0.001	0.6	0.6
	del s	118.9	6.5	125.4	0.000	0.09	0.09
`jess`	del c	125.9	157.7	283.6	0.013	70.3	70.3
	del m	122.1	156.0	278.1	0.002	21.8	21.8
	del s	122.1	155.8	277.9	0.001	0.06	0.06
`db`	del c	118.8	12.2	131.0	0.007	0.4	0.4
	del m	119.1	12.2	131.3	0.001	0.5	0.5
	del s	120.2	12.4	132.6	0.000	0.01	0.01
`javac`	del c	125.4	223.4	348.8	0.032	45.5	45.5
	del m	124.8	224.0	348.8	0.006	21.5	21.5
	del s	125.1	226.5	351.6	0.002	4.93	4.93
`mpegaudio`	del c	124.7	27.0	151.7	0.040	8.4	8.4
	del m	121.5	31.0	152.4	0.003	6.5	6.5
	del s	120.8	29.2	150.0	0.001	0.09	0.09
`mtrt`	del c	120.0	28.6	148.6	0.014	2.9	2.9
	del m	118.4	27.2	145.5	0.001	2.4	2.4
	del s	119.3	25.1	144.4	0.001	0.32	0.32
`jack`	del c	118.2	31.0	149.2	0.026	2.5	2.5
	del m	118.4	31.6	150.0	0.001	2.1	2.1
	del s	115.3	27.4	142.7	0.000	0.49	0.49
`avrora`	del c	38.9	15.1	54.0	0.009	1.3	1.3
	del m	37.9	16.8	54.7	0.001	1.6	1.6
	del s	38.7	15.4	54.1	0.001	0.14	0.14
`batik`	del c	141.7	148.9	290.6	0.014	7.3	7.3
	del m	137.4	145.9	283.3	0.003	7.5	7.5
	del s	138.9	141.2	280.1	0.003	0.04	0.04
`fop`	del c	192.0	372.5	564.5	0.065	134.7	134.7
	del m	191.6	378.4	569.9	0.006	28.7	28.7
	del s	190.4	366.4	556.8	0.001	0.11	0.12
`lusearch`	del c	38.0	59.7	97.7	0.010	4.6	4.6
	del m	44.4	63.1	107.5	0.002	4.6	4.6
	del s	38.8	61.8	100.6	0.000	2.03	2.03
`sunflow`	del c	123.3	32.3	155.6	0.018	3.4	3.4
	del m	130.6	28.2	158.7	0.002	4.4	4.4
	del s	126.7	31.0	157.7	0.002	0.48	0.49
`tradebeans`	del c	210.1	256.0	466.1	0.068	23.5	23.6
	del m	214.1	255.3	469.4	0.023	36.5	36.6
	del s	211.4	246.1	457.5	0.001	5.91	5.91
`xalan`	del c	39.2	20.6	59.8	0.009	1.9	1.9
	del m	38.8	20.6	59.4	0.002	1.9	1.9
	del s	36.9	20.3	57.2	0.002	0.02	0.02

Our incremental analysis is much faster for all the benchmarks under three different levels of code changes. The average speedups range from 4X to a factor reaching several thousands. This is also true even if only the query time alone is used as a reference, since QT2 is a small fraction of QT. In addition, the falsification process is very fast and negligible relative to QT2. For a single deletion of a class/method/statement, the average speedup is 78.3X/60.1X/3195.4X.

As the library code of a program is traced at package-level, our analysis consumes only 11 MB more memory than REFINEPTS in the worst case.

Our incremental analysis is designed to handle small and frequent code changes. Nevertheless, we have stress-tested it with some major changes, involving a deletion of 100 randomly selected methods in a program, as shown in Table 3. While the percentage of valid queries is smaller than the case when only small changes are made, our analysis still outperforms REFINEPTS by 1.8X on average.

Our incremental analysis is developed to avoid recomputing unaffected queries after program changes. To understand the sources of performance gains, we have plotted the percentage of unaffected queries, including the "major" changes (with 100 methods deleted) in Fig. 8. On average, 99.1% of the queries are unaffected after a statement deletion. The percentage becomes 93.1% (91.9%) when a method (class) is deleted, respectively. In the case of the major changes, only 33.4% queries are unaffected. Note

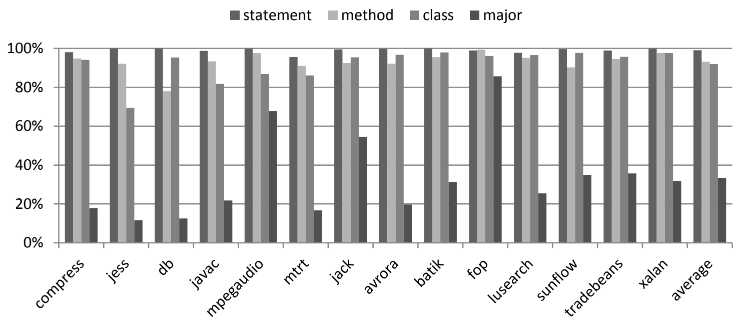


Fig. 8. Percentage of unaffected queries after program changes

Table 3. Stress testing of our analysis with "major" changes (deleting 100 methods)

Benchmark	Falsification	QT	#Unaffected Queries (%)	Speedup over REFINEPTS
`compress`	0.020	1.743	17.9	6.1
`jess`	0.043	141.896	11.6	1.1
`db`	0.024	3.611	12.5	3.3
`javac`	0.156	193.899	21.8	1.2
`mpegaudio`	0.095	19.911	67.7	1.3
`mtrt`	0.024	14.576	16.7	1.8
`jack`	0.030	15.364	54.5	1.8
`avrora`	0.040	12.326	19.7	1.4
`batik`	0.041	115.812	31.3	1.2
`fop`	0.531	327.090	85.7	1.2
`lusearch`	0.044	43.284	25.4	1.4
`sunflow`	0.022	18.925	35.0	1.7
`tradebeans`	0.118	212.521	35.7	1.2
`xalan`	0.052	16.826	31.9	1.4
`average`	0.052	81.270	33.4	1.8

that neither "method" nor "class" is consistently better than the other in terms of the percentage of affected queries. This may be due to the randomness of our experiments.

5.4 Trading Time for Space

For large programs, tracing the application code of a program at variable-level can be space-prohibitive. Our analysis allows it to be traced at coarser granularities to trade off analysis time for memory usage. As shown in Fig. 9 for a single method deletion, the average trace size (measured in terms of PAG nodes) per query increases as the trace policy becomes coarser. The percentage of unaffected queries for variable-level, method-level and class-level are 93.1%, 87.4% and 74.3%, respectively, on average. As a result, our analysis becomes slower but remains to be at least one order of magnitude faster than a from-scratch analysis. As discussed earlier, our analysis is 60.1X faster than REFINEPTS at variable-level. Its performance speedups has only dropped now to 24.2X and 18.0X at method-level and class-level, respectively.

At the two coarser trace policies, the largest analysis time increases are observed at `mtrt`, which takes 2.438 secs at variable-level but now 9.232 secs at method-level and 13.437 secs at class-level. The speedup of our analysis over REFINEPTS has dropped from 59.7X at variable-level to 15.8X at method-level and 10.X at class-level.

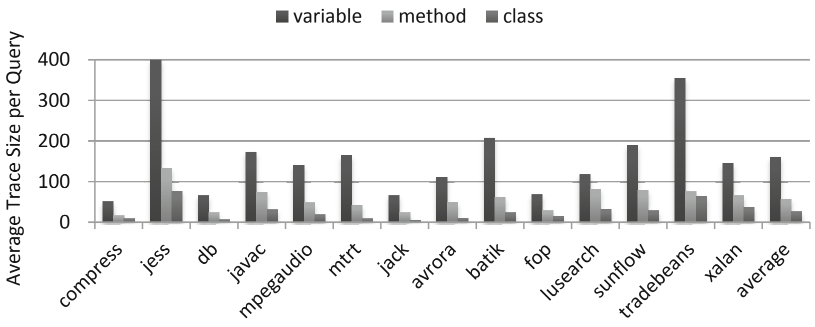


Fig. 9. Trace sizes at three different granularities for a single method deletion

6 Related Work

In recent years, there has been a large body of research devoted to points-to analysis. We restrict our discussion to three related areas: context-sensitive points-to analysis, incremental analysis and change impact analysis. As demonstrated via a null dereferencing client in our experiments, context sensitivity is needed for Java because many queries issued will not be positively answered otherwise.

Whole-program points-to analysis exhaustively computes points-to information for all its variables, which achieves context sensitivity by cloning [33] or summarisation [12,34,39,31,29]. Demand-driven points-to analysis [11] reduces the cost of analysis by only computing points-to information that is needed by its client analysis or optimisation. The state-of-the-art algorithms for Java [27,26,36] and C [41] are formulated in terms of CFL-reachability initially introduced in [23]. Given a CFL-reachability formulation, demand-driven analyses answer points-to queries as described in Section 2.

Pointer analyses based on CFL-reachability are precise, but they do not scale well to answer many queries for large programs. Sridharan et al. [28,27] proposed a refinement-based analysis to give an initial approximation and then gradually refine it until the client is satisfied. This strategy is useful for clients that can be satisfied early enough. Xu et al. [36] used an imprecise but cheap pre-analysis to find non-aliasing pairs to reduce redundancy in the subsequent points-to analysis. Zheng and Rugina [41] described a memory alias CFL-reachability formulation, answering alias queries without computing the complete points-to sets. Shang et al. [26] proposed a technique to summarise local points-to relations within a method. Such procedural CFL-reachability summaries may be reused later by the points-to analysis in the same or different calling contexts. In [25], they have also reported preliminary experience of using this technique to summarise the whole program and allow each procedural summary to be updated independently in response to edits from an IDE, achieving a limited form of incrementality. However, it does not allow points-to information to be reused. Therefore, points-to queries are always answered by recomputing from scratch. In contrast, our trace-based incremental algorithm presented in this paper allows previously computed points-to results to be reused, by recomputing only the queries that are falsified by code changes. Our technique is orthogonal to previous ones for improving the scalability of points-to analysis based on CFL-reachability. It may be possible to use our algorithm in conjunction with other techniques such as pre-analysis and procedural summarisation.

Many incremental algorithm have been developed for data-flow analysis problems. Some incremental analyses use the elimination method [3,5], some are based on the technique of restarting iterations [20] and some are hybrids of the two techniques [18]. A comparison of incremental iterative algorithm can be found in [4].

Incremental points-to analysis has been considered for C programs. Yur et al. [40] introduced an incremental approximation of their previous flow- and context-sensitive alias analysis [15] for C, by falsifying the aliases affected by the changed statements. Their algorithm handles addition/deletion of one single statement, achieving a 6-fold speedup for programs with 1 – 25K LOC. Their analysis is less precise than the reanalysis from scratch (with a solution agreement on 75% of tests on average). In contrast, our incremental algorithm produces exactly the same results as their full-analysis counterpart, and naturally handles multiple changes efficiently.

Kodumal and Aiken [13] considered for a limited form of incremental analysis via backtracking in their Banshee toolkit, which allows constraint systems to be rolled back to any previous state for a code change and reanalyses the program from that point forward. Their coarse-grained analysis is fast but imprecise due to its lack of support for context sensitivity. Saha and Ramakrishnan [24] extended [11], also for C, based on techniques for incremental evaluation of logic programs. When context sensitivity is considered, their analysis is slow, by consuming 50 – 73% of the from-scratch time.

Change impact analysis determines the effects of code changes to support the planning, implementation and validation of code changes in software evolution and maintenance. A taxonomy for impact analysis can be found in [16]. Recent approaches [1,6,8,21] rely on slicing, dependence analysis, dynamic tracing and history mining. In general, impact analysis requires fast and precise points-to information to be effective, which may benefit from our incremental points-to analysis.

7 Conclusion

Incremental points-to analysis is important in large projects where it is necessary to maintain a global analysis in the presence of small edits. We have described an incremental approach via tracing graph reachability, a mechanism that is efficient and simple to implement, for modern demand-driven context-sensitive points-to analyses. We have shown experimentally that tracing CFL-reachability is very effective in avoiding reanalysis of points-to information in Java. Our next step is to study the behaviour of real-world changes and to integrate our analysis into an interactive programming environment. We want to study changes made by real programmers, so that the sequence of changes we test will reflect more accurately modifications likely to be made in practice.

Acknowledgements. This research is supported by Australian Research Grants, DP0987236 and DP130101970.

References

1. Acharya, M., Robinson, B.: Practical change impact analysis based on static program slicing for industrial software systems. In: ICSE 2011 (2011)
2. Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., Mezini, M.: Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In: ICSE 2011 (2011)
3. Burke, M.G.: An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. ACM Trans. Program. Lang. Syst. 12(3) (1990)
4. Burke, M.G., Ryder, B.G.: A critical analysis of incremental iterative data flow analysis algorithms. IEEE Trans. Software Eng. 16(7) (1990)
5. Carroll, M.D., Ryder, B.G.: Incremental data flow analysis via dominator and attribute updates. In: POPL 1988 (1988)
6. Ceccarelli, M., Cerulo, L., Canfora, G., Di Penta, M.: An eclectic approach for change impact analysis. In: ICSE 2010 (2010)
7. Chaudhuri, S.: Subcubic algorithms for recursive state machines. In: POPL 2008 (2008)
8. Goeritzer, R.: Using impact analysis in industry. In: ICSE 2011 (2011)
9. Hardekopf, B., Lin, C.: Flow-sensitive pointer analysis for millions of lines of code. In: CGO 2011 (2011)
10. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. In: POPL 2009 (2009)
11. Heintze, N., Tardieu, O.: Demand-driven pointer analysis. In: PLDI 2011 (2001)
12. Kahlon, V.: Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In: PLDI 2008 (2008)
13. Kodumal, J., Aiken, A.: Banshee: A Scalable Constraint-Based Analysis Toolkit. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 218–234. Springer, Heidelberg (2005)
14. Kodumal, J., Aiken, A.: The set constraint/CFL reachability connection in practice. In: PLDI 2004 (2004)
15. Landi, W., Ryder, B.G.: A safe approximate algorithm for interprocedural aliasing. In: PLDI 1992 (1992)
16. Lehnert, S.: A taxonomy for software change impact analysis. In: IWPSE-EVOL 2011 (2011)
17. Lhoták, O., Hendren, L.: Scaling Java Points-to Analysis Using SPARK. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 153–169. Springer, Heidelberg (2003)
18. Marlowe, T.J., Ryder, B.G.: An efficient hybrid algorithm for incremental data flow analysis. In: POPL 1990 (1990)
19. Nguyen, P.H., Xue, J.: Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. In: ACSC 2005 (2005)
20. Pollock, L.L., Soffa, M.L.: An incremental version of iterative data flow analysis. IEEE Trans. Software Eng. 15(12) (1989)
21. Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.: Chianti: a tool for change impact analysis of Java programs. In: OOPSLA 2004 (2004)
22. Reps, T.: Program analysis via graph reachability. In: ILPS 1997 (1997)
23. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL 1995 (1995)
24. Saha, D., Ramakrishnan, C.: Incremental and demand-driven points-to analysis using logic programming. In: PPDP 2005 (2005)
25. Shang, L., Lu, Y., Xue, J.: Fast and precise points-to analysis with incremental CFL-reachability summarisation: preliminary experience. In: ASE 2012 (2012)
26. Shang, L., Xie, X., Xue, J.: On-demand dynamic summary-based points-to analysis. In: CGO 2012 (2012)
27. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for Java. In: PLDI 2006 (2006)

28. Sridharan, M., Gopan, D., Shan, L., Bodík, R.: Demand-driven points-to analysis for Java. In: OOPSLA 2005 (2005)
29. Sui, Y., Li, Y., Xue, J.: Query-directed adaptive heap cloning for optimizing compilers. In: CGO 2013 (2013)
30. Sui, Y., Ye, D., Xue, J.: Static memory leak detection using full-sparse value-flow analysis. In: ISSTA 2012 (2012)
31. Sui, Y., Ye, S., Xue, J., Yew, P.-C.: SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse SSA. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 155–171. Springer, Heidelberg (2011)
32. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot: a java bytecode optimization framework. In: CASCON 2010 (2010)
33. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PLDI 2004 (2004)
34. Wilson, R.P., Lam, M.S.: Efficient context-sensitive pointer analysis for C programs. In: PLDI 1995 (1995)
35. Xiao, X., Zhang, C.: Geometric encoding: forging the high performance context sensitive points-to analysis for Java. In: ISSTA 2011 (2011)
36. Xu, G., Rountev, A., Sridharan, M.: Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 98–122. Springer, Heidelberg (2009)
37. Yan, D., Xu, G., Rountev, A.: Demand-driven context-sensitive alias analysis for Java. In: ISSTA 2011 (2011)
38. Yannakakis, M.: Graph-theoretic methods in database theory. In: PODS 1990 (1990)
39. Yu, H., Xue, J., Huo, W., Feng, X., Zhang, Z.: Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In: CGO 2010 (2010)
40. Yur, J.-S., Ryder, B.G., Landi, W.: An incremental flow- and context-sensitive pointer aliasing analysis. In: ICSE 1999(1999)
41. Zheng, X., Rugina, R.: Demand-driven alias analysis for C. In: POPL 2008 (2008)

FESA: Fold- and Expand-Based Shape Analysis*

Holger Siegel and Axel Simon

Technische Universität München, Institut für Informatik II, Garching, Germany
`firstname.lastname@in.tum.de`

Abstract. A static shape analysis is presented that can prove the absence of `NULL`- and dangling pointer dereferences in standard algorithms on lists, trees and graphs. It is conceptually simpler than other analyses that use symbolically represented logic to describe the heap. Instead, it represents the heap as a single graph and a Boolean formula. The key idea is to summarize two nodes by calculating their common points-to information, which is done using the recently proposed *fold* and *expand* operations. The force of this approach is that both, *fold* and *expand*, retain relational information between points-to edges, thereby essentially inferring new shape invariants. We show that highly precise shape invariants can be inferred using off-the-shelf SAT-solvers. Cheaper approximations may augment standard points-to analysis used in compiler optimisations.

1 Introduction

Performing a shape analysis can help optimizing compilers to perform certain program specializations, such as inlining of virtual functions, partial evaluation (e.g. for fusing producers and consumers in functional languages [4]), or simply to eliminate `NULL` pointer tests.

In order to make shape analysis amenable for compilers, an analysis should be flexible in its precision to allow tuning its scalability. In this work we present a shape analysis whose performance depends on a numeric domain that can be implemented using off-the-shelf convex approximations [20], although we represent the state exactly using a SAT solver to illustrate its precision.

Our shape analysis can be seen as an instance of TVLA with one crucial simplification: The logic inherent in our analysis avoids a recursive transitive closure operator (RTC), as it is used in TVLA to encode reachability. For instance, a linked list with head H and a summarized tail T starting in a stack variable `x` is commonly encoded in TVLA as $x(H) = t_n(H,H) = t_n(H,T) = 1 \wedge t_n(T,T) = \frac{1}{2}$, meaning that `x` points to H, and from H we can reach H and T by following the n field zero or more times, from T we may reach T by following the n field zero or more times. The t_n predicate models transitive reachability and is defined using the RTC operator. Retaining this linked list invariant requires nontrivial integrity constraints on t_n, since the $\frac{1}{2}$ value of the $t_n(T,T)$ predicate proliferates during the evaluation of transfer functions. Rather than encoding reachability

* This work was supported by DFG Emmy Noether programme SI 1579/1.

a)
```
x = y = new_node();
while(rnd()) {
  *y = new_node();
  y = *y;
}
```

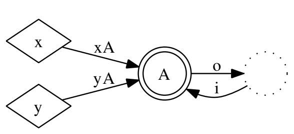
b)


Fig. 1. allocating a linked list

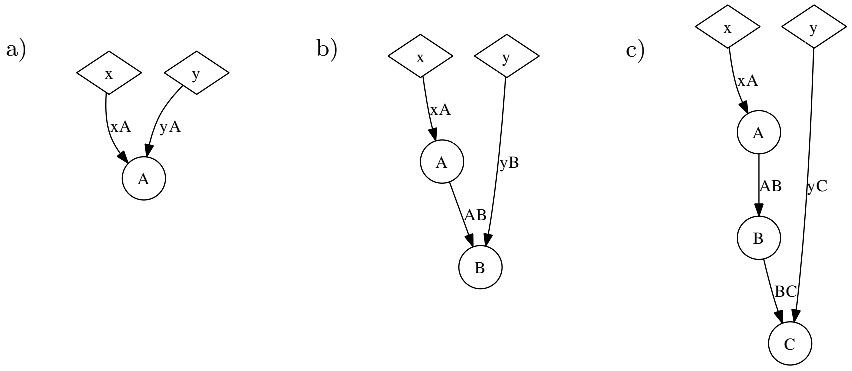


Fig. 2. calculating the state in the `while`-loop

directly, we only encode node-local information: (1) the existence of nodes and points-to edges, (2) the number of incoming nodes and (3) the number of outgoing edges. We show that these three properties suffice. In fact, we replace the set of graphs and their three-valued interpretation used in TVLA by a set of two-valued interpretations of a single graph. Thus, the omission of the RTC operator allows us to define a property-based shape analysis where a state is simply described by one graph and a single Boolean formula. The contribution of our work lies in describing how to perform summarization and materialization of nodes using sets of two-valued interpretations. Specifically, the insight in this paper is that invariant relations between neighboring nodes can be inferred during summarization using a relational *fold* operation that was recently defined for numeric domains [19]. A symmetric *expand* operation duplicates these relations when a summary cell is materialized back into two heap cells, one of which is a concrete non-summary cell. While the semantic information aggregated and duplicated by these two functions is nontrivial, their implementation is straightforward. This simplicity stands in contrast to previous work on using Boolean formulae as interpretation in the context of predicate abstraction [15] that requires sophisticated abstraction refinement techniques to obtain sufficient precision. Although our analysis does not explicitly maintain the linked list invariant, by virtue of the inferred relations between these predicates, it is precise enough to distinguish between lists, trees and graphs, which is not possible in TVLA without defining specific properties. We now demonstrate how inference of relational information replaces the validation of TVLA's linked list invariants.

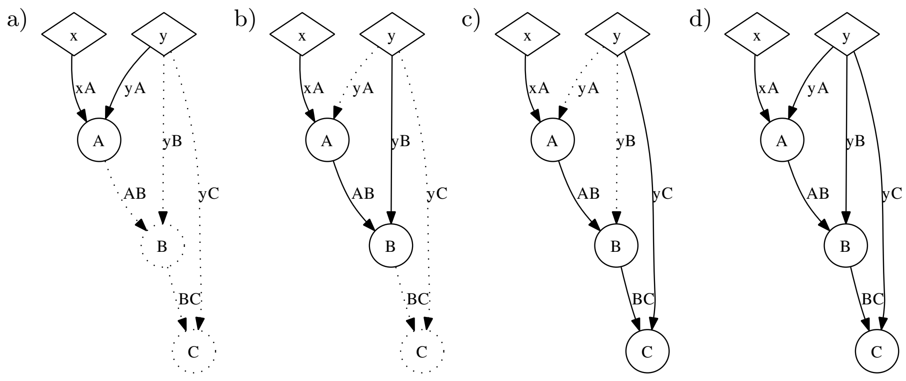


Fig. 3. Joining states

Consider the C program in Fig. 1a) that constructs a singly linked list. We will first show how the allocated cells can be summarized into a single summary node A, resulting in the heap in Fig. 1b). For the sake of exposition, we only show information pertaining to points-to edges.

Before the loop in Fig. 1a) is entered for the first time, `x` and `y` both point to a single node allocated by `new_node()`. The corresponding heap in Fig. 2a) depicts program variables as diamonds and heap-allocated cells as circles. Figure 2b) shows the state after one iteration. Here, `y` points to the newly allocated heap cell B. Figure 2c) shows the state after another iteration.

In order to represent the three states in a single abstract state, we represent heaps using a points-to map (a graph) and a numeric state. The points-to map takes each heap cell or program variable to a set of points-to edges and is shown as a directed graph. Each points-to edge is then further qualified by a flag that is mapped to zero or to one by the numeric domain. In particular, a flag f_{AB} maps to one iff the edge from node A to node B exists. For instance, the state before the loop consists of the points-to map shown in Fig. 2a) and the numeric state $\langle f_{xA}, f_{yA}\rangle \in \{\langle 1,1\rangle\}$. The state after one iteration consists of the points-to map shown in Fig. 2b) and the numeric state $\langle f_{xA}, f_{yB}, f_{AB}\rangle \in \{\langle 1,1,1\rangle\}$. After another iteration, the state consists of the points-to map in Fig. 2c) and the numeric state $\langle f_{xA}, f_{yC}, f_{AB}, f_{BC}\rangle \in \{\langle 1,1,1,1\rangle\}$.

These three states cannot be merged directly, since their sets of edges and nodes are different. We therefore make them *compatible* to each other by adding missing edges and nodes. Figure 3a) shows how adding nodes B and C and edges yB, yC, AB and BC to Fig. 2a) gives a compatible points-to graph with numeric state $\langle f_{xA}, f_{yA}, f_{yB}, f_{yC}, f_{AB}, f_{BC}\rangle \in \{\langle 1,1,0,0,0,0\rangle\}$. Figure 3b) and c) show how the states in Fig. 2b) and c) are made compatible with the corresponding numeric states $\{\langle 1,0,1,0,1,0\rangle\}$ and $\{\langle 1,0,0,1,1,1\rangle\}$, respectively. The merge of these compatible states is completed by joining the three numeric states into a single state $b := \{\langle 1,1,0,0,0,0\rangle\,\langle 1,0,1,0,1,0\rangle\,\langle 1,0,0,1,1,1\rangle\}$. The benefit of this representation is that the three heap configurations are all encoded by the graph in Fig. 3d) and the numeric state b.

a)
```
do {
  z = x;
  x = *x;
  free(z);
} while(x);
```

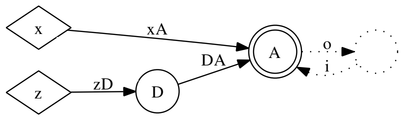
b)


Fig. 4. Deallocating a linked list

The latter graph and state b can now be transformed into a graph where all list nodes are summarized into one node as shown in Fig. 1b). To this end, our analysis uses the points-to graph to determine which edges to overlay and then applies a relational *fold* operation from [19] in order to summarize the corresponding dimensions. As a result, the dimensions $\langle f_{xA}, f_{yA}, f_{yB}, f_{yC}, f_{AB}, f_{BC}\rangle$ corresponding to Fig. 3d) are mapped to the dimensions $\langle f_{xA}, f_{yA}, f_i, f_o\rangle$ corresponding to Fig. 1b). This summary state represents the common points-to properties of all nodes, where the edges f_i and f_o represent the in- and outgoing edges connecting A with another instance of A. This instance is drawn with a dotted circle and is henceforth called the *ghost node* of A.

Interestingly, if we were to summarize a list with up to four nodes, we would obtain the same summarized state. Indeed, the summarized state is a fixpoint for the loop. This, in turn, implies that the summary represents a singly linked list of arbitrary size. The analysis has thus inferred that the loop constructs a singly linked list that commences in x and ends in the node pointed to by y. The numeric state of the fixpoint is quite subtle and describes the various rôles the summary node can take on: the list head A in Fig. 3a); the list head A in b), c); the unreachable nodes B, C in a) and C in b); the final node B in b) and C in c); and the inner node B in c). The key observation is that the relational *fold* is able to automatically synthesize these rôles and that the Boolean function maintains the distinction between them, thereby inferring very strong shape invariants. Note, though, that the invariant does not state which roles of A are possible. Thus, it might be that A only occurs in its role as a middle element of a list, thereby forming a cyclic list in the heap. Only the fact that x points to A and the relational information on A forces A to take on the role as list head and as list tail, thereby stating that it is an acyclic list. The inference of a precise relational summary using *fold* therefore replaces the verification of a linked-list invariant done by TVLA.

We now consider the loop in Fig. 4a) that deallocates a linked list. In our analysis, a summary node is materialized each time it is accessed. Thus, all modifications to heap nodes are performed on materialized nodes, which ensures that the abstract transformers are easy to derive since they closely follow the concrete transformers. Consider the assignment `x = *x` in the third line. Since the dereference `*x` would access a summary node, our analysis materializes a node D from the summary before executing the assignment. This materialization results in a state with the points-to set depicted in Fig. 4b).

In this particular case it is possible to free the node D by simply removing it from the graph. After z goes out of scope, the resulting graph is already compatible with that at the loop head. A fixpoint for the numeric domain is observed after one further iteration.

When evaluating the expression `*x`, the analysis checks that the numeric domain maps at least one edge in the points-to set of `x` to one. If this is not the case, a warning is emitted, stating that `x` can be `NULL`. In the example, our analysis is precise enough to verify the absence of `NULL`-pointer dereferences. Indeed, it can infer invariants that distinguish lists from trees from graphs. In contrast to other analyses [7,12], no extra effort is needed to make our analysis robust with respect to variations of these basic data structures (position of pointer fields, use of sentinel nodes instead of `NULL` values or the use of back pointers).

Overall, our shape analysis lies at a sweet-point between precision and simplicity by building on the following contributions presented in this work:

– a shape representation using points-to relations qualified by $\{0, 1\}$-vectors, thus substituting the common approach of representing heaps using a logic with simple transformers operating on a single graph and a numeric state
– a shape analysis for which transformers are easy to derive since they never operate on summary nodes and thus directly follow the concrete transformers
– the use of relational *fold* and *expand* operators [19] to summarize and materialize heap cells, allowing for a highly precise *inference* of new shapes; indeed, we can synthesize the strongest invariant when summarizing two heap cells
– the observation that only two extra properties suffice to distinguish common classes of data structures [21] as long as these are tracked with high precision; allowing us to verify programs operating on lists, trees and graphs

We present the principles of our shape analysis before Sect. 3 formalizes it for a C-like language. An analysis of trees is presented in Sect. 4 before Sect. 5 details our implementation. Section 6 discusses related work before Sect. 7 concludes.

2 Shape Analysis Using Numeric Domains

A shape analysis finitely summarizes a set of potentially unbounded, *concrete shape graphs* into an abstract representation. Before we define the abstract domain of points-to sets and numeric states, we consider concrete heap shapes.

Let $\mathcal{A}$ be a set of symbolic addresses. With each memory cell M we associate a unique symbolic address $A_M \in \mathcal{A}$. A *concrete shape graph* is a partial map $c : \mathcal{A} \to \wp(\mathcal{A})$ that maps each allocated memory cell to its *points-to set* $c(A_M)$: When $c(A_M) = \{A_N\}$ and $A_N \in \text{dom}(c)$ then memory cell M contains a pointer to memory cell N. When $c(A_M) = \emptyset$ then the memory cell M contains `NULL`. A points-to graph may also contain cells whose content does not represent a proper pointer value: When $|c(A_M)| > 1$ or $c(A_M) = \{A_N\}$ with $N \notin \text{dom}(c)$ then M contains an *invalid pointer*, namely, one that should not be dereferenced.

The remainder of the section presents the abstract states that represent sets of concrete shape graphs; it discusses summarization and materialization of nodes and addresses the correctness of these operations.

2.1 Abstract Shape Graphs

The building block of our shape analysis is a points-to set that is further qualified by a set of $\{0, 1\}$-vectors. In particular, we define a flow-sensitive points-to analysis [11] by associating each memory cell M with a points-to set of the form $\{\langle f_1, A_1\rangle, \dots \langle f_k, A_k\rangle\} \subseteq \mathcal{X} \times \mathcal{A}$ where $\mathcal{X}$ is a set of flags. A numeric domain associates each flag with a value drawn from $\{0, 1\}$. The idea is that $A_i \in c(A_M)$ if the flag f_i is mapped to one [20]. Thus, a memory cell can be dereferenced if exactly one flag in its points-to set is mapped to one by the numeric domain. In order to define this combined domain, we first present the numeric domain.

The Numeric Domain. The possible configurations of flags are given by a numeric domain $\mathcal{B}_n := \wp(\{0, 1\}^n)$ that holds sets of $\{0, 1\}$-vectors of dimension n. As a numeric domain, $\mathcal{B}_n$ can be modified by removing, adding and swapping dimensions or by restricting their values. The removal of dimension i from $b \in \mathcal{B}_n$ is defined by $drop_i(b) = \{\langle v_1, \dots, v_{i-1}, v_{i+1}, \dots, v_n\rangle \mid \langle v_1, \dots v_n\rangle \in b\}$. We write $drop_{i_1 \dots i_k} = drop_{i_1} \circ \dots \circ drop_{i_k}$ for ascending sequences $i_1 \dots i_k$ of indices. A dimension is added to $b \in \mathcal{B}_n$ using $add_i(b) = \{\langle v_1, \dots, v_{i-1}, v, v_i, \dots, v_n\rangle \mid \langle v_1, \dots, v_n\rangle \in b,\ v \in \{0, 1\}\}$. We write $add_{i_1 \dots i_k} = add_{i_k} \circ \dots \circ add_{i_1}$ for ascending sequences $i_1 \dots i_k$ of indices. A swap of two dimensions i and j in $b \in \mathcal{B}_n$ is denoted as $swap_{i,j}(b)$. It lifts naturally to two sequences of equal length.

For presentational purposes, we use flag names as synonyms of vector indices and assume sequences of dimensions to be in ascending order wherever they occur. Moreover, we write $add_f(b)$ to associate a new dimension with the flag name f. We omit the number of dimensions of the numeric domain, as it is equal to the number of stored flags. Using this convention, restricting a numeric state $b \in \mathcal{B}$ is denoted by $b[\![expr]\!] = \{\boldsymbol{b} \in b \mid expr\}$. For instance, $b[\![f = 0]\!]$ denotes all vectors $\boldsymbol{b} \in b$ in which the dimension associated with flag f maps to zero, and $add_{f_2}(b)[\![f_2 = 1 - f_1]\!]$ denotes an assignment of $1 - f_1$ to a fresh dimension f_2.

The Points-to Domain. A points-to state $p : \mathcal{A} \to \wp(\mathcal{X} \times \mathcal{A})$ maps (the address of) each memory cell to its points-to set. The set of points-to states is denoted by P. Writing a points-to set v to a memory cell at address A in state $p \in P$ is denoted by the update $p[A \mapsto v]$. We use a combined abstract state $p \rhd b \in P^{\mathcal{B}}$ where $P^{\mathcal{B}}$ is a new abstract points-to domain in which p is refined (qualified) by a numeric domain $b \in \mathcal{B}$. An operation on a state $p \rhd b$ adjusts p, thereby modifying the set of flags in the points-to sets, which, in turn, requires adjustments to $b \in \mathcal{B}$.

An abstract heap description must be able to represent concrete heaps c_1, c_2 with different numbers of cells, that is, $\mathrm{dom}(c_1) \neq \mathrm{dom}(c_2)$. For example, a heap cell may be deallocated in one state but not in another. In order to ensure that accessing a non-allocated region can be flagged as "dangling pointer" error, the numeric domain tracks an additional flag $f_{\exists M}$ for each heap-allocated cell M that is one iff M is allocated, that is, if $A_M \in \mathrm{dom}(c)$. For brevity, we generally omit the $f_{\exists X}$ flags in the presentation of the upcoming examples whenever they are constant one in the numeric state $b \in \mathcal{B}$, but we consider this example first:

Example 1. The points-to domain of the linked list in Fig. 5a) can be given by $p = [A_x \mapsto \{\langle f_{xA}, A_A\rangle\},\ A_A \mapsto \{\langle f_{AB}, A_B\rangle\},\ A_B \mapsto \{\langle f_{BC}, A_C\rangle\}]$ where A_x, A_A, A_B and A_C are the addresses of memory cells `x`, A, B and C, respectively. The numeric domain $\langle f_{xA}, f_{AB}, f_{BC}, f_{\exists A}, f_{\exists B}, f_{\exists C}\rangle \in \{\langle 1, 1, 1, 1, 1, 1\rangle\}$

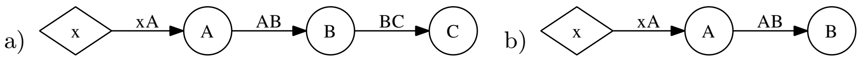


Fig. 5. Two linked lists

assures that dereferencing the contents of x, A and B is safe since in each points-to set exactly one flag has value one and all heap cells are allocated.

Lattice Operations on Abstract States. Whenever the control flow merges with states $p_1 \rhd b_1$ and $p_2 \rhd b_2$, the analysis continues with the joined state $p_1 \rhd b_1 \sqcup p_2 \rhd b_2$, which is $p_1 \rhd b_1 \cup b_2$ if $p_1 = p_2$. Checking for fixpoints when analyzing loops requires a subset test $p_1 \rhd b_1 \sqsubseteq p_2 \rhd b_2$ that reduces to $b_1 \subseteq b_2$ if $p_1 = p_2$. An analogous definition for $\sqcap$ yields the complete lattice $\langle P^{\mathcal{B}}, \sqsubseteq, \sqcup, \sqcap \rangle$.

In general, the points-to domains p_1 and p_2 may be different: Before they can be joined or compared, they must be made *compatible* by adding edges and nodes to p_1, p_2: When p_i contains a node M that is not present in p_j, the missing node is added to p_j using $p_j[A_M \mapsto \emptyset]$ and $f_{\exists M} = 0$ is added to b_j, indicating that M is not allocated. When p_i contains an edge from some node M to some node N that is not present in p_j, the edge is added to the points-to set of M in p_j and the corresponding numeric flag f_{MN} is introduced in b_j with value zero. Further adjustments are necessary for summary nodes and instrumentation predicates which are defined analogously. Consider again the lists in Fig. 5:

Example 2. Joining the heaps of Fig. 5a) and Fig. 5b) yields the points-to state of Fig. 5a) with $\langle f_{xA}, f_{AB}, f_{BC}, f_{\exists A}, f_{\exists B}, f_{\exists C} \rangle \in \{\langle 1, 1, u, 1, 1, u \rangle \mid u \in \{0, 1\}\}$.

2.2 Summarizing Nodes

This section illustrates how the previously defined operations are used to summarize two heap cells, which is a three-step process: First, both nodes are made *compatible*, then the edges between them are removed, and finally the numeric flags of one cell have to be merged with those of the other cell. Materialization applies the last two steps in reverse order. We consider each step in turn.

Making Nodes Compatible. Suppose that we summarize nodes A and B of the three-element linked list in Fig. 5. In order to merge the numeric information stored for the points-to information of A and B, both must have the same set of incoming and outgoing edges. To this end, they are made *compatible* by complementing the edge from x to A with a new edge from x to B and adding the flag $f_{xB} = 0$. Similarly, the edges from B to C and from A to B are complemented with edges from A to C and from C to B, yielding the state in Fig. 6a).

Disconnecting the Nodes. The points-to information between A and B can no longer be represented as edges between different nodes of the graph once they are summarized. We therefore introduce a *ghost node* to which these edges point, indicating that they point to a different instance of the summarized node. To

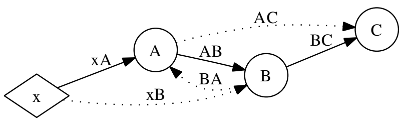
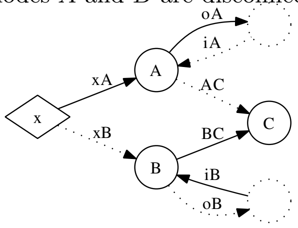
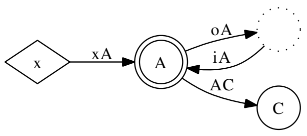
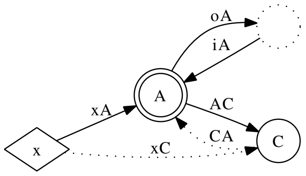
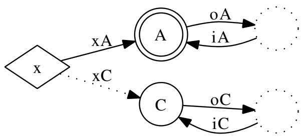
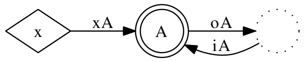
a) nodes A and B are compatible  $\langle f_{xA}, f_{xB}, f_{AB}, f_{BA}, f_{AC}, f_{BC}\rangle \in \{\langle 1,0,1,0,0,1\rangle\}$	b) nodes A and B are disconnected  $\langle f_{xA}, f_{xB}, f_{AC}, f_{iA}, f_{oA}, f_{BC}, f_{iB}, f_{oB}\rangle \in \{\langle 1,0,0,0,1,1,1,0\rangle\}$
c) node B is folded onto node A  $\langle f_{xA}, f_{AC}, f_{iA}, f_{oA}\rangle \in \{\langle x, 1-x, 1-x, x\rangle \mid x \in \{0,\ 1\}\}$	d) nodes A and C are compatible  $\langle f_{xA}, f_{xC}, f_{AC}, f_{CA}, f_{iA}, f_{oA}\rangle \in \{\langle x, 0, 1-x, 0, 1-x, x\rangle \mid x \in \{0,\ 1\}\}$
e) nodes A and C are disconnected  $\langle f_{xA}, f_{xC}, f_{iA}, f_{oA}, f_{iC}, f_{oC}\rangle \in \{\langle y, 0, x, 1-x, 1-y, y\rangle \mid x,y \in \{0,\ 1\}\}$	f) node C is folded onto node A  $\langle f_{xA}, f_{iA}, f_{oA}\rangle \in \{\langle y, x, 1-x\rangle \mid x,y \in \{0,\ 1\}\}$

Fig. 6. Folding and expanding a linked list

this end, we assign the value of f_{AB} to the flag f_{oA} that represents the out-edge from A to its ghost node and to flag f_{iB} that represents the in-edge from its ghost node to B. In the same way, we assign the value of f_{BA} to f_{iA} and to f_{oB}. Finally the edges between A and B are discarded, giving the state in Fig. 6b).

Summarizing Numeric Information. Note that the points-to sets of A and B now contain the same set of addresses, so that each flag associated with B can be merged with the respective flag of A. Merging and duplication of flags is based on *fold* and *expand* which is defined as follows [19]:

Definition 1. *Given the k dimensions $i_1 \ldots i_k$ and k dimensions $j_1 \ldots j_k$, define $\mathit{fold}_{i_1\ldots i_k, j_1\ldots j_k} : \mathcal{B}_{n+k} \to \mathcal{B}_n$ and $\mathit{expand} : \mathcal{B}_n \to \mathcal{B}_{n+k}$ as follows:*

$$\mathit{fold}_{i_1\ldots i_k, j_1\ldots j_k}(b) = \mathit{drop}_{j_1\ldots j_k}(b \cup \mathit{swap}_{i_1\ldots i_k, j_1\ldots j_k}(b))$$
$$\mathit{expand}_{i_1\ldots i_k, j_1\ldots j_k}(b) = \mathit{add}_{j_1\ldots j_k}(b) \cap \mathit{swap}_{i_1\ldots i_k, j_1\ldots j_k}(\mathit{add}_{j_1\ldots j_k}(b))$$

Intuitively, the *fold* operation merges the information over $j_1 \ldots j_k$ with that over $i_1 \ldots i_k$ and removes $j_1 \ldots j_k$. This process discards all information between

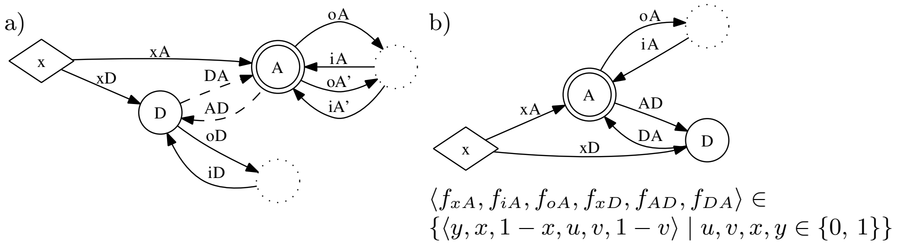

a)
x
xA
xD
A
D
DA
AD
oA
iA
oA'
iA'
oD
iD
b)
x
xA
xD
A
D
AD
DA
oA
iA
$\langle f_{xA}, f_{iA}, f_{oA}, f_{xD}, f_{AD}, f_{DA}\rangle \in$
$\{\langle y, x, 1-x, u, v, 1-v\rangle \mid u, v, x, y \in \{0, 1\}\}$

Fig. 7. Materializing D from A, given the state of Fig. 6f)

$i_1 \dots i_k$ and $j_1 \dots j_k$ but retains any relational information that holds for both $i_1 \dots i_k$ and $j_1 \dots j_k$. Symmetrically, *expand* retains relations within $i_1 \dots i_k$ when duplicating them to $j_1 \dots j_k$ but, unlike an assignment $j_1 := i_1$, induces no equality between j_1 and i_1. In the example, the flags $f_{xB}, f_{BC}, f_{iB}, f_{oB}$ are folded onto the flags $f_{xA}, f_{AC}, f_{iA}, f_{oA}$ by applying $\mathit{fold}_{f_{xA} f_{AC} f_{iA} f_{oA}, f_{xB} f_{BC} f_{iB} f_{oB}}$ to the numeric state. Note that this operation retains the relations that exist in each group of flags as shown in Fig. 6c). Here, the summarized state retained that the node that points to C is not the one pointed to by `x` since $f_{xA} \neq f_{AC}$.

Summarizing Summaries. Now we merge the summary node A with node C which has to be made compatible first as shown in Fig. 6d). Since A already has a ghost node, the flags f_{iA}, f_{oA} are already present in the numeric domain. Therefore, the information of f_{iA}, f_{oA} has to be merged with that of f_{AC}, f_{CA}. The intuition is that after summarizing A and C, there is no distinction between the edges from A to C and edges from A to nodes that have been previously merged with A. On the numeric state, flags f_{iC}, f_{oC} are introduced with $f_{iC} = f_{AC}$, $f_{oC} = f_{CA}$ and flags f_{CA}, f_{AC} are folded onto f_{iA}, f_{oA} by operation $\mathit{fold}_{f_{iA} f_{oA}, f_{CA} f_{AC}}$, yielding the state in Fig. 6e). Now all flags of node C are folded onto those of A by operation $\mathit{fold}_{f_{xA} f_{iA} f_{oA}, f_{xC} f_{iC} f_{oC}}$, giving the state in Fig. 6f).

2.3 Materializing Nodes from Summaries

In this section we discuss how to materialize a node from a summary node in two steps: first, the summary node is duplicated by applying *expand* to its numeric flags; second, the edges between the summary and the new node are synthesized using the edges to the ghost node. Consider expanding D from node A in Fig. 6f).

The first step applies $\mathit{expand}_{f_{xA} f_{iA} f_{oA}, f_{xD} f_{iD} f_{oD}}$ to the numeric state, resulting in $\langle f_{xA}, f_{iA}, f_{oA}, f_{xD}, f_{iD}, f_{oD}\rangle \in \{\langle y, x, 1-x, u, v, 1-v\rangle \mid u, v, x, y \in \{0, 1\}\}$. After that, it remains to reconstruct the edges between A and D: since node D is concrete and node A is a summary, we may assume that the flags f_{DA}, f_{AD} are equal to flags f_{oD}, f_{iD} and also equal to flags that have been summarized with f_{iA}, f_{oA} in the process of summarization. Thus, we first add the edges between A and D and their corresponding flags f_{AD}, f_{DA} by renaming flags f_{iD}, f_{oD} in Fig. 7a). In order to assert the equality with an *instance* of the edges f_{iA}, f_{oA}, we duplicate them to f'_{iA}, f'_{oA} using $\mathit{expand}_{f_{iA} f_{oA}, f'_{iA} f'_{oA}}$ and enforcing the equality by restricting the numeric state b to $b[\![f_{AD} = f'_{oA} \wedge f_{DA} = f'_{iA}]\!]$. The flags f'_{iA}, f'_{oA} are removed using *drop*, resulting in the state shown in Fig. 7b).

In contrast to the initial state in Fig. 6a), in the materialized state variable `x` may also contain `NULL` (since $f_{xA} = f_{xD} = 0$ is possible) or an invalid pointer (since $f_{xA} = f_{xD} = 1$ is possible). This precision loss is caused by summarizing the flags f_{xB} and f_{xC} onto f_{xA}, which makes them indistinguishable. Sect. 4.1 details how the validity of pointers can be maintained despite summarization. We conclude this section with an observation on the soundness of summarization.

2.4 Soundness of Summarization

We assume the existence of a function *summarize* that summarizes two nodes and a function *materialize* that expands summary nodes as described above and leaves non-summary nodes unchanged:

Definition 2. *Define summarize* : $\mathcal{A} \times \mathcal{A} \times P^{\mathcal{B}} \to P^{\mathcal{B}}$ *such that in state* $\mathit{summarize}(A_M, A_N, p \rhd q)$ *node* M *is a summary of* M *and* N *of state* $p \rhd q$*.*

Define materialize : $\mathcal{A} \times P^{\mathcal{B}} \to P^{\mathcal{B}}$ *such that for* $p' \rhd b' = \mathit{materialize}(A_N, p \rhd b)$*, some new address* $A_C \in \mathrm{dom}(p') \setminus \mathrm{dom}(p)$ *points to the materialized concrete node in state* $p' \rhd b'$ *if* N *is a summary in* $p \rhd b$*, and* $p' \rhd b' = p \rhd b$ *otherwise.*

Note that we assume that *materialize* recognizes a node M as summary node by observing the existence of the flags f_{iM}, f_{oM} in the numeric domain. As shown in [19], the pair $(\mathit{fold}_{i_1 \ldots i_k, j_1 \ldots j_k}, \mathit{expand}_{i_1 \ldots i_k, j_1 \ldots j_k})$ forms a Galois connection, from which follows that folding dimensions $j_1 \ldots j_k$ onto dimensions $i_1 \ldots i_k$ of a numeric state and then re-expanding dimensions $j_1 \ldots j_k$ yields an over-approximation of the initial numeric state. The following observations states that this extends to summarizing and then re-expanding a pair of nodes in the abstract shape graph.

Observation 1. *For all states* $p \rhd b \in P^{\mathcal{B}}$ *and addresses* $A_1, A_2 \in \mathrm{dom}(p)$ *there is* $p \rhd b \sqsubseteq \mathit{materialize}(A_1, \mathit{summarize}(A_1, A_2, p \rhd b))$ *up to the renaming of the materialized node.*

We will allow our analysis to summarize nodes in one sequence and afterwards materialize them in a different sequence. To ensure soundness, the result of summarizing and expanding a set of nodes must therefore be independent of the chosen sequence. It suffices that changing the order in which two nodes are materialized does not change the result, which is asserted as follows:

Observation 2. *For all states* $p \rhd b \in P^{\mathcal{B}}$ *and addresses* $A_1, A_2 \in \mathrm{dom}(p)$*,* $\mathit{materialize}(A_1, \mathit{materialize}(A_2, p \rhd b)) = \mathit{materialize}(A_2, \mathit{materialize}(A_1, p \rhd b))$ *holds up to the renaming of the materialized nodes.*

As a result, a shape analysis that relies on the operations *summarize* and *materialize* may summarize heap cells at any point and re-expand them on demand, namely when accessing the content of a summarized heap cell. The next section puts this into practice by defining an abstract interpreter for a C-like language which is later enriched to work on summaries.

$$\begin{aligned}
[\![\texttt{x=NULL}]\!]^\natural c &= c[A_x \mapsto \emptyset] \\
[\![\texttt{x=y}]\!]^\natural c &= c[A_x \mapsto c(A_y)] \\
[\![\texttt{x=\&y}]\!]^\natural c &= c[A_x \mapsto \{A_y\}] \\
[\![\texttt{x=malloc()}]\!]^\natural c &= c[A_x \mapsto \{A_N\}, A_N \mapsto \emptyset] \text{ where } A_N \text{ fresh} \\
[\![\texttt{x=*y}]\!]^\natural c &= c[A_x \mapsto c(A)] \text{ where } \{A\} = c(A_y) \\
[\![\texttt{*x=y}]\!]^\natural c &= c[A \mapsto c(A_y)] \text{ where } \{A\} = c(A_x) \\
[\![\texttt{free(x)}]\!]^\natural c &= c \setminus A \text{ where } \{A\} = c(A_x) \\
[\![\texttt{if (x==NULL) t else e}]\!]^\natural c &= \begin{cases} [\![\texttt{t}]\!]^\natural c \text{ if } c(A_x) = \emptyset \\ [\![\texttt{e}]\!]^\natural c \text{ otherwise} \end{cases}
\end{aligned}$$

Fig. 8. Concrete semantics of a C-like language

3 Shape Analysis of a C-Like Language

We present a language with explicit memory management using `malloc` and `free` which mimic their C counterparts. In contrast to C, we assume that program variables and heap cells are initialized to `NULL` and that they hold only one value; a simplification for the sake of clarity which is not present in our implementation.

The semantics of our language in Fig. 8 operates on concrete heap shapes as defined in Sect. 2. In a heap c, a cell M is allocated iff $A_M \in \text{dom}(c)$. Thus $[\![\texttt{x=malloc()}]\!]^\natural$ adds a mapping for a new address A_N to c and $[\![\texttt{free(x)}]\!]^\natural$ removes the mapping for A, denoted by $c \setminus A$. Analogously, stack variables `x`,`y`,... are stored at $A_x, A_y, \ldots$ where $A_x \in \text{dom}(c)$ iff `x` is in scope. Note that a state c containing an invalid pointer in cell M (with $|c(A_M)| > 1$ or $c(A_M) = \{A_N\}$ with $A_N \notin \text{dom}(c)$) is not an error, only dereferencing M is. This is addressed by the rules $[\![\texttt{x=*y}]\!]^\natural$, $[\![\texttt{*x=y}]\!]^\natural$, and $[\![\texttt{free(x)}]\!]^\natural$ that are undefined if the dereferenced cell does not hold exactly one pointer or if the pointed-to cell at A is not allocated, that is, if $A \notin \text{dom}(c)$. The goal of the analysis is to prove the absence of undefined behavior. Note that invalid pointers cannot arise in the concrete semantics, but may arise due to approximations in the abstract semantics, which is presented next.

3.1 Abstract Semantics

An abstract interpretation is sound if the abstract semantics $[\![\texttt{s}]\!]^\sharp$ of each statement `s` approximates its concrete semantics $[\![\texttt{s}]\!]^\natural$, that is, if $\overline{[\![\texttt{s}]\!]^\natural} \circ \gamma \subseteq \gamma \circ [\![\texttt{s}]\!]^\sharp$ with $\overline{[\![\texttt{s}]\!]^\natural}(C) := \{[\![\texttt{s}]\!]^\natural(c) \mid c \in C\}$ [9]. In order to derive this semantics, we first define a concretization function γ_0 that relates each abstract cell M to one concrete memory cell if A_M exists (that is, if its $f_{\exists M}$ flag is one):

Definition 3. *Function $\gamma_0 : P^{\mathcal{B}} \to \wp(\mathcal{A} \to \wp(\mathcal{A}))$ is given by*

$$\gamma_0(p \rhd b) = \big\{[A_M \mapsto \{A_i \mid \langle f_i, A_i\rangle \in p(A_M) \wedge \boldsymbol{b}(f_i) = 1\}]_{A_M \in \text{dom}(p) \wedge \boldsymbol{b}(f_{\exists M}) = 1} \mid \boldsymbol{b} \in b\big\}$$

where $\boldsymbol{b}(f_i)$ denotes dimension f_i of a vector $\boldsymbol{b}$.

$$\begin{array}{ll}
[\![\texttt{x=NULL}]\!]^\sharp(p \rhd b) & = p[A_x \mapsto \emptyset] \rhd \mathit{drop}_{\{f \mid \langle f,\_\rangle \in p(x)\}}(b) \\
[\![\texttt{x=y}]\!]^\sharp(p \rhd b) & = p[A_x \mapsto \{\langle f_{x1}, A_1\rangle, \ldots, \langle f_{xn}, A_n\rangle\}] \rhd \\
& \quad \mathit{add}_{f_{x1} \ldots f_{xn}}(\mathit{drop}_{\{f \mid \langle f,\_\rangle \in p(x)\}}(b))[\![f_{x1} = f_{y1}, \ldots, f_{xn} = f_{yn}]\!] \\
& \quad \text{where } \texttt{x} \neq \texttt{y} \text{ and } p(y) = \{\langle f_{y1}, A_1\rangle \ldots, \langle f_{yn}, A_n\rangle\} \\
[\![\texttt{x=\&y}]\!]^\sharp(p \rhd b) & = p[A_x \mapsto \{\langle f_{xy}, A_y\rangle\}] \rhd \mathit{add}_{f_{xy}}(\mathit{drop}_{\{f \mid \langle f,\_\rangle \in p(x)\}}(b))[\![f_{xy} = 1]\!] \\
[\![\texttt{x=malloc()}]\!]^\sharp(p \rhd b) & = [\![\texttt{x=\&N}]\!]^\sharp(p[A_N \mapsto \emptyset] \rhd \mathit{add}_{f_{\exists N}}(b)[\![f_{\exists N} = 1]\!]) \text{ where } A_N \text{ fresh} \\
[\![\texttt{*x=y}]\!]^\sharp(p \rhd b) & = \bigsqcup_{\langle A_z, p' \rhd b'\rangle \in \mathit{deref}(A_x,(p \rhd b))} [\![\texttt{z=y}]\!]^\sharp(p' \rhd b') \\
[\![\texttt{x=*y}]\!]^\sharp(p \rhd b) & = \bigsqcup_{\langle A_z, p' \rhd b'\rangle \in \mathit{deref}(A_y,(p \rhd b))} [\![\texttt{x=z}]\!]^\sharp(p' \rhd b') \\
[\![\texttt{free(x)}]\!]^\sharp(p \rhd b) & = \bigsqcup_{\langle A_z, p' \rhd b'\rangle \in \mathit{deref}(A_x,(p \rhd b))} p' \rhd b'[f_{\exists z} \mapsto 0] \\
[\![\texttt{if (x==NULL) t else e}]\!]^\sharp(p \rhd b) & = [\![\texttt{t}]\!]^\sharp(p \rhd b[\![f_{x1} = 0 \wedge \ldots \wedge f_{xn} = 0]\!]) \\
& \quad \sqcup [\![\texttt{e}]\!]^\sharp(p \rhd b[\![f_{x1} = 1 \vee \ldots \vee f_{xn} = 1]\!]) \\
& \quad \text{where } p(A_x) = \{\langle f_{x1}, A_1\rangle, \ldots \langle f_{xk}, A_k\rangle\}
\end{array}$$

Fig. 9. Abstract semantics

The abstract semantics $[\![\texttt{s}]\!]^\sharp$ that lifts the concrete semantics $[\![\texttt{s}]\!]^\natural$ of a statement `s` to the abstract domain $P^{\mathcal{B}}$ is shown in Figure 9. Here, $[\![\texttt{x=NULL}]\!]^\sharp$ removes all previous flags f of the points-to set of `x` from b using *drop* and empties $p(A_x)$. These stale flags are also removed in $[\![\texttt{x=y}]\!]^\sharp$ before the new flags $f_{x1}, \ldots f_{xn}$ are set to the values of $f_{y1}, \ldots f_{yn}$ of `y`. Analogously for $[\![\texttt{x=\&y}]\!]^\sharp$. Allocating heap cells with $[\![\texttt{x=malloc()}]\!]^\sharp$ chooses a fresh address A_N. The fact that A_N is a valid cell is stated by adding the flag $f_{\exists N}$ with value one to b. The result `x` is set to point to the new address A_N using $[\![\texttt{x=\&N}]\!]^\sharp$.

The next three statements dereference pointers. Each statement dereferences the pointer contents using function $\mathit{deref} : \mathcal{A} \times P^{\mathcal{B}} \to \wp(\mathcal{A} \times P^{\mathcal{B}})$. This function partitions the passed-in state depending on the address that is pointed to:

$$\mathit{deref}(A, p \rhd b) = \bigcup_{(\_,\, B) \in p(A)} \phi(A, B, p \rhd b) \setminus \{\mathit{warn}\}$$

Here, a call $\phi(A, B, p \rhd b)$ to the auxiliary function $\phi : \mathcal{A} \times \mathcal{A} \times P^{\mathcal{B}} \to \wp(\mathcal{A} \times P^{\mathcal{B}}) \cup \{\mathit{warn}\}$ returns a state in which the cell at A points to the address B:

$$\begin{aligned}
\phi(A, B, p \rhd b) = {} & \{(B,\ p \rhd b[\![f_1 + \ldots + f_n = 1 \wedge f_{AB} = 1 \wedge f_{\exists B} = 1]\!])\} && (1)\\
& \cup \{\mathit{warn} \mid b[\![f_1 + \ldots + f_n \neq 1]\!] \neq \emptyset\} && (2)\\
& \cup \{\mathit{warn} \mid b[\![f_1 + \ldots + f_n = 1 \wedge f_{AB} = 1 \wedge f_{\exists B} = 0]\!] \neq \emptyset\} && (3)
\end{aligned}$$

where $p(A) = \{\langle f_1, A_1\rangle, \ldots, \langle f_n, A_n\rangle\}$. Value *warn* indicates a possible run-time error that has to be reported by the abstract interpreter: it is issued whenever a points-to set may point to none or more than one memory cell (Eqn. 2), or when the heap cell pointed to is not allocated due to, for instance, `free()` (Eqn. 3).

For each tuple $\langle B, p' \rhd b'\rangle$ returned by function *deref*, $[\![\texttt{*x=y}]\!]^\sharp$ writes `y` to address B, $[\![\texttt{x=*y}]\!]^\sharp$ reads from address B, and $[\![\texttt{free(x)}]\!]^\sharp$ deallocates address B in state $p' \rhd b'$, and the results are joined. Finally, the rule for the conditional partitions the state $p \rhd b$ such that `x` is `NULL` and non-`NULL`.

We note that our abstract semantics fulfills the soundness condition from above:

Observation 3. *For any statement* s*, there is* $\overline{[\![s]\!]^\natural} \circ \gamma_0 = \gamma_0 \circ [\![s]\!]^\sharp$*. Given* $\alpha_0 : \wp(\mathcal{A} \to \wp(\mathcal{A})) \to P^{\mathcal{B}}$ *with* $\alpha_0(c) = \sqcap\{a \mid c \subseteq \gamma(a)\}$ *there is* $\alpha_0 \circ \overline{[\![s]\!]^\natural} = [\![s]\!]^\sharp \circ \alpha_0$*.*

Moreover, the observation states that the abstract semantics exactly mirrors the concrete semantics. However, abstract states can grow arbitrarily large, and therefore fixpoint computations must apply a widening to ensure termination, as detailed in the next section.

3.2 Widening and Materialization on Access

An infinite growth of the abstract state space can be avoided by inserting a widening operator into every loop of the program [9] which ensures that any increasing sequence of states eventually stabilizes. We implement widening by summarization. By materializing memory cells on access, we are able to retain our abstract semantics as is. We detail both operations in turn.

Widening by Summarization. A challenge in analyzing loops is to ensure that the set of heap cells remains finite. We address this by summarizing nodes that are *abstract reachable* from the same program variables. This *abstract reachability* only considers the points-to graph and not the information in the numeric domain. For instance, the nodes A and B in Fig. 3a) are abstract reachable from `x` and `y`, even though B is not reachable according to the numeric state.

Our widening consists of two steps and is applied to a single state. In the first step, a partitioning of the heap-allocated nodes is calculated: nodes that are *abstract reachable* from the same set of stack variables are put into one partition. This ensures finiteness of fixpoint computation since the set of partitions is determined by the number of stack variables in scope. In the second step, each partition of heap nodes is collapsed into one summary node by summarizing all its members with the oldest member. This ensures that the symbolic addresses remain stable under the fixpoint calculations which enables the detection of a fixpoint using $\sqsubseteq$.

Materialization on Access. Materialization is performed as part of dereferencing pointers. To this end, each use of *deref* in Fig. 9 is replaced by *deref′* (defined below). For every summary node, *deref′* returns a state in which the node is materialized and a state in which the node is turned into a concrete node by simply removing the ghost node. The latter is necessary in case a summary node represents only one concrete node. The two cases are reflected by two calls to ϕ:

$$\mathit{deref}'(A, p \rhd b) \;=\; \bigcup_{(f,\,B) \in p(A)} \phi(A, B', p_1 \rhd b_1) \cup \phi(A, B, p_2 \rhd b_2) \setminus \{\mathit{warn}\}$$

where $p_1 \rhd b_1 = \mathit{materialize}(B, p \rhd b)$ is the result of materializing node B to $\{B'\} = \mathrm{dom}(p_1) \setminus \mathrm{dom}(p)$ and $p_2 \rhd b_2 = p \rhd \mathit{drop}_{f_{iB}, f_{oB}}(b)$ is the state in which B is no longer a summary node. Since the state $p_1 \rhd b_1$ returned by function *materialize* is unchanged if B is not a summary, $\mathit{deref}'(A, p \rhd b)$ is identical to $\mathit{deref}(A, p \rhd b)$ when the points-to set of A only refers to concrete nodes.

We conclude this section by defining a concretization γ that takes an abstract state in $P^{\mathcal{B}}$ to concrete states in $\wp(\mathcal{A} \to \wp(\mathcal{A}))$.

Definition 4. *Concretization* $\gamma : P^{\mathcal{B}} \to \wp(\mathcal{A} \to \wp(\mathcal{A}))$ *is given by* $\gamma = \gamma_0 \circ \tau$ *where* $\tau : P^{\mathcal{B}} \to P^{\mathcal{B}}$ *is given by* $\tau(s) = \mathit{fix}_s(\lambda t \,.\, t \sqcup \bigsqcup_{A \in \mathcal{A}} \{u \mid A \in \mathcal{A},\ (\_, u) \in \mathit{materialize}(A, t)\}))$ *where* $\mathit{fix}_s(f)$ *is the least fixpoint of* f *greater than* s.

Here, function τ calculates the reflexive transitive closure of applying arbitrary materializations. Note that for abstract states with at least one summary node, τ returns a points-to domain with an infinite number of nodes paired with a numeric domain with an infinite number of dimensions. Materialization on access retains the following property which implies that all precision loss is incurred by widening:

Observation 4. *For any statement* s, *there is* $\overline{[\![s]\!]^{\natural}} \circ \gamma = \gamma \circ [\![s]\!]^{\sharp}$.

We now enhance this basic analysis by two predicates that are sufficient to distinguish between list, trees and graphs.

4 Two Simple Instrumentations

For the sake of presentation, we generalize the numeric domain from {0, 1}-vectors to $\mathbb{N}^n$. We define two simple numeric instrumentation variables and describe how they can be approximated by {0, 1}-flags.

4.1 Counting Outgoing Edges

The shape graph in Fig. 7b) shows the result of materializing a heap cell D from a summary A. The obtained numeric state is $\langle f_{xA}, f_{iA}, f_{oA}, f_{xD}, f_{AD}, f_{DA} \rangle \in \{\langle y, x, 1-x, u, v, 1-v \rangle \mid u, v, x, y \in \{0, 1\}\}$, which allows flags f_{xA} and f_{xD} to be zero or one at the same time, indicating `NULL` or an invalid pointer value in variable `x`. The reason for this precision loss is that when summarizing two nodes M and N the flags f_{xM} and f_{xN} become indistinguishable due to folding.

We rectify this precision loss by tracking the number of outgoing edges of each node N in a numeric counter c_N^{out}. As this counter reflects the number of outgoing edges in the concrete graph, summarizing two nodes does not change this counter. However, when summarizing node O onto P, the flags f_{NO} and f_{NP} are merged into f'_{NP} which may be smaller than $f_{NO} + f_{NP}$. Thus, the sum $s_N := f_{N1} + \cdots + f_{Nk}$ of N's points-to flags may be smaller than c_N^{out}. We apply the information in c_N^{out} by adding the assertion $s_N = c_N^{out} = 1$ to deref' and by enforcing $s_M \leq c_M^{out}$ whenever $p(A_M)$ receives new edges due to materialization.

Since the concrete semantics in Fig. 8 precludes points-to sets with more than one element, the invariant $c_N^{out} \leq 1$ can be enforced on the abstract state without losing soundness. This reduction with respect to the concrete semantics has been dubbed “hygiene condition” [18]. As a consequence, we implement only the lowest bit of the c_N^{out} counter as {0, 1}-flag.

a)
```
while(a != NULL) {
  tree * b = a->l;
  if(b==NULL) {
    b = a->r;
    free(a);
    a = b;
  } else {
    a->l = b->r;
    b->r = a->r;
    a->r = b; }}
```

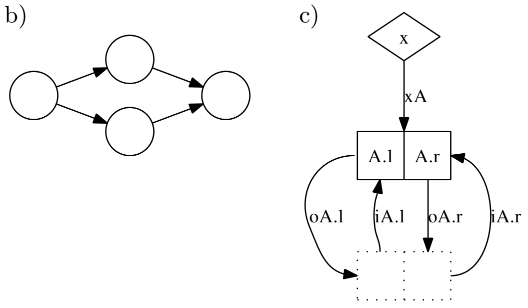


Fig. 10. Expanding a binary tree

4.2 Counting Incoming Edges

Fig. 10a) shows a C program that iteratively deallocates a binary tree pointed to by variable `a` by tree rotation. It requires that `a` holds a tree, that is, neither cycles nor diamond-shaped subgraphs as in Fig. 10b) are reachable from `a`, i.e., no heap cell can be accessed via more than one path. We guarantee the latter by simple *reference counting*: every memory cell M is equipped with a reference counter c_M^{in} that counts the references coming from *heap-allocated* cells. Whenever an entry $\langle f_{NM}, A_M \rangle$ is added to the points-to set of a node N, counter c_M^{in} is incremented by the value of f_{NM}. Analogously, the counter is decremented by the respective flag when an entry is removed from the points-to set. We then assert $c_M^{in} \geq 0$.

Our analysis is easily extended to compound memory cells like, for example, the nodes of a binary tree: summarization is done by making the components compatible one-by-one and then folding the compound cells onto each other. Fig. 10c) shows the abstract representation of an arbitrary binary tree. Summarization can infer the invariant that $f_{xA} = 1 - c_A^{in} \in \{0, 1\}$, that is, every heap cell has exactly one incoming edge. Indeed, this instrumentation is nearly sufficient to prove the absence of double-free-errors for the algorithm in Fig. 10a).

Since the numeric domain of our implementation is restricted to $\{0, 1\}$-values, we implement the counters c_N^{in} by a saturating binary 2-bit counter. We now show how the analysis can be efficiently implemented using a SAT solver.

5 Implementation and Experimental Results

The numeric state $b \in \mathcal{B}_n$ can be represented by a Boolean formula over n variables. We therefore implemented the functor domain $p \triangleright b \in P^{\mathcal{B}_n}$ as a tuple consisting of a map for the points-to sets $p \in P$ and a single formula φ for $b \in \mathcal{B}_n$ that is either in conjunctive or in disjunctive normal form.

Since *expand* calculates an intersection of states, it can be straightforwardly implemented on a CNF formula by duplicating clauses. Analogously, *fold* has a straightforward implementation on a DNF formula. Join (resp. meet, in particular $b[\![\cdot]\!]$) is calculated by merging the DNF (resp. CNF) representations. The semantics of most statements requires the removal of dimensions from the vector set using $drop_{f_i}$, which corresponds to removing the quantifier in the formula

Table 1. Evaluation of our implementation

	instr. pred.	projections	avg. size vars/clauses	time	mem	loop iterations	warnings
three-element list	–	0	–	2 ms	10 MB	-	1
	+	0	–	3 ms	10 MB	-	0
singly linked list	–	33	20 / 36	30 ms	12 MB	4 / 3	6
	+	42	36 / 55	132 ms	14 MB	5 / 3	0
doubly linked list	–	12	23 / 37	13 ms	10 MB	2 / 2	7
	+	15	47 / 76	49 ms	13 MB	3 / 2	0
summarize tree	–	0	–	2 ms	12 MB	-	0
	+	0	–	2 ms	13 MB	-	0
access tree	–	38	15 / 15	48 ms	10 MB	3	6
	+	12	131 / 298	32 ms	17 MB	2	0
deallocate tree	–	95	32 / 45	380 ms	25 MB	5	28
	+	80	52 / 95	1.511 ms	26 MB	5	1
deallocate graph	–	96	34 / 51	375 ms	23 MB	5	44
	+	139	57 / 104	1.425 ms	26 MB	5	1

Table 2. Comparison with TVLA

TVLA	predicates	time	mem	iterations	warnings
singly linked list	16	176 ms	14 MB	40	0
deallocate tree	19	3,391 ms	20 MB	66	26

$\exists f_i \, . \, \varphi$. Rather than removing each dimension using *drop*, f_i is simply renamed to a fresh variable. Unused variables are eventually removed during the conversion between a CNF and a DNF formula. For the latter, we use the projection method of Brauer et al. [5], which is based on SAT-solving and calculates a minimal DNF representation from a CNF formula and vice versa.

Our implementation is written in Java using MINISAT v2.2. Table 1 shows the results for: summarizing a three-elemented list and accessing its first element as described in Sect 2.2; allocating and then deallocating a singly linked list; dto. for a doubly linked list of arbitrary length; summarizing a seven-element binary tree; accessing the leftmost innermost element of a binary tree; running the algorithm of Fig. 10a) on the summarized tree; running it on the summary of a diamond-shaped graph as in Fig. 10b). Each task is shown with instrumentation flags enabled (+) and disabled (–).

The columns show the number of calls to the projection function and the average size of the formulae (number of variables / number of clauses) at each projection. Running times and memory consumption on an Intel Core i7 with 2,66 GHz on Mac OS X 10.6 follow. The next two columns show how many iterations are required to find a fixpoint for the loop(s) in the examples. A "-" indicates that the example has no loop. For all examples except the last two, we could verify the absence of `NULL`- and dangling pointer dereferences. The warnings that occur in the deallocation examples arise when accessing a freed heap node N through a stack variable. In the *tree* deallocation example, $c^{in}_N = 1$ but no summary node M pointing to N can be materialized with $f_{MN} = 1$. That is, this state is contradictory in itself and could be removed by a reduction step. Thus, while our analysis is expressive enough, an explicit reduction is required to remove this spurious warning. Since this reduction is orthogonal to the shape analysis itself, it has been omitted. Note that the warning in the last row of Table 1 is a true error.

We compare our prototype implementation with the latest TVLA 3 analyzer. Table 2 shows the running time of two of our examples when reformulated with TVLA predicates. The verification of the list example is slightly slower. A TVLA tree example that mimics our deletion algorithm for trees is shown in the second row. Although TVLA provides predicates that can express the invariant, it is unable to prove that a node is only freed once and, hence, emits warnings. This is curious, since TVLA provides several tree invariants that are silently enforced by integrity constraints.

6 Related Approaches to Shape Analysis

Using Boolean functions for analyzing points-to sets is a re-occurring theme in the literature, although they are often represented as binary decision diagrams rather than CNF formulae [3]. Our work shows how points-to analysis can be enriched with summaries of heap structures, thereby giving a new answer to the question of how to merge the regions created at call sites of `malloc` [14]. Moreover, our analysis could replace ad-hoc forms of shape analysis such as summarizing all heap cells but the last one allocated [1]. The latter is used to allow strong updates on heap cells that are allocated in a loop. By materializing on access and summarizing through widening, our analysis refines this ad-hoc strategy by a dynamic, semantics-driven strategy.

One peculiarity of our analysis is the ability to distinguish lists, trees, and graphs using only relational information associated with each node. This design simplifies the abstract transfer functions in that they only have to update information of the nodes that are actually accessed, we call this a node-local analysis. One motivation for using separation logic for shape analysis is exactly this ability, namely that an update in separation logic retains a so-called frame that describes the part of the heap that is not being accessed. Moreover, the inductively defined predicates of separation logic [17] are also local in that they relate each node to a fixed number of neighboring nodes. Using these predicates, a linked list is specified as $\mathit{list}(x) \equiv \mathbf{emp} \vee (\exists y \,.\, x \mapsto y \bullet \mathit{list}(y))$ where the *separating conjunction* $h_1 \bullet h_2$ expresses that heaps h_1 and h_2 do not overlap. Interestingly, a summary node N and its ghost node live at different addresses and, thus, they can be seen as being separated by the separating conjunction.

While automatic analysis using separation logic relies on a symbolic representation of heap shapes, our approach is based on a Boolean domain whose join operation can infer new invariants. Inferring new invariants (predicates) in the context of separation logic is in general not yet possible [8]. However, templates, namely higher-order predicates, have been automatically instantiated in order to infer hierarchical data structures [2]. Since our current approach already infers any node-local invariant, future work should address the inference of hierarchical shapes, for instance, to verify operations on a list of independent circular lists.

A different approach to shape analysis is the TVLA framework [18] where the shape of the heap is described by a set of core predicates such as $n(x, C)$ (indicating that $x.n$ points to C). Other, so-called instrumentation predicates, such as $r_x(C)$ (node C is reachable from x) are defined in terms of core predicates. Transfer functions that describe the new value after a program statement must be given at least for all core predicates. Two heap cells A_1 and A_2 are summarized by merging the truth values of the predicates that mention them: for example, if $v_i = n(x, A_i)$, then the merged value is v_1 if $v_1 = v_2$ or $\frac{1}{2}$ if $v_1 \neq v_2$. Indeed, TVLA's three-valued interpretation approximates our set of Boolean vectors b by using the value $\frac{1}{2}$ for a predicate p iff $\boldsymbol{b}(p)$ is not constant for all $\boldsymbol{b} \in b$. This is troublesome when, for example, re-evaluating $r_x(C)$ after summarizing two nodes that lie on a path from x to node C: although C is still reachable from x, its re-evaluation on the core predicates yields $\frac{1}{2}$. Indeed, devising precise transfer functions for instrumentation predicates in TVLA is considered a "black art" [16, Sect. 4]. This triggered work on automatic synthesis of transfer function [16].

For certain predicates it is particularly challenging to define a precise transfer function, one of them being $r_x(C)$. The reason is that these predicates use a recursive, transitive closure operator, whose calculation in general requires the whole heap state and, hence, incurs the imprecision of core predicates over summary nodes. In contrast, our analysis only requires node-local information, thereby eschewing the need to perform calculations using information in summary nodes. This strength comes at the cost of rather unintuitive invariants: For instance, in TVLA a predicate would directly state that a summary node A represents an acyclic list, whereas in our analysis the relation $f_{oA} \neq f_{iA}, f_{xA} \neq f_{iA}$ with $[A_x \mapsto \{\langle f_{xA}, A_A\rangle\}]$ states that A is a list which is acyclic if and only if `x` is pointing to it (here f_{iA} and f_{oA} decorate the edges to the ghost node of A).

The flag $f_{\exists N}$ (node N is allocated) resembles the TVLA property *present* of [13]. While our c_N^{in} counter can be seen as a generalization of TVLA's $is(N)$ predicate (is shared, indicating that N has more than one incoming edge), it is actually motivated by work on classifying data structures by Tsai [21]. Indeed, our diamond-shaped subgraph in Fig. 10 can be classified as a shared, acyclic set of heap nodes. We follow Tsai in using reference counting to detect this sharing.

The use of Boolean formulae for a TVLA-style shape analysis was also advocated by Wies et al. [15]. Updates in their predicate abstraction approach are also node-local. However, their analysis does not consider summary nodes. Furthermore, due to the lack of an adequate projection algorithm [5] they deliberately destroy relational information using cartesian abstraction.

An interesting approach to shape analysis is given by Calcagno et al. [7] who propose to perform a backward analysis using abduction. While it is well-known [6] that Boolean functions lend themselves to this kind of task, future work has to address if the combined points-to and Boolean domain also allows for abduction.

A natural extension is the use of a more generic numeric domain like polyhedra [10] in which our instrumentation counters c_N^{in} and c_N^{out} require no special encoding. This would also raise the question of how relational numeric invariants between a summary and its ghost node can be inferred, for instance, to deduce that a list is sorted [12,8]. Future work will address these challenges.

7 Conclusion

We proposed and formalized a fully automatic shape analysis that expresses the heap shape using a single graph and a Boolean function. Our analysis is highly precise by exploiting the ability of Boolean formulae to express relations between heap properties. Due to this relational information, our analysis distinguishes lists from trees from graphs by using only predicates pertaining to the existence of nodes and edges and the number of incoming and outgoing edges.

The key insight is that this relational information can be precisely inferred using a relational *fold* and *expand* [19] that we adapted to Boolean functions. Using these operations, our shape analysis has the ability to infer new shape invariants automatically. We have shown how an efficient implementation of the analysis is possible using SAT solving.

References

1. Balakrishnan, G., Reps, T.: Recency-Abstraction for Heap-Allocated Storage. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 221–239. Springer, Heidelberg (2006)
2. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape Analysis for Composite Data Structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
3. Berndl, M., Lhoták, O., Qian, F., Hendren, L., Umanee, N.: Points-to analysis using BDDs. In: Programming Language Design and Implementation, pp. 103–114. ACM, San Diego (2003)
4. Boquist, U., Johnsson, T.: The Grin Project: A Highly Optimising Back end for Lazy Functional Languages. In: Kluge, W.E. (ed.) IFL 1996. LNCS, vol. 1268, pp. 58–84. Springer, Heidelberg (1997)
5. Brauer, J., King, A., Kriener, J.: Existential Quantification as Incremental SAT. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 191–207. Springer, Heidelberg (2011)
6. Brauer, J., Simon, A.: Inferring Definite Counterexamples through Under-Approximation. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 54–69. Springer, Heidelberg (2012)
7. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional Shape Analysis by means of Bi-Abduction. In: Principles of Programming Languages, Savannah, Georgia, USA, ACM (2009)
8. Chang, B.-Y.E., Rival, X.: Relational Inductive Shape Analysis. In: Principles of Programming Languages, pp. 247–260. ACM (2008)

9. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: Principles of Programming Languages, pp. 269–282. ACM, San Antonio (1979)
10. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Constraints among Variables of a Program. In: Principles of Programming Languages, Tucson, Arizona, USA, pp. 84–97. ACM (1978)
11. Hind, M., Pioli, A.: Which Pointer Analysis Should I Use? In: International Symposium on Software Testing and Analysis, Portland, Oregon, USA, pp. 113–123. ACM (2000)
12. McCloskey, B., Reps, T., Sagiv, M.: Statically Inferring Complex Heap, Array, and Numeric Invariants. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 71–99. Springer, Heidelberg (2010)
13. McCloskey, B.: Practical Shape Analysis. PhD thesis, EECS Department, University of California, Berkeley (May 2010)
14. Nystrom, E.M., Kim, H.S., Hwu, W.W.: Importance of Heap Specialization in Pointer Analysis. In: Flanagan, C., Zeller, A. (eds.) Program Analysis for Software Tools and Engineering. ACM, Washington, DC (2004)
15. Podelski, A., Wies, T.: Boolean Heaps. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 268–283. Springer, Heidelberg (2005)
16. Reps, T., Sagiv, M., Loginov, A.: Finite Differencing of Logical Formulas for Static Analysis. Transactions on Programming Languages and Systems 32, 24:1–24:55 (2010)
17. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science, Copenhagen, Denmark, pp. 55–74. IEEE (2002)
18. Sagiv, M., Reps, T., Wilhelm, R.: Parametric Shape Analysis via 3-Valued Logic. Transactions on Programming Languages and Systems 24(3), 217–298 (2002)
19. Siegel, H., Simon, A.: Summarized Dimensions Revisited. In: Mauborgne, L. (ed.) Workshop on Numeric and Symbolic Abstract Domains, ENTCS, Venice, Italy. Springer (2011)
20. Simon, A.: Splitting the Control Flow with Boolean Flags. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 315–331. Springer, Heidelberg (2008)
21. Tsai, M.-C.: Categorization and Analyzing Linked Structures. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, Illinois, USA (1994)

Simple and Efficient Construction of Static Single Assignment Form

Matthias Braun[1], Sebastian Buchwald[1], Sebastian Hack[2], Roland Leißa[2], Christoph Mallon[2], and Andreas Zwinkau[1]

[1] Karlsruhe Institute of Technology
{matthias.braun,buchwald,zwinkau}@kit.edu
[2] Saarland University
{hack,leissa,mallon}@cs.uni-saarland.de

Abstract. We present a simple SSA construction algorithm, which allows *direct* translation from an abstract syntax tree or bytecode into an SSA-based intermediate representation. The algorithm requires no prior analysis and ensures that even during construction the intermediate representation is in SSA form. This allows the application of SSA-based optimizations during construction. After completion, the intermediate representation is in minimal and pruned SSA form. In spite of its simplicity, the runtime of our algorithm is on par with Cytron et al.'s algorithm.

1 Introduction

Many modern compilers feature intermediate representations (IR) based on the static single assignment form (SSA form). SSA was conceived to make program analyses more efficient by compactly representing use-def chains. Over the last years, it turned out that the SSA form not only helps to make analyses more efficient but also easier to implement, test, and debug. Thus, modern compilers such as the Java HotSpot VM [14], LLVM [2], and libFirm [1] exclusively base their intermediate representation on the SSA form.

The first algorithm to efficiently construct the SSA form was introduced by Cytron et al. [10]. One reason, why this algorithm still is very popular, is that it guarantees a form of *minimality* on the number of placed ϕ functions. However, for compilers that are entirely SSA-based, this algorithm has a significant drawback: Its input program has to be represented as a control flow graph (CFG) in non-SSA form. Hence, if the compiler wants to construct SSA from the input language (be it given by an abstract syntax tree or some bytecode format), it has to take a detour through a non-SSA CFG in order to apply Cytron et al.'s algorithm. Furthermore, to guarantee the minimality of the ϕ function placement, Cytron et al.'s algorithm relies on several other analyses and transformations: To calculate the locations where ϕ functions have to be placed, it computes a dominance tree and the iterated dominance frontiers. To avoid placing dead ϕ functions, liveness analyses or dead code elimination has to be performed [7]. Both, requiring a CFG and relying on other analyses, make it inconvenient to use this algorithm in an SSA-centric compiler.

Modern SSA-based compilers take different approaches to construct SSA: For example, LLVM uses Cytron et al.'s algorithm and mimics the non-SSA CFG by putting all local variables into memory (which is usually not in SSA-form). This comes at the cost of expressing simple definitions and uses of those variables using memory operations. Our measurements show that 25% of all instructions generated by the LLVM front end are of this kind: immediately after the construction of the IR they are removed by SSA construction.

Other compilers, such as the Java HotSpot VM, do not use Cytron et al.'s algorithm at all because of the inconveniences described above. However, they also have the problem that they do not compute minimal and/or pruned SSA form, that is, they insert superfluous and/or dead ϕ functions.

In this paper, we

– present a simple, novel SSA construction algorithm, which does neither require dominance nor iterated dominance frontiers, and thus is suited to construct an SSA-based intermediate representation directly from an AST (Section 2),
– show how to combine this algorithm with on-the-fly optimizations to reduce the footprint during IR construction (Section 3.1),
– describe a post pass that establishes minimal SSA form for arbitrary programs (Section 3.2),
– prove that the SSA construction algorithm constructs pruned SSA form for all programs and minimal SSA form for programs with reducible control flow (Section 4),
– show that the algorithm can also be applied in related domains, like translating an imperative program to a functional continuation-passing style (CPS) program or reconstructing SSA form after transformations, such as live range splitting or rematerialization, have added further definitions to an SSA value (Section 5),
– demonstrate the efficiency and simplicity of the algorithm by implementing it in Clang and comparing it with Clang/LLVM's implementation of Cytron et al.'s algorithm (Section 6).

To the best of our knowledge, the algorithm presented in this paper is the first to construct minimal and pruned SSA on reducible CFGs without depending on other analyses.

2 Simple SSA Construction

In the following, we describe our algorithm to construct SSA form. It significantly differs from Cytron et al.'s algorithm in its basic idea. Cytron et al.'s algorithm is an eager approach operating in forwards direction: First, the algorithm collects all definitions of a variable. Then, it calculates the placement of corresponding ϕ functions and, finally, pushes these definitions down to the uses of the variable. In contrast, our algorithm works backwards in a lazy fashion: Only when a variable is used, we query its reaching definition. If it is unknown at the current location, we will search backwards through the program. We insert ϕ functions at join points in the CFG along the way, until we find the desired definition. We employ memoization to avoid repeated look-ups.

This process consists of several steps, which we explain in detail in the rest of this section. First, we consider a single basic block. Then, we extend the algorithm to whole CFGs. Finally, we show how to handle incomplete CFGs, which usually emerge when translating an AST to IR.

2.1 Local Value Numbering

When translating a source program, the IR for a sequence of statements usually ends up in a single basic block. We process these statements in program execution order and for each basic block we keep a mapping from each source variable to its current defining expression. When encountering an assignment to a variable, we record the IR of the right-hand side of the assignment as current definition of the variable. Accordingly, when a variable is read, we look up its current definition (see Algorithm 1). This process is well known in literature as *local value numbering* [9]. When local value numbering for one block is finished, we call this block *filled*. Particularly, successors may only be added to a filled block. This property will later be used when handling incomplete CFGs.

```
a ← 42;                 v_1: 42
b ← a;
c ← a + b;              v_2: v_1 + v_1
                        v_3: 23
a ← c + 23;             v_4: v_2 + v_3
c ← a + d;              v_5: v_4 + v_?
```

(a) Source program (b) SSA form

Fig. 1. Example for local value numbering

```
writeVariable(variable, block, value):
  currentDef[variable][block] ← value

readVariable(variable, block):
  if currentDef[variable] contains block:
    # local value numbering
    return currentDef[variable][block]
  # global value numbering
  return readVariableRecursive(variable, block)
```

Algorithm 1. Implementation of local value numbering

A sample program and the result of this process is illustrated in Figure 1. For the sake of presentation, we denote each SSA value by a name v_i.[1] In a concrete implementation, we would just refer to the representation of the expression. The names have no meaning otherwise, in particular they are not local variables in the sense of an imperative language.

[1] This acts like a `let` binding in a functional language. In fact, SSA form is a kind of functional representation [3].

Now, a problem occurs if a variable is read before it is assigned in a basic block. This can be seen in the example for the variable d and its corresponding value $\mathsf{v}_?$. In this case, d's definition is found on a path from the CFG's root to the current block. Moreover, multiple definitions in the source program may reach the same use. The next section shows how to extend local value numbering to handle these situations.

2.2 Global Value Numbering

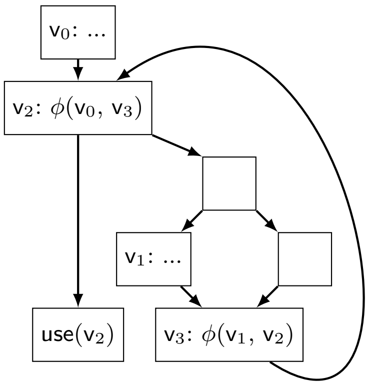
If a block currently contains no definition for a variable, we recursively look for a definition in its predecessors. If the block has a single predecessor, just query it recursively for a definition. Otherwise, we collect the definitions from all predecessors and construct a ϕ function, which joins them into a single new value. This ϕ function is recorded as current definition in this basic block.

```
x ← ...
while (...)

{
  if (...) {
    x ← ...
  }

}
use(x)
```

(a) Source Program

```
v0: ...
while (...)
  v2: φ(v0, v3)
{
  if (...) {
    v1: ...
  }
  v3: φ(v1, v2)
}
use(v2)
```

(b) SSA form



(c) Control flow graph

Fig. 2. Example for global value numbering

Looking for a value in a predecessor might in turn lead to further recursive look-ups. Due to loops in the program, those might lead to endless recursion. Therefore, *before recursing*, we first create the ϕ function without operands and record it as the current definition for the variable in the block. Then, we determine the ϕ function's operands. If a recursive look-up arrives back at the block, this ϕ function will provide a definition and the recursion will end. Algorithm 2 shows pseudocode to perform global value numbering. Its first condition will be used to handle incomplete CFGs, so for now assume it is always false.

Figure 2 shows this process. For presentation, the indices of the values v_i are assigned in the order in which the algorithm inserts them. We assume that the loop is constructed before x is read, i.e., v_0 and v_1 are recorded as definitions for x by local value numbering and only then the statement after the loop looks up x. As there is no definition for x recorded in the block after the loop, we perform a recursive look-up. The block has only a single predecessor, so no ϕ function is needed here. This predecessor is the loop header, which also has no definition

```
readVariableRecursive(variable, block):
  if block not in sealedBlocks:
    # Incomplete CFG
    val ← new Phi(block)
    incompletePhis[block][variable] ← val
  else if |block.preds| = 1:
    # Optimize the common case of one predecessor: No phi needed
    val ← readVariable(variable, block.preds[0])
  else:
    # Break potential cycles with operandless phi
    val ← new Phi(block)
    writeVariable(variable, block, val)
    val ← addPhiOperands(variable, val)
  writeVariable(variable, block, val)
  return val

addPhiOperands(variable, phi):
  # Determine operands from predecessors
  for pred in phi.block.preds:
    phi.appendOperand(readVariable(variable, pred))
  return tryRemoveTrivialPhi(phi)
```

Algorithm 2. Implementation of global value numbering

```
tryRemoveTrivialPhi(phi):
  same ← None
  for op in phi.operands:
    if op = same || op = phi:
      continue # Unique value or self−reference
    if same ≠ None:
      return phi # The phi merges at least two values: not trivial
    same ← op
  if same = None:
    same ← new Undef() # The phi is unreachable or in the start block
  users ← phi.users.remove(phi) # Remember all users except the phi itself
  phi.replaceBy(same) # Reroute all uses of phi to same and remove phi

  # Try to recursively remove all phi users, which might have become trivial
  for use in users:
    if use is a Phi:
      tryRemoveTrivialPhi(use)
  return same
```

Algorithm 3. Detect and recursively remove a trivial ϕ function

for x and has two predecessors. Thus, we place an operandless ϕ function v_2. Its first operand is the value v_0 flowing into the loop. The second operand requires further recursion. The ϕ function v_3 is created and gets its operands from its direct predecessors. In particular, v_2 placed earlier breaks the recursion.

Recursive look-up might leave redundant ϕ functions. We call a ϕ function v_ϕ *trivial* iff it just references itself and one other value v any number of times: $\exists v \in V\colon v_\phi\colon \phi(x_1, \ldots, x_n) \quad x_i \in \{v_\phi, v\}$. Such a ϕ function can be removed and the value v is used instead (see Algorithm 3). As a special case, the ϕ function might use no other value besides itself. This means that it is either unreachable or in the start block. We replace it by an undefined value.

Moreover, if a ϕ function could be successfully replaced, other ϕ functions using this replaced value might become trivial as well. For this reason, we apply this simplification recursively on all of these users.

This approach works for all acyclic language constructs. In this case, we can fill all predecessors of a block before processing it. The recursion will only search in already filled blocks. This ensures that we retrieve the latest definition for each variable from the predecessors. For example, in an if-then-else statement the block containing the condition can be filled before the then and else branches are processed. Accordingly, after the two branches are completed, the block joining the branches is filled. This approach also works when reading a variable after a loop has been constructed. But when reading a variable within a loop, which is under construction, some predecessors—at least the jump back to the head of the loop—are missing.

2.3 Handling Incomplete CFGs

We call a basic block *sealed* if no further predecessors will be added to the block. As only filled blocks may have successors, predecessors are always filled. Note that a sealed block is not necessarily filled. Intuitively, a filled block contains all its instructions and can provide variable definitions for its successors. Conversely, a sealed block may look up variable definitions in its predecessors as all predecessors are known.

```
sealBlock(block):
  for variable in incompletePhis[block]:
    addPhiOperands(variable, incompletePhis[block][variable])
  sealedBlocks.add(block)
```

Algorithm 4. Handling incomplete CFGs

But how to handle a look-up of a variable in an unsealed block, which has no current definition for this variable? In this case, we place an operandless ϕ function into the block and record it as proxy definition (see first case in Algorithm 2). Further, we maintain a set incompletePhis of these proxies per block. When later on a block gets sealed, we add operands to these ϕ functions (see Algorithm 4). Again, when the ϕ function is complete, we check whether it is trivial.

Sealing a block is an explicit action during IR construction. We illustrate how to incorporate this step by the example of constructing the while loop seen in Figure 3a. First, we construct the while header block and add a control flow edge from the while entry block to it. Since the jump from the body exit needs to be added later, we cannot seal the while header yet. Next, we create the body entry

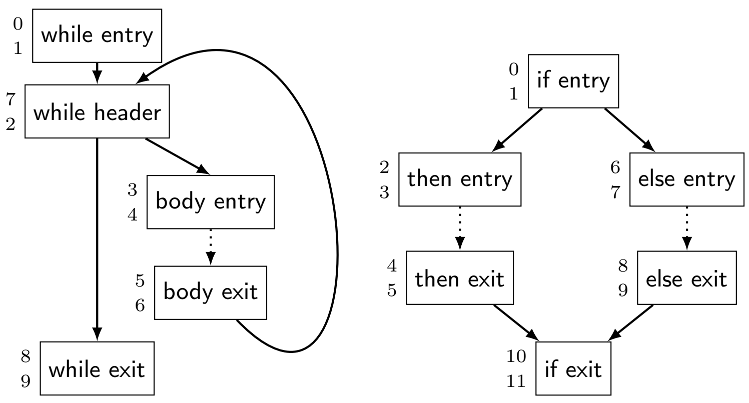


Fig. 3. CFG illustration of construction procedures. Dotted lines represent possible further code, while straight lines are normal control flow edges. Numbers next to a basic block denote the order of sealing (top) and filling (bottom).

and while exit blocks and add the conditional control flow from the while header to these two blocks. No further predecessors will be added to the body entry block, so we seal it now. The while exit block might get further predecessors due to break instructions in the loop body. Now we fill the loop body. This might include further inner control structures, like an if shown in Figure 3b. Finally, they converge at the body exit block. All the blocks forming the body are sealed at this point. Now we add the edge back to the while header and seal the while header. The loop is completed. In the last step, we seal the while exit block and then continue IR construction with the source statement after the while loop.

3 Optimizations

3.1 On-the-Fly Optimizations

In the previous section, we showed that we optimize trivial ϕ functions as soon as they are created. Since ϕ functions belong to the IR, this means we employ an IR optimization during SSA construction. Obviously, this is not possible with all optimizations. In this section, we elaborate what kind of IR optimizations can be performed during SSA construction and investigate their effectiveness.

We start with the question whether the optimization of ϕ functions removes all trivial ϕ functions. As already mentioned in Section 2.2, we recursively optimize all ϕ functions that have used a removed trivial ϕ function. Since a successful optimization of a ϕ function can only render ϕ functions trivial that use the former one, this mechanism enables us to optimize all trivial ϕ functions. In Section 4, we show that, for reducible CFGs, this is equivalent to the construction of minimal SSA form.

Since our approach may lead to a significant number of triviality checks, we use the following caching technique to speed such checks up: While constructing a ϕ function, we record the first two distinct operands that are also distinct from the ϕ function itself. These operands act as witnesses for the non-triviality of the ϕ function. When a new triviality check occurs, we compare the witnesses again. If they remain distinct from each other and the ϕ function, the ϕ function is still non-trivial. Otherwise, we need to find a new witness. Since all operands until the second witness are equal to the first one or to the ϕ function itself, we only need to consider operands that are constructed after both old witnesses. Thus, the technique speeds up multiple triviality checks for the same ϕ function.

There is a more simple variant of the mechanism that results in minimal SSA form for most but not all cases: Instead of optimizing the users of a replaced ϕ function, we optimize the unique operand. This variant is especially interesting for IRs, which do not inherently provide a list of users for each value.

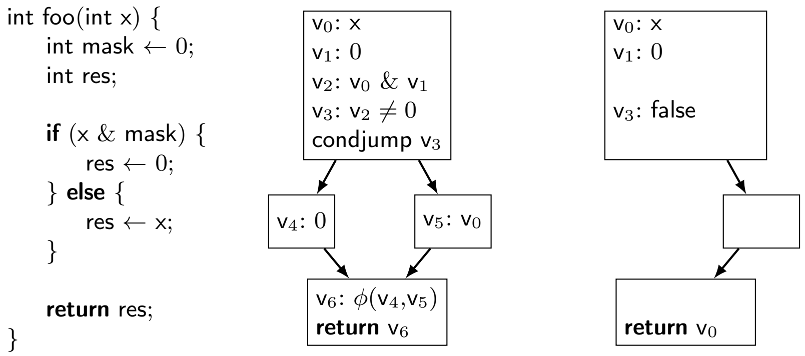
The optimization of ϕ functions is only one out of many IR optimizations that can be performed during IR construction. In general, our SSA construction algorithm allows to utilize conservative IR optimizations, i.e., optimizations that require only local analysis. These optimizations include:

Arithmetic Simplification. All IR node constructors perform peephole optimizations and return simplified nodes when possible. For instance, the construction of a subtraction x−x always yields the constant 0.

Common Subexpression Elimination. This optimization reuses existing values that are identified by local value numbering.

Constant Folding. This optimization evaluates constant expressions at compile time, e.g., `2*3` is optimized to `6`.

Copy Propagation. This optimization removes unnecessary assignments to local variables, e.g., `x = y`. In SSA form, there is no need for such assignments, we can directly use the value of the right-hand side.



```
int foo(int x) {
    int mask ← 0;
    int res;

    if (x & mask) {
        res ← 0;
    } else {
        res ← x;
    }

    return res;
}
```

(a) Program code (b) Unoptimized (c) Optimized

Fig. 4. The construction algorithm allows to perform conservative optimization during SSA construction. This may also affect control flow, which in turn could lead to a reduced number of ϕ functions.

Figure 4 shows the effectiveness of these optimizations. We want to construct SSA form for the code fragment shown in Figure 4a. Without on-the-fly optimizations, this results in the SSA form program shown in Figure 4b. The first difference with enabled optimizations occurs during the construction of the value v_2. Since a bitwise conjunction with zero always yields zero, the arithmetic simplification triggers and simplifies this value. Moreover, the constant value zero is already available. Thus, the common subexpression elimination reuses the value v_1 for the value v_2. In the next step, constant propagation folds the comparison with zero to `false`. Since the condition of the conditional jump is `false`, we can omit the `then` part.[2] Within the `else` part, we perform copy propagation by registering v_0 as value for res. Likewise, v_6 vanishes and in the end the function returns v_0. Figure 4c shows the optimized SSA form program.

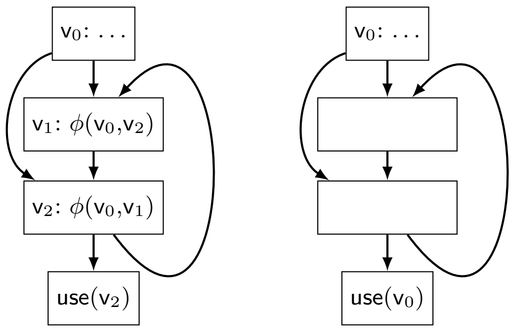
The example demonstrates that on-the-fly optimizations can further reduce the number of ϕ functions. This can even lead to fewer ϕ functions than required for minimal SSA form according to Cytron et al.'s definition.

3.2 Minimal SSA Form for Arbitrary Control Flow

So far, our SSA construction algorithm does not construct minimal SSA form in the case of irreducible control flow. Figure 5b shows the constructed SSA form for the program shown in Figure 5a. Figure 5c shows the corresponding minimal SSA form—as constructed by Cytron et al.'s algorithm. Since there is only one definition for the variable x, the ϕ functions constructed by our algorithm are superfluous.

```
x ← ...

if (...)
    goto second_loop_entry

while (...) {
    ...
second_loop_entry:
    ...
}

use(x)
```

(a) Program with irreducible control flow



(b) Intermediate representation with our SSA construction algorithm

(c) Intermediate representation in minimal SSA form

Fig. 5. Our SSA construction algorithm can produce extraneous ϕ functions in presence of irreducible control flow. We remove these ϕ functions afterwards.

In general, a non-empty set P of ϕ functions is *redundant* iff the ϕ functions just reference each other or one other value v: $\exists v \in V\ \forall v_i \in P\colon v_i\colon \phi(x_1,\ldots,x_n)\ x_i \in P \cup \{v\}$. In particular, when P contains only a single ϕ function, this

[2] This is only possible because the `then` part contains no labels.

```
proc removeRedundantPhis(phiFunctions):
    sccs ← computePhiSCCs(inducedSubgraph(phiFunctions))
    for scc in topologicalSort(sccs):
        processSCC(scc)

proc processSCC(scc):
    if len(scc) = 1: return # we already handled trivial $\phi$ functions

    inner ← set()
    outerOps ← set()
    for phi in scc:
        isInner ← True
        for operand in phi.getOperands():
            if operand not in scc:
                outerOps.add(operand)
                isInner ← False
        if isInner:
            inner.add(phi)

    if len(outerOps) = 1:
        replaceSCCByValue(scc, outerOps.pop())
    else if len(outerOps) > 1:
        removeRedundantPhis(inner)
```

Algorithm 5. Remove superfluous ϕ functions in case of irreducible data flow

definition degenerates to the definition of a trivial ϕ function given in Section 2.2. We show that each set of redundant ϕ functions P contains a strongly connected component (SCC) that is also redundant. This implies a definition of minimality that is independent of the source program and more strict than the definition by Cytron et al. [10].

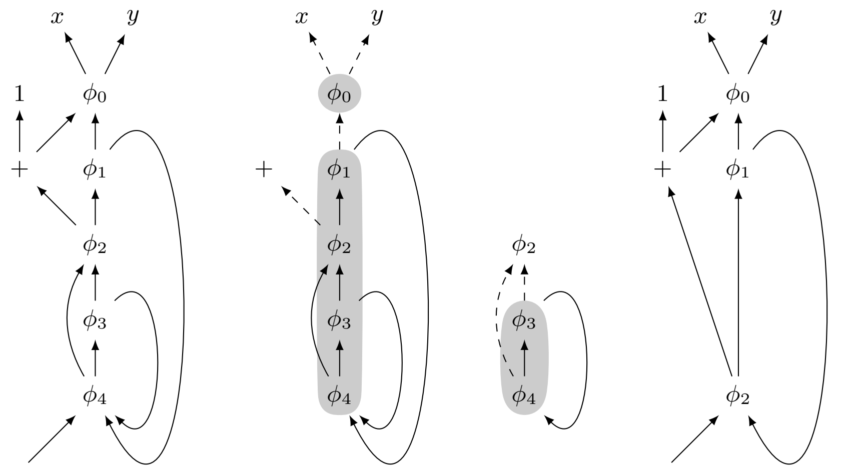
Lemma 1. *Let P be a redundant set of ϕ functions with respect to v. Then there is a strongly connected component $S \subseteq P$ that is also redundant.*

Proof. Let P' be the condensation of P, i.e., each SCC in P is contracted into a single node. The resulting P' is acyclic [11]. Since P' is non-empty, it has a leaf s'. Let S be the SCC, which corresponds to s'. Since s' is a leaf, the ϕ functions in S only refer to v or other ϕ functions in S. Hence, S is a redundant SCC. □

Algorithm 5 exploits Lemma 1 to remove superfluous ϕ functions. The function removeRedundantPhis takes a set of ϕ functions and computes the SCCs of their induced subgraph. Figure 6b shows the resulting SCCs for the data flow graph in Figure 6a. Each dashed edge targets a value that does not belong to the SCC of its source. We process the SCCs in topological order to ensure that used values outside of the current SCC are already contracted. In our example, this means we process the SCC containing only ϕ_0 first. Since ϕ_0 is the only ϕ function within its SCC, we already handled it during removal of trivial ϕ functions. Thus, we skip this SCC.

For the next SCC containing ϕ_1–ϕ_4, we construct two sets: The set inner contains all ϕ functions having operands solely within the SCC. The set outerOps contains all ϕ function operands that do not belong to the SCC. For our example, inner$= \{\phi_3, \phi_4\}$ and outerOps$= \{\phi_0, +\}$.

If the outerOps set is empty, the corresponding basic blocks must be unreachable and we skip the SCC. In case that the outerOps set contains exactly one value, all ϕ functions within the SCC can only get this value. Thus, we replace the SCC by the value. If the outerOps set contains multiple values, all ϕ functions that have an operand outside the SCC are necessary. We collected the remaining ϕ functions in the set inner. Since our SCC can contain multiple inner SCCs, we recursively perform the procedure with the inner ϕ functions. Figure 6c shows the inner SCC for our example. In the recursive step, we replace this SCC by ϕ_2. Figure 6d shows the resulting data flow graph.



(a) Original data flow graph (b) SCCs and their operands (c) Inner SCC (d) Optimized data flow graph

Fig. 6. Algorithm 5 detects the inner SCC spanned by ϕ_3 and ϕ_4. This SCC represents the same value. Thus, it gets replaced by ϕ_2.

Performing on-the-fly optimizations (Section 3.1) can also lead to irreducible data flow. For example, let us reconsider Figure 2 described in Section 2.2. Assume that both assignments to x are copies from some other variable y. If we now perform copy propagation, we obtain two ϕ functions v_2: $\phi(v_0, v_3)$ and v_3: $\phi(v_0, v_2)$ that form a superfluous SCC. Note that this SCC also will appear after performing copy propagation on a program constructed with Cytron's algorithm. Thus, Algorithm 5 is also applicable to other SSA construction algorithms. Finally, Algorithm 5 can be seen as a generalization of the local simplifications by Aycock and Horspool (see Section 7).

3.3 Reducing the Number of Temporary ϕ Functions

The presented algorithm is structurally simple and easy to implement. Though, it might produce many temporary ϕ functions, which get removed right away during IR construction. In the following, we describe two extensions to the algorithm, which aim at reducing or even eliminating these temporary ϕ functions.

Marker Algorithm. Many control flow joins do not need a ϕ function for a variable. So instead of placing a ϕ function before recursing, we just mark the block as visited. If we reach this block again during recursion, we will place a ϕ function there to break the cycle. After collecting the definitions from all predecessors, we remove the marker and place a ϕ function (or reuse the one placed by recursion) if we found different definitions. Using this technique, no temporary ϕ functions are placed in acyclic data-flow regions. Temporary ϕ functions are only generated in data-flow loops and might be determined as unnecessary later on.

SCC Algorithm. While recursing we use Tarjan's algorithm to detect data-flow cycles, i.e., SCCs [17]. If only a unique value enters the cycle, no ϕ functions will be necessary. Otherwise, we place a ϕ function into every basic block, which has a predecessor from outside the cycle. In order to add operands to these ϕ functions, we apply the recursive look-up again as this may require placement of further ϕ functions. This mirrors the algorithm for removing redundant cycles of ϕ functions described in Section 3.2. In case of recursing over sealed blocks, the algorithm only places necessary ϕ functions. The next section gives a formal definition of a necessary ϕ function and shows that an algorithm that only places necessary ϕ functions produces minimal SSA form.

4 Properties of Our Algorithm

Because most optimizations treat ϕ functions as uninterpreted functions, it is beneficial to place as few ϕ functions as possible. In the rest of this section, we show that our algorithm does not place dead ϕ functions and constructs minimal (according to Cytron et al.'s definition) SSA form for programs with reducible control flow.

Pruned SSA Form. A program is said to be in pruned SSA form [7] if each ϕ function (transitively) has at least one non-ϕ user. We only create ϕ functions on demand when a user asks for it: Either a variable being read or another ϕ function needing an argument. So our construction naturally produces a program in pruned SSA form.

Minimal SSA Form. Minimal SSA form requires that ϕ functions for a variable v only occur in basic blocks where different definitions of v meet for the first time. Cytron et al.'s formal definition is based on the following two terms:

Definition 1 (Path Convergence). *Two non-null paths $X_0 \rightarrow^+ X_J$ and $Y_0 \rightarrow^+ Y_K$ are said to* converge *at a block Z iff the following conditions hold:*

$$X_0 \neq Y_0; \tag{1}$$

$$X_J = Z = Y_K; \tag{2}$$

$$(X_j = Y_k) \Rightarrow (j = J \vee k = K). \tag{3}$$

Definition 2 (Necessary ϕ Function). *A ϕ function for variable v is necessary in block Z iff two non-null paths $X \rightarrow^+ Z$ and $Y \rightarrow^+ Z$ converge at a block Z, such that the blocks X and Y contain assignments to v.*

A program with only necessary ϕ functions is in *minimal SSA form*. The following is a proof that our algorithm presented in Section 2 with the simplification rule for ϕ functions produces minimal SSA form for reducible programs.

We say a block A *dominates* a block B if every path from the entry block to B passes through A. We say A *strictly dominates* B if A dominates B and $A \neq B$. Each block C except the entry block has a unique immediate dominator $idom(C)$, i.e., a strict dominator of C, which does not dominate any other strict dominator of C. The dominance relation can be represented as a tree whose nodes are the basic blocks with a connection between immediately dominating blocks.

Definition 3 (Reducible Flow Graph, Hecht and Ullmann [12]). *A (control) flow graph G is reducible iff for each cycle C of G there is a node of C, which dominates all other nodes in C.*

We now assume that our construction algorithm is finished and has produced a program with a reducible CFG. We observe that the simplification rule tryRemoveTrivialPhi of Algorithm 3 was applied at least once to each ϕ function with its current arguments. This is because we apply the rule each time a ϕ function's parameters are set for the first time. In the case that a simplification of another operation leads to a change of parameters, the rule is applied again. Furthermore, our construction algorithm fulfills the following property:

Definition 4 (SSA Property). *In an SSA-form program a path from a definition of an SSA value for variable v to its use cannot contain another definition or ϕ function for v. The use of the operands of ϕ function happens in the respective predecessor blocks not in the ϕ's block itself.*

The SSA property ensures that only the "most recent" SSA value of a variable v is used. Furthermore, it forbids multiple ϕ functions for one variable in the same basic block.

Lemma 2. *Let p be a ϕ function in a block P. Furthermore, let q in a block Q and r in a block R be two operands of p, such that p, q and r are pairwise distinct. Then at least one of Q and R does not dominate P.*

Proof. Assume that Q and R dominate P, i.e., every path from the start block to P contains Q and R. Since immediate dominance forms a tree, Q dominates R or R dominates Q. Without loss of generality, let Q dominate R. Furthermore, let S be the corresponding predecessor block of P where p is using q. Then there is a path from the start block crossing Q then R and S. This violates the SSA property. □

Lemma 3. *If a ϕ function p in a block P for a variable v is unnecessary, but non-trivial, then it has an operand q in a block Q, such that q is an unnecessary ϕ function and Q does not dominate P.*

Proof. The node p must have at least two different operands r and s, which are not p itself. Otherwise, p is trivial. They can either be:

– The result of a direct assignment to v.
– The result of a necessary ϕ function r'. This however means that r' was reachable by at least two different direct assignments to v. So there is a path from a direct assignment of v to p.
– Another unnecessary ϕ function.

Assume neither r in a block R nor s in a block S is an unnecessary ϕ function. Then a path from an assignment to v in a block V_r crosses R and a path from an assignment to v in a block V_s crosses S. They converge at P or earlier. Convergence at P is not possible because p is unnecessary. An earlier convergence would imply a necessary ϕ function at this point, which violates the SSA property.

So r or s must be an unnecessary ϕ function. Without loss of generality, let this be r.

If R does not dominate P, then r is the sought-after q. So let R dominate P. Due to Lemma 2, S does not dominate P. Employing the SSA property, $r \neq p$ yields $R \neq P$. Thus, R strictly dominates P. This implies that R dominates all predecessors of P, which contain the uses of p, especially the predecessor S' that contains the use of s. Due to the SSA property, there is a path from S to S' that does not contain R. Employing R dominates S' this yields R dominates S.

Now assume that s is necessary. Let X contain the most recent definition of v on a path from the start block to R. By Definition 2 there are two definitions of v that render s necessary. Since R dominates S, the SSA property yields that one of these definitions is contained in a block Y on a path $R \to^+ S$. Thus, there are paths $X \to^+ P$ and $Y \to^+ P$ rendering p necessary. Since this is a contradiction, s is unnecessary and the sought-after q. □

Theorem 1. *A program in SSA form with a reducible CFG G without any trivial ϕ functions is in minimal SSA form.*

Proof. Assume G is not in minimal SSA form and contains no trivial ϕ functions. We choose an unnecessary ϕ function p. Due to Lemma 3, p has an operand q, which is unnecessary and does not dominate p. By induction q has an unnecessary ϕ function as operand as well and so on. Since the program only has a finite number of operations, there must be a cycle when following the q chain. A cycle in the ϕ functions implies a cycle in G. As G is reducible, the control flow cycle contains one entry block, which dominates all other blocks in the cycle. Without loss of generality, let q be in the entry block, which means it dominates p. Therefore, our assumption is wrong and G is either in minimal SSA form or there exist trivial ϕ functions. □

Because our construction algorithm will remove all trivial ϕ functions, the resulting IR must be in minimal SSA form for reducible CFGs.

4.1 Time Complexity

We use the following parameters to provide a precise worst-case complexity for our construction algorithm:

- B denotes the number of basic blocks.
- E denotes the number of CFG edges.
- P denotes the program size.
- V denotes the number of variables in the program.

We start our analysis with the simple SSA construction algorithm presented in Section 2.3. In the worst case, SSA construction needs to insert $\Theta(B)$ ϕ functions with $\Theta(E)$ operands for each variable. In combination with the fact the construction of SSA form *within* all basic block is in $\Theta(P)$, this leads to a lower bound of $\Omega(P + (B + E) \cdot V)$.

We show that our algorithm matches this lower bound, leading to a worst-case complexity of $\Theta(P + (B + E) \cdot V)$. Our algorithm requires $\Theta(P)$ to fill all basic blocks. Due to our variable mapping, we place at most $\mathcal{O}(B \cdot V)$ ϕ functions. Furthermore, we perform at most $\mathcal{O}(E \cdot V)$ recursive requests at block predecessors. Altogether, this leads to a worst-case complexity of $\Theta(P + (B + E) \cdot V)$.

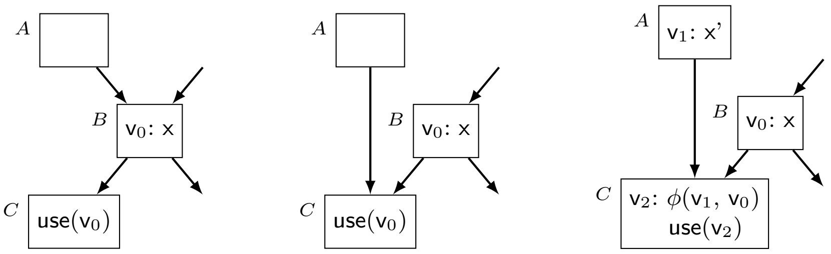
Next, we consider the on-the-fly optimization of ϕ functions. Once we optimized a ϕ function, we check whether we can optimize the ϕ functions that use the former one. Since our algorithm constructs at most $B \cdot V$ ϕ functions, this leads to $\mathcal{O}(B^2 \cdot V^2)$ checks. One check needs to compare at most $\mathcal{O}(B)$ operands of the ϕ function. However, using the caching technique described in Section 3.1, the number of checks performed for each ϕ functions amortizes the time for checking the corresponding ϕ function. Thus, the on-the-fly optimization of ϕ functions can be performed in $\mathcal{O}(B^2 \cdot V^2)$.

To obtain minimal SSA form, we need to contract SCCs that pass the same value. Since we consider only ϕ functions and their operands, the size of the SCCs is in $\mathcal{O}((B + E) \cdot V)$. Hence, computing the SCCs for the data flow graph is in $\mathcal{O}(P + (B + E) \cdot V)$. Computing the sets inner and outer consider each ϕ function and its operands exactly once. Thus, it is also in $\mathcal{O}((B + E) \cdot V)$. The same argument applies for the contraction of a SCC in case there is only one outer operand. In the other case, we iterate the process with a subgraph that is induced by a proper subset of the nodes in our SCC. Thus, we need at most $B \cdot V$ iterations. In total, this leads to a time complexity in $\mathcal{O}(P + B \cdot (B + E) \cdot V^2)$ for the contraction of SCCs.

5 Other Applications of the Algorithm

5.1 SSA Reconstruction

Some transformations like live range splitting, rematerialization or jump threading introduce additional definitions for an SSA value. Because this violates the SSA property, SSA has to be *reconstructed*. For the latter transformation, we run



(a) Original program (b) Optimized control flow (c) Fixed SSA form

Fig. 7. We assume that A always jumps via B to C. Adjusting A's jump requires SSA reconstruction.

```
int f(int x) {
  int a;
  if (x = 0) {
    a ← 23;
  } else {
    a ← 42;
  }

  return a;
}
```

(a) Source program

```
int f(int x) {

  if (x = 0) {

  } else {

  }
  v0: φ(23, 42)
  return v0
}
```

(b) SSA form

```
f(x : int, ret : int → ⊥) → ⊥ {
  let then := () → ⊥
      next(23)
    else := () → ⊥
      next(42)
    next := (a : int) → ⊥
      ret(a)
  in
    branch(x = 0, then, else)
}
```

(c) CPS version

Fig. 8. An imperative program in SSA form and converted to a functional CPS program

through an example in order to demonstrate how our algorithm can be leveraged for SSA reconstruction.

Consider an analysis determined that the basic block B in Figure 7a always branches to C when entered from A. Thus, we let A directly jump to C (Figure 7b). However, definition v_0 does not dominate its use anymore. We can fix this issue by first inserting a copy v_1 of v_0 into A. Then, we invoke writeVariable(V, A, x') and writeVariable(V, B, x) while V is just some handle to refer to the set of definitions, which represent the "same variable". Next, a call to readVariableRecursive(V, C) adds a necessary ϕ function and yields v_2 as new definition, which we can use to update v_0's original use (Figure 7c).

In particular, for jump threading, it is desirable to not depend on dominance calculations—as opposed to Cytron et al.'s algorithm: Usually, several iterations of jump threading are performed until no further improvements are possible. Since jump threading alters control flow in a non-trivial way, each iteration would require a re-computation of the dominance tree.

Note that SSA reconstruction always runs on complete CFGs. Hence, sealing and issues with non-sealed basic blocks do not arise in this setting.

5.2 CPS Construction

CPS is a functional programming technique that captures control flow in *continuations*. Continuations are functions, which never return. Instead, each continuation invokes another continuation in tail position.

SSA form can be considered as a restricted form of CPS [13,3]. Our algorithm is also suitable to directly convert an imperative program into a functional CPS program without the need for a third program representation. Instead of ϕ functions, we have to place parameters in local functions. Instead of adding operands to ϕ functions, we add arguments to the predecessors' calls. Like when constructing SSA form, on-the-fly optimizations, as described in Section 3.1, can be exploited to shrink the program. Figure 8 demonstrates CPS construction.

In a CPS program, we cannot simply remove a ϕ function. Rather, we would have to eliminate a parameter, fix its function's type and adjust all users of this function. As this set of transformations is expensive, it is worthwhile to not introduce unnecessary parameters in the first place and therefore use the extensions described in Section 3.3.

6 Evaluation

6.1 Comparison to Cytron et al.'s Algorithm

We implemented the algorithm presented in this paper in LLVM 3.1 [2] to compare it against an existing, highly-tuned implementation of Cytron et al.'s algorithm. Table 1 shows the number of constructed instructions for both algorithms. Since LLVM first models accesses to local variables with load and stores instructions, we also denoted the instructions immediately before SSA construction.

Table 1. Comparison of instruction counts of LLVM's normal implementation and our algorithm. *#mem* are alloca, load and store instructions. *Insn ratio* is the quotient between *#insn* of *before SSA construction* and *marker*.

Bench-	Before SSA Constr.			After SSA Constr.			Marker			Insn
mark	#insn	#mem	#phi	#insn	#mem	#phi	#insn	#mem	#phi	ratio
gzip	12,038	5,480	82	9,187	2,117	594	9,179	2,117	594	76%
vpr	40,701	21,226	129	27,155	6,608	1,201	27,092	6,608	1,201	67%
gcc	516,537	206,295	2,230	395,652	74,736	12,904	393,554	74,683	12,910	76%
mcf	3,988	2,173	14	2,613	658	154	2,613	658	154	66%
crafty	44,891	18,804	116	36,050	8,613	1,466	36,007	8,613	1,466	80%
parser	30,237	14,542	100	20,485	3,647	1,243	20,467	3,647	1,243	68%
perlbmk	185,576	86,762	1,764	140,489	37,599	5,840	140,331	37,517	5,857	76%
gap	201,185	86,157	4,074	149,755	29,476	9,325	149,676	29,475	9,326	74%
vortex	126,097	65,245	990	88,257	25,656	2,739	88,220	25,661	2,737	70%
bzip2	8,605	4,170	9	6,012	1,227	359	5,993	1,227	359	70%
twolf	76,078	38,320	246	58,737	18,376	2,849	58,733	18,377	2,849	77%
Sum	1,245,933	549,174	9,754	934,392	208,713	38,674	931,865	208,583	38,696	75%

In total, SSA construction reduces the number of instructions by 25%, which demonstrates the significant overhead of the temporary non-SSA IR.

Comparing the number of constructed instructions, we see small differences between the results of LLVM's and our SSA-construction algorithm. One reason for the different number of ϕ functions is the removal of redundant SCCs: 3 (out of 11) non-trivial SCCs do not originate from irreducible control flow and are not removed by Cytron et al.'s algorithm. The remaining difference in the number of ϕ functions and memory instructions stems from minor differences in handling unreachable code. In most benchmarks, our algorithm triggers LLVM's constant folding more often, and thus further reduces the overall instruction count. Exploiting more on-the-fly optimizations like common subexpression elimination as described in Section 3.1 would shrink the overall instruction count even further.

Table 2. Executed instructions for Cytron et al.'s algorithm and the Marker algorithm

Benchmark	Cytron et al.	Marker	$\frac{\text{Marker}}{\text{Cytron et al.}}$
164.gzip	969,233,677	967,798,047	99.85%
175.vpr	3,039,801,575	3,025,286,080	99.52%
176.gcc	25,935,984,569	26,009,545,723	100.28%
181.mcf	722,918,540	722,507,455	99.94%
186.crafty	3,653,881,430	3,632,605,590	99.42%
197.parser	2,084,205,254	2,068,075,482	99.23%
253.perlbmk	12,246,953,644	12,062,833,383	98.50%
254.gap	8,358,757,289	8,339,871,545	99.77%
255.vortex	7,841,416,740	7,845,699,772	100.05%
256.bzip2	569,176,687	564,577,209	99.19%
300.twolf	6,424,027,368	6,408,289,297	99.76%
Sum	71,846,356,773	71,647,089,583	99.72%

For the runtime comparison of both benchmarks, we count the number of executed x86 instructions. Table 2 shows the counts collected by the valgrind instrumentation tool. While the results vary for each benchmark, the marker algorithm needs slightly (0.28%) fewer instructions in total. All measurements were performed on a Core i7-2600 CPU with 3.4 GHz, by compiling the C-programs of the SPEC CINT2000 benchmark suite.

6.2 Effect of On-the-Fly Optimization

We also evaluated the effects of performing on-the-fly optimizations (as described in Section 3.1) on the speed and quality of SSA construction. Our libFirm [1] compiler library has always featured a variant of the construction algorithms described in this paper.

There are many optimizations interweaved with the SSA construction. The results are shown in Table 3. Enabling on-the-fly optimizations during construction results in an increased construction time of 0.84 s, but the resulting graph

Table 3. Effect of on-the-fly optimizations on construction time and IR size

Benchmark	No On-the-fly Optimizations			On-the-fly Optimizations		
	Time	IR Time	Instructions	Time	IR Time	Instructions
164.gzip	1.38 s	0.03 s	10,520	1.34 s	0.05 s	9,255
175.vpr	3.80 s	0.08 s	28,506	3.81 s	0.12 s	26,227
176.gcc	59.80 s	0.61 s	408,798	59.16 s	0.91 s	349,964
181.mcf	0.57 s	0.02 s	2,631	0.60 s	0.03 s	2,418
186.crafty	7.50 s	0.13 s	42,604	7.32 s	0.18 s	37,384
197.parser	5.54 s	0.06 s	19,900	5.55 s	0.09 s	18,344
253.perlbmk	25.10 s	0.29 s	143,039	24.79 s	0.41 s	129,337
254.gap	18.06 s	0.25 s	152,983	17.87 s	0.34 s	132,955
255.vortex	17.66 s	0.35 s	98,694	17.54 s	0.45 s	92,416
256.bzip2	1.03 s	0.01 s	6,623	1.02 s	0.02 s	5,665
300.twolf	7.24 s	0.18 s	60,445	7.20 s	0.27 s	55,346
Sum	147.67 s	2.01 s	974,743	146.18 s	2.86 s	859,311

has only 88.2% the number of nodes. This speeds up later optimizations resulting in an 1.49 s faster overall compilation.

6.3 Conclusion

The algorithm has been shown to be as fast as the Cytron et al.'s algorithm in practice. However, if the algorithm is combined with on-the-fly optimizations, the overall compilation time is reduced. This makes the algorithm an interesting candidate for just-in-time compilers.

7 Related Work

SSA form was invented by Rosen, Wegman, and Zadeck [15] and became popular after Cytron et al. [10] presented an efficient algorithm for constructing it. This algorithm can be found in all textbooks presenting SSA form and is used by the majority of compilers. For each variable, the iterated dominance frontiers of all blocks containing a definition is computed. Then, a rewrite phase creates new variable numbers, inserts ϕ functions and patches users. The details necessary for this paper were already discussed in Section 1.

Choi et al. [7] present an extension to the previous algorithm that constructs minimal and pruned SSA form. It computes liveness information for each variable v and inserts a ϕ function for v only if v is live at the corresponding basic block. This technique can also be applied to other SSA construction algorithms to ensure pruned SSA form, but comes along with the costs of computing liveness information.

Briggs et al. [6] present semi-pruned SSA form, which omits the costly liveness analysis. However, they only can prevent the creation of dead ϕ functions for variables that are local to a basic block.

Sreedhar and Gao [16] present a data structure, called DJ graph, that enhances the dominance tree with edges from the CFG. Compared to computing iterated dominance frontiers for each basic block, this data structure is only linear in the program size and allows to compute the blocks where ϕ functions need to be placed in linear time per variable. This gives an SSA construction algorithm with cubic worst-case complexity in the size of the source program.

There are also a range of construction algorithms, which aim for simplicity instead. Brandis and Mössenböck [5] present a simple SSA construction algorithm that directly works on the AST like our algorithm. However, their algorithm is restricted to structured control flow (no gotos) and does not construct pruned SSA form. Click and Paleczny [8] describe a graph-based SSA intermediate representation used in the Java HotSpot server compiler [14] and an algorithm to construct this IR from the AST. Their algorithm is in the spirit as the one of Brandis and Mössenböck and thus does construct neither pruned nor minimal SSA form. Aycock and Horspool present an SSA construction algorithm that is designed for simplicity [4]. They place a ϕ function for each variable at each basic block. Afterwards they employ the following rules to remove ϕ functions:

1. Remove ϕ functions of the form $v_i = \phi(v_i, \ldots, v_i)$.
2. Substitute ϕ functions of the form $v_i = \phi(v_{i_1}, \ldots, v_{i_n})$ with $i_1, \ldots, i_n \in \{i, j\}$ by v_j.

This results in minimal SSA form for reducible programs. The obvious drawback of this approach is the overhead of inserting ϕ functions at each basic block. This also includes basic blocks that are prior to every basic block that contains a real definition of the corresponding variable.

8 Conclusions

In this paper, we presented a novel, simple, and efficient algorithm for SSA construction. In comparison to existing algorithms it has several advantages: It does not require other analyses and transformations to produce minimal (on reducible CFGs) and pruned SSA form. It can be directly constructed from the source language without passing through a non-SSA CFG. It is well suited to perform several standard optimizations (constant folding, value numbering, etc.) already during SSA construction. This reduces the footprint of the constructed program, which is important in scenarios where compilation time is of the essence. After IR construction, a post pass ensures minimal SSA form for arbitrary control flow. Our algorithm is also useful for SSA reconstruction where, up to now, standard SSA construction algorithms where not directly applicable. Finally, we proved that our algorithm always constructs pruned and minimal SSA form.

In terms of performance, a non-optimized implementation of our algorithm is slightly faster than the highly-optimized implementation of Cytron et al.'s algorithm in the LLVM compiler, measured on the SPEC CINT2000 benchmark suite. We expect that after fine-tuning our implementation, we can improve the performance even more.

Acknowledgments. We thank the anonymous reviewers for their helpful comments. This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

References

1. libFirm – The FIRM intermediate representation library, http://libfirm.org
2. The LLVM compiler infrastructure project, http://llvm.org
3. Appel, A.W.: SSA is functional programming. SIGPLAN Notices 33(4), 17–20 (1998)
4. Aycock, J., Horspool, N.: Simple Generation of Static Single-Assignment Form. In: Watt, A. (ed.) CC 2000. LNCS, vol. 1781, pp. 110–125. Springer, Heidelberg (2000)
5. Brandis, M.M., Mössenböck, H.: Single-pass generation of static single-assignment form for structured languages. ACM Trans. Program. Lang. Syst. 16(6), 1684–1698 (1994)
6. Briggs, P., Cooper, K.D., Harvey, T.J., Simpson, L.T.: Practical improvements to the construction and destruction of static single assignment form. Softw. Pract. Exper. 28(8), 859–881 (1998)
7. Choi, J.D., Cytron, R., Ferrante, J.: Automatic construction of sparse data flow evaluation graphs. In: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1991, pp. 55–66. ACM, New York (1991)
8. Click, C., Paleczny, M.: A simple graph-based intermediate representation. In: Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations, IR 1995, pp. 35–49. ACM, New York (1995)
9. Cocke, J.: Programming languages and their compilers: Preliminary notes. Courant Institute of Mathematical Sciences, New York University (1969)
10. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. TOPLAS 13(4), 451–490 (1991)
11. Eswaran, K.P., Tarjan, R.E.: Augmentation problems. SIAM J. Comput. 5(4), 653–665 (1976)
12. Hecht, M.S., Ullman, J.D.: Characterizations of reducible flow graphs. J. ACM 21(3), 367–375 (1974)
13. Kelsey, R.A.: A correspondence between continuation passing style and static single assignment form. SIGPLAN Not. 30, 13–22 (1995)
14. Paleczny, M., Vick, C., Click, C.: The Java HotSpot$^{\mathrm{TM}}$server compiler. In: Symposium on Java$^{\mathrm{TM}}$ Virtual Machine Research and Technology Symposium, JVM 2001, pp. 1–12. USENIX Association, Berkeley (2001)
15. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global value numbers and redundant computations. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1988, pp. 12–27. ACM Press, New York (1988)
16. Sreedhar, V.C., Gao, G.R.: A linear time algorithm for placing ϕ-nodes. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1995, pp. 62–73. ACM, New York (1995)
17. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM Journal Computing 1(2), 146–160 (1972)

PolyGLoT: A Polyhedral Loop Transformation Framework for a Graphical Dataflow Language

Somashekaracharya G. Bhaskaracharya[1,2] and Uday Bondhugula[1]

[1] Department of Computer Science and Automation, Indian Institute of Science
[2] National Instruments, Bangalore, India

Abstract. Polyhedral techniques for program transformation are now used in several proprietary and open source compilers. However, most of the research on polyhedral compilation has focused on imperative languages such as C, where the computation is specified in terms of statements with zero or more nested loops and other control structures around them. Graphical dataflow languages, where there is no notion of statements or a schedule specifying their relative execution order, have so far not been studied using a powerful transformation or optimization approach. The execution semantics and referential transparency of dataflow languages impose a different set of challenges. In this paper, we attempt to bridge this gap by presenting techniques that can be used to extract polyhedral representation from dataflow programs and to synthesize them from their equivalent polyhedral representation.

We then describe PolyGLoT, a framework for automatic transformation of dataflow programs which we built using our techniques and other popular research tools such as Clan and Pluto. For the purpose of experimental evaluation, we used our tools to compile LabVIEW, one of the most widely used dataflow programming languages. Results show that dataflow programs transformed using our framework are able to outperform those compiled otherwise by up to a factor of seventeen, with a mean speed-up of 2.30$\times$ while running on an 8-core Intel system.

1 Introduction and Motivation

Many computationally intensive scientific and engineering applications that employ stencil computations, linear algebra operations, image processing kernels, etc. lend themselves to polyhedral compilation techniques [2, 3]. Such computations exhibit certain properties that can be exploited at compile time to perform parallelization and data locality optimization.

Typically, the first stage of a polyhedral optimization framework consists of polyhedral extraction. Specific regions of the program that can be represented using the polyhedral model, typically affine loop-nests, are analyzed. Such regions have been termed Static Control Parts (SCoPs) in the literature. Results of the analysis include an abstract mathematical representation of each statement in the SCoP, in terms of its iteration domain, schedule, and array accesses. Once dependences are analyzed, an automatic parallelization and locality optimization tool such as Pluto [16] is used to perform high-level optimizations. Finally, the transformed loop-nests are synthesized using a loop generation tool such as CLooG [4].

Regardless of whether an input program is written in an imperative language, a dataflow language, or using another paradigm, if a programmer does care about performance, it is important for the compiler not to ignore transformations that yield significant performance gains on modern architectures. These transformations include, for example, ones that enhance locality by optimizing for cache hierarchies and exploiting register reuse or those that lead to effective coarse-grained parallelization on multiple cores. It is thus highly desirable to have techniques and abstractions that could bring the benefit of such transformations to all programming paradigms.

There are many compilers, both proprietary and open-source which now use the polyhedral compiler framework [12, 15, 6, 16]. Research in this area, however, has predominantly focused on imperative languages such as C, C++, and Fortran. These tools rely on the fact that the code can be viewed as a sequence of statements executed one after the other. In contrast, a graphical dataflow program consists of an interconnected set of nodes that represent specific computations with data flowing along edges that connect the nodes, from one to another. There is no notion of a statement or a mutable storage allocation in such programs. Conceptually, the computation nodes can be viewed as consuming data flowing in to produce output data. Nodes become ready to be 'fired' as soon as data is available at all their inputs. The programs are thus inherently parallel. Furthermore, the transparency with respect to memory referencing allows such a program to write every output data value produced to a new memory location. Typically, however, copy avoidance strategies are employed to ensure that the output data is *inplace* to input data wherever possible. Such *inplaceness* decisions can in turn affect the execution schedule of the nodes.

The polyhedral extraction and code synthesis for dataflow programs, therefore, involves a different set of challenges to those for programs in an imperative language such as C. In this paper, we propose techniques that address these issues. Furthermore, to demonstrate their practical relevance, we describe an automatic loop transformation framework that we built for the LabVIEW graphical dataflow programming language, which uses all of these techniques. To summarize, our contributions are as follows:

- We provide a specification of parts of a dataflow program that lends itself to the abstract mathematical representation of the polyhedral model.
- We describe a general approach for extracting the polyhedral representation for such a dataflow program part and also for the inverse process of code synthesis.
- We present an experimental evaluation of our techniques for LabVIEW and comparison with the LabVIEW production compiler.

The rest of the paper is organized as follows. Section 2 provides the necessary background on LabVIEW, dataflow languages in general, the polyhedral model, and introduces notation used in the rest of the paper. Section 3 deals with extracting a polyhedral representation from a dataflow program, and Section 4 addresses the inverse process of code synthesis. Section 5 describes our PolyGLoT framework. Section 6 presents an experimental evaluation of our techniques. Related work and conclusions are presented in Section 7 and Section 8 respectively.

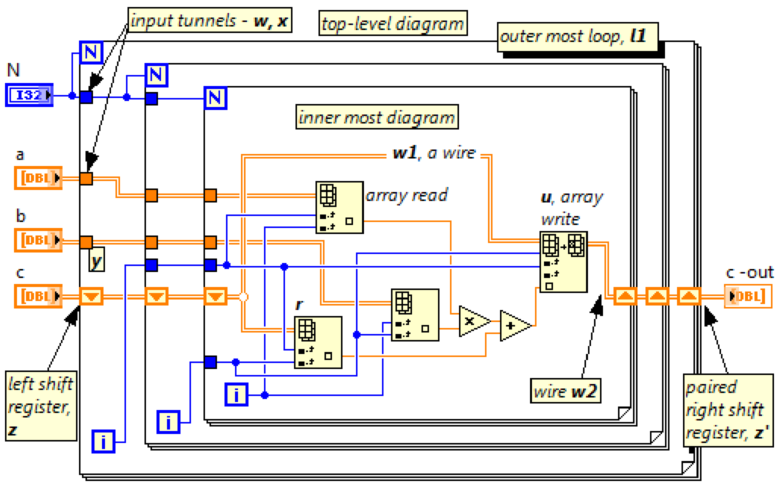


Fig. 1. *matmul* in LabVIEW. LabVIEW for-loops are unit-stride for-loops with zero-based indexing. A loop iterator node in the loop body (the [i] node) produces the index value in any iteration. A special node on the loop boundary (the N node) receives the upper loop bound value. The input arrays are provided by the nodes *a, b* and *c*. The output array is obtained at node *c-out*. The color of the wire indicates the type of data flowing along it e.g. blue for integers, orange for floats. Thicker lines are indicative of arrays.

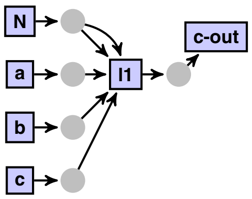


Fig. 2. DAG of the top-level diagram of *matmul*. In this abstract model, the gray nodes are wires. The 4 source nodes (N, a, b, c), the sink node (c-out) and the outermost loop are represented as the 6 blue nodes. Directed edges represent the connections from inputs/outputs of computation nodes to the wires, e.g. data from source node N flows over a wire into two inputs of the loop node. Hence the two directed edges from the corresponding wire node.

2 Background

2.1 LabVIEW – Language and Compiler

LabVIEW is a graphical, dataflow programming language from National Instruments Corporation (NI) that is used by scientists and engineers around the world. Typically, it is used for implementing control and measurement systems, and embedded applications. The language itself, due to its graphical nature, is referred to as the G language. A LabVIEW program called a Virtual Instrument (VI) consists of a front panel (the graphical user interface) and a block diagram, which is the graphical dataflow diagram. Instead of textual statements, the program consists of specific computation nodes. The flow of data is represented by a *wire* that links the specific output on a source node to the specific input on a sink node. The block diagram of a LabVIEW VI for matrix multiplication is shown in Figure 1.

Loop nodes act as special nodes that enclose the dataflow computation that is to be executed in a loop. Data that is only read inside the loop flows through a special node on the boundary of the loop structure called the *input tunnel*. A pair of boundary nodes called the *left* and *right shift registers* are used to represent loop-carried dependence. Data flowing into the right shift register in one iteration flows out of the left shift register in the subsequent iteration. The data produced as a result of the entire loop computation flows out of the right shift register. Additionally, some boundary nodes are also used for the loop control. In addition to being inherently parallel because of the dataflow programming paradigm, LabVIEW also has a parallel for loop construct that can be used to parallelize the iterative computation [7].

The LabVIEW compiler first translates the G program into a Data Flow Intermediate Representation (DFIR) [14]. It is a high-level, hierarchical and graph-based representation that closely corresponds to the G code. Likewise, we model the dataflow program as being conceptually organized in a hierarchy of diagrams. It is assumed that the diagrams are free of dead-code.

2.2 An Abstract Model of Dataflow Programs

Suppose N is the set of computation nodes and W is the set of wires in a particular diagram. Each diagram is associated with a directed acyclic graph (DAG), $G = (V, E)$, where $V = N \cup W$ and $E = E_N \cup E_W$. $E_N \subseteq N \times W$ and $E_W \subseteq W \times N$. Essentially, E_N is the set of edges that connect the output of the computation nodes to the wires that will carry the output data. Likewise, E_W is the set of edges that connect the input of computation nodes to the wires that propagate the input data. We follow the convention of using small letters v and w to denote computation nodes and wires respectively. Any edge (v, w) represents a particular output of node v and any edge (w, v) represents a particular input of node v. So, the edges correspond to memory locations. The wires serve as copy nodes, if necessary.

For every $n \in N$ that is a loop node, it is associated with a DAG, $G_n = (V_n, E_n)$ which corresponds to the dataflow graph describing the loop body. The loop inputs and outputs are represented as source and sink vertices. The former have no incoming edges, whereas the latter have no outgoing edges. Let I and O be the set of inputs and outputs. Furthermore, a loop output vertex may be paired with a loop input vertex to signify a loop-carried data dependence i.e., data produced at the loop output in one iteration flows out of the input for the next iteration (Fig 1).

Inplaceness. In accordance with the referential transparency of a dataflow program, each edge could correspond to a new memory location. Typically, however, a copy-avoidance strategy may be used to re-use memory locations. For example, consider the array element write node u in Figure 1, and its input and output wires, w_1 and w_2. The output array data flowing along w_2 could be stored in the same memory location as the input array data flowing along w_1. The output data can be *inplace* to the input data. The *can-inplace* relation $(w_1, u) \rightsquigarrow (u, w_2)$ is said to hold.

In general, for any two edges (x, y) and (y, z), $(x, y) \rightsquigarrow (y, z)$ holds iff the data inputs or outputs that these edges correspond to *can* share the same memory location (regardless of whether a specific copy-avoidance strategy chooses to re-use the memory location or not). The can-inplace relation is an equivalence relation. A path $\{x_1, x_2, \ldots, x_n\}$ in a graph $G = (V, E)$, such that $(x_{i-1}, x_i) \rightsquigarrow (x_i, x_{i+1})$ for all $2 \le i \le n-1$, is said to be a *can-inplace* path. Note that by definition, the can-inplace relation $(w_1, v) \rightsquigarrow (v, w_2)$ implies that the node v can overwrite the data flowing over w_1. And in such a case, we say that the relation $v\mathrm{X}w_1$ holds. However, the can-inplace relation $(v_1, w) \rightsquigarrow (w, v_2)$ does not necessarily imply such a destructive operation as the purpose of a wire is to propagate data, not to modify it.

Suppose $<_s$ is a binary relation on V which specifies a total ordering of the computation nodes. The relation $<_s$ specifies a *valid* execution schedule iff $(v_1 <_s v_2)$ implies that there does not exist a directed path in graph G, from v_2 to v_1 for any $v_1, v_2 \in V$ i.e., the schedule respects all dataflow dependences. As we shall see later, the problem of scheduling the computation nodes is closely related to inplaceness. Memory re-use due to copy-avoidance can create additional dependences. A conjunction of scheduling relations $\bigwedge(v_1 <_s v_2)$ is said to be *consistent* with a conjunction of can-inplace relations $\bigwedge((x, y) \rightsquigarrow (y, z))$, for $x, y, z \in N \cup V$, iff such a schedule does not violate the dependences imposed by such an inplaceness choice.

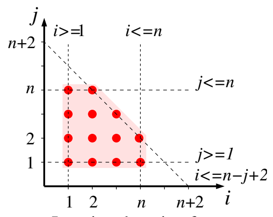
Loop Inputs and Outputs. Data flowing into and out of a loop is classified as either *loop-invariant input data* or *loop-carried data*. Loop-invariant input data is that which is only read in every iteration of the loop. Let *Inv* be the set of loop-invariant data inputs to the loop. The LabVIEW equivalent for such an input is an input tunnel. In Figure 1, for the outermost loop l_1, $Inv = \{w, x, y\}$. Loop-carried data is that which is part of a loop-carried dependence inducing dataflow. The paired loop inputs and outputs represent such a dependence. Let *ICar, OCar* be sets of these loop inputs and outputs. The loop-carried dependence is represented by the one-to-one mapping $lcd : OCar \rightarrow ICar$. The LabVIEW equivalent for such a pair are the left and right shift registers. In Figure 1, for loop l_1, $ICar = \{z\}$, $OCar = \{z'\}$, $(z, z') \in lcd$.

Array Accesses. In Figure 1, the array read access is a node that takes in an array and the access index values to produce the indexed array element value. The array write access, takes the same set of inputs and the value to be written to produce an array with the indexed element overwritten. We model the array read and write accesses similarly. Notice that the output array of an array write, v need not be inplace to the input array flowing through a wire w_1. If it is, then $v\mathrm{X}w_1$.

2.3 Overview of the Polyhedral Model

The polyhedral model provides an abstract mathematical model to reason about program transformations. Consider a program part that is a sequence of statements with zero or more loops surrounding each statement. The loops may be imperfectly nested. The dynamic instances of a statement S, are represented by the integer points of a polyhedron whose dimensions correspond to the enclosing loops. The set of dynamic instances of a statement is called its *iteration domain*, D. It is represented by the polyhedron, defined by a conjunction of affine inequalities that involve the enclosing loop

```
for (i=1;i<=n;i++){
  for (j=1;j<=n;j++){
    if (i<=n-j+2){
      S1;
    }
  }
}
```



Iteration domain of S1

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \geq \begin{pmatrix} 1 \\ -n \\ 1 \\ -n \\ -n-2 \end{pmatrix}$$

Iteration domain of S1

Fig. 3. Polyhedral representation of a loop-nest in geometric and linear algebraic form

```
for(i=1;i<=n−1;i++)
  for(j=i+1;j<=n;j++)
/*S1*/ c[i][j]=a[j][i]/a[i][i];
  for(j=i+1;j<=n;j++)
    for(k=i+1;k<=n;k++)
/*S2*/ a[j][k]−=c[i][j]*a[i][k];
```

(a) Original code

$\theta_{S1}(i,j) = (i,0,j,0)$

$\theta_{S2}(i,j,k) = (i,1,j,k)$

(b) Initial schedule

$\theta_{S_1}(i,j) = (i+j,j,0)$

$\theta_{S_2}(i,j,k) = (i+j,j,1,k)$

(c) New schedule

```
for(t1=3;t1<=2*n−1;t1++)
  for(t2=ceild(t1+1,2);t2<=min(n,t1−1);t2++)
    c[t1−t2][t2]=a[t2][t1−t2]/a[t1−t2][t1−t2];
    for(t3=t1−t2+1;t3<=n;t3++)
      a[t2][t3]−=c[t1−t2][t2]*a[t1−t2][t3];
```

(d) Transformed code

Fig. 4. An example showing the input code, the corresponding original schedule, a new schedule that fuses j loops of both statements while skewing the outermost loop with respect to the second outermost one, and code generated with the new schedule.

iterators and global parameters. Each dynamic instance is uniquely identified by its *iteration vector*, i.e., the vector $\boldsymbol{i}_S$ of enclosing loop iterator values. Figure 3 shows the polyhedral representation of a loop-nest in its geometric and linear algebraic form.

Schedules. Each statement, or more precisely its domain, has an associated schedule, which is a multi-dimensional affine function mapping each integer point in the statement's domain to a unique time point that determines when it is to be executed. Code generated from the polyhedral representation scans integer points corresponding to all statements globally in the lexicographic order of the time points they are mapped to. For example, $\theta_S(i,j,k) = (i+j,j,k)$ is a schedule for a 3-d loop nest with original loop indices i, j, k. Changing the schedule to $(i+j,k,j)$ would interchange the two inner loops. The reader is referred to [3] for more detail on the polyhedral representation.

The initial schedule which is extracted, corresponding to the original execution order, is referred to as an *identity schedule*, i.e., if it is not modified, code generation will lead to the same code as the one from which the representation was extracted. A dimension of the multi-dimensional affine scheduling function is called a *scalar dimension* if it is a constant. In Figure 4(b), the second dimension of both statements' schedules are scalar dimensions. In Figure 4(c) schedules the third dimension is a scalar one. Polyhedral optimizers have models to pick the right schedule among valid ones. A commonly used model that minimizes dependence distances in the transformed space [5], thereby optimizing locality and parallelism simultaneously is implemented in Pluto [16].

3 Extracting the Polyhedral Representation

The polyhedral representation of a SCoP typically consists of an abstract mathematical description of the iteration domain, schedule and array accesses for each statement.

The array accesses are also classified as either read or write accesses. Each of these are expressed as affine functions of the enclosing loop iterators and symbolic constants.

3.1 Challenges

As mentioned earlier, a graphical dataflow program has no notion of a statement. The program is a collection of nodes that represent specific computations, with data flowing along edges that connect one node to another. Referential transparency ensures that each edge could be associated with its own distinct memory location. Generally, copy-avoidance strategies are used to maximize inplaceness of output and input data. However, the exact memory allocation depends on the specific strategy used. Additionally, the problem of copy-avoidance is closely tied with the problem of scheduling the computation nodes. In the matmul program (Figure 1), consider the array write u and the array read r that share the same data source (say, v). If no array copy is to be created, the read must be scheduled ahead of the write, i.e., $u <_s r$ is not *consistent* with $(v, w_1) \rightsquigarrow (w_1, u) \wedge (w_1, u) \rightsquigarrow (u, w_2) \wedge (v, w_1) \rightsquigarrow (w_1, r)$. If the write is scheduled first, the read must work on a copy of the array as the write is likely to overwrite the array input. Abu-Mahmeed et al. [1] have looked into the problem of scheduling to maximize the inplaceness of aggregate data. To summarize, the main challenges in the extraction of the polyhedral representation for a graphical dataflow program are as follows:

1. A graphical dataflow program cannot be viewed as a sequence of statements executed one after the other.
2. While the access expressions could be analyzed just like parse trees, it is difficult to relate the access to a particular array definition as the exact memory allocation depends on the specific copy-avoidance strategy used.
3. The actual execution schedule of the computation nodes determined depends on the copy-avoidance decisions.

A trivial polyhedral representation can be extracted by treating each node of a graphical data flow program as a statement analogue while making the conservative assumption that data is copied over each edge. As most compilers make use of copy-avoidance strategies, such a polyhedral representation most certainly over-estimates the amount of data space required. This also results in an over-estimation of the computation e.g. an array copy. Therefore, the problem of polyhedral transformation in such a representation begins with a serious limitation in terms of dataspace and computation over-estimation. In essence, extraction of a polyhedral representation of a dataflow program part cannot negate the copy-avoidance optimizations. The inplaceness opportunities in the dataflow program must be factored into the analysis.

3.2 Static Control Dataflow Diagram (SCoD)

A SCoP is defined as a maximal set of consecutive statements without while loops, where loop bounds and conditionals may only depend on invariants within this set of statements. Analogous to this, we now characterize a canonical graphical dataflow program, a Static Control Dataflow Diagram (SCoD), which lends itself well to existing polyhedral techniques for program transformation. The reasoning behind each individual characteristic is provided later.

1. It is a maximal dataflow diagram without constructs for loops that are not countable, where the countable loop bounds and conditionals, in any diagram, only depend on parameters that are invariant for that diagram. Nodes in the SCoD (and its nested diagrams) must be functional, without causing run-time side-effects or relying on any run-time state.
2. The only array primitives that feature as nodes in a SCoD and its nested diagrams are those which read an array element or write to an array element. More importantly, primitives that output array data that cannot be inplace to an input array data cannot be present in the diagrams.
3. For an array data source in any diagram, (v_1, w), there exists at most one node v_2 such that $(v_1, w) \rightsquigarrow (w, v_2) \wedge v_2 X w$.
4. Data flowing into a loop in any diagram is either loop-invariant data or loop-carried scalar data or loop-carried array data that has an associated can-inplace path through the loop body, which creates the loop-carried dependence, i.e., loop input $x \in I \Rightarrow x \in Inv \vee (x \in ICar \wedge (isScalarType(x, w_x) \vee (isArrayType(x, w_x) \wedge (x, w_x) \rightsquigarrow (w_y, y))))$ where $y = lcd^{-1}(x)$, w_x and w_y the input and output wires.
5. In any diagram, there is no can-inplace path from a loop-invariant data input to the loop-carried data input of a inner loop or to the array input of an array element write node, i.e., in any DAG, $G = (N, E)$ that corresponds to the body of a loop, if (v_1, w_1) is the loop invariant input, then there does not exist any edge (w_2, v_2) such that $(v_1, w_1) \rightsquigarrow (w_2, v_2) \wedge v_2 \mathrm{X} w_2$.

The first characteristic is closely tied with the characterization of a SCoP. The rest of the characterization specifies a canonical form of dataflow diagram which has can-inplace relations that facilitate polyhedral extraction. As explained earlier, a naive implementation of a dataflow language could write each new output into a new memory location. The question of whether a particular wire vertex gets a new memory allocation or not depends on the actual copy-avoidance strategy employed by the compiler. The problem of extracting the polyhedral representation of an arbitrary dataflow diagram, therefore depends on the copy-avoidance strategy. In order to make the polyhedral extraction independent of it, we canonicalize the dataflow in a given diagram in accordance with the above characteristics.

An operation such as appending an element to an input array data is a perfectly valid dataflow operation. Clearly, the output array cannot be inplace to the input array. (2) ensures that such array operations are disallowed. Furthermore, it is possible in a dataflow program to overwrite multiple, distinct copies of the same array data. In such a case, a copy-avoidance strategy would inplace only one of the copies with the original data and the rest of them would be separate copies of data. (3) precludes such a scenario. It is important to note that it however, still allows multiple writes. (4) ensures that loop-carried dependence involving array data is tied to a single array data source. Assuming the absence of (5), data flowing from loop-invariant source vertex to a loop-carried input of an inner loop would necessitate a copy because the source data would have a pending read in subsequent iterations of the outer loop.

Theorem 1. *In any diagram of the SCoD, G = (V, E), there exists a schedule $<_s$ of the computation nodes V, which is consistent with the conjunction of all possible can-inplace relations, $\bigwedge_{x,\,y,\,z\,\in\,V}$ ((x, y) $\rightsquigarrow$ (y, z)), where isArrayType(x, y) $\wedge$ isArrayType(y, z) holds.*

Proof. Consider an array data source *(v, w)* in G with $v \in I$. If $v_1, v_2, \ldots, v_n$ are the nodes that consume the array data, then in accordance with characteristic (3), there is at most one node v_i such that $(v, w) \rightsquigarrow (w, v_i) \wedge v_i X w$. Without any loss of generality, we can assume that $i = n$. This implies that any valid schedule $<_s$ where $v_1 <_s v_2 <_s v_3 \ldots <_s v_n$ holds is consistent with $\bigwedge_{i=1}^{n} ((v, w) \rightsquigarrow (w, v_i))$. Similar scheduling constraints can be inferred for the nodes that consume the array data produced by v_n and so on, thereby ensuring that all the can-inplace relations are satisfied for array dataflow. The new constraints inferred cannot contradict an existing constraint as the graph is acyclic. Therefore, any valid schedule $<_s$ where all the inferred scheduling constraints are satisfied is consistent with maximum array inplaceness in the diagram. □

Essentially, in a SCoD, it is possible to schedule the computation nodes such that no new memory allocation need be performed for any array data inside the SCoD, i.e., all the array data consumed inside the SCoD will then have an inplace source that ultimately lies outside the SCoD.

Lemma 1. *In any diagram of the SCoD, G = (V, E), for any sink vertex $t \in O$, that has array data flowing into it, a can-inplace path exists from a source vertex $s \in I$ to t.*

Proof. There must exist a node v_1 which produces the array data flowing into *t* through wire *w*. So, $(v_1, w) \rightsquigarrow (w, t)$ holds. In accordance with the model, v_1 can either be an array write node or a loop. In either case, there must exist a node v_2 which produces the array data flowing into v_1 and so on until a source vertex *s* is encountered. The path traversed backwards from *t* to *s* clearly constitutes a can-inplace path. □

3.3 A Multi-dimensional Schedule of Compute-Dags

A *compute-dag*, $T = (V_T, E_T)$ in a diagram $G = (V, E)$, is a sub-graph of *G* where there exists a node, $r \in V_T$ such that for every other $x \in V_T$ there exists a path from *x* to *r* in *T* (the node *r* will hereafter be referred to as the root node). As it is possible to pick inplace opportunities such that no array data need be copied on any edge in the SCoD, any diagram in the SCoD can be viewed as a sequence of computations that write on the incoming array data. Instead of statements, compute-dags, which are essentially dags of computation nodes can be identified. Consider an array write node or a loop node, both of which can overwrite an input array. Starting with a dag that is just this node as the root, the compute-dag can be built recursively by adding nodes which produce data that flows into any of the nodes in the dag. Such a recursive sweep of the graph stops on encountering another array write or loop node. However, while identifying compute-dags in a diagram, it is necessary to account for all the data produced by the nodes in the diagram.

Theorem 2. *In any diagram of the SCoD, $G = (V, E)$, for every edge $(x, y) \in E$ where x is a computation node, there exists a compute-dag $T_i = (V_i, E_i)$ in the set $\Sigma' = \{T_1, T_2, \ldots, T_m\}$ of compute-dags rooted at array write or loop nodes, such that $x \in V_i$ iff only array data flows out of every diagram.*

Proof. In accordance with Lemma 1, for any sink vertex $t \in O$, with array data flowing into it, there exists a can-inplace path $p_{s \to t} = \{s, \ldots, v, w, t\}$ from a source vertex s to t. Also, every node x, which is not dead-code, must have a path $q_{x \to t_j} = \{x, y, \ldots, v_j, w_j, t_j\}$ to at least one sink vertex $t_j \in O$. If only array data flows into every sink vertex, consider the first vertex z at which the path $q_{x \to t_j}$ overlaps with $p_{s_j \to t_j}$. z must either be a loop-node or an array write node. In either case x must be part of a compute-dag rooted at z (in the former case, if $x \neq z$, notice also that the data flowing along (x, y) must be the intermediate result of a computation that produces the loop-invariant data for the loop z). On the other hand, if scalar data can flow into a sink vertex t, clearly, the path from node x to t is not guaranteed to have either an array write node or a loop node. Consequently, the node x is not guaranteed to be part of any compute-dag. Likewise, if loops can have scalar loop-carried data since a scalar loop-carried output is represented as a sink vertex in the loop body DAG. □

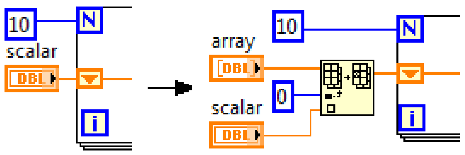
In order to address the consequences of Theorem 2, it is necessary to treat scalar data flowing out of the diagram as a single-element array. This results in a compute-dag that accounts for the scalar dataflow. Likewise, loop-carried scalar data must also be treated as single-element array. The dataflow into the loop-carried input is treated as a write to the array resulting in a corresponding compute-dag (refer Fig 5(a)). (Hereafter, we assume in the following discussion, that scalar data flowing out of a diagram and loop-carried scalar data are treated specially in this way as single-element arrays).

Each diagram in a SCoD is analyzed for compute-dags, starting from the top-level diagram. Suppose θ is the scheduling function. At each diagram level, d, the set of roots of the compute-dags in $\Sigma' = \{T_1, T_2, \ldots, T_m\}$ are ordered as follows:

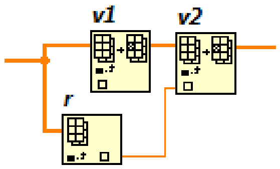
- If data produced by a root node n_1 is consumed by root node n_2, then $\theta^d_{n_1} \prec \theta^d_{n_2}$.
- In accordance with Theorem 1, if there is an array write in a compute-dag rooted at n_1 and an array read in a compute-dag rooted at n_2, both of which are dependent on the same array data source, then $\theta^d_{n_1} \succ \theta^d_{n_2}$. Scheduling n_1 ahead of n_2 in the polyhedral representation would be unsafe. Such a schedule would only be possible if n_2 were to read a copy of the input array, allowing n_1 to overwrite the input array. The safe schedule ensures that an array copy is not required.
- If neither of the above hold for the two root nodes, either $\theta^d_{n_1} \succ \theta^d_{n_2}$ or $\theta^d_{n_1} \prec \theta^d_{n_2}$ should hold true.

Each diagram in the diagram hierarchy of the SCoD contributes to a dimension in the global schedule. Each loop encountered adds an additional dimension. The total order on the compute-dag roots in any diagram determines the time value at which each compute-dag can be scheduled in that dimension. The global schedule is obtained by appending its time value in the owning diagram to the schedule of the owning loop, if any, together with the loop dimension.

Apart from ensuring that all the data produced by the nodes in a diagram are accounted for, it must also be possible to schedule the compute-dag roots in a total order.



(a) Treating scalar loop-carried data as single-element array



(b) Scheduling compute-dags rooted at v_1 and v_2 in total order is not possible

Fig. 5. Single-element arrays and contradiction in schedule of compute-dags

Theorem 3. *In any diagram of the SCoD, G = (V, E), it is possible to schedule the set of roots of the compute-dags in $\Sigma' = \{T_1, T_2, \ldots, T_m\}$ in a total order if for every path $p_{v_1 \to v_2}$ between a pair of roots, v_1 and v_2, there does not exist an array read node, r in the compute-dag T(v_2), such that r and v_1 share the same data source (x, w) with $v_1 X w$ and a path $q_{r \to v_2}$ exists that does not include v_1 on it.*

Proof. Consider a pair of root nodes v_1 and v_2 (refer Fig 5(b)). In accordance with the scheduling constraints specified above, $\theta^d_{v_1} \prec \theta^d_{v_2}$ if a path $p_{v_1 \to v_2}$ exists in G. This scheduling order is contradicted only if for some reason $\theta^d_{v_1} \succ \theta^d_{v_2}$ must hold, which can only happen if the compute-dag *T(v_2)* contains a node that must be schedule ahead of v_1, i.e., an array read node that shares the same source as v_1. If such a node does not exist, then the contradiction never arises leading to a total ordering of the compute-dag roots. Similarly, there is no contradiction in schedule order if a path $p_{v_1 \to v_2}$ does not exist. □

In order to address the consequence of Theorem 3, while building compute-dag rooted at v_2, it is also necessary to stop on encountering an array read node *r* when there is a node v_1 with the same array source, overwriting the incoming array, such that there exists a path from *r* to v_2 which does not include v_1. A separate compute-dag rooted at such an array read must be identified, thereby breaking the compute-dag that would have been identified otherwise (rooted at v_2) into two different dags.

The set of actual statement analogues is $\Sigma = \{T_1, T_2, \ldots, T_n\}$ such that the *root(T_i)* for any $T_i \in \Sigma'$ is not a loop. Algorithm 1 provides a procedure for identifying the set of statement analogues in a given dataflow graph G = (V, E) of a particular diagram. It is possible for two statement analogues to have common sub-expressions. However, the nodes in a SCoD are functional, making the common sub-expressions also so.

Analysis of Iteration Domains. We assume that loop normalization has been done, i.e., all for-loops have a unit stride and a lower bound of zero. Analyzing the iteration domain of a for loop only involves the analysis of the dataflow computation tree that computes the upper bound of the for loop. This analysis is very similar to parsing an expression tree. Symbolic constants are identified as scalar data sources that lie outside the SCoD. Loop iterators and constant data sources are explicitly represented as nodes in our model.

Analysis of Array Accesses. The access expression trees for the array reads and array writes which are present in the compute-dags of the statement analogues are analyzed to

Algorithm 1. *identify-compute-dags(G = (V, E))*

```
Require: Treat scalar data flowing out of diagram as single element arrays
Require: Loop-invariant computations have not been code-motioned out into an
   enclosing diagram
1: procedure IDENTIFY-COMPUTE-DAGS($G = (V, E)$)
2:    $\Sigma$ = $\varnothing$
3:    for all $n \in V$ | isArrayWriter(n) $\wedge$ !isloop(n) do
4:       $\Sigma$ =$\Sigma$ $\cup$ build-compute-dag(n, G)      ▷ compute-dag from G, with root n
5:    for all $n \in V$ | root-candidate[n] do
6:       $\Sigma$ = $\Sigma$ $\cup$ build-compute-dag(n, G)
7:    return $\Sigma$

8: procedure BUILD-COMPUTE-DAG($n$, $G = (V, E)$)
9:    $V_T = \{n\}, E_T = \varnothing$
10:   while (x, y) = get-new-node-for-dag(n, T = ($V_T$, $E_T$), G) do
11:      $V_T = V_T \cup x, E_T = E_T \cup \{(x, y)\}$
12:   return $T = (V_T, E_T)$

13: procedure GET-NEW-NODE-FOR-DAG($n$, $T = (V_T, E_T)$, $G = (V, E)$)
14:   for each $(x, y) \in E$ do
15:      if (x, y) $\notin E_T \wedge y \in V_T \wedge$ !isArrayWriter(x) $\wedge$ !isloop(x) then
16:         if isArrayReader(x) then
17:            z = get-array-write-off-same-source-if-any(x)
18:            if there exists a path $p_{z \to n}$ then
19:               root-candidate[x] = true
20:               continue
21:         return (x, y)
22:   return $\varnothing$
```

obtain the access functions. The most important problem of tying the array access to a particular memory allocation is resolved easily. Due to a carefully determined scheduling order, which schedules array reads ahead of an array write having the same source, all the accesses can be uniquely associated with array data sources that lie outside the SCoD. This is regardless of the actual copy-avoidance strategy that may be used. Additionally, the scalar data produced by an array read that is the root of its own compute-dag and a node in another compute-dag is treated as a single-element array, thereby encoding the corresponding dependence in the array accesses of both the compute-dags. So, each statement analogue has exactly one write access.

4 Code Synthesis

A polyhedral optimizer can be used to perform the required program transformations on the polyhedral representation of the SCoD. We now consider the problem of synthesizing a SCoD given its equivalent polyhedral representation.

4.1 Input

The input polyhedral representation must capture the iteration domain, access and scheduling information of the statement analogues i.e., the set $\Sigma = \{T_1, T_2, \ldots, T_n\}$ of compute-dags, which are also available as input. Each compute-dag, derived perhaps from an earlier polyhedral extraction phase, has exactly one array write node, which is the root of the dag.

Algorithm 2. *Synthesize-SCoD()*

```
1:  Convention: If $s$ represents a source vertex, the paired sink is $s'$
2:  procedure SYNTHESIZE-SCOD( )
3:      Let $G_0$ be the DAG of the top level diagram, $G_0 = (\varnothing, \varnothing)$
4:      create-source-vertex-for-each-global-parameter($G_0$)
5:      for each statement analogue, $T$ in global schedule order do
6:          Read domain (D), identity schedule ($\theta$) and access (A) matrices
7:          l = create-or-get-loop-nest($G_0$, D, $\theta$)            ▷ l, innermost loop
8:          add-compute-dag(l, T, $G_0$)
9:          for each (variable, read access) pair (v, a) in A do
10:             $(s_0, s'_0)$ = create-source-and-sink-vertices-if-none(v, $G_0$)
11:             create-or-get-dataflow($s_0$, $s'_0$, l, $G_0$, READ)
12:             array-read-node-access(a, T, l, $G_0$)   ▷ node reads data flowing into
        l through loop-invariant input or data flowing into the loop carried output if
        it exists. Create array index expression tree using a
13:         (v, a) = get-variable-write-access-pair(A)
14:         ($s_0$, $s'_0$) = create-source-and-sink-vertices-if-none(v, $G_0$)
15:         create-or-get-dataflow($s_0$, $s'_0$, l, $G_0$, WRITE)
16:         insert-array-write-node(a, T, l, $G_0$)           ▷ node is added to flow
        path so that it overwrites the data flowing into the loop-carried output of l.
        Create array index expression tree
17:         create-dataflow-from-parameters-and-iterators(c, $G_0$)
18:     return $G_0$
```

The polyhedral representation must have identity schedules. Any polyhedral representation with non-identity schedules can be converted to one with identity schedules by performing code generation and extracting the generated code again into the polyhedral representation. In this manner, scheduling information gets into statement domains and the schedule extracted from the generated code is an identity one. Once an equivalent polyhedral representation in this form has been obtained, the approach described in the rest of this section is used to synthesize a SCoD.

4.2 Synthesizing a Dataflow Diagram

The pseudocode for synthesizing a dataflow diagram is presented in Algorithms 2 and 3. The statement analogues are processed in their global schedule order (line 2.5). The iteration domain and scheduling information of a statement analogue are together used to create the surrounding loop-nest (line 2.7). Lower and upper bounds are inferred for each loop iterator. In case the for-loop is a normalized for-loop as in our abstract model, the actual upper bound will be a difference of the minimum and maximum of the inferred upper and lower bounds plus one. Built-in primitives for various operations such as max, min, floor, ceil etc. may be used to set up the loop-control. Note that if the required loop-nest has been created already for a statement analogue scheduled earlier, it need not be created again. The compute-dag is then added to the dataflow graph of the enclosing loop (line 2.8).

Inherent Parallelism – the Factor to Consider. Dataflow programs are inherently parallel. A computation node is ready to be fired for execution as soon as all its inputs are available. It is essential to exploit this inherent parallelism during code synthesis. In order to infer such parallelism and exploit it, we reason in terms of *coalesced dependences*. A coalesced dependence is the same as a regular data dependence except that two accesses are considered to be in conflict if they even access the same variable (potentially an aggregate data type), as opposed to the same location in the aggregate data.

For example, an array access that writes to odd locations does not conflict with another that reads from even locations. However, a coalesced dependence exists between the two. Analogous to regular data dependences, we now also use the terms flow, anti, and output coalesced dependences.

A unique source-sink vertex pair (s_0, s'_0) is created in the top-level DAG, G_0, of the top-level diagram for each variable v whose access is described in the access matrices (lines 2.10, 2.14). A dataflow path is also created from s_0 to s'_0. The problem of synthesizing a dataflow diagram is essentially a problem of synthesizing the dependences between the given set $\Sigma = \{T_1, T_2, \ldots, T_n\}$ of compute-dags in terms of edges that will connect them together. Specifically, as all the dependences involve array variables (may be single-element), these interconnecting edges represent the dataflow between array read or write nodes in the compute-dags, through intervening loops. Consider set of array write nodes $U = \{u_1, u_2, \ldots, u_n\}$, which correspond to write accesses on the same variables in a particular time dimension such that u_i is scheduled ahead of u_j for all $i < j$ (i.e., the corresponding compute-dags).

Theorem 4. *All coalesced output dependences on a variable in the polyhedral representation are satisfied by a synthesized dataflow diagram if in any diagram, all array write nodes $u_1, u_2, \ldots, u_n$ corresponding to write accesses to that variable lie on the same can-inplace path $p_{u_1 \to u_n}$.*

Proof. Suppose all the nodes in $U = \{u_1, u_2, \ldots, u_n\}$ are scheduled in the outermost diagram. A coalesced output dependence exists between any pair of write nodes scheduled in this diagram, thereby defining a total ordering on the set U. Therefore, all the corresponding array write nodes must be inserted along the can-inplace path $p_{s_0 \to s'_0}$. Now consider a write node u scheduled in an inner loop. A coalesced output dependence exists between u and any array write node $u_i \in U$. This is ensured by inserting the inner loop along the path $p_{s_0 \to s'_0}$, in accordance with its schedule order relative to the other writes nodes on the path. The incoming and outgoing edges of the loop node on the can-inplace path must correspond to the loop-carried input and its paired output, which in turn serve as the source and sink vertices in the DAG of the loop body. □

Theorem 5. *A coalesced flow dependence in the polyhedral representation is satisfied by a synthesized dataflow diagram if the array write node and read node associated with the dependence lie on the same can-inplace path.*

Proof. Each of the array write nodes $u_1, u_2, \ldots, u_m$ lies on the can-inplace path $\mathrm{p}_{u_1 \to u_m}$ due to Theorem 4. A coalesced flow dependence exists between the the write access u_m and read access r. Therefore, there must be a path $\mathrm{p}_{u_m \to r}$, which means that all of these nodes must lie on the same can-inplace path $\mathrm{p}_{u_1 \to r}$. If a read access r is the only access to a variable inside an inner loop l, the coalesced flow dependence between r and any u_i scheduled earlier is satisfied by a can-inplace path $p_{u_1 \to l}$. The incoming edge to l on this path need only correspond to a loop-invariant input. It acts as a data source for r in the loop body. □

Together, from Theorem 4 and Theorem 5, it can be seen that the path $p_{u_1 \to r}$ diverges from the path $\mathrm{p}_{u_1 \to u_n}$ at u_m i.e., the last write scheduled ahead of r. This enables the concurrent execution of the array write node u_{m+1} and r, thereby exposing the inherent parallelism in a dataflow diagram discussed earlier. There is no coalesced output or coalesced flow dependence between u_{m+1} and r. Also, just as the output array of an array write node can be inplace to the input array, loop-carried array outputs of a loop node can be inplace to the corresponding input. Similarly, a loop-invariant array input corresponds to the array input of a read node, as they do not have a corresponding output that can be inplace. Due to this symmetric relationship, based on coalesced dependences, we can infer inherent parallelism in the following scenarios:

– Consider two compute-dags, T_1 and T_2, scheduled in the same time dimension, d, such that $\theta^d_{T_1} \prec \theta^d_{T_2}$ with no coalesced output or coalesced flow dependence between them e.g. the two compute-dags have array accesses on disjoint sets of arrays. T_1 and T_2 then constitute two tasks that can executed in parallel in a dataflow program.
– Consider two loops, l_x and l_y, scheduled in the same time dimension such that there is no coalesced output or coalesced flow dependence between compute-dags in one loop and those of the other e.g. compute-dags in l_x only read a particular array variable, where those in l_y only write to it. The two loops can be executed as parallel tasks. This can be particularly crucial in obtaining good performance.
– Similarly, a loop and a compute-dag scheduled in the same time dimension with no coalesced output or coalesced flow dependence between the compute-dag and those in the loop.

Note that coalesced anti-dependences do not inhibit parallelism. The read and write access on the same variable may share the same data source. The read access can be performed on a copy of the data, while the write access is performed on the source data.

A dataflow diagram synthesized as described in the proofs for Theorem 4 and 5 is indeed a SCoD. The characteristics (1) and (2) are trivially satisfied. The dataflow diagram also meets characteristic (3) as all the array write nodes are serialized in accordance with Theorem 4. Furthermore, the construction described in the proof for Theorem 4 also ensures that whenever a loop-carried input-output pair is created, the corresponding source and sink vertices have an associated can-inplace path, thereby ensuring characteristic (4). Finally, the proof for Theorem 5 also implies a loop-carried input for a particular variable access is created on a loop only when all the accesses to a variable inside the loop-nest are read accesses. Therefore, a flow path from a loop-invariant source vertex to a loop-carried input never exists, ensuring characteristic (6).

Algorithm 2 processes the read accesses of a statement analogue first and then the write access. Algorithm 3, briefly explained below, describes the creation of the array dataflow paths for the corresponding read and write nodes in the compute-dag.

Read Accesses: Suppose the array read node is scheduled to execute in loop l_m. The closest enclosing loop l_c that has an array write node (for a write access on the same variable) in its body, and therefore, an associated loop-carried input s_c is found (line 3.4). A dataflow through loop-invariant inputs is then created to propagate the data flowing into the loop-carried output s'_c (line 3.12) to the inner loop l_m (line 3.14). This is the data produced by the write node associated with the last write access on the variable. However, if part of such a flow through loop-invariant inputs already exists for an intervening loop, it is extended to reach l_m (line 3.13).

Algorithm 3. *Creation of loop-carried and loop-invariant dataflow*

```
1: procedure CREATE-OR-GET-DATAFLOW($s_0, s_0', l_m, G_0, access-type$)
2:    $\{l_1, \dots, l_m\}$ = get-enclosing-loops($l_m$)          ▷ $\{G_1, \dots, G_m\}$ be their DAGs
3:    sources = get-inflow-if-any($s_0, l_m, G_0$)
4:    $c$ = max $i \mid i \in \{0, 1, \dots, |sources|\}$ and $s_i \in ICar[l_i]$ for $i > 0$
5:    if access-type == WRITE then
6:       if $c < m$ then
7:          find $v \mid (v, w), (w, s_c') \subset E_{|c|}$
8:          create flow path from $v$ to $s_c'$ through loop $l_m$ via loop-carried
   inputs/outputs (transforming $s_{c+1}, \dots, s_{|sources|}$ into loop-carried inputs)
9:          replace flow path $(v, w, s_c')$ with this new flow path
10:    else if $|sources| < m$ then                       ▷ must be a read access
11:       if $c$ == $|sources|$ then             ▷ use data overwritten in outer loop
12:          find $v \mid (v, w), (w, s_c') \subset E_{|c|}$
13:       else $v = s_{|sources|}$ ▷ extend loop-invariant flow
14:       create a flow path through loop-invariant inputs from $v$ to $l_m$
15:    return

16: procedure GET-INFLOW-IF-ANY($s_0, l_m, G_0$)
17:    $\{l_1, \dots, l_m\}$ = get-enclosing-loops($l_m$)                    ▷ $l_1$ outermost
18:    $s = s_0$, $H = G$, $U = V$, $F = E$, sources = $\varnothing$
19:    for $i \leftarrow 1, m$ do
20:       $w_i$ = wire carrying data from $s$
21:       if $\exists\ w \in U \mid (w, l_i) \in F \wedge (s, w_i) \rightsquigarrow (w, l_i)$ then
22:          $H$ = DAG that describes body of loop $l_i$
23:          $s$ = source vertex in $H$ that corresponds to loop input $(w, l_i)$
24:          sources = append $s$ to the sources list
25:       else break
26:    return sources
```

Write Accesses: Suppose the array write node is scheduled in a loop l_m. As in the case of a read access, the loop l_c is found (line 3.4). Any flow of data through loop-invariant inputs of intervening loops, from the source v of the loop-carried output s_c' is transformed to a flow of data through loop-carried inputs to the inner loop l_m. The newly created data flow through loop-carried inputs and outputs replaces the existing flow path *(v, w, s_c')* (line 3.7-line 3.9).

Once the dataflow from the variable source vertex is created to the loop enclosing the access node, it can read or write the data flowing in. The required access computation trees are created using the access information (usually represented by a matrix). The data outputs from these trees serve as the index inputs to access node.

Loop Iterators and Global Parameters: Besides the variable accesses, considered so far, there might still be other nodes whose input dataflow is yet to be created. The sources of these node inputs are either the loop iterators or global parameters for the SCoD e.g. consider the compute-dag that corresponds to `(b[i] = a[i] + i)`, the `i` input to the add node in the compute-dag still needs an input dataflow. Two mappings, `paramSource` and `iteratorSource`, from the set of node inputs to the sets of global parameters and loop iterators, can be used to create the input dataflow from the corresponding source vertices. In an actual implementation, these mappings have to be derived from the earlier phases of polyhedral extraction and optimization.

5 The PolyGLoT Auto-transformation Framework

We employed the techniques described so far to build PolyGLoT, a polyhedral automatic transformation framework for LabVIEW. The LabVIEW compiler translates the source dataflow diagram into a hierarchical, graph-based dataflow intermediate representation (DFIR). Several standard compiler optimizations are performed on this intermediate representation. We implemented a separate pass that uses PolyGLoT to perform polyhedral extraction, auto-transformation and dataflow diagram synthesis in that order. The optimized DFIR graph is then translated to the LLVM IR, which is then further optimized by the LLVM backend compiler that finally emits the machine code.

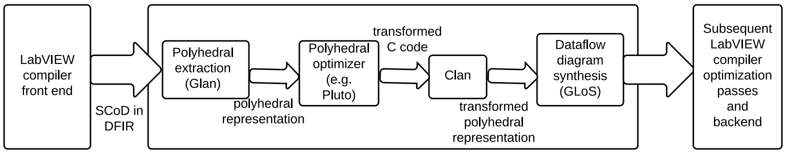

LabVIEW compiler front end
SCoD in DFIR
Polyhedral extraction (Glan)
polyhedral representation
Polyhedral optimizer (e.g. Pluto)
transformed C code
Clan
transformed polyhedral representation
Dataflow diagram synthesis (GLoS)
Subsequent LabVIEW compiler optimization passes and backend

Fig. 6. A high-level overview of PolyGLoT

PolyGLoT consists of four stages. The first stage extracts the polyhedral representation from a user-specified SCoD using the techniques described in Section 3. The translation is performed on DFIR. Glan (named after its C counterpart), a G loop analysis tool, was implemented to serve this purpose. The polyhedral representation extracted is used as an input to Pluto, an automatic parallelizer and locality optimizer. Pluto then applies a sequence of program transformations that include loop interchange, skewing, tiling, fusion, and distribution. Pluto internally calls into CLooG to output the transformed program as C code. Glan was used to produce a representative text (encoding a compute-dag id and also text describing the array accesses) for each compute-dag. Thus, we ensured that the transformed C code produced by Pluto included statements that could be matched with the computed-dags identified during extraction.

Clan was used to extract the polyhedral representation of the transformed C code, which was finally used as the input for GLoS (G loop synthesis). GLoS is a tool that synthesizes DFIR from the input polyhedral representation as per techniques developed in Section 4. Pluto was also used to produce scheduling information of loops that it would parallelize using OpenMP. This information was used by GLoS to parallelize the corresponding loops in the synthesized DFIR using the LabVIEW parallel for loop feature.

6 Experimental Evaluation

For the purpose of experimental evaluation, we implemented many of the benchmarks in the publicly available Polybench/C 3.2 [17] suite in LabVIEW. The matmul and ssymm benchmarks from the example test suite in Pluto were also used. Each of these benchmarks were then compiled using five different configurations.

– *lv-noparallel* is the configuration that simply uses the LabVIEW production compiler. This is the baseline for configurations that do not parallelize loops.

Table 1. Summary of performance (sequential and parallel execution on an 8-core machine)

Benchmark	Problem size	Execution time (seq)		Speedup	Execution time (8 cores)			Speedup over *lv-par*	
		lv-nopar	*pg-loc*	(local)	*lv-par*	*pg-par*	*pg-loc-par*	*pg-par*	*pg-loc-par*
atax	NX=4096, NY=4096	0.456s	0.567s	0.80	0.707s	0.642s	0.167s	1.10	4.23
bicg	NX=4096, NY=4096	0.409s	0.689s	0.59	0.409s	0.220s	0.093s	1.86	4.40
doitgen	NQ=NR=NP=128	7.476s	7.344s	1.02	0.976s	0.999s	0.934s	0.98	1.04
floyd-warshall	N=1024	86.06s	91.89s	0.94	82.76s	13.64s	4.909s	6.07	16.9
gemm	NI=NJ=NK=1024	60.40s	24.20s	2.50	7.026s	5.473s	3.628s	1.28	1.94
gesummv	N=4096	0.488s	0.536s	0.91	0.078s	0.069s	0.074s	1.13	1.05
matmul	N=2048	688.5s	196.3s	3.51	89.49s	94.70s	27.44s	0.94	3.26
mvt	N=4096	1.248s	0.828s	1.51	0.195s	0.334s	0.105s	0.58	1.86
seidel	N=1024, T= 1024	44.82s	44.79s	1.00	45.03s	9.797s	8.364s	4.60	5.38
ssymm	N=2048	122.8s	177.4s	0.69	15.03s	55.45s	23.85s	0.27	0.63
syr2k	NI=1024, NJ=1024	34.03s	30.86s	1.10	4.190s	4.423s	4.223s	0.95	0.99
syrk	NI=1024, NJ=1024	24.44s	22.01s	1.11	2.974s	3.118s	2.793s	0.95	1.06
trmm	N=2048	231.7s	64.62s	3.59	41.29s	39.94s	11.42s	1.03	3.62

- *pg-loc* uses the LabVIEW compiler but with our transformation pass enabled to perform locality optimizations.
- *lv-parallel* again uses the LabVIEW production compiler, but with loop parallelization. The parallel for loop feature in LabVIEW [19, 7] is used to parallelize loops when possible in the G code.
- *pg-par* is with our transformation pass enabled to perform auto-parallelization but without any locality optimizing transformations. In order to realize a parallel loop identified as parallelizable, Bordelon et al [7]'s solution is used. The parallel loops in the transformed code are identified using Pluto [16].
- *pg-loc-par* is with our transformation pass enabled to perform both locality optimizations and auto-parallelization.

The comparisons of the runtime performance with various configurations can be found in Table 1. The performance numbers were obtained on on a dual-socket Intel Xeon CPU E5606 (2.13 GHz, 8 MB L3 cache) machine with 8 cores in all, and 24 GB of RAM. LLVM 2.8 was the final backend compiler used by LabVIEW.

Table 1 shows that the benchmarks gemm, matmul, mvt, syr2k, syrk and trmm benefit from locality-enhancing optimizations, in particular, loop tiling, and in addition, loop fusion and other unimodular transformations [2, 18]. Table 1 also shows the effect of locality optimizations in conjunction with loop parallelization. It can be seen that for floyd-warshall and seidel, loop skewing exposes loop parallelism that could not have been exploited without it. The benchmarks, atax, bicg, floyd-warshall, gemm, matmul, mvt, seidel, syrk, trmm benefit from more coarse-grained parallelism, i.e., a reduced frequency of shared-memory synchronization between cores as a result of loop tiling. In some cases, we see a slow down with PolyGLoT, often by about 10%. We believe that this is primarily due to transformed code generated by PolyGLoT not being optimized by subsequent passes within LabVIEW and the backend compiler (LLVM) as well as the baseline (*lv-noparallel* and *lv-parallel*). This is also partly supported by the fact that *pg-loc* itself produces this slow down, for example, for ssymm. Better downstream optimization within LabVIEW and in LLVM after PolyGLoT has been run can address this. In addition, the loop fusion heuristic used by Pluto can be tailored for LabVIEW code to obtain better performance. Overall, we see a mean speedup of 2.30$\times$ with PolyGLoT (*pg-loc-par*) over the state-of-the-art (*lv-parallel*).

7 Related Work

Much work has been done on using polyhedral techniques in the compilation of imperative languages [9, 11, 5]. Clan is a widely used research tool for extracting a polyhedral representation from C static control parts [3]. Production compilers with polyhedral framework implementations include IBM XL [6], RSTREAM [15], and LLVM [12].

Ellmenreich et al. [8] have considered the problem of adapting the polyhedral model to parallelize a functional program in Haskell. The source program is analyzed to obtain a set of parallelizable array definitions. Dependence analysis on each array set is then performed to parallelize all the computations within the set. Johnston et al. [13] review the advances in dataflow programming over the decades. Ample work has been done on parallelizing dataflow programs. It includes the work on loop parallelization analysis by Yi et al. [19]. Dependences between array accesses are analyzed using standard techniques to determine if a given user-specified loop in a graphical dataflow program can be parallelized. In contrast to these works, the focus of our work is not really on parallelization but on leveraging existing polyhedral compilation techniques to perform dataflow program transformations. Parallelism detection is but a small component of a loop-nest optimization framework. The complete polyhedral representation that we extract from a given dataflow program part can be used to drive automatic transformations, many of which can actually aid parallelization. Furthermore, to the best of our knowledge, no prior art exists that tackles this problem and the problem of dataflow program part synthesis from an equivalent polyhedral representation by exploiting the inplaceness opportunities that can be inferred from the dataflow program. The work of Yi et al. [19] is commercially available as the parallel for loop feature in LabVIEW, and we compared with it through experiments in Section 6. Given an iterative construct in a dataflow program that is marked parallel, Bordelon et al. [7] studied the problem of parallelizing and scheduling it on multiple processing elements. Our system uses it to eventually realize parallel code from the transformed DFIR.

The interplay between scheduling and maximizing the inplaceness of aggregate data has been studied by Abu-Mahmeed et al. [1]. Recently, Gerard et al. [10] have built on this work to provide a solution for inter-procedural inplaceness using language annotations that express inplace modifications. The soundness of such an annotation scheme is guaranteed by a semi-linear type system, where a value of a semi-linear type can be read multiple times and then updated once. For any array data source in any diagram of the SCoD, there is at most one node that can overwrite it. During the polyhedral extraction, by scheduling a write node after all the read nodes which share the same data source, we in effect choose semi-linear type semantics on the array data in the dataflow diagram. It also allows us to infer an inplace path of array updates. The inplace path is used for associating the accesses to an array definition in the polyhedral representation, which can have multiple write accesses to the same definition.

8 Conclusions

We have addressed the problem of extracting polyhedral representations from graphical dataflow programs that can be used to perform high-level program transformations automatically. Additionally, we also studied the problem of synthesizing dataflow diagrams from their equivalent polyhedral representation. To the best of our knowledge, this is the first work which tackles these problems, and does this while exploiting inplaceness opportunities inherent in a dataflow program. We also demonstrated that our techniques are of practical relevance by building an automatic transformation framework for the LabVIEW compiler that uses them. In several cases, programs compiled through our framework outperformed those compiled otherwise by significant margins, sometimes by a factor as much as seventeen. A mean speed-up of 2.30× was observed over state-of-the-art.

Acknowledgments. We thank Anand Kodaganur and Praveen Shivananda from National Instruments, Bangalore for their support. We are also thankful all anonymous reviewers and Albert Cohen (INRIA) for very valuable feedback.

References

[1] Abu-Mahmeed, S., McCosh, C., Budimlić, Z., Kennedy, K., Ravindran, K., Hogan, K., Austin, P., Rogers, S., Kornerup, J.: Scheduling Tasks to Maximize Usage of Aggregate Variables in Place. In: de Moor, O., Schwartzbach, M.I. (eds.) CC 2009. LNCS, vol. 5501, pp. 204–219. Springer, Heidelberg (2009)

[2] Aho, A.V., Sethi, R., Ullman, J.D., Lam, M.S.: Compilers: Principles, Techniques, and Tools, 2nd edn. Prentice-Hall (2006)

[3] Bastoul, C.: Clan: The Chunky Loop Analyzer. The Clan User Guide

[4] Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT, pp. 7–16 (September 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., Gunluk, O., Dash, S., Renganarayanan, L.: A model for fusion and code motion in an automatic parallelizing compiler. In: PACT. ACM (2010)

[7] Bordelon, A., Dye, R., Yi, H., Fletcher, M.: Automatically creating parallel iterative program code in a data flow program (20100306733) (December 2010), http://www.freepatentsonline.com/y2010/0306733.html

[8] Ellmenreich, N., Lengauer, C., Griebl, M.: Application of the Polytope Model to Functional Programs. In: Carter, L., Ferrante, J. (eds.) LCPC 1999. LNCS, vol. 1863, pp. 219–235. Springer, Heidelberg (2000)

[9] Feautrier, P.: Some efficient solutions to the affine scheduling problem: Part I, one-dimensional time. International Journal of Parallel Programming 21(5), 313–348 (1992)

[10] Gérard, L., Guatto, A., Pasteur, C., Pouzet, M.: A modular memory optimization for synchronous data-flow languages: application to arrays in a lustre compiler. In: LCTES, pp. 51–60 (2012)

[11] Griebl, M.: Automatic Parallelization of Loop Programs for Distributed Memory Architectures. University of Passau, Habilitation thesis (2004)

[12] Grosser, T., Zheng, H., Aloor, R., Simbrger, A., Grolinger, A., Pouchet, L.-N.: Polly: Polyhedral optimization in LLVM. In: IMPACT (2011)

[13] Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages. ACM Comput. Surv. 36(1) (March 2004)

[14] NI LabVIEW Compiler: Under the Hood,
http://www.ni.com/white-paper/11472/en
[15] Meister, B., Vasilache, N., Wohlford, D., Baskaran, M.M., Leung, A., Lethin, R.: R-stream compiler. In: Encyclopedia of Parallel Computing, pp. 1756–1765. Springer (2011)
[16] PLUTO: An automatic polyhedral parallelizer and locality optimizer for multicores, http://pluto-compiler.sourceforge.net
[17] Polybench, http://polybench.sourceforge.net
[18] Wolfe, M.: High Performance Compilers for Parallel Computing. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
[19] Yi, H., Fletcher, M., Dye, R., Bordelon, A.: Loop parallelization analyzer for data flow programs (20100306753) (December 2010),
http://www.freepatentsonline.com/y2010/0306753.html

Architecture-Independent Dynamic Information Flow Tracking[*]

Ryan Whelan[1], Tim Leek[2], and David Kaeli[1]

[1] Department of Electrical and Computer Engineering
Northeastern University, Boston, MA USA
{rwhelan,kaeli}@ece.neu.edu
[2] Cyber System Assessments Group
MIT Lincoln Laboratory, Lexington, MA USA
tleek@ll.mit.edu

Abstract. Dynamic information flow tracking is a well-known dynamic software analysis technique with a wide variety of applications that range from making systems more secure, to helping developers and analysts better understand the code that systems are executing. Traditionally, the fine-grained analysis capabilities that are desired for the class of these systems which operate at the binary level require tight coupling to a specific ISA. This places a heavy burden on developers of these systems since significant domain knowledge is required to support each ISA, and the ability to amortize the effort expended on one ISA implementation cannot be leveraged to support other ISAs. Further, the correctness of the system must carefully evaluated for each new ISA.

In this paper, we present a *general* approach to information flow tracking that allows us to support multiple ISAs without mastering the intricate details of each ISA we support, and without extensive verification. Our approach leverages binary translation to an intermediate representation where we have developed detailed, architecture-neutral information flow models. To support advanced instructions that are typically implemented in C code in binary translators, we also present a combined static/dynamic analysis that allows us to accurately and automatically support these instructions. We demonstrate the utility of our system in three different application settings: enforcing information flow policies, classifying algorithms by information flow properties, and characterizing types of programs which may exhibit excessive information flow in an information flow tracking system.

Keywords: Binary translation, binary instrumentation, information flow tracking, dynamic analysis, taint analysis, intermediate representations.

[*] This work was sponsored by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

1 Introduction

Dynamic information flow tracking (also known as dynamic taint analysis) is a well-known software analysis technique that has been shown to have wide applicability in software analysis and security applications. However, since dynamic information flow tracking systems that operate at the binary level require fine-grained analysis capabilities to be effective, this means that they are generally tightly coupled with the ISA of code to be analyzed.

In order to implement a fine-grained analysis capability such as information flow tracking for an ISA of interest, an intimate knowledge of the ISA is required in order to accurately capture information flow for each instruction. This is especially cumbersome for ISAs with many hundreds of instructions that have complex and subtle semantics (e.g., x86). Additionally, after expending the work required to complete such a system, the implementation only supports the single ISA, and a similar effort is required for each additional ISA. To overcome this challenge, we've elected to take a compiler-based approach by translating architecture-specific code into an architecture-independent intermediate representation where we can develop, reuse, and extend a single set of analyses.

In this work, we present Pirate: a Platform for Intermediate Representation-based Analyses of Tainted Execution. Pirate decouples the tight bond between the ISA of code under analysis and the additional instrumentation code, and provides a general taint analysis framework that can be applied to a large number of ISAs. Pirate leverages QEMU [4] for binary translation, and LLVM [14] as an intermediate representation within which we can perform our architecture-independent analyses. We show that our approach is both *general* enough to be applied to multiple ISAs, and *precise* enough to provide the detailed kind of information expected from a fine-grained dynamic information flow tracking system. In our approach, we define detailed byte-level information flow models for 29 instructions in the LLVM intermediate representation which gives us coverage of thousands of instructions that appear in translated guest code. We also apply these models to complex guest instructions that are implemented in C code. To the best of our knowledge, this is the first implementation of a binary level dynamic information flow tracking system that is general enough to be applied to multiple ISAs without requiring source code.

The contributions of this work are:

- A framework that leverages dynamic binary translation producing LLVM intermediate representation that enables architecture-independent dynamic analyses, and a language for precisely expressing information flow of this IR at the byte level.
- A combined static/dynamic analysis to be applied to the C code of the binary translator for complex ISA-specific instructions that do not fit within the IR, enabling the automated analysis and inclusion of these instructions in our framework.

– An evaluation of our framework for x86, x86_64, and ARM, highlighting three security-related applications: 1) enforcing information flow policies, 2) characterizing algorithms by information flow, and 3) diagnosing sources of state explosion for each ISA.

The rest of this paper is organized as follows. In Section 2, we present background on information flow tracking. Sections 3 and 4 present the architectural overview and implementation of PIRATE. Section 5 presents our evaluation with three security-related applications, while Section 6 includes some additional discussion. We review related work in Section 7, and conclude in Section 8.

2 Background

Dynamic information flow tracking is a dynamic analysis technique where data is labeled, and subsequently tracked as it flows through a program or system. Generally data is labeled and tracked at the byte level, but this can also happen at the bit, word, or even page level, depending on the desired granularity. The labeling can also occur at varying granularities, where each unit of data is also accompanied by one bit of data (tracked or not tracked), one byte of data (accompanied by a small number), or a data structure that tracks additional information. Tracking additional information is useful for the cases when *label sets* are propagated through the system. In order to propagate the flow of data, major components of the system need a shadow memory to keep track of where data flows within the system. This includes CPU registers, memory, and in the case of whole systems, the hard drive also. When information flow tracking is implemented for binaries at a fine-grained level, this means that propagation occurs at the level of the ISA where single instructions that result in a flow of information are instrumented. This instrumentation updates the shadow memory accordingly when tagged information is propagated.

Information flow tracking can occur at the hardware level [7,26,28], or in software through the use of source-level instrumentation [12,15,31], binary instrumentation [10,13,22], or the use of a whole-system emulator [9,20,24]. In general, hardware-based approaches are faster, but less flexible. Software-based approaches tend to have higher overheads, but enable more detailed dynamic analyses. Source-level approaches tend to be both fast and flexible, but are sometimes impractical when source code is not available. These techniques have proven to be effective in a wide variety of applications, including detection and prevention of exploits, malware analysis, debugging assistance, vulnerability discovery, and network protocol reverse engineering.

Due to the popularity of the x86 ISA, and the tight bond of these binary instrumentation techniques with the ISA under analysis, many of these systems have been carefully designed to correctly propagate information flow only for the instructions that are included in x86. This imposes a significant limitation on dynamic information flow tracking since a significant effort is required to support additional ISAs. PIRATE solves this problem by decoupling this analysis technique from the underlying ISA, without requiring source code or higher-level semantics.

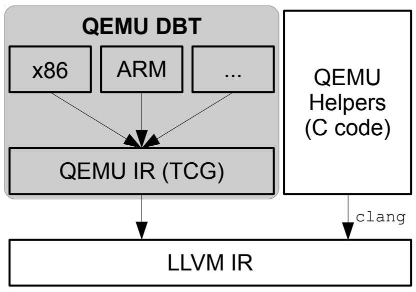


Fig. 1. Lowering code to the LLVM IR with QEMU and Clang

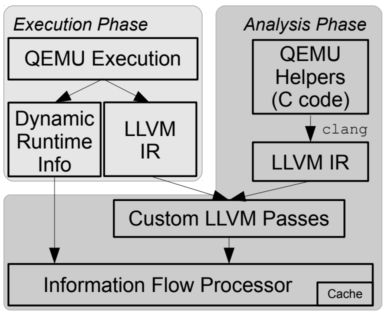


Fig. 2. System architecture split between execution and analysis

3 System Overview

At a high level, PIRATE works as follows. The QEMU binary translator [4] is at the core of our system. QEMU is a versatile dynamic binary translation platform that can translate 14 different guest architectures to 9 different host architectures by using its own custom intermediate representation, which is part of the Tiny Code Generator (TCG). In our approach, we take advantage of this translation to IR to support information flow tracking for multiple guest architectures. However, the TCG IR consists of very simple RISC-like instructions, making it difficult to represent complex ISA-specific instructions. To overcome this limitation, the QEMU authors implement a large number of guest instructions in *helper functions*, which are C implementations of these instructions. Each guest ISA implements hundreds of instructions in helper functions, so we have devised a mechanism to automatically track information flow to, from, and within these helper functions.

3.1 Execution and Analysis

We perform dynamic information flow tracking on Linux binaries using the following approach, which is split between execution and analysis. In the execution phase, we run the program as we normally would under QEMU. We have augmented the translation process to add an additional translation step, which translates the TCG IR to the LLVM IR and then executes on the LLVM just-in-time (JIT) compiler. This translation occurs at the level of basic blocks of guest code. In Figure 1, we show the process of lowering guest code and helper function code to the LLVM IR to be used in our analysis.

Figure 2 shows the architecture of our system. Once we have the execution of guest code captured in the LLVM IR, we can perform our analysis over each basic block of guest code with custom IR passes we have developed within the LLVM infrastructure. The first part of our analysis is to derive information flow operations to be executed on our abstract information flow processor. This is

```
void glue(helper_pshufw, SUFFIX) (Reg *d, Reg *s, int order){
    Reg r;
    r.W(0) = s->W(order & 3);
    r.W(1) = s->W((order >> 2) & 3);
    r.W(2) = s->W((order >> 4) & 3);
    r.W(3) = s->W((order >> 6) & 3);
    *d = r;
}
```

Fig. 3. QEMU helper function implementing the `pshufw` (packed shuffle word) MMX instruction for x86 and x86_64 ISAs

an automated process that emits information flow operations that we've specified for each LLVM instruction. After deriving the sequence of information flow operations for a basic block of code, we execute them on the information flow processor to propagate tags and update the shadow memory. This allows us to keep our shadow memory updated on the level of a guest basic block. When we encounter system calls during our processing, we treat I/O-related calls as sources or sinks of information flow. For example, the `read()` system call is a source that we begin tracking information flow on, and a `write()` system call is a sink where we may want to be notified if tagged information is written to disk.

3.2 Optimizations

QEMU has a *translation block cache* mechanism that allows it to keep a copy of translated guest code, avoiding the overhead of re-translating frequently executed code repeatedly. This optimization permeates to our analysis phase; a guest basic block that is cached may also be executed repeatedly in the LLVM JIT, but it only appears once in the LLVM module that we analyze. This optimization in QEMU also provides us the opportunity to cache information flow operations. As we analyze translated guest code in the representation of basic blocks that may be cached, we can also perform the derivation of information flow tracking operations once, and then cache; we refer to these as *taint basic blocks*. Once we derive a taint basic block for a guest basic block of code, we deposit it into our *taint basic block cache*.

3.3 Static Analysis of QEMU Helper Functions

Since QEMU helper functions perform critical computations on behalf of the guest, we need to include them in our analysis as well. To do this, we use Clang [1] to translate helper functions to the LLVM IR. From there, we can use the exact same analysis that we apply to translated guest code to derive information flow operations. Since this is a static analysis, we can emit the corresponding information flow operations for each helper function into a *persistent cache*. Figure 3 shows the helper function that implements the `pshufw` MMX instruction. Automatically analyzing functions like these takes a significant burden off of developers of information flow tracking systems.

4 Implementation

Next, we present implementation details of PIRATE. The implementation consists of several major components: the execution and trace collection engine, the information flow processor which propagates tagged data, the shadow memory which maintains tagged data, the analysis engine which performs analysis and instrumentation passes over the LLVM IR, and the caching mechanism that caches information flow operations and reduces overhead. In this paper, we support information flow tracking for the x86, x86_64, and ARM ISAs on Linux binaries, but our approach can be extended in a straightforward manner to other ISAs that are supported by QEMU. Our system is implemented with QEMU 1.0.1 and LLVM 3.0.

4.1 Dynamic Binary Translation to LLVM IR

At the core of our approach is the translation of guest code to an ISA-neutral IR. Much like a standard compiler, we want to perform our analyses in terms of an IR, which allows us to decouple from the ISA-specific details. We take advantage of the fact that QEMU's dynamic binary translation mechanism translates guest code to its own custom IR (TCG), but this IR is not robust enough for our analyses. In order to bridge the gap between guest code translated to the TCG IR and helper functions implemented in C, we chose to perform our analysis in the LLVM IR. Since both the TCG and LLVM IR consist of simple RISC-like instructions, we have a straightforward translation from TCG to LLVM. For this translation, we leverage the TCG to LLVM translation module included as part of the S2E framework [8]. LLVM also enables us to easily translate helper functions to its IR (through the Clang front end), and it provides a rich set of APIs for us to work with.

By performing information flow tracking in the LLVM IR, we abstract away the intricate details of each of our target ISAs, leaving us with only 29 RISC-like instructions that we need to understand in great detail and model correctly for information flow analysis. These 29 LLVM instructions describe all instructions that appear in translated QEMU code, and all helper functions that we currently support. Developing detailed information flow tracking models for this small set of LLVM instructions that are semantically equivalent to guest code means that our information flow tracking will also be semantically equivalent. Additionally, since the system actually executes the translated IR, we can rely on the correctness of the translation to give us a degree of assurance about the completeness and correctness of our analysis. While formally verifying the translation to LLVM IR is out of scope of this work, we assume that this translation is correct since we can execute programs on the LLVM JIT and obtain correct outputs.

4.2 Decoupled Execution and Analysis

In PIRATE, we decouple the execution and analysis of code in order to give us flexibility in altering our analyses on a single execution. We capture a compact

Table 1. Information flow operations

Operation	Semantics
`label(a,l)`	$L(a) \leftarrow L(a) \cup l$
`delete(a)`	$L(a) \leftarrow \emptyset$
`copy(a,b)`	$L(b) \leftarrow L(a)$
`compute(a,b,c)`	$L(c) \leftarrow L(a) \cup L(b)$
`insn_start`	Bookkeeping info
`call`	Begin processing a QEMU helper function
`return`	Return from processing a QEMU helper function

dynamic trace of the execution in the LLVM bitcode format, along with dynamic values from the execution that include memory access addresses, and branch targets. We obtain these dynamic values by instrumenting the IR to log every address of loads and stores, and every branch taken during execution. The code we capture is in the format of an LLVM bitcode module which consists of a series of LLVM functions, each corresponding to a basic block of guest code. We also capture the order in which these functions are executed. Our trace is compact in the sense that if a basic block is executed multiple times, we only need to capture it once.

Once we've captured an execution, we leverage the LLVM infrastructure to perform our analysis directly on the LLVM IR. Our analysis is applied in the form of an LLVM analysis pass, where we specify the set of *information flow operations* for each LLVM instruction in the execution. We perform this analysis at the granularity of a guest basic block, and our analysis emits a *taint basic block*. Our abstract information flow processor then processes these taint basic blocks to update the shadow memory accordingly.

4.3 Shadow Memory, Information Flow Processor

Shadow Memory. Our shadow memory is partitioned into the following segments: virtual memory, architected registers, and LLVM registers (which includes multiple calling scopes). The virtual memory portion of the shadow memory keeps track of information flow through the process based on virtual addresses. The architected state portion keeps track of general purpose registers, program counters, and also some special purpose registers (such as MMX and XMM registers for x86 and x86_64) – this is the only architecture-specific component of our system. The LLVM shadow registers are how we keep track of information flow between LLVM IR instructions, which are expressed in static single assignment form with infinite registers. Currently, our shadow memory models 2,000 abstract registers, which is sufficient for our analysis. We maintain multiple scopes of abstract LLVM registers in our shadow memory to accommodate the calling of helper functions, which are explained in more detail in Section 4.5. The shadow memory is configurable so data can be tracked at the binary level (tagged or untagged), or positionally with multiple labels per address, which we refer to as a *label set*. Since we are modeling the entire address space of a process in our shadow memory, it is important that we utilize an efficient implementation. For 32-bit ISAs, our shadow memory of the virtual address space consists of a two-level structure that maps a directory to tables with tables that map to pages, similar to x86 virtual addressing. For 64-bit ISAs, we instead use a five-level structure in order to accommodate the entire 64-bit address space. To save memory overhead, we only need to allocate shadow guest memory for memory pages that contain tagged information.

Deriving Information Flow Operations. On our abstract information flow processor, we execute information flow operations in order to propagate tags and update the shadow memory. These operations specify information flow at the byte level. An address can be a byte in memory, a byte in an architected register, or a byte in an LLVM abstract register. The set of information flow operations can be seen in Table 1. Here, we describe them in more detail:

- `label`: Associate label l with the set of labels that belong to address a.
- `delete`: Discard the label set associated with address a.
- `copy`: Copy the label set associated with address a to address b.
- `compute`: Address c gets the union of the label sets associated with address a and address b.
- `insn_start`: Maintains dynamic information for operations. For loads and stores, a value from the dynamic log is filled in. For branches, a value from the dynamic log is read to see which branch was taken, and which basic block of operations needs to be processed next.
- `call`: Indication to process information flow operations for a QEMU helper function. Shift information flow processor from caller scope to callee scope, which has a separate set of shadow LLVM registers. Retrieve information flow operations from the persistent cache. If the helper function takes arguments, propagate information flow of arguments from caller scope to callee scope.
- `return`: Indication that processing of a QEMU helper function is finished. Shift information flow processor from callee scope to caller scope. If the helper function returns a value, propagate information flow to shadow return value register.

The information flow models we've developed allow us to derive the sequence of information flow operations for each LLVM function using our LLVM analysis pass. In this pass, we iterate over each LLVM instruction and populate a buffer with the corresponding information flow operations. In Figure 4, we show sequences of information flow operations for the LLVM `xor` and `load` instructions.

4.4 Caching of Information Flow Tracking Operations

One of the main optimizations that QEMU implements is the *translation block cache* which saves the overhead of retranslating guest code to host code for frequently executed basic blocks. We took a similar approach for our *taint basic*

LLVM Instruction:

```
%32 = xor i32 %30, %31;
```

Information Flow Operations:

```
compute(%30[0], %31[0], %32[0]);
compute(%30[1], %31[1], %32[1]);
compute(%30[2], %31[2], %32[2]);
compute(%30[3], %31[3], %32[3]);
```

LLVM Instruction:

```
%7 = load i32* %2;
```

Information Flow Operations:

```
// get load address from dynamic
// log, and fill in next
// four operations
insn_start;

// continue processing operations
copy(addr[0], %7[0]);
copy(addr[1], %7[1]);
copy(addr[2], %7[2]);
copy(addr[3], %7[3]);
```

Fig. 4. Examples of byte-level information flow operations for 32-bit `xor` and `load` LLVM instructions

blocks and developed a caching mechanism to eliminate the need to repeatedly derive information flow operations. This means we only need to run our pass once on a basic block, and as long as it is in our cache, we simply process the information flow operations.

Our caching mechanism works as follows. During our analysis pass, we leave dynamic values such as memory accesses and taken branches empty, and instead fill them in at processing time by using our `insn_start` operation, as illustrated in Figure 4. In the case of a branch, the `insn_start` operation tells the information flow processor to consult the dynamic log to find which branch was taken, and continue on to process that *taint basic block*. This technique enables us to process cached information flow operations with minor preprocessing to adjust for dynamic information.

4.5 Analysis and Processing of QEMU Helper Functions

Instrumentation and Analysis. Because important computations are carried out in helper functions, we need some mechanism to analyze them in a detailed, correct way. Because there are hundreds of helper functions in QEMU, this process needs to be automated. We have modified the QEMU build process to automatically derive information flow operations for a subset of QEMU helper functions, and save them to a *persistent cache*. Here, we describe that process in more detail:

1. **Translate helper function C code to LLVM IR using Clang.** The Clang compiler [1], which is a C front end for the LLVM infrastructure, has an option to emit a LLVM bitcode file for a compilation unit. We have modified the QEMU build process to do this for compilation units which contain helper functions we are interested in analyzing.
2. **Run our LLVM function analysis pass on the helper function LLVM.** Once we have helper function code in the LLVM format, we can compute

information flow operations using the same LLVM analysis pass that we have developed for use on QEMU translated code.

3. **Instrument the LLVM IR to populate the dynamic log.**
   In order for us to perform our analysis on the helper function LLVM, we need this code to populate the dynamic log with load, store, and branch values. We have developed a simple code transformation pass that instruments the helper function IR with this logging functionality.
4. **Emit information flow operations into a persistent cache.**
   Helper function information flow operations can be emitted into a persistent cache because they are static, and because runtime overhead will be reduced by performing these computations at compile time. This cache is now another by-product of the QEMU build process.
5. **Compile and link the instrumented LLVM.**
   Since the instrumented IR should populate the dynamic log during the trace collection, we create an object file that can be linked into QEMU. Again, we can use Clang to translate our instrumented LLVM bitcode into an object file, and then link that file into the QEMU executable during the QEMU build process.

Processing. Integration of helper function analysis into PIRATE works as follows. During analysis of QEMU generated code, we see a call to a helper function, arguments (in terms of LLVM registers, if any), and return value (in terms of a LLVM register, if any). When we see a call instruction in our analysis pass on translated code, we propagate the information flow of the arguments to the callee's scope of LLVM registers, if necessary. For example, assume in the caller's scope that there is a call to `foo()` with values `%29` and `%30` as arguments. In the scope of the helper function, the arguments will be in values `%0` and `%1`. So the information flow of each argument gets copied to the callee's scope, similar to how arguments are passed to a new scope on a stack. We then insert our `call` operation, which tells the information flow processor which function to process, and the pointer to the set of corresponding taint operations that are in the cache. The information flow processor then processes those operations until return. On return, a helper function may or may not return a value to the previous scope. For return, we emit a `return` operation to indicate that we are returning to the caller's scope. If a value is returned, then its information will be present in the LLVM return value register in our shadow memory so if there are any tags on that value, they will be propagated back to the caller's scope correctly.

5 Evaluation

In our evaluation, we show that PIRATE is decoupled from a specific ISA, bringing the utility of information flow tracking to software developers and analysts regardless of the underlying ISA they are targeting. We demonstrate the following three applications for x86, x86_64, and ARM: enforcing information flow policies, algorithm characterization, and state explosion characterization. We

Table 2. Functions which operate on tagged data

Program	x86	x86_64	ARM
Hello World	10/104 (9.62%)	11/93 (11.83%)	10/100 (10.00%)
Gzip Compress	17/150 (11.33%)	15/147 (10.20%)	17/150 (11.33%)
Bzip2 Compress	16/167 (9.58%)	16/153 (10.46%)	17/165 (10.30%)
Tar Archive	2/391 (0.51%)	2/372 (0.54%)	2/361 (0.55%)
OpenSSL AES Encrypt	8/674 (1.19%)	7/655 (1.07%)	7/671 (1.04%)
OpenSSL RC4 Encrypt	4/672 (0.59%)	4/653 (0.61%)	5/679 (0.73%)
Kernighan-Lin Graph Partition	29/132 (21.97%)	63/122 (51.64%)	32/134 (23.88%)

performed our evaluation on Ubuntu 64-bit Linux 3.2, and in each case, we compiled programs with GCC 4.6 with default options for each program.

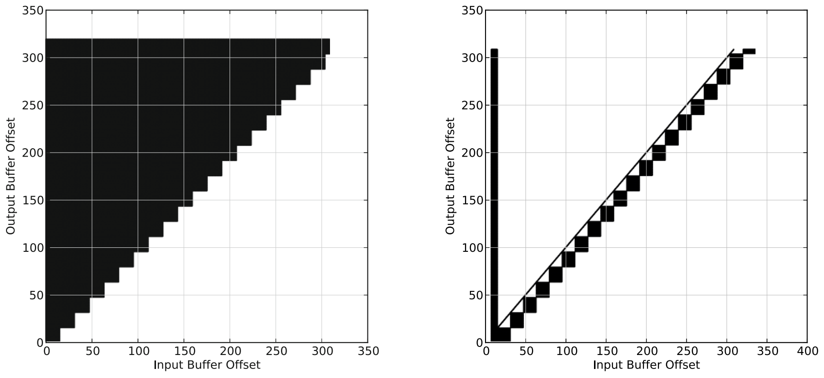
5.1 Enforcing Information Flow Policies

One important application of dynamic information flow tracking is to define information flow policies for applications, and ensure that they are enforced within the application. For example, one may define a policy that a certain subset of program data is not allowed to be sent out over the network, or that user-provided data may not be allowed to be passed to security-sensitive functions. A universal information flow policy that most programs enforce is that user-provided data may not be used to overwrite the program counter. However, there is a lack of information flow tracking systems that support embedded systems employing ARM, MIPS, PowerPC, and even x86_64, so defining and verifying these policies without modifying source code is difficult or impossible with existing information flow tracking systems.

Our system enables software developers to define and enforce these information flow policies, regardless of the ISA they are developing for. In one set of experiments, we carried out a buffer overflow exploit for a vulnerable program and our system was able to tell us exactly which bytes from our input were overwriting the program counter for x86, x86_64, and ARM.

In addition to telling the developer where in the program these information flow policies are violated, Pirate can also tell the developer each function in the program where tagged data flows. This can assist the developer in identifying parts of the program that operate directly on user input so they can more clearly identify where to focus when ensuring the security of their program. In Table 2, we present results for the ratio of functions in several programs that operate on tagged data. These ratios indicate the percentage of functions in the program that operate on tagged data compared with every function executed in the dynamic trace. For most of the programs we evaluated, these ratios are under 25%. The exception is KL graph partition for x86_64, which shows effects of state explosion. This is addressed in more detail in Section 5.3.

With this enhanced security capability, software developers can more easily identify parts of their programs that may be more prone to attacks. This



(a) Encryption (b) Decryption

Fig. 5. AES CBC mode input/output dependency graphs

capability can also be used in the context of vulnerability discovery when source code isn't available in binaries compiled for different ISAs.

5.2 Algorithm Characterization

Recent work has shown that dynamic analysis techniques such as information flow tracking can help malware analysts better understand the obfuscation techniques employed by malware [6,17]. However, these approaches suffer the same limitations as other systems, where they are tightly-coupled with a single ISA (x86). As embedded systems are becoming increasingly relevant in the security community, it is becoming more desirable for analysts to leverage the power of dynamic information flow tracking for these embedded ISAs.

Here, we highlight the capability of our system to characterize encryption algorithms based on *input/output dependency graphs*. We generate these graphs by positionally labeling each byte in the buffer after the `read()` system call, and tracking how each byte of the input is propagated through the encryption algorithm. By subsequently interposing on the `write()` system call, we can inspect each byte in the buffer to see the set of input bytes that influences each output byte. For these experiments, we chose OpenSSL 1.0.1c [2] as a test suite for two modes of AES block cipher encryption, and RC4 stream cipher encryption for x86, x86_64, and ARM. The OpenSSL suite is ideal for demonstrating our capability because most of the encryption algorithms have both C implementations and optimized handwritten assembly implementations.

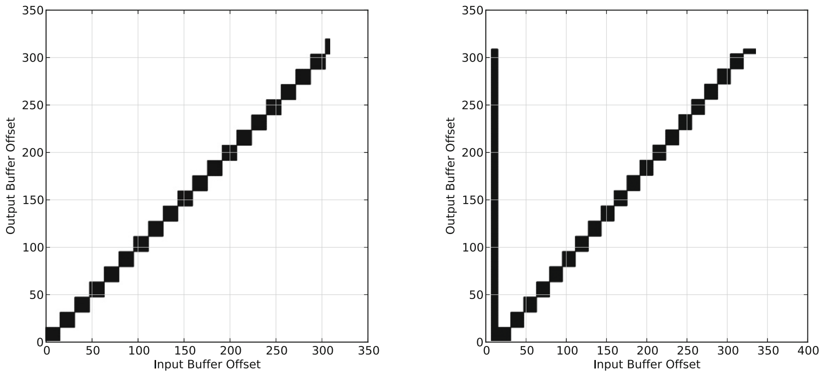
AES, Cipher Block Chaining Mode. AES (Advanced Encryption Standard) is a block encryption algorithm that operates on blocks of 128 bits of data, and allows for key sizes of 128, 192, and 256 bits [25]. As there are a variety of encryption modes for AES, cipher block chaining mode (CBC) is one of the stronger modes. In CBC encryption, a block of plaintext is encrypted, and then the resulting ciphertext is passed through an exclusive-or operation with the subsequent block of plaintext before that plaintext is passed through the block cipher. Inversely, in CBC decryption, a block of ciphertext is decrypted, and then passed through an exclusive-or operation with the previous block of ciphertext in order to retrieve the plaintext.

Figure 5 shows our input/output dependency graphs for AES encryption and decryption. In these figures, we can visualize several main characteristics of the AES CBC cipher: the block size (16 bytes), and the encryption mode. In Figure 5(a), the first block of encrypted data is literally displayed as a block indicating complicated dependencies between the first 16 bytes. We see the chaining pattern as each subsequent block depends on all blocks before it in the dependency graph. In Figure 5(b), we can see that each value in the output is dependent on the second eight bytes in the input; this corresponds to the salt value, which is an element of randomness that is included as a part of the encrypted file. We can also see the chaining dependency characteristic of CBC decryption, where each block of ciphertext is decrypted, and then passed through an exclusive-or operation with the previous block of ciphertext. This series of exclusive-or operations is manifested as the diagonal line in Figure 5(b). With PIRATE, we were able to generate equivalent dependency graphs for x86, x86_64, and ARM, for both handwritten and C implementations.

This result highlights the versatility of our approach based on the wide variety of implementations of AES in OpenSSL. In particular, the x86 handwritten version is implemented using instructions from the MMX SIMD instruction set. Our automated approach for deriving information flow operations for these advanced instructions allows us to support these instructions without the manual effort that other systems require.

AES, Electronic Code Book Mode. Electronic Code Book (ECB) mode is similar to CBC mode, except that it performs block-by-block encryption without the exclusive-or chaining of CBC mode [25]. The input/output dependency graphs we've generated to characterize this algorithm can be seen in Figure 6. Here, we see that our system can accurately tell us the block size and the encryption mode, without the chaining dependencies from the previous figures. We again see the dependence on the bytes containing the salt value in Figure 6(b).

For AES in ECB mode, we were able to generate equivalent dependency graphs for each ISA (x86, x86_64, and ARM) and implementation (handwritten and C implementation), with the exception of the ARM C implementation for decryption, and the x86 handwritten assembly implementation for encryption. In these exceptional cases, we see a similar input/output dependency graph with some additional apparent data dependence. This highlights a design decision of our system, where we over-approximate information flow transfer of certain LLVM instructions in order to prevent the incorrect loss of tagged information. This over-approximation can manifest itself as additional information flow spread, but we've made the decision that it is better to be conservative rather than miss an exploit, especially in the context of security-critical applications of this system.



(a) Encryption (b) Decryption

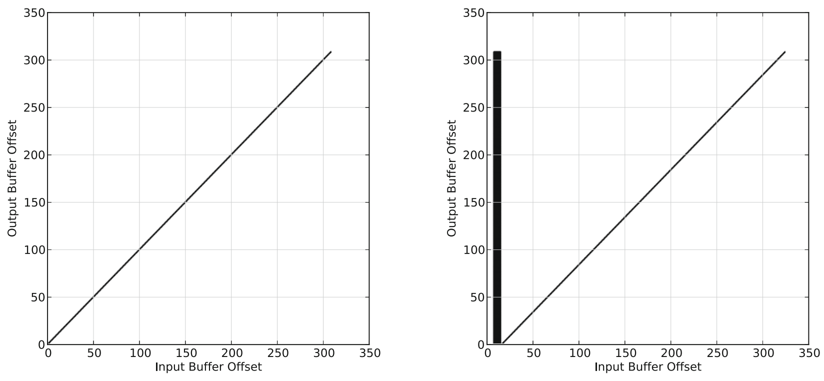
Fig. 6. AES ECB mode input/output dependency graphs

RC4. RC4 is a stream cipher where encryption occurs through byte-by-byte exclusive-or operations. The algorithm maintains a 256 byte state that is initialized by the symmetric key [25]. Throughout the encryption, bytes in the state are swapped pseudo-randomly to derive the next byte of the state to be passed through an exclusive-or operation with the next byte of the plaintext.

The input/output dependency graphs for RC4 encryption can be seen in Figure 7. Since we only track information flow of the input data and not the key, we can see from these figures that there is a linear dependence from input to output, based on the series of exclusive-or operations that occur for each byte in the file. As with the previous figures for decryption, we can see the dependence on the salt value that is in the beginning of the encrypted file in Figure 7(b). For RC4 encryption and decryption, we were able to generate equivalent dependency graphs for encryption and decryption for each ISA (x86, x86_64, and ARM) and implementation (handwritten and C implementation) with the exception of x86_64 encryption and decryption handwritten implementations. For these cases, we see an equivalent dependency with additional information, again due to the conservative approach we take in terms of information flow tracking.

5.3 Diagnosing State Explosion

One limitation of information flow tracking systems is that they are subject to state explosion where tagged data spreads (i.e., grows) uncontrollably, increasing the amount of data that needs to be tracked as it flows through the system. This is especially true when pointers are tracked in the same way as data [23]. Despite this limitation, it is necessary to track pointers to detect and prevent certain kinds of attacks, such as those involved in read or write overflows but where no control flow divergence occurs [7], or those that log keyboard input which is passed through lookup tables [9]. In PIRATE, we've implemented tagged pointer



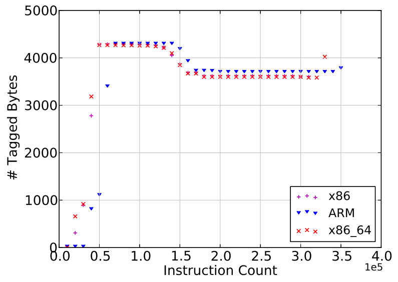
(a) Encryption (b) Decryption

Fig. 7. RC4 input/output dependency graphs

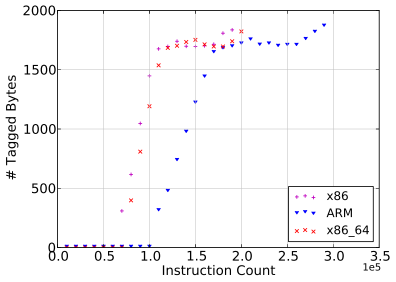
tracking as a configurable option. When this option is turned on, we propagate information for loads and stores not only from the addresses that are accessed, but also from the values that have been used to calculate those addresses. PIRATE allows us to evaluate the effects of state explosion between CISC and RISC ISAs since we support x86, x86_64, and ARM. It also allows us to evaluate the rate of state explosion for different software implementations of the same application.

To perform this evaluation, we've experimentally measured the amount of tagged information throughout executions of four programs that make extensive use of pointer manipulations with our tagged pointer tracking turned on. These programs are bzip2, gzip, AES CBC encryption, and the Kernighan-Lin (KL) graph partitioning tool (obtained from the pointer-intensive benchmark suite [3]). The bzip2 and gzip programs make extensive use of pointers in their compression algorithms. Part of the AES encryption algorithm requires lookups into a static table, known as the S-box. For these three programs, tagged pointer tracking is required to accurately track information flow through the program, or else this tagged information is lost due to the indirect memory accesses that occur through pointer and array arithmetic and table lookups. In addition, the KL algorithm implementation utilizes data structures like arrays and linked lists extensively for graph representation.

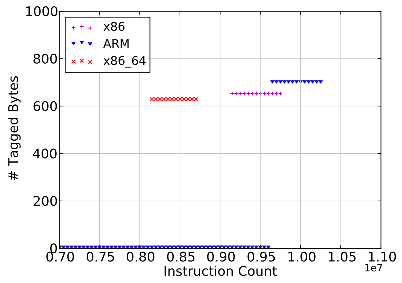
The measurements of information spread for pointer-intensive workloads can be seen in Figure 8 for x86, x86_64, and ARM. Figures 8(a), 8(b), and 8(c) show similar results for each ISA in terms of the number of tagged bytes in memory, but we can see offsets in instruction counts that highlights one of the differences of these CISC vs. RISC implementations. Figures 8(a) and 8(b) show the results of compressing the same file, which was approximately 300 bytes. These figures show the extent of information spread for these workloads; the peak number of tagged bytes reaches 14x the original file size for bzip2 on average across each ISA, and 6x the original file size for gzip on average across each ISA. For bzip2, the drastic growth in tagged data occurs as soon as the block sorting starts in



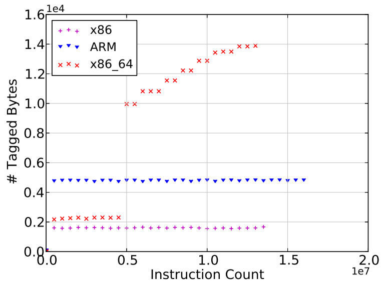
(a) Bzip2 compression



(b) Gzip compression



(c) AES CBC encryption



(d) KL graph partition

Fig. 8. Tagged information spread throughout execution

the algorithm. For gzip, there is a more gradual increase in tagged data as soon as the compression function begins compressing the data. These patterns are indicative of the complex manipulations that are made on files as the tagged data flows through these compression algorithms. On the contrary, while many complex manipulations occur on files through AES encryption, Figure 8(c) shows that the amount of tagged data increases just over 2x for each ISA. Overall, these three pointer-intensive algorithms show similar patterns of state explosion for information flow tracking, regardless of the underlying ISAs that we've evaluated.

Figure 8(d) on the other hand shows major discrepancies in the amount of tagged information across the various ISAs. For this experiment, we processed a file of size 1260 bytes. For x86 and ARM, we can see an initial increase of tagged information followed by a gradual increase up to a maximum of 1.5x and 4.1x the original tagged data, respectively. For x86_64, it is clear that a form of state explosion occurs causing the amount of tagged information to spread dramatically, reaching 11x the amount of original tagged data. Looking more closely at the x86_64 instance, we found that this initial explosion occurs inside of C library functions. One reason for this state explosion is that tagged data propagated to a global variable or base pointer, resulting in subsequent propagation with every access of that variable or base pointer. The fact that this explosion occurs inside of the C library implementation explains why we see the discrepancies across ISAs.

6 Discussion

Currently, our system provides the capability to perform dynamic information flow tracking for several of the major ISAs supported by the QEMU binary translator. It is straightforward to support more of these ISAs since we already have the ability to translate from the TCG IR to the LLVM IR. Additional work required for this involves properly tracking changes to CPU state (architecture-specific general purpose registers), and modeling those registers in the shadow memory. We plan to support more of the ISAs included in QEMU as future work.

We also plan to extend our decoupled execution and analysis approach to work with systems at runtime. This will enable us to perform dynamic detection and prevention of exploits, as well as other active defenses. Additionally, having a runtime system will allow us to perform a detailed performance evaluation, identify major sources of overhead, and optimize accordingly. Developing optimization passes over information flow operations is one way that we hope will help to improve performance.

One limitation of the QEMU user mode emulator is that there is limited support for multi-threaded programs. To deal with this, we plan to extend our system to support the QEMU whole-system emulator. With this enhancement, we will have the ability to perform detailed security analyses for entire operating systems, regardless of the ISA that they are compiled to run on. This will enable studies in the area of operating system security, including exploit detection and vulnerability discovery. Our architecture-independent approach will allow us to perform important analyses for embedded systems ISAs, where support for dynamic information flow tracking is limited.

7 Related Work

Dynamic information flow tracking has been shown to have a wide variety of real world applications, including the detection and prevention of exploits for binary programs [7,20,31] and web applications [27]. Applications to malware analysis include botnet protocol reverse engineering [6,30], and identifying cryptographic primitives in malware [17]. For debugging and testing assistance, dynamic information flow tracking can be used to visualize where information flows in complex systems [18], or to improve code coverage during testing [15]. Additionally, this technique can be used for automated software vulnerability discovery [12,29].

Information flow tracking has been implemented in a variety of ways, including at the hardware level [26,28], in software through the use of source-level instrumentation [12,15,31], binary instrumentation [10,13,22], or the use of a whole-system emulator [9,20,24]. Additionally, it can also be implemented at the Java VM layer [11]. Between all of these different implementations, the most practical approach for analyzing real-world software (malicious and benign) when source code is not available is to perform information flow tracking at the binary level. PIRATE is the first information flow tracking system that operates at the binary level, supports multiple ISAs, and can be extended in a straightforward manner to at least a dozen ISAs.

Existing systems that are the most similar to ours are Argos [20], BitBlaze [24], Dytan [10], and Libdft [13]. These systems have contributed to significant results in the area of dynamic information flow tracking. Argos and BitBlaze are implemented with QEMU [4], while Dytan and Libdft are implemented with Pin [16]. Even though QEMU and Pin support multiple guest ISAs, each of these information flow tracking systems are tightly coupled with x86, limiting their applicability to other ISAs.

Intermediate representations have been shown to be useful not only in compilers, but also in software analyses. Valgrind [19] employs an intermediate representation, but it is also limited to user-level programs which would prevent us from extending our work to entire operating systems. Valgrind also employs a shadow memory, but no tools exist that perform information flow tracking with the detail that we do in an architecture-neutral way. BAP [5] defines an intermediate representation for software analysis, but that system currently can only analyze x86 and ARM programs, and it doesn't have x86_64 support. CodeSurfer/x86 [21] shows how x86 binaries can be statically lifted to an intermediate representation enabling various static analyses on x86 binaries.

8 Conclusion

In this paper, we have presented PIRATE, an architecture-independent information flow tracking framework that enables dynamic information flow tracking at the binary level for several different ISAs. In addition, our combined static and dynamic analysis of helper function C code enables us to track information that flows through these complex instructions for each ISA. PIRATE enables us to decouple all of the useful applications of dynamic information flow tracking from specific ISAs without requiring source code of the programs we are interested in analyzing. To demonstrate the utility of our system, we have applied it in three security-related contexts; enforcing information flow policies, characterizing algorithms, and diagnosing sources of state explosion. Using PIRATE, we can continue to build on the usefulness of dynamic information flow tracking by bringing these security applications to a multitude of ISAs without requiring extensive domain knowledge of each ISA, and without the extensive implementation time required to support each ISA.

References

1. Clang: A C language family frontend for LLVM, http://clang.llvm.org
2. OpenSSL cryptography and SSL/TLS toolkit, http://openssl.org
3. Austin, T., Breach, S.E., Sohi, G.S.: Efficient detection of all pointer and array access errors. Tech. Rep., University of Wisconsin-Madison (1993)
4. Bellard, F.: QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference (2005)
5. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: A Binary Analysis Platform. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 463–469. Springer, Heidelberg (2011)

6. Caballero, J., Poosankam, P., Kreibich, C., Song, D.: Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In: Proceedings of the 16th ACM Conference on Computer and Communications Security (2009)
7. Chen, S., Xu, J., Nakka, N., Kalbarczyk, Z., Iyer, R.K.: Defeating memory corruption attacks via pointer taintedness detection. In: International Conference on Dependable Systems and Networks (2005)
8. Chipounov, V., Kuznetsov, V., Candea, G.: S2E: A platform for in-vivo multi-path analysis of software systems. In: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (2011)
9. Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., Rosenblum, M.: Understanding data lifetime via whole system simulation. In: Proceedings of the 13th USENIX Security Symposium (2004)
10. Clause, J., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis (2007)
11. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (2010)
12. Ganesh, V., Leek, T., Rinard, M.: Taint-based directed whitebox fuzzing. In: IEEE 31st International Conference on Software Engineering (2009)
13. Kemerlis, V.P., Portokalidis, G., Jee, K., Keromytis, A.D.: libdft: Practical dynamic data flow tracking for commodity systems. In: Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (2012)
14. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (2004)
15. Leek, T., Baker, G., Brown, R., Zhivich, M., Lippman, R.: Coverage maximization using dynamic taint tracing. Tech. Rep. MIT Lincoln Laboratory (2007)
16. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (2005)
17. Lutz, N.: Towards Revealing Attackers' Intent by Automatically Decrypting Network Traffic. Master's thesis, ETH Zurich (2008)
18. Mysore, S., Mazloom, B., Agrawal, B., Sherwood, T.: Understanding and visualizing full systems with data flow tomography. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (2008)
19. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (2007)
20. Portokalidis, G., Slowinska, A., Bos, H.: Argos: an emulator for fingerprinting zero-day attacks. In: Proceedings of the 1st ACM SIGOPS European Conference on Computer Systems (2006)
21. Reps, T., Balakrishnan, G., Lim, J.: Intermediate-representation recovery from low-level code. In: Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (2006)

22. Saxena, P., Sekar, R., Puranik, V.: Efficient fine-grained binary instrumentation with applications to taint-tracking. In: Proceedings of the 6th IEEE/ACM International Symposium on Code Generation and Optimization (2008)
23. Slowinska, A., Bos, H.: Pointless tainting? evaluating the practicality of pointer tainting. In: Proceedings of the 4th ACM European Conference on Computer Systems (2009)
24. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A New Approach to Computer Security via Binary Analysis. In: Sekar, R., Pujari, A.K. (eds.) ICISS 2008. LNCS, vol. 5352, pp. 1–25. Springer, Heidelberg (2008)
25. Stallings, W., Brown, L.: Computer Security: Principles and Practice. Pearson Prentice Hall (2008)
26. Suh, G.E., Lee, J.W., Zhang, D., Devadas, S.: Secure program execution via dynamic information flow tracking. In: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (2004)
27. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: TAJ: Effective taint analysis of web applications. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (2009)
28. Venkataramani, G., Doudalis, I., Solihin, Y., Prvulovic, M.: FlexiTaint: A programmable accelerator for dynamic taint propagation. In: Proceedings of the 14th International Symposium on High Performance Computer Architecture (2008)
29. Wang, T., Wei, T., Gu, G., Zou, W.: TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: Proceedings of the 31st IEEE Symposium on Security & Privacy (2010)
30. Wang, Z., Jiang, X., Cui, W., Wang, X., Grace, M.: ReFormat: Automatic Reverse Engineering of Encrypted Messages. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 200–215. Springer, Heidelberg (2009)
31. Xu, W., Bhatkar, S., Sekar, R.: Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In: Proceedings of the 15th USENIX Security Symposium (2006)

On the Determination of *Inlining Vectors* for Program Optimization

Rosario Cammarota[1], Alexandru Nicolau[1], Alexander V. Veidenbaum[1], Arun Kejariwal[2], Debora Donato[2], and Mukund Madhugiri[2]

[1] University of California Irvine
{rosario.c,nicolau,alexv}@ics.uci.edu
[2] Yahoo! Inc.
{akej,donato,mukundm}@yahoo-inc.com

Abstract. In this paper we propose a new technique and a framework to select inlining heuristic constraints - referred to as an *inlining vector*, for program optimization. The proposed technique uses machine learning to model the correspondence between inlining vectors and performance (*completion time*). The automatic selection of a machine learning algorithm to build such a model is part of our technique and we present a rigorous selection procedure. Subject to a given architecture, such a model evaluates the benefit of inlining combined with other global optimizations and selects an inlining vector that, in the limits of the model, minimizes the completion time of a program.

We conducted our experiments using the GNU GCC compiler and optimized 22 combinations *(program, input)* from SPEC CINT2006 on the state-of-the-art Intel Xeon Westmere architecture. Compared with optimization level, i.e., `-O3`, our technique yields performance improvements ranging from 2% to 9%.

1 Introduction

The widespread use of object-oriented programming models and software engineering methodologies often leads to complex program structures that are composed of a multitude of functions and source files. The presence of these files and functions unfortunately limits the scope of global optimizations and their forced separate compilation reduces the performance in complex and unpredictable ways. As a result, when relying on current compiler technologies and rigid compiler heuristics, programs achieve in practice only a portion of the performance that they could in principle achieve on a given architecture.

Function inlining [1] provides a simple - in principle - way of overcoming these barriers to program optimization - it removes the boundaries of function calls by expanding call sites with the body of the callee - but it too is not without pitfalls. Prior studies acknowledge the potential that function inlining offers at compile time to other optimizations [2–4], parallelization [5] and vectorization [6]. The importance of function inlining is also evidenced by the fact it is used in all optimizing compilers - e.g., the GNU GCC, Intel ICC, IBM XLC - as a first pass of their aggressive optimization levels such as `-O3`. Unfortunately, inlining in an effort to enable maximal optimization to the code has been shown to be NP-Complete [7, 8] and while heuristics are implemented by all good optimizing compilers, the number of constraints and their combinations to the process are extremely large - making the achievement of best, or even good inlining for a given application, practically unachievable. The above represents a limitation to the direct application of specialized search techniques [9, 10] to the case of static compilation of production software written in C or C++. For a given program, searching the space of inlining vectors would involve a large number of expensive recompilations and runs. For example, the number of inlining constraints vary from compiler to compiler - GCC v3.4 has 4 inlining constraints, whereas GCC v4.5 has 7 inlining constraints - and while in practice the range of values of each constraint is somewhat limited (because of compiler malfunctioning), the number of combinations is still enormous. In this work - refer to Section 4.2, we estimate the number inlining vectors to be $> 10^{15}$.

Despite the importance of function inlining, the influence of inlining constraints on performance of arbitrarily complex programs and on high-performance architectures had received little attention in the past. Cavazos and O'Boyle in [9] developed a technique for dynamically tuning the JikesRVM [11] inlining heuristics and selectively inline frequently used functions. However, in the case of static compilation, inlining decisions incrementally influence any inlineable functions within each module [7, 10], making the problem of selecting inlining vectors more complex than that of inlining frequently used functions. Cooper et. al [10] proposed a new heuristic for function inlining for C programs and a technique for auto-tuning its parameters using genetic algorithms. While such an heuristic performs better inlining decisions compared with the inlining heuristics implemented in production compilers, e.g., GCC v3.3, the technique proposed in [10] does not account for separate compilation, where additional barriers to global optimization are the boundaries of source files.

In this paper we develop a new technique for function inlining, given a compiler and its parameterized heuristic. The proposed technique attempts to achieve the best settings of the inlining constraints - referred to as an *inlining vector*, that will result in the most efficient (*fastest*) execution of the code produced by this compiler. Our technique is suitable to be used in industry settings as a support to production compilers as it does not modify the compiler. On the contrary, the technique proposed in this work suggests inlining vectors for program optimization and provides an estimation of the performance that can be achieved after recompiling a program using the predicted inlining vectors.

The proposed technique is implemented in as a performance framework whose design is shown in Figure 1 and is composed of two main passes. In the first pass, it uses machine learning to model the relation between inlining vectors - **iv**s, and performance (completion time) - p. The fact that the completion time of a program is a continuous quantity leads us to the adoption of supervised learning algorithms to build regression models. As part of our technique we provide a quantitative procedure to evaluate and select a regression algorithm, amongst

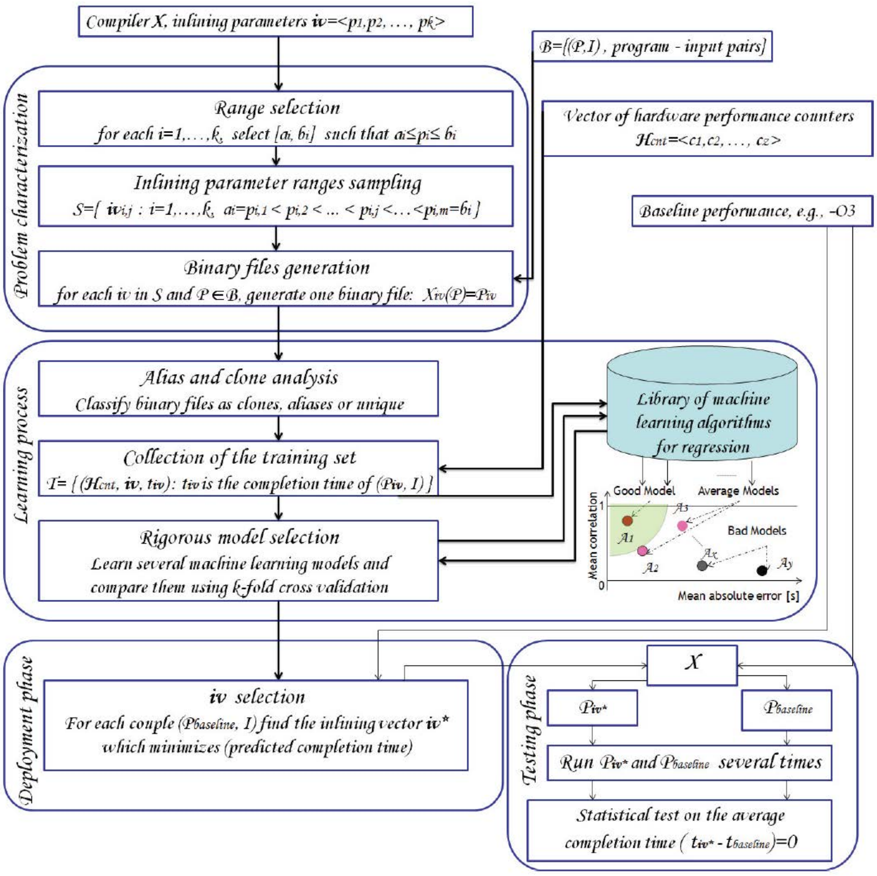

Compiler X, inlining parameters iv=<p1,p2, ..., pk>
B={(P,I), program - input pairs}
Problem characterization
Range selection
for each i=1,...,k, select [ai, bi] such that ai≤pi≤ bi
Inlining parameter ranges sampling
S={ ivi,j : i=1,...,k, ai=pi,1 < pi,2 < ... < pi,j <...<pi,m=bi }
Binary files generation
for each iv in S and P∈B, generate one binary file: Xiv(P)=Piv
Vector of hardware performance counters
Hcnt=<c1,c2, ..., cz>
Baseline performance, e.g., -O3
Learning process
Alias and clone analysis
Classify binary files as clones, aliases or unique
Collection of the training set
T= { (Hcnt, iv, tiv): tiv is the completion time of (Piv, I) }
Rigorous model selection
Learn several machine learning models and compare them using k-fold cross validation
Library of machine learning algorithms for regression
Good Model
Average Models
Bad Models
Mean correlation
Mean absolute error [s]
A1
A2
A3
Ax
Ay
Deployment phase
iv selection
For each couple (Pbaseline, I) find the inlining vector iv* which minimizes (predicted completion time)
Testing phase
X
Piv*
Pbaseline
Run Piv* and Pbaseline several times
Statistical test on the average completion time (tiv* - tbaseline)=0

Fig. 1. Performance framework design

many regression algorithms, that best models the relation $\hat{h} : \mathbf{iv} \rightarrow p$ subject to a number of training samples - referred to as *training set*. Such a procedure is based on 10-fold cross validation [12] that, in our technique, is used to evaluate and compare different models in terms of mean absolute error - i.e., the average accuracy of performance prediction, and mean coefficient of correlation - i.e., the average correlation/alignment between a batch of performance predictions and the actual batch of performance - refer to Appendix A. Once assigned a threshold on the maximum admissible mean absolute error, the model corresponding to the algorithm with maximum mean coefficient of correlation is selected amongst the models whose mean absolute error is within the assigned threshold.

In the second pass of our technique, the selected model is leveraged to search for an inlining vector - $\mathbf{iv}^*$, which is able to maximize predicted and ultimately achieved performance of a program subject to a given workload (input). At the cost of training a model using a relatively limited number of compilations and runs, such a model supplies for the intractability of searching the space of inlining vectors without performing additional recompilations and runs.

In our experiments, we used GNU GCC version 4.5 - referred to as GCC v4.5 - to optimize 22 combinations *(program, input)* on the state-of-the-art Intel Xeon Westmere architecture. These combinations belong to four programs selected from SPEC CINT2006 (refer to Section 4.1 and [13]) because these programs have a larger number of input files, compared with other applications in the same benchmark suite, and because their behavior is highly influenced by the selection of a particular input file [14]. While the optimization of a program subject to a workload may seem a limiting factor of the proposed technique, in industry settings programs are purposely manually optimized for certain a certain class of workloads [14]. We train our technique using a limited number of instances of the 22 combinations ($program, input$), for a limited number of training inlining vectors - refer to Section 3.2. Our technique identifies random forest - `M5P` [15] - as a suitable algorithm to model the relation between inlining vectors and completion time. Next, the performance model is used to predict inlining vectors to optimize the performance of each combination ($program, input$). Lastly, we recompile our programs using the inlining vectors predicted above. Experimental results show: (1) Performance improvements ranging from 2% to 9% on the state-of-the-art Intel Xeon Westmere architecture and without manually modifying neither the structure of the program source code nor that of the compiler; (2) Performance results outperform both the baseline performance and the best performance recorded during the training phase.

To the best of our knowledge, this is the first paper investigating the influence of inlining vectors on program performance and in the presence of separate compilation. This is also the first paper proposing a rigorous procedure to build and enforce machine learning based performance models. The contributions of this paper can be summarized as follows:

I - We propose a new machine-learning based technique to select inlining vectors for program optimization and implement the proposed technique in the performance framework illustrated in Figure 1. Our technique relies on the selection of a performance model to quantify the influence of inlining vectors on program performance. The presence of such a model represents a practical mean to perform a fast exploration of the a large set of inlining vectors, whereas the model is built using a limited number of training examples - refer to Section 3.3.

II - We propose a rigorous procedure to select a machine learning algorithm to build the model above. Using 10-fold cross-validation, such a procedure focuses on algorithms providing models whose performance prediction accuracy is below an admissible threshold - which is a parameter of our framework. Amongst these model, our procedure selects the model with highest mean coefficient of correlation.

The rest of the paper is organized as follows: In Section 2 we describe the opportunity of improving performance by determining inlining vectors and motivate the need of using machine learning to address the problem of selecting

Table 1. `401.bzip2`, hot functions

Source file:Function name	Size [kB]	# of calls	Coverage
`compress.c:BZ2_compressBlock`	15.76	143	28.75
`blocksort.c:fallbackSort`	2.48	34	27.72
`decompress.c:BZ2_decompress`	10.92	17681	16.00
`blocksort.c:mainSort`	4.45	143	9.71
`blocksort.c:mainGtU`	0.55	1515464	7.96

Table 2. `401.bzip2`, hot paths on the precise dynamic call graph

Source file:Caller name	Source file:Callee name	# of edge traversals	Coverage [%]
`bzip2.c:main`	`bzip2.c:compressStream`	3	79.14
`bzlib.c:BZ2_bzCompress`	`bzlib.c:handle_compress`	36345	78.95
`bzlib.c:compressStream`	`bzlib.c:BZ2_bzWrite`	18876	77.99
`bzlib.c:BZ2_bzWrite`	`bzlib.c:BZ2_bzCompress`	35986	77.93
`bzlib.c:handle_compress`	`compress.c:BZ2_compressBlock`	143	74.61

inlining vectors. In Section 3 we present our technique, including (a) a quantitative procedure to build and evaluate a suitable performance prediction model to select inlining vectors; (b) the detailed discussion on how to build the training set. In Section 4, we describe the experimental setup, the experiments and discuss experimental results. We highlight both the significance of our performance improvements and trends that are common to the predicted inlining vectors, subject to the architecture and the applications in use. Related work is discussed in Section 5. Key highlights and conclusion are remarked in Section 6.

2 Motivation

In this Section we use the program `401.bzip2` from the industry standard benchmark suite SPEC INT2006 [13] as an example program to show opportunities to improve program performance by properly selecting inlining vectors. We compile the program using the compiler optimization level `-O3` and refer the corresponding binary file to as $\texttt{401.bip2}_{\texttt{O3}}$. We use Trin-Trin [16] to collect the dynamic call graph of the program subject to one of the reference inputs. Trin-Trin allows the identification of hot functions - refer to Table 1, and hot paths - refer to Table 2. Figure 2 shows a snapshot of the call graph concentrated in the proximity of the hot functions and including the hot paths. `401.bzip2` has a peaked profile [17] as it spends most of its execution time in the functions `blocksort.c:fallbackSort` and `compress.c:BZ2_compressBlock`. The dynamic call graph of the binary $\texttt{401.bip2}_{\texttt{O3}}$ is composed of 59 nodes - where a node represent a function, 69 edges, $2,454,278$ directs call and 36 indirect calls. Given the limited number of hot functions and hot paths, we manually searched for opportunities for performance improvement coming from the reduction of the number of function calls. In Table 1, Table 2 and Figure 2 we list the name of the source file containing the implementation of a given function, along with the name of the function. We can distinguish three groups of functions belonging to the source files `bzip2.c`, `bzlib.c` and `compress.c`. The function `BZ2_compressBlock` calls only the function `BZ_blockSort` withing a single loop. The inlining of the latter function in the former would save 143 function calls, with an increase of the size of the caller of $\approx$ 16 `kB`. However, because these two functions appear in different source files (and modules), the decision to inlining `BZ_blockSort` in its caller is never evaluated by the inliner. [1] Nevertheless, within the same source file (or module) there can be inlining decisions that the inliner misses because of inappropriate settings of its inlining parameters. For example, the function `blocksort.c:mainGtu` is a leaf function in the call graph that is called 1,515,464 times. The large call count stems from three different call sites embodied in three different loops. In this case there is a trade-off between increasing the size of the caller by roughly 1.65 `kB` and reducing the overhead of 1,515,464 function calls. This is a missed inlining opportunity for improving performance due to the default inlining settings.

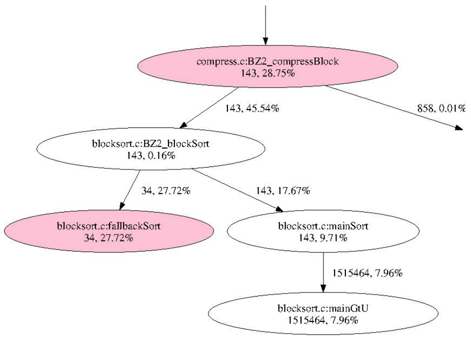


Fig. 2. `401.bzip2` call graph snapshot

The analysis that we carried out for `401.bzip2` would not have been possible for other programs from SPEC CINT2006, because of the large number of functions and files composing their source code. For example, the dynamic call graph of `403.gcc` accounts 2,072 nodes, 7,868 edges, 216,947,768 function calls and a rather flat profile [17]. Such a complexity requires to approach our problem from different angles and with automatic methodologies that can learn from examples and are capable of predicting optimal inlining settings per program.

Lastly, we show that an obvious solution to tune inlining heuristics is not suitable to approach program optimization. For this, we study the variation in size and performance of `401.bzip2` with the increase of one inlining parameter ruling indiscriminate inlining, `max_inlining_insns_auto`. All the functions whose estimated cost - i.e., pseudo-instructions count [19] - is less than the value assigned to `max_inlining_insns_auto` are inlined. Therefore, the larger is the value assigned to `max_inlining_insns_auto`, the more functions are inlined, independently of the values of other inlining parameters. [2] One could potentially argue that the more functions calls are eliminated, the more performance of a program varies and improves. To show that this is not the case, we assigned

[1] In commercial and open source compilers the inliner works on the intermediate representation and so the inliner acts separately on single files or on a single module. The case of link-time optimization - e.g., GCC LTO [18] - is left to future work as we believe that our technique would speedup performance of GCC LTO.

[2] The list of inlining parameters of `GCCv4.5` is at `http://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/Optimize-Options.html`

values from 10 to 190, with step 10, to the parameter `max_inlining_insns_auto` and compile 19 binary versions of `401.bzip2`. These 19 versions falls in four bins of different sizes 75 `kB`, 79 `kB`, 83 `kB` and 87 `kB` and not all the version of the binary files are unique. After 110 there is no variation in size, whereas after 190 the compiler exhibits malfunctioning. The performance of the binaries generated with different values of `max_inlining_insns_auto` as discussed above is slower ≈ 3 to 6% than performance achieved by the baseline `-O3`. Only in one case and for the first input there is a speedup of $\approx 2\%$.

3 Technique

In this section we first establish the background to build our technique, including the notations and the terminology used through the paper, and then we describe our technique to select inlining vectors for program optimization.

3.1 Notations

In this paper we denote an inlining vector as $\mathbf{iv}$, whereas we denote an inlining vector selected for optimizing a certain program as $\mathbf{iv}^*$. The set of all possible inlining vectors is denoted as F . While F is in principle an unbounded set - as each component of the inlining vector can assume any positive integer value, in practice the search for inlining vectors is confined into a subset of F that is practically bounded due to compiler malfunctioning. Any limited the subset of F that our technique explores to determine inlining vectors for program optimization is denoted as $\hat{F}$.

A machine learning algorithm is denoted as $\mathcal{A}$, whereas the model built with the algorithm $\mathcal{A}$ is denoted as $\hat{h}_{\alpha,\rho}$ - and is referred to as an hypothesis. An hypothesis is characterized by a number of parameters indicating its quality. As a measure of the quality of an hypothesis, in this work we use the mean absolute error - denoted as α, and the mean coefficient of correlation - denoted as ρ. The training set - that is the set of sample runs used to train our model - is denoted as T and is composed of pairs $(program_{\mathbf{iv}}, p)$, where several instances of programs subject to different inputs are compiled with a limited number of inlining vectors and executed. p is the completion time corresponding to a particular execution. During the execution of an instance of $program_{\mathbf{iv}}$, a vector of hardware performance counters is collected. Such a vector is denoted as $\mathbf{cnt}_{\mathbf{iv}}$ and features the run-time behavior of the program subject to an inlining vector. The set of $\mathbf{iv}$s used to build the training set is within a limited subset of $\hat{F}$ and is denoted as $\mathcal{M}$.

3.2 Collection of the Training Set

The training set for building our model is populated as follows. A set of programs is compiled using $\mathcal{M}$ inlining vectors to generate an equal number binary files per program. A subset of these binary files are executed. A vector of hardware performance counters and the completion time of each run is recorded. Each component of these inlining vectors assumes values in a sequence of integer values extracted from a limited range of integer values. These ranges are upper-bounded in practice by the maximum integer value that produces a valid binary file for a given program. Within each range we select a number of consecutive values uniformly spaced and the combinations of these values, from all the ranges, compose the set of $\mathbf{iv} \in \mathcal{M}$. However, not all these combinations of inlining parameters produce distinct binary files and amongst distinct binary files not all of these have distinct file sizes. In this work, distinct binary files with distinct sizes are referred to as *unique* files; distinct binary files with same sizes are referred to as *alias* files; other binary files that are identical byte-by-byte are referred to as *clone* files. To build the training set our technique first produces $\mathcal{M}$ binary files and second it dissects these files in *unique*, *alias* and *clone* files. While the process still involves a large number of compilations, such a dissection reduces the number of runs to the execution of only the *unique* files to populate the training set - *alias* files achieve nearly equal performance subject to the same workload. Therefore we decided not to include aliases in the training set.

The procedure to dissect the binary files in *unique*, *alias* and *clone* files is implemented as follows in our framework. The available binary files are scanned and a table indexed using the binary size is maintained. Each entry of the table is a tuple `<key, list of values>`, where the key is binary size and the list of values contains the names of alias and clone files. When a new binary is examined, its size is measured and used to access the table. If there is not an entry in the table for this key, then the binary is marked as *unique* and a new entry `<key, list of values>` is added to the table. If such an entry exists, the binary is appended to the list of values and marked as an *alias*. To test if the new binary is *unique*, it is compared byte-to-byte with the binary file corresponding to the first entry of the list.[3] If the comparison is positive, the new binary is re-marked as a clone. The list of values is kept to provide statistics about the clone, alias and unique files, whereas the first element of each list is a unique binary that will be used to populate the training set. The training set is populated as follows: for each element in the first column of the table, execute the corresponding binary file and record a vector of hardware performance counters - a parameter of our framework - and the corresponding performance.

3.3 Hypothesis Selection

Once the training set is available, our technique builds hypotheses from different types of regression algorithms. Let us denote these algorithms as $\mathcal{A}_1, \mathcal{A}_2, \cdots, \mathcal{A}_S$. For example, $\mathcal{A}_1$ could represent a regression tree such as C4.5 [20], $\mathcal{A}_2$ could be support vector regression [21], etc. Each algorithm is trained using the training set to build one (or more) hypotheses. Each hypothesis is subsequently validated

[3] Byte-to-byte comparison is a simple, yet convenient way to classify binary files as *unique*, *alias* and *clones*. Alternatively, a digest such as `SHA-2` (Secure Hash Algorithm version 2) can be used for such a classification.

using 10-fold cross-validation, as discussed in the next Section and Appendix A, and is characterized with a coefficient of correlation and a mean absolute error. We denote the hypothesis associated to each algorithm as $\hat{h}_{\alpha_i,\rho_i}$, with $i = 1, 2, \cdots, S$.

Given a threshold z - corresponding to the maximum acceptable mean absolute error - each hypothesis whose mean absolute error is greater than the z and whose coefficient of correlation is negative is rejected. In our technique, we assign a value to z according to the range of the outcomes of our experiments at the baseline - the baseline is a parameter of our framework. For example, let us assume `-O3` to be the baseline. If our $(program_{\texttt{O3}}, input)$ pair runs for 500 seconds, a threshold tolerating an error of 1% on performance prediction would be set as $z = 5$ seconds. Therefore, all the hypotheses exhibiting an average mean absolute error lower than 5 seconds and exhibiting a positive coefficient of correlation are good candidates to model our experiments.

Amongst the remaining hypotheses, the one with maximum mean coefficient of correlation - and not necessarily with minimum mean absolute error - is selected.[4]

3.4 Model Validation

The evaluation of one or more regression algorithms is usually performed via a standard statistical technique called k-fold cross-validation [12]. In k-fold cross-validation, the training set is first partitioned into k nearly equally sized segments or folds. Subsequently k iterations of training and validation are performed such that within each iteration a different fold of the data is taken out for validation while the remaining $k-1$ folds are used to train a regression model. At each iteration, one or more regression algorithms are trained using $k-1$ folds of data to build one or more hypotheses. These hypotheses are subsequently used to make predictions using the features in the validation fold - refer to Appendix A for an example.

The performance of each learning algorithm on each fold can be assessed using performance metrics, such as the mean correlation coefficient and mean absolute error. At the end of the cross-validation process, k samples of each performance metric are available for each regression model. Different methodologies, such as the average of the k samples, can be used to obtain an aggregate measure from these samples, or these samples can be used in a statistical hypothesis test to show that one regression model is better to another.

In this paper we consider mean absolute error and mean coefficient of correlation as metrics to evaluate and to select a suitable regression model for our analysis. The former represents the average accuracy of performance prediction, whereas the latter represents the average correlation/alignment between a batch of performance predictions and the actual batch of performance - refer to Appendix A.

[4] In the case of hypotheses with the same mean absolute error and/or the coefficient of correlation, a test of significance can be executed to select an hypothesis with the most significant mean [22].

3.5 Selection of an Inlining Vector

Without additional recompilations and runs, the hypothesis enables a rapid search of the space of inlining vectors and the determination of the best inlining vector to optimize performance of a given program subject to a given workload. Let us denote the selected hypothesis $\hat{h}^*$. The hypothesis predicts performance for unseen inlining vectors in $\hat{F}$. Therefore the hypothesis is used to search for the inlining vector that maximizes predicted performance and outperforms an assigned baseline. In other words our technique leverages the hypothesis to solve the following problem: select one inlining vector to minimize the completion time of a given combination $(program, input)$. Eventually, our framework verifies the efficacy of the predicted inlining vector by recompiling the program using the inlining vector $\mathbf{iv}^*$ and measuring its performance on an average of several runs compared with the performance of the baseline.

4 Experiments

We implemented the technique presented in Section 3 as a performance framework written in Perl [23] and R [24–26]. The experiments we carried out to evaluate the efficacy of the proposed technique are presented in this section.

4.1 Experimental Setup

We used the state-of-the-art Intel Xeon Westmere architecture for our experiments - refer to Table 3. We evaluated the proposed technique optimizing 22 combinations $(program, input)$ from SPEC CINT2006 - refer to Table 4. The combinations above belong to four programs selected from SPEC CINT2006 [13], because these programs have a larger number of input files, compared with other applications in the same benchmark suite, and because their behavior is highly influenced by the selection of a particular input file [14]. The components of the inlining vector for GNU GCC v4.5 and the ranges estimated by our technique are shown in Table 5.

To represent the run-time behavior of a binary compiled with a certain inlining vector, we select a vector of hardware counters - **cnt** - composed of the following components:

Table 3. System configuration

Model	Intel X5680 (Westmere) @ 3.33GHz
L1 I/D cache [kB]	32
L2 cache [kB]	256
L3 cache (shared) [MB]	12
Main Memory [GB]	24
Compiler	GNU GCC 4.5
Baseline optimization level	`-O3`
Operating system	Linux Red Hat AS 4 update 7

Table 4. Combinations *(program, input)*

Program name	**Application domain**	**Input**
401.bzip2	Compression	I1:chicken.jpg I2:control I3:input.source I4:liberty.jpg I5:text.html
403.gcc	C Language Optimizing Compiler	I1:166.i I2:200.i I3:c-typeck.i I4:cpdecl.i I5:expr.i I6:expr2.i I7:g23.i I8:s04.i I9:scilab.i
445.gobmk	Artificial intelligence	I1:13x13.tst I2:nngn.tst I3:score2.tst I4:trevorc.tst I5:trevord.tst
464.h264ref	Video compression	I1:foreman_ref_encoder_baseline.cfg I2:foreman_ref_encoder_main.cfg I3:sss_encoder_main.cfg

Table 5. Practical search space and $\hat{F}$ for our programs

iv component	**Range**
`inline_call_cost`	{10}
`max_inline_insns_auto`	[10-190]
`large_function_insns`	[1100-3100]
`large_function_growth`	[20-100]
`large_unit_insns`	[6,000-16,000]
`inline_unit_growth`	[30-300]
`inline_recursive_depth`	[4-8]

– `CPI` : Cycles per instruction.
– `Br rate` [%] : Number of branch retired as a percentage of the total instructions retired.
– `L2 miss` [‰] : The count of L2 cache misses per thousand instructions.
– `L3 miss` [‰] : The count of L3 cache misses per thousand instructions.

While the second component of the vector of counters, i.e., `Br rate`, includes the variation in the number of function calls due to a different amount of inlining, the other components indicates indirectly the influence of global optimizations to local performance, i.e., `CPI`, and the memory hierarchy behavior, i.e., `L2 and L3 miss`. Admittedly, the vector of hardware counters selected in this study may be complemented with other counters aimed to capture more specific run-time features. However, for the architecture considered in this paper, the counters above can be collected in a single run. This alleviates the problem(s) which may arise due to various sources of inaccuracies in the process of collection of the hardware performance counters [27]. Furthermore, the counters are normalized to instructions retired with different scale factors - % in the case of branch retired and ‰ in the case of the cache misses - because for ordinary programs,

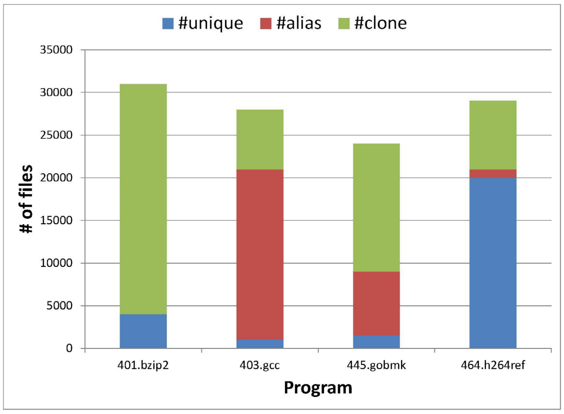


Fig. 3. Binary files dissection in *unique*, *alias* and *clones*

the cache miss count is usually one order of magnitude lower than the count of other hardware events [28]. We use Intel VTune [29] to collect **cnt**.

4.2 Inlining Parameter Ranges and the Set $\hat{F}$

We compile $\approx 30,000$ binary file per program - the size of $\mathcal{M} \approx 30,000$ - and dissect the number of unique, clone and alias files as it is shown in Figure 3.

In our technique, the set of inlining vectors used to build the training set is limited to the unique and not alias binary files for each program to optimize - $\approx 26,000$ unique binary files in total, corresponding to $\approx 20,000$ inlining vectors. However, inlining vectors for program optimization are determined in a larger set $\hat{F}$ composed of $> 10^{15}$ inlining vectors - refer to Table 5.

4.3 Baseline

In our experiments we set `-O3` as the baseline performance for our framework. Our framework compiles the programs using `-O3` and runs them to assess the baseline performance for each reference input. **cnt** is collected for the baseline. In the program selected for our experiments, each reference input exercises a different and/or the same path of the call graph of a program, but in different ways that are captured by the values of the components of **cnt**. For example, (`403.gcc`, I6)=`<1.24, 25.11, 1.98, 0.12>` and (`403.gcc`, I7)=`<1.42, 26.32, 2.89, 0.22>`. Furthermore, our framework provides the percentage of total cycles spent in program routines versus external library/system routines - referred as cycles breakdown. The cycles breakdown is an indication of opportunities for performance optimization exploitable at compile-time. Arguably, the more the program runs into external/system routines the less opportunities exist for performance optimization exploitable at compile-time. Our framework shows that the average number of cycles - average on the reference input - spent within program routines ranges from 45% to 98%. It follows the presence of opportunities for program optimization for our set of programs.

4.4 Collection of the Training Set

Our framework compiles each program times the size of $\mathcal{M}$ - where size of $\mathcal{M} \approx 30,000$. According to the analysis explained in Section 3 and for each program, our framework separates unique binary files from the binary files compiled. Next, our framework populate the training set. For each combination $(program_{\mathbf{iv}}, input)$, the framework runs the unique binary files and collects (a) the completion time expressed in second, and (b) the vector **cnt**.

4.5 Model Selection

Once the training set is available, the framework selects the regression algorithm that best models our experimental setup.[5] In this section we report the comparison between three regression algorithms - Least Median Square - `LeastMedSq` [30], Radial Basis Function network - `RBFnetwork` [31], and Random Forest for regression - `M5P` model tree [15]. These algorithms belong to three different types of regression algorithms. `LeastMedSq` builds a linear function as an hypothesis, whereas `RBFnetwork` builds a form of artificial neural network for regression, whereas `M5P` builds a form of random forests for regression. The framework compares `LeastMedSq`, `RBFnetwork` and `M5P` in terms of their mean absolute errors and coefficients of correlation - refer to Table 6. The model selection is favorable to the adoption of `M5P`, which is the algorithm exhibiting the lower prediction error and the maximum coefficient of correlation.

Table 6. Hypotheses comparison and selection

`Learning algorithm`	(α, ρ)
`LeastMedSq`	$(0.27, 93.17)$
`RBFnetwork`	$(0.77, 38.67)$
`M5P`	$(0.99, 2.15)$

4.6 iv* Determination

After the model is selected, the framework leverages the model to find inlining vectors that minimizes the predicted completion time for each combination $(program_{\mathtt{O3}}, input)$. The predicted inlining vectors are listed in Table 7. The predicted inlining vectors never correspond to that of the baseline or to any others present in the training set.

On the current architecture the components of the predicted inlining vectors exhibit the following properties (*trends*):

[5] Our framework explores all the regression algorithms available in the default package Rweka [24] for the language R.

Table 7. $\mathbf{iv}^*$ determined per combination $(program, input)$; `Speedup` $= \frac{p_{\mathtt{O3}}}{p_{\mathtt{iv}^*}}$; $\mathbf{iv}_{\mathtt{O3}} = < 10, 120, 2000, 300, 10000, 150, 8 >$

$(program, input)$	$\mathbf{iv}^*$	$p_{\mathtt{O3}}$ [s]	$p_{\mathtt{iv}^*}$ [s]	Speedup
(401.bzip2,I1)	<10, 125, 2043, 197, 10000, 103, 8>	40.92	39.63	1.03
(401.bzip2,I2)	<10, 142, 2042, 299, 10000, 115, 8>	363.00	354.24	1.02
(401.bzip2,I3)	<10, 134, 2021, 185, 10000, 73, 8>	116.60	116.60	1.00
(401.bzip2,I4)	<10, 132, 2032, 147, 10000, 64, 8>	64.98	63.47	1.02
(401.bzip2,I5)	<10, 144, 2079, 159, 10000, 128, 8>	141.75	140.27	1.01
(403.gcc,I1)	<10, 126, 2415, 297, 6000, 135, 8>	36.15	35.34	1.02
(403.gcc,I2)	<10, 165, 2405, 296, 6000, 66, 8>	48.21	47.43	1.02
(403.gcc,I3)	<10, 127, 2524, 46, 6000, 103, 8>	69.07	66.70	1.04
(403.gcc,I4)	<10, 167, 2697, 207, 6000, 105, 8>	43.52	39.94	1.09
(403.gcc,I5)	<10, 167, 2693, 33, 6000, 95, 8>	48.56	47.61	1.02
(403.gcc,I6)	<10, 168, 2684, 183, 6000, 145, 8>	67.98	65.50	1.04
(403.gcc,I7)	<10, 152, 2373, 298, 6000, 106, 8>	85.07	83.18	1.02
(403.gcc,I8)	<10, 150, 2637, 127, 6000, 149, 8>	86.73	84.13	1.03
(403.gcc,I9)	<10, 138, 2691, 299, 6000, 44, 8>	16.25	16.08	1.01
(445.gobmk,I1)	<10, 129, 2004, 179, 6000, 40, 8>	72.54	70.78	1.02
(445.gobmk,I2)	<10, 169, 2690, 285, 6000, 149, 8>	183.71	179.00	1.03
(445.gobmk,I3)	<10, 157, 2070, 199, 6000, 123, 8>	97.62	91.36	1.07
(445.gobmk,I4)	<10, 134, 2088, 164, 6000, 148, 8>	71.54	70.00	1.02
(445.gobmk,I5)	<10, 156, 2603, 296, 6000, 97, 8>	97.90	95.50	1.03
(464.h264ref,I1)	<10, 154, 2688, 297, 6000, 149, 8>	81.48	81.41	1.00
(464.h264ref,I2)	<10, 169, 2053, 295, 6000, 124, 8>	60.78	60.00	1.01
(464.h264ref,I3)	<10, 120, 2078, 271, 6000, 75, 8>	527.06	515.00	1.02

- `max_inline_insns_auto`: the predicted values suggest that inlining functions indiscriminately is never the best option. However, it is beneficial to assign a value to this parameter that is larger than the default value.
- `large_function_insns`: the predicted values suggest that the presence of larger functions (large in terms of instructions count after inlining) is beneficial to performance. Indeed, it potentially exposes more opportunities for global optimizations.
- `large_function_growth`: the predicted values limits the size of these functions imposing more restrictive constraints than the baseline.
- `large_unit_insns`: the predicted values separate the programs in two groups. In the first group there is `401.bzip2`, where the default value corresponds to the predicted one. This indicates that this program is not sensitive to the variation of this parameter. The second group contains the other programs. The value of `large_unit_insns` is lower than that of the baseline suggesting that the inliner must exclude more functions from the inlining decisions to deliver better program performance.
- `large_unit_growth`: the predicted values suggest that the presence of smaller modules after compilation benefits performance on the target architecture.
- `inline_recursive_depth`: this parameter always assumes the default value, although there are recursive functions inlined within both `401.bzip2` or `403.gcc`. This indicates the performance variation induced by varying this parameter is predicted to be negligible.

4.7 Performance Results

Our framework feeds the compiler with the inlining vectors in Table 7 and recompiles the programs in Table 4. The compilation produces one binary file for each combinations (*program*, *input*). Hence, five versions of the program `401.bzip2`, nine versions of the program `403.gcc`, five versions of the program `445.gobmk` and three versions of the program `464.h264ref` are produced by our framework. Performance results - shown in Table 7 - exhibit modest yet positive speedups. This is in part due to the features of the architecture in use. Indeed, when compared with prior generations, Westmere has a deeper pipeline and large micro-architectural buffers to mask more miss-events; consistent chunks of the working set sizes fit into the large last level of cache; the presence of a cross-bar interconnection network, which replace the old front-side bus, provides fast access to main memory. Despite the massive presence of training examples that were not outperforming the baseline performance for each program, the proposed procedure for the hypothesis selection allows leveraging the information embedded in the model and selecting inlining vectors outperforming in both the best inlining vectors seen during the training phase and the baseline performance.

To support the performance results in Table 7 we repeated the execution of the baseline and the optimized program for > 50 times. Then we applied `t-test` [32] to evaluate the hypothesis of equality of the average execution times of the baseline and the predicted inlining vector. As a result, our statistical test rejects the null hypothesis with a confidence level of 95%. This, in turns assesses the significance of performance improvements from 2% to 9%. Furthermore, our technique never provides performance losses to any of the combinations (*program*, *input*) considered in this work.

5 Related Work

Prior studies acknowledge the potential that function inlining offers at compile time to other optimizations [2–4, 33], parallelization [5] and vectorization [6]. Despite its importance, most of the efforts of prior work on function inlining propose techniques to improve the accuracy of inlining decisions. Sheifler in [7] profile the cost of function calls and provide this information to the inliner. Dean and Chambers in [34] propose inlining trials. After inlining a call site, the caller is compiled and executed so to measure the profitability of the current inlining decisions. The inliner queries a database of inlining trials to decide to inlining a call site. This technique requires a compiler to be capable of compiling, execute and replay pieces of code. Hazelwood and Grove in [35] drive function inlining and specialization using call contexts. The technique proposed in this work focuses on the selection of inlining constraints, i.e., an inlining vector, to minimize the completion time of a program subject to a given workload on any architecture. Little attention has been paid in the past to investigate the influence of inlining constraints on performance of arbitrarily complex programs, i.e., composed of several functions and source files. In particular, the techniques referenced above do not account for the mutual influence among inlining decisions during the compilation of a module. In this work, we deal explicitly with such a complex aspect of automatic inlining with the assistance of a machine learning model to select inlining vectors for program optimization.

Cavazos and O'Boyle in [9] presented a technique that uses genetic search to dynamically tune JikesRVM inlining heuristics and selectively inline frequently used methods of Java programs. Profiling of frequently used methods in programs with long uptime - e.g., application server, ensures in practice the convergence of the search within the uptime of the application. Cooper et al. in [10] extended the source-to-source C inliner proposed by Davidson and Holler in [1], introducing the notion of `condition string` - a vector of static and dynamic program properties designed to characterize the call sites of a program and to allow fine-grain inlining decisions. Selecting condition strings is as complex as selecting inlining vectors and specialized search methods may represent a solution to the problem. Hill climbing is used in [10] to explore the space of condition strings. The work in [10] is concerned on characterizing the space of condition strings rather than reducing the time of the search. However, the large number of compilations and runs involved in the search, for each program, shows that the direct application of specialized search techniques is not practical for optimize statically compiled programs. Furthermore, the technique proposed in [10] does not account for separate compilation, which further reduces attainable performance.

In contrast, the technique proposed in this work - that extends our prior work in [36] - deals with separate compilation and approaches the problem of selecting inlining constraints in two passes. First our technique selects and builds a performance prediction model using a small number of training examples and then uses the model to search the inlining vector which optimizes performance of a program.

Prior work [37–39] interested in predicting near-optimal compiler settings for program optimization explore only on-off compiler setting, including those enabling/disabling inlining, but do not account for the selection of inlining constraints that directly influence the behavior of the inliner. To the best of our knowledge this is the first paper investigating automatic selection of inlining constraints, given a compiler and its parameterized heuristic. Differently from prior work utilizing machine learning algorithms to predict near-optimal on-off compiler settings [40–45, 38, 46, 39] our technique leaves the selection of the model unspecified until the training set is available and automatically selects the model using the rigorous and quantitative procedure described in Sections 3.3 and 3.4.

6 Conclusion

We proposed a new technique to determine inlining vectors for program optimization. Our technique uses regression algorithms to learn the relation between inlining vectors and completion time. Such a relation is subsequently used to predict inlining vectors able to optimize programs subject to a given workload and targeting a given architecture.

Key highlights of our technique are as follows: (1) The selection of the regression algorithm is unspecified until the training set is available and many algorithms are evaluated; (2) The rigorous selection of the most suitable regression algorithm follows a procedure based 10-fold cross-validation; (3) Our technique is suitable to be used in industry settings as a support for production compilers. In such settings, applications are routinely tested, hence the training data required to train our technique is often times already available, whereas application performance is often times optimized for specific classes of workloads rather than for any workload.

The effectiveness of our technique is shown by the experimental results with GNU GCC. Our technique was able to select inlining vectors to optimize 22 combinations $(program, input)$ on the state-of-the-art Intel Xeon Westmere. Compared with the optimization level `-O3` and its default inlining vector, our technique yielded performance improvements ranging from 2% to 9%.

Acknowledgments. This work was partially supported by Yahoo! Inc, the NSF grant number CCF-0811882 and the NSF Variability Expedition Grant number CCF-1029783. The authors would like to thank the anonymous reviewers for their valuable comments.

References

1. Davidson, J.W., Holler, A.M.: A study of a C function inliner. Software: Practice and Experience 18(8) (1988)
2. Ball, J.E.: Predicting the effects of optimization on a procedure body. In: Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction (1979)
3. Cooper, K.D., Hall, M.W., Torczon, L.: An experiment with inline substitution. Softw. Pract. Exper. 21(6) (May 1991)
4. Cooper, K.D., Hall, M.W., Torczon, L.: Unexpected side effects of inline substitution: a case study. ACM Lett. Program. Lang. Syst. 1(1) (March 1992)
5. Guo, J., Stiles, M., Yi, Q., Psarris, K.: Enhancing the role of inlining in effective interprocedural parallelization. In: Proceedings of the International Conference on Parallel Processing (2011)
6. Allen, R., Johnson, S.: Compiling C for vectorization, parallelization, and inline expansion. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1988)
7. Scheifler, R.W.: An analysis of inline substitution for a structured programming language. Commun. ACM 20(9) (September 1977)
8. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman (1979)
9. Cavazos, J., O'Boyle, M.F.P.: Automatic tuning of inlining heuristics. In: Proceedings of the ACM/IEEE Conference on Supercomputing (2005)
10. Cooper, K.D., Harvey, T.J., Waterman, T.: An Adaptive Strategy for Inline Substitution. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 69–84. Springer, Heidelberg (2008)

11. Alpern, B., Augart, S., Blackburn, S.M., Butrico, M., Cocchi, A., Cheng, P., Dolby, J., Fink, S., Grove, D., Hind, M., McKinley, K.S., Mergen, M., Moss, J.E.B., Ngo, T., Sarkar, V.: The Jikes research virtual machine project: building an open-source research community. IBM Syst. J. 44(2) (January 2005)
12. Stone, M.: Cross-validatory choice and assessment of statistical predictions. Journal of the Royal Statistical Society B 36(1), 111–147 (1974)
13. Henning, J.L.: Spec cpu2006 benchmark descriptions. SIGARCH Comput. Archit. News 34(4) (September 2006)
14. Berube, P., Amaral, J.M.: Aestimo: a feedback-directed optimization evaluation tool. In: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (2006)
15. Quinlan, R.J.: Learning with continuous classes. In: 5th Australian Joint Conference on Artificial Intelligence (1992)
16. Jalan, R., Kejariwal, A.: Trin-trin: Who's calling? a pin-based dynamic call graph extraction framework. International Journal of Parallel Programming 40(4) (2012)
17. Weicker, R.P., Henning, J.L.: Subroutine profiling results for the cpu2006 benchmarks. SIGARCH Comput. Archit. News 35(1) (March 2007)
18. Glek, T., Hubicka, J.: Optimizing real-world applications with gcc link time optimization. In: GCC Developers Summit (2010)
19. Hubicka, J.: Interprocedural optimization framework in GCC. In: GCC Developers Summit (2007)
20. Quinlan, J.R.: C4.5: programs for machine learning. Morgan Kaufmann Publishers Inc., San Francisco (1993)
21. Smola, A.J., Schölkopf, B.: A tutorial on support vector regression. Statistics and Computing 14(3) (August 2004)
22. Siegel, S., Castellan, N.J.: Nonparametric Statistics for the Behavioral Sciences, 2nd edn. McGraw-Hill (1988)
23. Wall, L., Christiansen, T., Orwant, J.: Programming Perl, 3rd edn. O'Reilly Media (2000)
24. Hornik, K., Buchta, C., Zeileis, A.: Open-source machine learning: R meets Weka. Computational Statistics 24(2) (2009)
25. Williams, G.J.: Rattle: A data mining GUI for R. The R Journal 1(2) (December 2009)
26. R Development Core Team. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2011)
27. Zaparanuks, D., Jovic, M., Hauswirth, M.: Accuracy of performance counter measurements. In: IEEE International Symposium on Performance Analysis of Systems and Software (2009)
28. Levinthal, D.: Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors (2009)
29. Reinders, J.: VTune Performance Analyzer Essentials: Measurement and Tuning Techniques for Software Developers. Engineer to Engineer Series. Intel Press (2005)
30. Rousseeuw, P.J., Leroy, A.M.: Robust regression and outlier detection. John Wiley & Sons, Inc. (1987)
31. Moody, J., Darken, C.J.: Fast learning in networks of locally-tuned processing units. Neural Comput. 1(2) (1989)
32. Student: The Probable Error of a Mean. Biometrika 6(1) (1908)
33. Ayers, A., Schooler, R., Gottlieb, R.: Aggressive inlining. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (1997)

34. Dean, J., Chambers, C.: Towards better inlining decisions using inlining trials. In: Proceedings of the ACM Conference on LISP and Functional Programming (1994)
35. Hazelwood, K., Grove, D.: Adaptive online context-sensitive inlining. In: Proceedings of the International Symposium on Code Generation and Optimization (2003)
36. Cammarota, R., Kejariwal, A., Donato, D., Nicolau, A., Veidenbaum, A.V.: Selective search of inlining vectors for program optimization. In: Proceedings of the 9th Conference on Computing Frontiers (2012)
37. Pan, Z., Eigenmann, R.: Fast and effective orchestration of compiler optimizations for automatic performance tuning. In: Proceedings of the International Symposium on Code Generation and Optimization (2006)
38. Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'Boyle, M.F.P., Temam, O.: Rapidly selecting good compiler optimizations using performance counters. In: Proceedings of the International Symposium on Code Generation and Optimization (2007)
39. Fursin, G., Temam, O.: Collective optimization: A practical collaborative approach. ACM Trans. Archit. Code Optim. 7(4) (December 2010)
40. Monsifrot, A., Bodin, F., Quiniou, R.: A Machine Learning Approach to Automatic Production of Compiler Heuristics. In: Scott, D. (ed.) AIMSA 2002. LNCS (LNAI), vol. 2443, pp. 41–50. Springer, Heidelberg (2002)
41. Stephenson, M., Amarasinghe, S., Martin, M., O'Reilly, U.: Meta optimization: improving compiler heuristics with machine learning. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (2003)
42. Pan, Z., Eigenmann, R.: Rating compiler optimizations for automatic performance tuning. In: Proceedings of the ACM/IEEE Conference on Supercomputing (2004)
43. Haneda, M., Knijnenburg, P.M.W., Wijshoff, H.A.G.: Optimizing general purpose compiler optimization. In: Proceedings of the 2nd Conference on Computing Frontiers (2005)
44. Stephenson, M., Amarasinghe, S.: Predicting unroll factors using supervised classification. In: Proceedings of the International Symposium on Code Generation and Optimization (2005)
45. Haneda, M., Knijnenburg, P.M.W., Wijshoff, H.A.G.: On the impact of data input sets on statistical compiler tuning. In: Proceedings of the 20th International Conference on Parallel and Distributed Processing (2006)
46. Dubach, C., Jones, T.M., Bonilla, E.V., Fursin, G., O'Boyle, M.F.P.: Portable compiler optimisation across embedded programs and microarchitectures using machine learning. In: Proceedings of the IEEE/ACM International Symposium on Microarchitecture (2009)
47. Pearson, K.: On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arised from random sampling. Philosophical Magazine Series 5 50 (1900)
48. Pearson, K.: Notes on the History of Correlation. Biometrika 13(1) (October 1920)
49. Rodgers, J.L., Nicewander, A.W.: Thirteen ways to look at the correlation coefficient. The American Statistician 42 (1988)

APPENDIX A - Example of Cross-Validation

Let us assume that our training set is composed of 1000 (size of $\mathcal{M}$) samples and that we want to compare the two hypotheses $\hat{h}_I$ and $\hat{h}_{II}$ corresponding to the regression algorithms I and II respectively. We carry out k-fold cross-validation using $k = 10$. We divide our training set into 10 folds. Each fold is composed of 100 examples. The outcomes of the features in the fold j are arranged as a vector as in Equation 1.

$$\mathbf{p}_j = (p_{j,1}; \cdots ; p_{j,100}),\ \textit{with } j = 1, 2, \cdots , 10 \tag{1}$$

To compare the two hypotheses at the pass j of the procedure, we take the fold j out of the training set, train both the models I and II with the remaining 9 folds and use the corresponding $\hat{h}_{I,j}$ and $\hat{h}_{II,j}$ to predict the outcomes corresponding to the (*unseen*) features in the fold j - see Equation 2.

$$\hat{\mathbf{p}}_j = (\hat{p}_{j,1}; \cdots ; \hat{p}_{j,100}),\ \textit{with } j = 1, 2, \cdots , 10 \tag{2}$$

The mean absolute error between $\mathbf{p}_j$ and $\hat{\mathbf{p}}_j$ is computed as in Equation 3, whereas the Pearsons coefficient of correlation [47–49] between $\mathbf{p}_j$ and $\hat{\mathbf{p}}_j$ is computed as in Equation 4.

$$\alpha_j = \frac{|p_{j,1} - \hat{p}_{j,1}| + \cdots + |p_{j,100} - \hat{p}_{j,100}|}{100} \tag{3}$$

$$\rho_j = \frac{\sum_{s=1}^{100} (p_{j,s} - \mu_j)(\hat{p}_{j,s} - \hat{\mu}_j)}{\sqrt{\sum_{s=1}^{100} (p_{j,s} - \mu_j)^2 \times \sum_{s=1}^{100} (\hat{p}_{j,s} - \hat{\mu}_j)^2}} \tag{4}$$

where

$$\mu_j = \frac{\sum_{s=1}^{100} p_{j,s}}{100} \quad \textit{and} \quad \hat{\mu}_j = \frac{\sum_{s=1}^{100} \hat{p}_{j,s}}{100}$$

At the end of the cross-validation process we obtain a sequence of 10 mean absolute errors and a sequence of 10 coefficients of correlation. The two models, I and II, are compared using the average values of the mean absolute errors and of the coefficients of correlation as indicated in Equations 5 and 6.

$$\mu_I^\alpha = \frac{\sum_{s=1}^{10} \alpha_{j,s}}{10} \quad \textit{and} \quad \mu_I^\rho = \frac{\sum_{s=1}^{10} \rho_{j,s}}{10} \tag{5}$$

$$\mu_{II}^\alpha = \frac{\sum_{s=1}^{10} \alpha_{j,s}}{10} \quad \textit{and} \quad \mu_{II}^\rho = \frac{\sum_{s=1}^{10} \rho_{j,s}}{10} \tag{6}$$

By definition $\alpha > 0$ and $|\rho| \leq 1$ and too are the quantities in Equations 5 and 6. Model selection occur as follows. Let assume, for example, that $\mu_I^\alpha < \mu_{II}^\alpha$ and $\mu_I^\rho > \mu_{II}^\rho > 0$. Both models have positive coefficients of correlation - indicating the presence of a linear relation between the real and the estimated values during cross-validation. However, the mean absolute error of the model I is lower than that of the model II. Consequently, one would select/prefer model I to model II.

Automatic Generation of Program Affinity Policies Using Machine Learning

Ryan W. Moore and Bruce R. Childers

University of Pittsburgh,
Pittsburgh, USA
{rmoore,childers}@cs.pitt.edu

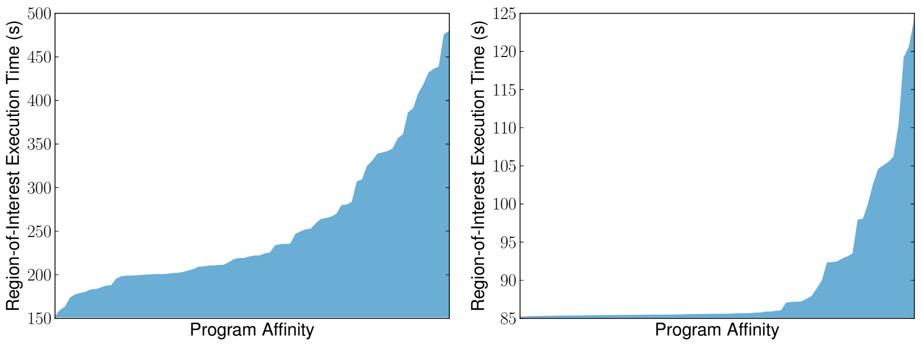
Abstract. Modern scientific and server programs require multisocket, multicore machines to achieve good performance. Maximizing the performance of these programs requires careful consideration of program behavior and careful management of hardware resources. In particular, a program's affinity can have a critical performance effect. For such machines, there are many possible affinities for a multithreaded program. In this paper, we present AutoFinity, a solution to automatically generate program affinity policies that consider program behavior and the target machine. The policies are constructed with machine learning and used online to select an affinity. We implemented AutoFinity on a 4-processor, 48-core machine and evaluated it on 18 multithreaded programs with varying thread counts. Our results show that in 12 out of 15 cases where affinity impacts runtime, the policy generated by AutoFinity chose affinities that improved performance versus assignments that do not consider program and machine behavior.

Keywords: policy generation, runtime adaptation, parallel performance.

1 Introduction

Today's computers for scientific and server workloads are large multisocket, multicore machines, capable of large amounts of parallelism. The cores in these machines share multiple resources, such as caches, memory controllers, and interconnects. For a multithreaded program, allocation decisions can be made for threads to share the hardware resources. Alternatively, threads can be given exclusive access to resources (e.g., executing with their own last-level cache). Because resource allocation impacts performance, it must be performed carefully and in response to runtime conditions, such as core availability, thread count, and workload demand. The assignment of program threads to cores, i.e. *program affinity*, is one way resources can be allocated.

It is well known that there is no single "best" program affinity to use across all programs. For example, Fig. 1a shows the program *streamcluster*'s region-of-interest (ROI) execution time when executed with different affinities [1]. The ROI is the program's parallel section where the "work" is done. This figure shows how performance (y-axis) changes with different affinities (x-axis). The program



(a) *Streamcluster* (b) *Swaptions*

Fig. 1. Execution times of programs' region-of-interest across different program affinities on a 48-core machine (each program has 8 threads)

affinities are sorted by ROI execution time. *Streamcluster* uses 8 threads on a 48-core AMD machine in these experiments.

For *streamcluster* (Fig. 1a) the first 5% of affinities achieve an execution time of around 150 seconds. Beyond this point, quality diminishes, resulting in an execution time of about 200 seconds (33% slower). The remaining execution times steeply increase as the affinity quality continues to diminish. The worst affinity has an execution time of over 450 seconds! Because so few affinities are good, it is unlikely that a randomly chosen affinity will result in good performance.

In contrast, *swaptions* (Fig. 1b) is relatively insensitive to affinity [1]. The majority of affinities have nearly identical performance. If an affinity was randomly selected, the resulting execution time would likely be good because *most* affinities performed well. However, some affinities cause *swaptions* to perform poorly. The worst affinity has an execution time of 124 seconds (45% slower than the best one). Even in this insensitive program, it can be beneficial to select its affinity because there are cases that should be avoided.

It is important to understand why *streamcluster* and *swaptions* behave so differently (Fig. 1a and 1b). Certain affinities allow programs to communicate more quickly (e.g., by sharing cache space). However, sharing caches among threads can reduce the cache space, causing the working set to overflow the cache. Other affinities give a thread more cache space, by spreading the threads across sockets and caches, at the cost of communication speed. Cache and communication demands of a program directly influence which affinities are best.

In a study of PARSEC, *streamcluster* made frequent use of barriers [1]. Because affinity affects communication, and thus, the speed at which threads reach and pass through barriers, *streamcluster* is affected dramatically by its affinity. *Swaptions* does not make much use of barriers or locks [1]. Furthermore, its working set size is relatively small: A thread's working set can fit in our experimental machine's private L2 cache. Thus, *swaptions* is more resilient to affinity changes.

As these figures show, there is diversity in program behavior across affinities. The runtime resource manager (e.g., operating system) should support the user and select a well-behaving affinity for the user's program. Furthermore, selection should support any thread count as well as never-before-executed programs.

In this paper, we present *AutoFinity*, an automated system that generates an *affinity policy* based on machine learning and training data. The generated policy is used at runtime to select a program's affinity. The policy can handle programs that were not part of the training and/or thread counts that have not been considered by training. AutoFinity can update its policy as it discovers and records new information about programs.

A policy chooses a thread-count-independent affinity class, which we call an *affinity hint*. The system resource manager (e.g., the operating system) uses a hint to select an affinity. Affinity hints enable policies and affinity selection across thread counts in the presence of runtime resource constraints.

We discuss the choices and trade-offs behind AutoFinity's design. We evaluate several design decisions on how AutoFinity can improve program performance. The contributions of this paper are:

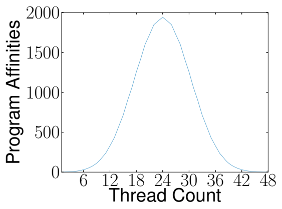
1. We show the importance of choosing program affinities;
2. We present AutoFinity, an automated, thread-independent solution to select program affinity;
3. We provide guidelines to find the proper set of hardware performance counters (HPCs); and,
4. We evaluate AutoFinity and demonstrate its ability to produce policies that guide affinity settings across a range of programs and thread counts.

This paper is organized in the following way. Section 2 discusses the design and use of AutoFinity. Section 3 performs an evaluation of AutoFinity. Section 4 discusses related work. Section 5 concludes.

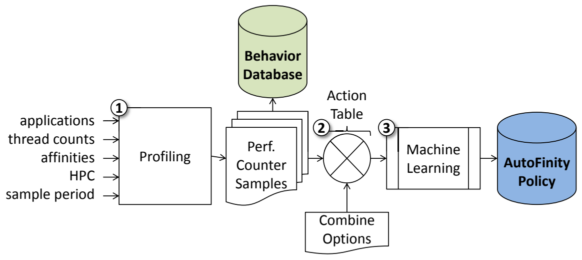
2 The Design of an Affinity Policy Generator

This work focuses on choosing program affinity for CPU/memory-bound programs. These programs will most benefit from proper affinity selection. There are several challenges to choosing program affinity. First, the number of affinity settings is large, as shown in Fig. 2a. With 24 threads there are 1941 settings on a 48-core machine. Selecting program affinities across thread counts is even more difficult. Second, the selection process must support unknown programs and yet quickly select a program affinity without evaluating, through trial and error, program affinities. Finally, affinity selection should handle runtime resource constraints (e.g., an unavailable socket). We assume at least one available core per thread and that, if is the system is shared between users, machine resources (e.g., sockets) are statically partitioned to provide strong isolation.

AutoFinity addresses these challenges. It is built on *REEact*, a framework for virtual execution management [13]. AutoFinity automatically builds an affinity policy to maximize a user-defined metric. The policy is consulted at runtime on unseen programs to determine an *affinity hint*, a set of possible affinities that shall work well. A hint is independent from thread count and is used by the resource manager (e.g., operating system) as a guide for satisfying runtime conditions.



(a) Affinity counts



(b) Policy generation steps

Fig. 2. Available affinities and building a policy to help choose one

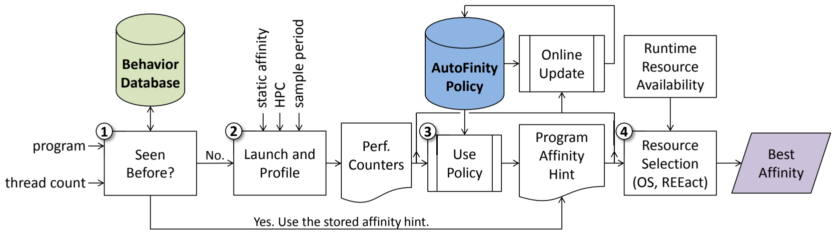


Fig. 3. Policy usage steps

Fig. 2b shows the steps to build a policy. First, AutoFinity profiles training programs for a range of affinities (step 1). This step gathers data that indicates how well particular affinities behave. Program behavior is captured by hardware performance counters (HPC) gathered during the program's ROI. The HPC values are recorded regularly, creating multiple samples. Each sample is timestamped to facilitate the examination of cross-affinity program behavior (e.g., two samples each from the beginning of a program's execution). A sample observes program behavior over a fixed amount of time. As such, multiple consecutive sample periods capture a program's ROI behavior.

The samples are analyzed and transformed into an *action table* (step 2). The action table is built to maximize a user-defined metric (e.g., performance). A row in the action table states what affinity hint (discussed in Sect. 2.1) should be used if a program exhibits a particular behavior at runtime. The samples are also archived in a behavior database to build future policies.

Machine learning is used to compact the rows of the action table thus building the *affinity policy* while also handling contradictory or missing information in the table (step 3). Machine learning resolves the conflicts and "holes" through its statistical analysis.

The generated policy is used at runtime to select an affinity hint. Fig. 3 shows this process. To use the policy, the program's thread count must be known. The thread count can be discovered on the command line, in an environment variable, or by monitoring thread creation syscalls.

With the thread count, the behavior database is consulted. If the program and its thread count have been previously observed then the behavior database specifies the program's affinity hint (step 1). Otherwise, the policy is consulted to obtain the program's affinity hint. To consult the policy, program behavior must be observed. A starting affinity will be used and the program's parallel behavior will be observed. In this paper, we examine the program's behavior once, during a single sample period (step 2). Continuous sampling and adjustment are naturally supported by our approach, but require a migration cost model (future work).

The sampled HPC values are used by the AutoFinity policy to select an affinity hint (step 3). The HPC values and selected hint are saved into the behavior database. The affinity hint can be used to choose a new program affinity. To use an affinity hint, the program's resource controller (e.g., operating system), considers the hint and allocates resources (cores), ultimately choosing a program affinity for the program (step 4).

Because AutoFinity continuously records program behavior, it can generate a new policy as program information is accumulated (e.g., when the behavior database has grown by some percent threshold). If a stored affinity hint corresponds to an old policy, the hint will be removed.

2.1 Affinity Hints

Program affinities explicitly state which cores a program may use (e.g., cores 0–5). For a particular thread count, there may be thousands of possible affinity choices. It is difficult to directly pick an appropriate affinity, and selecting an affinity is made harder by supporting all thread counts. To reduce the number of affinity options that a policy must consider, it is necessary to generalize program affinities into classes (e.g., affinity hints).

An affinity hint suggests a relationship between cores (e.g., cores should be distributed across different cache domains). The generalization of program affinities enables techniques which work *across* thread counts. Using a hint, a program's resource manager will select an affinity based on available resources (e.g., sockets) and number of active threads.

There are two requirements for affinity hints. First, affinity hints should not be overly specific: A hint should be realized at runtime, even under runtime resource constraints (e.g., an unavailable processor socket). Second, affinity hints must capture the effects of core assignments on a thread's: a) cache space, b) communication, and c) access to main memory (DRAM).

Threads may benefit from a large amount of effective cache space (e.g., to hold their private working set). The space available to a thread may be negatively impacted by the presence of one or more threads in the same last-level cache domain (LLC). However, sharing LLCs also allows threads to more quickly communicate. Regardless of whether cores share one or more levels of cache, a thread will occasionally have cache misses. It is important to consider the impact of affinity on NUMA accesses. For example, programs may be able to better exploit memory bandwidth if spread across NUMA domains.

Affinity Hint Parameter	**Parameter Description**	**Values**	**Symbol**
spreadAcrossSockets	Prefer use of many sockets	{True, False}	S
spreadAcrossLLCs	Prefer to not share LLCs	{True, False}	L
socketOptionGetsPriority	Socket preference is priority	{True, False}	P

Fig. 4. Affinity hint parameters

```
GetAffinity(resources, threadCount, spreadAcrossSockets, spreadAcrossLLCs,
socketOptionGetsPriority)
affs = AllAffinities(resources, threadCount)
if spreadAcrossSockets
    socketSortFunc = PrioritizeByLargerUsedSocketCount
else
    socketSortFunc = PrioritizeBySmallerUsedSocketCount
if spreadAcrossLLCs
    LLCSortFunc = PrioritizeBySmallerLLCOccupancy
else
    LLCSortFunc = PrioritizeByHigherLLCOccupancy
if socketOptionGetsPriority
    prioritizedAffs = Sort(affs, socketSortFunc then by LLCSortFunc)
else
    prioritizedAffs = Sort(affs, LLCSortFunc then by socketSortFunc)
return HighestPriority(prioritizedAffs)
```

Fig. 5. Algorithm for selecting a program affinity

To meet these requirements, the affinity hints have three boolean parameters, leading to eight possible affinity hints (Fig. 4). The first affinity hint parameter dictates whether threads should be distributed across sockets (*spreadAcrossSockets*, symbol: S). The use of multiple sockets may allow greater memory bandwidth, by simultaneously employing memory banks. On the contrary, assigning threads to the same sockets allows better communication in the same memory domain, potentially avoiding the need to access remote nodes.

The second parameter, *spreadAcrossLLCs* (symbol: L), controls whether threads should be assigned to cores that share a LLC. Threads that share a cache may communicate with each other more quickly or may prefetch data for the sharers. However, sharers may contend for cache space. The third parameter of an affinity hint (*socketOptionGetsPriority*, symbol: P) captures whether the sharing sockets or sharing LLC parameter is more important.

The three parameters define eight hints. We use boolean notation to represent each hint. For example, $\mathrm{S}\overline{\mathrm{LP}}$ specifies *spreadAcrossSockets* = True, *spreadAcrossLLCs* = False, and *socketOptionGetsPriority* = False. This hint dictates that application threads share as few LLCs as possible ($\overline{\mathrm{L}}$)[1]. The LLCs are preferred to be on separate sockets (S). The priority in this example hint is to pack threads onto as few LLCs as possible ($\overline{\mathrm{P}}$).

[1] There is always at most one thread per core.

With an affinity hint, the OS can consider runtime resource availability to choose an affinity. This process is shown in Fig. 5. GetAffinity takes a list of available resources (cores) and the amount needed (*threadCount*). A list of affinities that can be made from the available resources is obtained (line 1). Only affinities that use as many cores as threads are considered. The value of *spreadAcrossSockets* is used to prioritize the potential affinities (lines 2–5). If threads should be spread across sockets then affinities which utilize more sockets are prioritized, otherwise affinities which use fewer sockets will be preferred. In lines 6–9 *spreadAcrossLLCs* is similarly processed. A hint which states that threads should spread across LLCs will prioritize affinities accordingly. Next, the potential affinities are sorted based on *socketOptionGetsPriority*'s value (lines 10–13). Ties are broken by the secondary priority. The highest priority affinity (i.e., the one that best matches the affinity hint) is returned (line 14)[2].

2.2 Action Table Generation

AutoFinity analyzes program behavior to generate a policy that specifies an affinity hint for a program under runtime conditions. This process consists of two steps: 1) building the action table and 2) condensing the table into a policy.

The action table contains observed program behaviors and the affinity hint(s), that will improve the user-defined performance metric in those situations. It is built with initial training data: programs in various affinities will have their behavior (HPC values) recorded. Additional behaviors can be gathered online.

Fig. 6 shows BuildActionTable. This function condenses performance information (*observations*) into an action table. BuildActionTable has two more parameters: *condense* (a function) and *leeway* (an integer). *Condense* resolves different performance values coming from similar HPC samples across multiple programs. This phenomenon is expected to occur occasionally, as programs may have similar behavior (e.g., cache misses) but different performances.

GetAffinity selects one configuration per affinity hint (assuming a fixed thread count). Each of the eight affinity hints will correspond to a best match affinity (eight affinities total). To allow training on affinities which *almost* would be selected by GetAffinity, we introduce *leeway*. *Leeway* is an integer that allows an affinity hint to correspond to additional affinities, beyond its best match. A leeway of 0 means that only the best match affinities are considered.

Leeway is a maximum edit distance between two affinities. A leeway of l allows a program affinity to be considered as belonging to an affinity hint if, by moving up to l threads across LLC domains, the affinity becomes identical to that affinity hint's best match affinity. The movement of threads across LLC domains may also move threads across sockets. Therefore leeway also places a limit on affinity similarity in terms of socket usage.

For example, an affinity that has six threads share a LLC might be selected for the hint $\overline{\text{SLP}}$ as its corresponding best match affinity (for 6 threads). With

[2] Multiple affinities may tie for the same priority. In this case, the affinities will be isomorphic and one will be chosen arbitrarily.

```
BuildActionTable(observations, condense, leeway)
 1  $possibilities = \emptyset$, $actionTable = \emptyset$
 2  for $program \in observations$
 3      for $samplePeriod \in program$
 4          for $aff \in samplePeriod$
 5              $behavior = aff.observedPerformanceCounters$
 6              for $destAff \in samplePeriod$
 7                  $perf = destAff.perf$
 8                  $possibilities[behavior][aff][destAff].append(perf)$
 9  for $behavior \in possibilities$
10      for $aff \in behavior$
11          for $destAff \in aff$
12              $affinityHints$ = AffinityHintFromConf($destAff$, $leeway$)
13              if $affinityHints$ = NONE
14                  continue // Configuration does not map to an affinity hint.
15              $consequence = condense(possibilities[behavior][aff][destAff])$
16              // An affinity may map to > 1 affinity hints (e.g., due to leeway).
17              for $hint \in affinityHints$
18                  $actionTable[behavior][aff][hint] = consequence$
19  for $behavior \in actionTable$
20      for $aff \in behavior$
21          $bestAffinityHint$ = MAX($actionTable[behavior][aff]$)
22          WriteTrainAction($behavior$, $bestAffinityHint$)
```

Fig. 6. Algorithm for building the action table

a leeway of 1, the affinity which causes 5 threads to share a LLC while putting another thread on a separate socket, would still be considered as corresponding to $\overline{\text{SLP}}$ because only 1 thread was moved as compared to the best match affinity.

In Fig. 6 the first set of loops (lines 2–8), build up a table, *possibilities*, which stores the affinity (*aff*) and its associated *behavior* (HPC values). For each affinity with recorded behavior, alternate affinities (with the same thread count) are considered. These are considered to be destination affinities (*destAff*) that the program *could have* been in. For each alternative affinity, the performance metric of the program in that sample period affinity is recorded ($perf$).

Programs may exhibit similar behavior, and therefore, the first set of loops accumulate a list of multiple performance metrics (line 8). List items are reduced into one entry using the *condense* function (lines 9–18). BuildActionTable also converts destination affinities to their corresponding affinity hint(s), with AffinityHintfromConf (the inverse of GetAffinity). After this step, the action table contains a list of affinities and associated behaviors, affinity hints that could have applied, and the consequent performance for the affinity hints.

Finally, the action table is ready to be written. In lines 19–22 the action table is iterated over. For each type of behavior seen under each experienced affinity, BuildActionTable finds the best affinity hint to be in. These results are written to a file as a series of rules (WriteTrainAction). Each rule states an affinity that should be used if a particular behavior is observed. Contradictions

```
AffinityHintPolicySW($dCacheAccesses$, $invalidDCacheLinesEvicted$,
                     $exclusiveReadRequestsToL3CacheFromAnyCore$, $retiredUops$)
if $retiredUops \in (\text{BoundLower0}, \text{BoundUpper0}]$
    if $dCacheAccesses < \text{Bound1}$ and $invalidDCacheLinesEvicted < \text{Bound2}$
        return $\overline{\text{S}}\text{LP}$
    if $invalidDCacheLinesEvicted < \text{Bound3}$ and
            $dCacheAccesses \in (\text{BoundLower4}, \text{BoundUpper4}]$ and
            $exclusiveReadRequestsToL3CacheFromAnyCore < \text{Bound5}$
        return $\overline{\text{S}}\text{L}\overline{\text{P}}$
return $\overline{\text{SL}}\text{P}$
```

Fig. 7. Example affinity hint policy

between rules are expected, and may be due to noise and/or different programs exhibiting similar behavior, yet operating best in different affinity hints.

2.3 Building the Affinity Hint Policy

To convert the action table to a policy that can be consulted at runtime, we use JRIP from WEKA [4]. JRIP is an implementation of Repeated Incremental Pruning to Produce Error Reduction (RIPPER), a propositional rule learner that produces a series of rules for classification. Each rule is a set of conditions joined by `and` operators. RIPPER fills in "holes" due to incomplete data (i.e., combinations of HPC values that were not observed). It also prunes rules that are statistically insignificant. We use RIPPER because it creates rules for missing data, produces simple, human-readable policies, and its rules are easily converted into a code implementation to use at runtime. We use WEKA's default settings.

An example policy produced by JRIP is shown in Fig. 7. This policy is used by *swaptions* in Sect. 3. For readability, numeric values have been replaced with constants. The policy's parameters are the values of four HPCs. The selection of these counters is discussed later (Sect. 3.1). Depending on the values of these counters, one of three affinity hints will be selected ($\overline{\text{S}}\text{LP}$, $\overline{\text{S}}\text{L}\overline{\text{P}}$, or $\overline{\text{SL}}\text{P}$).

These affinity hints have been selected by the RIPPER algorithm as the only necessary affinity hints (i.e., there is no fourth affinity hint that the policy will select). As more program performance data is obtained and the policy is rebuilt, additional affinity hints may become available for selection.

3 Evaluation

The previous section described how AutoFinity generates affinity hints, builds the action table and its policy, and uses its policy. This section presents experimental results for AutoFinity on a diverse set of programs and thread counts.

All evaluations were performed on the machine described in Fig. 8. The machine is a 48-core AMD Opteron 6164 NUMA system. Each of the four sockets has 2 NUMA domains (8 NUMA domains total). Each NUMA domain has 6

Machine Component	**Component Details**
Processors	4 AMD Opteron 6164 HE sockets (48 cores total)
Socket	2 NUMA nodes
NUMA Node	1 L3 cache (LLC)
L3 Cache	5 MiB cache, 6x cores
Core	1.7 GHz, private L1 $ (64 KiB), private L2 $ (512 KiB)
Operating System	Linux 2.6.39.1

Fig. 8. Experimental Machine

cores that share an L3. The cores have private L1 and L2 caches. Programs are executed on an otherwise idle system.

The 18 evaluated benchmarks are culled from PARSEC 2.1 [1], OmpSCR 2.0 [3], and the NAS Parallel Benchmark Suite (NPB) 3.3.1 [7]. The PARSEC benchmarks use the largest input size ("native") and the NPB benchmark uses the "A" input size. Because OmpSCR does not provide official inputs, we configured its programs to execute in a similar amount of time as the PARSEC benchmarks (about one to two minutes with eight threads).

We use the following PARSEC programs: *blackscholes* (BS), *bodytrack* (BT), *canneal* (CN), *dedup* (DD), *facesim* (FS), *fluidanimate* (FA), *freqmine* (FM), *raytrace* (RA), *streamcluster* (SC), *swaptions* (SW), *vips* (VP), and *x264* (X). From OMPSCR we use *c_fft* (FT), *c_fft6* (FT6), *c_lu* (LU), *c_mandel* (MN), and *c_md* (MD). Lastly, we use *dc* (DC) from NAS. Some programs (e.g., PARSEC's *ferret*) were not used due to compilation errors, framework incompatibilities, or short execution times (i.e., a few seconds).

We use AutoFinity to maximize instructions per second (IPC), consequently minimizing program execution time. We chose this metric because it captures the rate of application progress and is easy to obtain via hardware performance monitoring. As we show later, choosing IPC works well to reduce execution time.

Before using AutoFinity to select a program's affinity hint and program affinity, we remove that program's data from the training data. This causes the program to be treated as a never-before-executed program. The AutoFinity policy is built on the remaining programs' data (leave-one-out cross-validation).

To evaluate the quality of the generated policy, we launch the program in a fixed affinity and observe one sample's HPC values (as shown in Fig. 3). The AutoFinity policy uses the HPC values to select an affinity hint. An affinity is selected from the hint. We then execute the program with that affinity and observe its ROI runtime. To find average performance, we take the geometric mean of each program executed under their selected affinity.

By default, previously unexecuted programs and thread counts are executed under a *distributed* affinity. A distributed affinity allows the AutoFinity policy to make better affinity hint suggestions. A distributed affinity spreads a program's threads across sockets and LLCs.

3.1 Choosing and Gathering Hardware Performance Counters

Before an AutoFinity policy can be used, the policy must be built. This requires profiling programs using HPCs. We use feature selection to choose appropriate counters automatically. The chosen counters will be measured during the initial training period as well as online (e.g., to consult the policy). To determine which counters were the best to help AutoFinity maximize IPC, we profiled several programs multiple times, collecting HPC values periodically with 60 manually selected counters.

Our experimental machine supports recording up to four counters at a time. Additional counters bring dwindling returns in terms of utility. The machine learning algorithm (i.e., RIPPER) will ignore counters and/or counter values that are statistically insignificant.

We used WEKA's `CfsSubsetEval` feature selection algorithm to choose the four performance counters that best correlate with our objective metric, IPC. This method evaluates features by considering the individual predictive ability of each feature along with the degree of redundancy between them. Features were selected and "grown" with a greedy search method.

Out of the 60 counters considered, the feature selection process selected the following features: 1) data cache accesses, 2) invalid data cache lines evicted, 3) exclusive read requests to the LLC from any core, and 4) retired micro-ops. These HPCs capture well how affinities affect IPC due to:

1. Data cache accesses indicate the memory demands of a program.
2. Cache lines become invalid if a shared line becomes modified by another core. Therefore, this counter indicates thread communication.
3. Exclusive read requests to the LLC cache also indicates communication and memory demand. A cache line in the exclusive state may soon become shared if another thread accesses that line. Exclusive LLC read requests may also allow one thread to prefetch another's data.
4. Retired micro-ops counts the number of operations completed by the processor. Our experiment machine, an x86-64 processor, executes instructions that decode into one or more RISC-like operations. Therefore, this metric captures application performance (IPC).

Lastly, for our techniques to work across a range of thread counts and sampling period lengths, we normalize each performance counter value to the number of threads and the number of seconds over which the counter was gathered.

Knowing which four HPCs were necessary to observe and maximize IPC, we gathered the selected counters and IPCs for each program. The programs are executed with 8 threads and counters are recorded for each of the 78 program affinities (8 threads). This step gathers the initial training data.

3.2 Selecting a Discretization Method

To build the action table, the training data must be put into discrete bins. After discretization, the data is analyzed and the action table is constructed (as discussed in Sect. 2.2). We consider two methods to discretize the data:

Method	Geo. Mean
equal-width	79.4
equal-frequency	83.1

(a) Impact of the discretization method

Function	Geo. Mean
average	80.6
gmean	80.6
hmean	80.6
max	81.5
median	79.4
min	83.1
sum	80.6

(b) Impact of the condense function

Fig. 9. Effects on average program execution time

1. Equal-width binning: Each bin corresponds to a constant range over the possible values of a HPC. Equal-width binning uniformly covers the range of observed performance counter values.
2. Equal-frequency binning: Bin widths are uneven to allow higher resolution where HPC values are most concentrated. Bins have the same item counts.

While we tried a range of numbers, we found that five bins for each method worked well, without making individual bins too wide or small. WEKA's equal-width discretization supports the automatic adjustment of the number of bins to be data-appropriate. We enabled this feature. Equal-frequency binning does not support such an option. In our experience, the adjustment is influenced by the number of requested bins.

To evaluate the discretization method, we perform cross-validation. The geometric mean of the program execution times is computed. Because the method of discretization is not the only adjustable parameter, we fix the other parameters (e.g., the condense function) to their best values (described later).

Fig. 9a shows the results of varying the discretization method. The results show that equal-width binning improves the geometric mean by approximately 5% (a mean of 79.4s versus 83.1s). Equal-width binning does better because equal-frequency binning creates very wide bins to allow for occasional narrow bins. However, a policy built with equal-frequency binning may be unable to later differentiate between HPC values which, though the difference between them is large, are grouped into the same wide bins. Thus, we conclude that equal-width binning is the best discretization method.

3.3 Selecting a Condense Function

We considered multiple condense functions to determine which method works best. The condense function combines performance measures from similar behaving samples gathered from the same affinity. Similar to Sect. 3.2, we set the other parameters to their best values. The condense function can be: a) arithmetic mean (AVERAGE), b) geometric mean (GMEAN), c) harmonic mean (HMEAN), d) MAX, e) MEDIAN, f) MIN, and g) SUM.

For example, suppose that program a is similar to b in that both have high data cache miss rates among other counter values. After analyzing their performance in different affinities, it is determined that both programs would execute fastest in affinity hint h. However, due to differences in program behavior, a will obtain an IPC in h of 2, whereas b will obtain an IPC of 1.5.

During action table and policy construction, AutoFinity must consider the expected IPC of a program, c, running under hint h, that is similar to a and b (e.g., c is a not-yet-encountered program). The action table could consider c to have a runtime of 2 (MAX), 1.5 (MIN), or 1.75 (AVERAGE). The condense function determines the expected value.

Fig. 9b shows the results of varying the condense function. The best function, MEDIAN, has a geometric mean of 79.4s. The worst condense function is MIN, with a geometric mean of 83.1s, about 5% worse than MEDIAN's result. MIN results in the worst performance because it creates a pessimistic policy. It penalizes types of behavior which are exhibited by multiple programs (e.g., if three programs exhibit the same behavior, the worst performing one is the only one whose performance will be considered).

Although MIN makes a pessimistic policy, and MAX makes an overly optimistic one, MEDIAN function is balanced and works well. Similarly, each of the means had good behavior.

3.4 Other Parameters

There are three other parameters for AutoFinity: 1) whether to normalize IPCs, 2) whether to include affinity-insensitive programs in the training data, and 3) leeway.

A program's IPC can be normalized to the smallest IPC recorded for that program. This allows for easier comparison of IPCs across programs. For example, if using a particular affinity hint can boost a program's IPC by 0.1, then this may be a relatively minor performance increase (e.g., 3% if the original IPC is 3.0) or a relatively larger increase (e.g., 10% if the original IPC is 1.0). Normalization allows these trends to be captured.

However, using the best parameter settings for the other policy parameters (e.g., the best condense function, the best discretization method) causes normalization to have no significant benefit. Nevertheless, we normalize IPCs because it did, for suboptimal parameter settings, result in a better policy.

The second parameter is whether affinity-insensitive programs are included in the training data. We define program insensitive programs as programs whose ROI execution time standard deviation is less than 1% of the program's average execution time across all affinities. Intuitively, programs whose behavior is insensitive to affinity contain no useful information to guide affinity selection. At worst, the data from insensitive programs may dilute important information.

As with IPC normalization, we found that using the best possible parameter settings causes the removal or inclusion of affinity-insensitive programs to make little difference (less than 1%). We did find it was occasionally beneficial for less

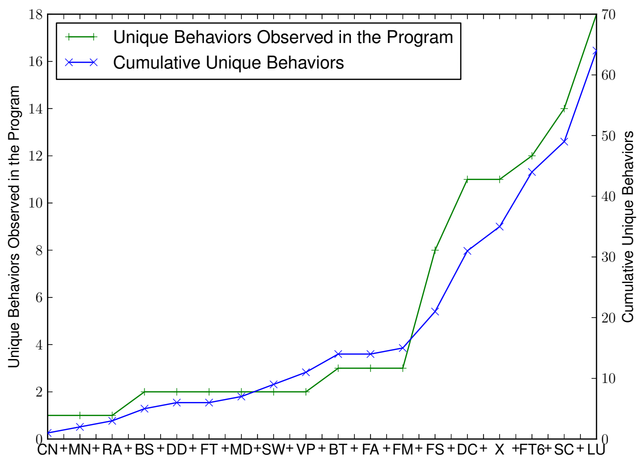


Fig. 10. Cumulative and per-program discretized behavior coverage

optimal settings. Therefore in our later evaluation, programs that are affinity insensitive are removed from the training data.

Lastly, we set leeway to 2. This leeway setting was determined experimentally (not presented due to space constraints). For 8 threads, this value resulted in 31 possible destination affinities while building the initial AutoFinity policy.

3.5 Program Behavior Coverage

Exposure to diverse program behavior allows the action table and policy to appropriately respond to unseen programs. To gain insight into this diversity, we analyzed program behavior across all possible affinities for 8 threads.

We define a program as exhibiting a behavior, b, if during the training process it produces a discretized HPC sample that is equal to b. Each program, p, produces a set of unique behaviors, B_p, where $B_p = \{b_0, b_1, ...\}$. Examining the discretized behaviors, we can determine the diversity of a program ($|B_p|$) and whether programs exhibit the same number of behaviors.

Other questions that we sought to address include whether some programs have more diverse behavior than others (i.e., whether $|B_{p_i}| < |B_{p_j}|$), and what is the effect of a program's behavior on the cumulative training information.

Figure 10 shows how coverage is influenced by the programs. The figure's x-axis is sorted by the number of unique behaviors in each program. It has two plots. The "+" line shows the number of unique behaviors in each program, while the "×" line shows the number of cumulative behaviors as the programs are added (left to right).

From the figure, we broadly place each program into one of two categories: 1) programs that have few unique behaviors and 2) programs that exhibit a diverse range of behavior. Approximately half of the programs, those before and including FM (*freqmine*), have few behaviors, and therefore, exhibit consistent behavior across their lifetime and affinities. This is shown by the low height of the "unique behaviors" line. Even though 12 programs fall into this category, each with an average of 2 unique behaviors per program (24 total), there are only 15 unique behaviors between them (i.e., at the point on the x-axis for FM, the cumulative unique behaviors line has a y-value of 15). Each program, therefore, contributes on average only 1.25 unique behaviors to the cumulative training data (15 behaviors over 12 programs = 1.25 behaviors per program).

The other programs (after FM) have great diversity. This diversity may be due to phases and/or sensitivity to affinity. These 6 programs contribute 49 unique behaviors to the training data, for an average of more than 8 unique behaviors per program. *C_lu* has the most unique behaviors. It exhibits 18 unique behaviors and contributes 15 to the cumulative list. *Streamcluster* has the second highest number of unique behaviors (14) and contributes 5 unique behaviors.

From this data, we believe that only a few training programs (i.e., those that exhibit a range of behavior) may be necessary to produce a good policy.

3.6 Performance Evaluation

Next, we evaluate how the policies from AutoFinity choose affinity hints for unknown programs. We compare the execution times of the programs using the AutoFinity-selected affinities against two static policies:

1. *Packed*: This policy places threads together, causing them to share LLCs whenever possible, and preferring to use as few sockets as possible ($\overline{\text{SLP}}$).
2. *Distributed*: This policy distributes threads uniformly across sockets and LLCs (SLP).

AutoFinity uses a leeway of 2 and affinity insensitive programs are removed from the training data. IPCs from a program are normalized to the slowest IPC from that program. HPC monitoring uses a sample size of 10 seconds for the four counters described in Sect. 3.1. Lastly, the condense function is MEDIAN.

Fixed Thread Count Evaluation. First, we compare the runtime of each program with 8 threads under the affinities chosen by AutoFinity. Training data also consisted of programs with 8 threads. Figure 11 shows these results.

The y-axis is ROI execution time (lower is better). The x-axis shows each program, with the geometric mean at the far right. Subscripts indicate thread count. For each program, three bars are shown. The first bar is the runtime under a packed affinity. The second bar is runtime under a distributed affinity. Finally, the third bar is runtime for the affinity chosen by the AutoFinity policy.

Some programs are insensitive to the policy. *Blackscholes*, *freqmine*, *c_md*, *c_mandel*, *raytrace*, *swaptions*, *vips*, and *x264* are not significantly affected by affinity. To a lesser extent, *fluidanimate* is insensitive too. However for most programs, the affinity is important. Five of the programs prefer a packed affinity (*bodytrack*, *canneal*, *dedup*, *c_fft*, *streamcluster*). These programs' threads communicate and/or share data. For *bodytrack*, *dedup*, and *c_fft*, AutoFinity selects an affinity that performs nearly as good or just as good as the best static policy.

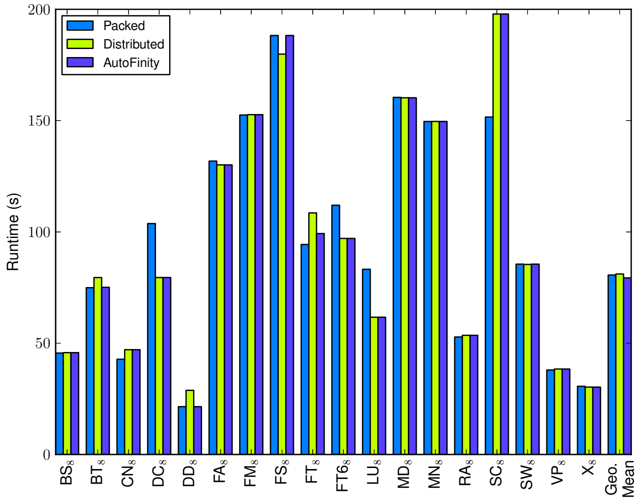


Fig. 11. AutoFinity, packed, and distributed ROI execution times for 8 threads

The AutoFinity policy selects a poor affinity hint for *streamcluster* ($\text{S}\overline{\text{LP}}$). The result is due to *streamcluster*'s unique behavior. As shown in Fig. 10 (discussed in Sect. 3.5), *streamcluster* has the second highest number of unique behaviors. Because *streamcluster* exhibits such different behavior (as compared to the other programs), AutoFinity did not build an appropriate rule to handle programs like it. Additional training data (i.e., from *streamcluster* or programs which exhibit behavior like it) would, we believe, allow the AutoFinity policy to make better affinity hint selections.

Dc, *facesim*, *c_fft*6, and *c_lu* prefer a distributed affinity. These programs have large memory footprints, as evidenced by the distributed policy preference. For *dc*, *c_fft*6, and *c_lu*, AutoFinity chooses an affinity hint and program affinity which performs just as well as the distributed static policy.

In 7 out of 9 cases involving affinity sensitive programs, AutoFinity chose a program affinity which performed nearly as well or just as well as their preferred static policy. The geometric mean (last set of bars in Fig. 11) shows that AutoFinity has a slight runtime advantage over either static policy. The advantage to AutoFinity is the user does not have to manually select the proper affinity.

We also compared AutoFinity's performance against an oracle policy, built through exhaustive experimentation. On average, AutoFinity achieved 96% of the performance (ratio of the oracle and AutoFinity's geometric mean). For three programs, it tied with the oracle policy (*dc*, *c_fft*6, *c_lu*).

Varying Thread Count. Next, we evaluate AutoFinity on each program with a range of thread counts (4, 8, 16, and 24 threads)[3]. Training data was obtained

[3] *Facesim* and *fluidanimate* do not support executing with 24 threads.

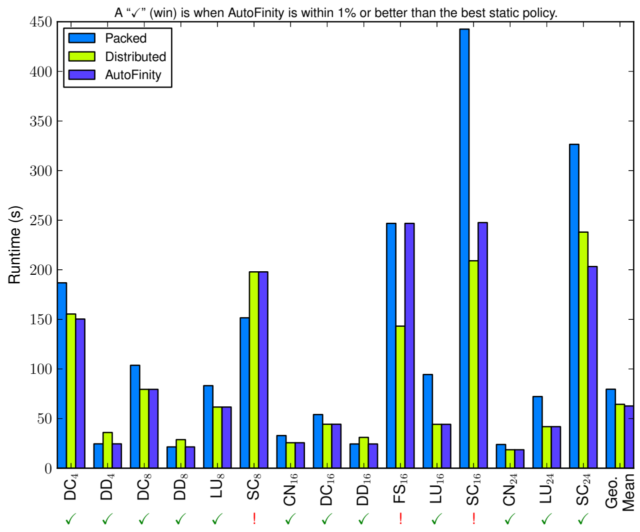


Fig. 12. AutoFinity and static policy ROI execution time comparisons across programs and thread counts for interesting cases

only for a thread count of 8. To test AutoFinity on a program, we remove that program from the training data before building the AutoFinity policy.

As previously shown, some programs have behavior that is not affected by affinity. Many programs, however, do show widely variable behavior. Affinity insensitivity is even sometimes thread count dependent. We deem the cases that have at least a 20% ROI runtime difference between their packed and distributed affinities, as "interesting cases." Interesting cases have a potential for bad or good affinity choices. Due to space constraints, we show and discuss only the interesting cases across the thread counts. There are 15 interesting cases.

Figure 12 shows the interesting cases. The x-axis shows the program's name with subscripts indicating thread count. The y-axis is the runtime of the program's ROI (lower is better). The first bar is the runtime for the packed affinity, the second bar is the runtime for the distributed affinity, and the third bar is the runtime of the program under the affinity chosen by AutoFinity.

The figure is annotated to indicate whether AutoFinity chooses an affinity that is within 1% of the best static policy. Wins (i.e., when AutoFinity is within 1%) are marked by "✓." If AutoFinity does not have a performance within 1% of the best static policy, we mark the graph with "!."

Dedup (DD) is an interesting program, appearing 3 times (4, 8, and 16 threads). Each time, *dedup* does best under a packed affinity. Because it is a pipeline parallel program, there is communication between threads as data is passed from stage to stage [1]. Therefore, it is natural that *dedup* would benefit from packed affinities, which cause threads to share cache space. In each case, the AutoFinity policy chooses an appropriate program affinity, and thus allowing *dedup* to execute quickly even though the AutoFinity policy was not built on data obtained from observing *dedup*.

Dc also appears 3 times in Fig. 12 (using 4, 8, 16 threads). Each time, it prefers a packed affinity. If *dc* is executed with 4 threads, AutoFinity chooses an affinity hint that performs better than the best static affinity ($\mathrm{S}\overline{\mathrm{LP}}$).

Streamcluster (SC) is another interesting case. If run with 8 threads (SC_8), it benefits most from a packed affinity. Unfortunately, AutoFinity chooses an affinity whose runtime is similar to the distributed affinity's runtime. If using 16 threads (SC_{16}), *streamcluster* prefers a distributed affinity. AutoFinity chooses an affinity which performs almost like (in terms of execution time) the preferred distributed affinity, but is not within 1%, and therefore, it is not classified as a win. If 24 threads are used (SC_{24}), *streamcluster* still prefers a distributed affinity. AutoFinity actually beats both the packed and distributed policies and chooses an affinity hint that performs better than either ($\overline{\mathrm{S}}\mathrm{LP}$).

Finally, in *facesim* (FS_{16}) the packed affinity is almost 75% slower than the distributed affinity. Unfortunately, in this case AutoFinity chooses an affinity hint that behaves like the packed affinity. Additional behavior data would improve AutoFinity's hint selection.

Overall, AutoFinity chooses a good affinity in most cases (12 out of 15 cases). It even beats the distributed and packed static policies in two cases. These results across thread counts show that AutoFinity policies and affinity hints work well.

4 Related Work

Wang et al., like our work, use machine learning to choose affinities [14]. Their approach requires a special compiler or dynamic binary instrumentation to extract features. Their offline approach requires a single-threaded execution of a previously-unseen program. Fengguang et al. also use dynamic binary instrumentation to choose affinities. They use an analytical model and gathered information to predict program performance in various thread settings [9]. We do not require dynamic binary instrumentation or special training runs before an affinity can be chosen.

Tam et al. [10] propose a system that uses hardware performance counters to decide thread affinity settings. Their work requires hardware support to track data sharing on a cache line basis. Our system supports responding to runtime resource availability by providing thread-count-independent affinity hints. We present guidelines for using our techniques on new systems, by discussing feature selection, sensitivity studies, and making use of more generally available performance counters. Our techniques adapt online and are more generic, supporting multiple performance metrics (not just IPC). Evaluating AutoFinity on other performance metrics is future work.

Klug et al. [5] propose *autopin*, a tool which uses hardware performance counters to choose thread affinities. Autopin uses a set, evaluate, iterate strategy: threads are given an affinity and the program behavior is measured. Another affinity is then evaluated and eventually the tool chooses an affinity. They ultimately reach well-performing affinity settings. Radojković et al. have a similar approach (e.g., resource monitoring, trial and error) but focus on network applications [8]. Our approaches decide on an affinity after a single sample.

AbouGhazaleh et. al use machine learning to build a runtime policy that examines hardware performance counters. However, their work is applied to embedded single-core systems, and focuses on optimizing energy-delay product [8].

Our techniques currently assume program behavior with a particular thread count does not change across invocations. Some programs have changing behavior across inputs. Techniques exist to model and predict how an input will affect a program's behavior [12]. Such techniques could be integrated into the AutoFinity flow, and enable the reuse of learned program behavior.

Under AutoFinity, programs whose behavior is not stored in the behavior database may dynamically change their program affinities (i.e., after sampling an affinity hint is chosen). For some programs, dynamic affinities can cause poor NUMA behavior. Blagodurov et al., propose dynamic techniques to improve NUMA page placement, thus solving this potential issue [2]. Terboven et al. also examined data placement and proposed OpenMP-based solutions [11].

Lee et al. discuss why a program might use different thread counts across invocations [6]. They provide an automatic system to change program thread count. This motivates the need for our work.

Although some programs' were insensitive to the affinity setting, Zhang et al. show how to modify programs to better share caches [15]. With their techniques, programs can have improved performance. The modified programs will then, we expect, require good affinity selection like that provided by AutoFinity.

5 Conclusion

Program affinity can have a large impact on performance; it is important to select the "right" affinity to maximize performance. In this paper, we present AutoFinity, a method to automatically generate a policy that can select at runtime the affinity of a previously unknown program. AutoFinity uses machine learning to derive thread-count-independent policy from training data. We evaluate AutoFinity on previously unknown programs, across a range of thread counts on which AutoFinity may not have been trained. In 12 of 15 cases where affinity has a significant impact on performance, we show that AutoFinity's selection performs within 1% of, or better than, fixed assignments that do not consider program behavior. In two of these cases, AutoFinity outperforms the fixed assignments. AutoFinity is a practical and successful solution to the problem of choosing program affinity.

References

[1] Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: Characterization and architectural implications. In: Proc. of the 17th Int'l Conf. on Parallel Architectures and Compilation Techniques (October)

[2] Blagodurov, S., Zhuravlev, S., Dashti, M., Fedorova, A.: A case for NUMA-aware contention management on multicore systems. In: Proc. of the 2011 USENIX Conf. on USENIX Annual Tech. Conf., USENIXATC 2011. USENIX Assoc., Berkeley (2011)
[3] Dorta, A., Rodriguez, C., de Sande, F.: The OpenMP source code repository. In: 13th Euromicro Conf. on Parallel, Distributed and Network-Based Processing, PDP 2005 (February 2005)
[4] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: an update. SIGKDD Explor. Newsl. (November)
[5] Klug, T., Ott, M., Weidendorfer, J., Trinitis, C.: `autopin` – Automated Optimization of Thread-to-Core Pinning on Multicore Systems. In: Stenström, P. (ed.) Transactions on HiPEAC III. LNCS, vol. 6590, pp. 219–235. Springer, Heidelberg (2011)
[6] Lee, J., Wu, H., Ravichandran, M., Clark, N.: Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In: Proc. of the 37th Annual Int'l Symp. on Computer Architecture, ISCA 2010. ACM (2010)
[7] NAS Parallel Benchmarks Team: NAS parallel benchmarks 3.3.1 (2009)
[8] Radojković, P., Čakarević, V., Verdú, J., Pajuelo, A., Cazorla, F.J., Nemirovsky, M., Valero, M.: Thread to strand binding of parallel network applications in massive multi-threaded systems. In: Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPoPP 2010. ACM (2010)
[9] Song, F., Moore, S., Dongarra, J.: Analytical modeling and optimization for affinity based thread scheduling on multicore systems. In: IEEE Int'l Conference on Cluster Computing and Workshops, CLUSTER 2009 (2009)
[10] Tam, D., Azimi, R., Stumm, M.: Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In: Proc. of the 2nd ACM SIGOPS/EuroSys European Conf. on Comp. Systems, EuroSys 2007 (2007)
[11] Terboven, C., an Mey, D., Schmidl, D., Jin, H., Reichstein, T.: Data and thread affinity in openmp programs. In: Proc. of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?, MAW 2008. ACM (2008)
[12] Tian, K., Jiang, Y., Zhang, E.Z., Shen, X.: An input-centric paradigm for program dynamic optimizations. In: Proc. of the ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications, OOPSLA 2010. ACM (2010)
[13] Wang, W., Dey, T., Moore, R.W., Aktasoglu, M., Childers, B.R., Davidson, J.W., Irwin, M.J., Kandemir, M., Soffa, M.L.: REEact: a customizable virtual execution manager for multicore platforms. In: Proc. of the 8th ACM SIGPLAN/SIGOPS Conf. on Virtual Execution Environments, VEE 2012. ACM (2012)
[14] Wang, Z., O'Boyle, M.F.: Mapping parallelism to multi-cores: a machine learning based approach. In: Proc. of the 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPoPP 2009. ACM (2009)
[15] Zhang, E.Z., Jiang, Y., Shen, X.: Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In: Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPoPP 2010. ACM (2010)

Compiler-Guided Identification of Critical Sections in Parallel Code

Stefan Kempf, Ronald Veldema, and Michael Philippsen

University of Erlangen-Nuremberg, Computer Science Dept., Programming Systems Group,
Martensstr. 3, 91058 Erlangen, Germany
{stefan.kempf,veldema,philippsen}@cs.fau.de

Abstract. There is a huge body of sequential legacy code that needs to be refactored for multicore processors. Especially for control code for embedded systems it is often easy to split the program into multiple threads. But it is difficult to identify critical sections to avoid data races as the legacy code hides its synchronization in a static schedule, priorities and interrupts.

To ease refactoring, this paper presents a new static data-dependence analysis that identifies necessary critical sections in thread-parallel code that does not yet contain any synchronization between threads. A novel optimization pass then breaks up and shrinks the identified critical sections to maximize parallelism while preserving correctness. Our technique proved to be successful in refactoring sequential assembly-like legacy codes in an industry-sponsored project.

But as refactoring projects are hard to evaluate quantitatively and as the domain specific low-level language is of limited interest, we use a standard benchmark suite for which the optimum, i.e., the minimal set of the necessary atomic block annotations is known. We removed the annotations and let the compiler attempt to rediscover them. For 5 out of 7 benchmarks, our compiler identified the same critical sections as the original programmers did by hand. For the other two benchmarks, the compiler found slightly larger (but also correct) critical sections. In all cases, the versions of the benchmarks that the compiler annotated achieved the original run-time performance.

1 Introduction

With the increasing use of multicore processors, the refactorization of sequential legacy applications becomes more important. For certain codes the division of the program into multiple threads is relatively easy. This is especially true for low-level control codes in embedded systems where all the concurrent control loops are sequentialized into a (sequential) static schedule. When refactoring such code for use on multicores, the control loops are obvious but it is difficult to identify the necessary critical sections (atomic blocks) to avoid data races, because these have been implicitly implemented in the static schedule or are hidden behind priorities and interrupts. It is likely to miss a critical section, make it too small or too large. If critical sections are implemented with mutexes, it is also too easy to introduce deadlocks.

To solve this problem, we leave the identification of critical sections to the compiler. It analyzes parallel, but unsynchronized code and determines where critical sections are necessary. By means of static data-dependence techniques our algorithm determines the set of variables that concurrent threads share and it also analyzes the correlations between those variables. Of course, every access to a shared variable must be atomic. But correlations between variables also dictate that these variables are to be accessed within an enclosing critical section. Hence, sets of shared variables and correlations among them determine where critical sections need to be placed in the code. We will show that there are two types of correlations which we call explicit and implicit relationships. Explicit relationships can be detected by the compiler by means of dependence analyses. Implicit relationships must be annotated by the programmer. In order to keep critical sections small, a novel optimization pass breaks up the found critical sections into several smaller ones, if possible. In general, these optimizations work by determining which data dependences do not indicate variable correlations and can therefore be ignored by the compiler. Furthermore, we allow a programmer to use annotations that further help to make critical sections smaller.

With this compiler tool at hand, the refactoring workflow for legacy applications is then as follows. A programmer starts to parallelize the code by dividing it into multiple threads, but without adding critical sections. Next, s/he asks the tool to suggest critical sections. In the (rare) presence of implicit relationships, the critical sections may be too small (the compiler might have placed accesses of correlated variables in different critical sections instead of putting them together into one section), but the programmer at least knows all places where races may occur, instead of having to find them on his own. S/He can add annotations to mark implicit relationships in the code. If a suggested critical section seems to be too large, the programmer can think of ways to restructure the code to allow for smaller critical sections or can add annotations to indicate which data dependences can be safely ignored by the compiler analysis. That way, adding synchronization becomes an iterative process that is less error-prone in comparison to having to add all synchronization manually to large legacy code bases.

Another potentially useful application of the techniques presented in this paper would be a checking-tool. Assume a programmer that writes a new parallel program including the critical sections. The checking-tool would ignore these critical sections provided by the programmer, analyze the unsynchronized code, identify critical sections, and present differences between the generated and the programmer-provided critical sections. The programmer can then use this diff to reason about the correctness of his/her solution.

As the industry automation programs we worked with are written in a proprietary domain specific assembly-like language, we present our technique for a broader audience in a C dialect. We have frontends for both languages that generate the same intermediate representation. The C frontend is a source-to-source C compiler that uses POSIX threads to generate parallel code. It accepts C extended with the following extensions for parallel programming. The statement $t = \mathit{spawn}\ f()$ creates a new thread that starts execution in function f and returns a handle of the thread in t. There is also a statement $\mathit{join}\ t$ that waits until a thread finishes. In order to be able to coordinate the work of multiple threads and to divide it into multiple computational phases, the language also supports barriers. For convenience, the dialect supports a parallel for loop where each iteration spawns a new thread and every thread executes the body of the loop. At the end of the loop, there is an implicit join statement that waits for the threads to finish. Note that our compiler does not automatically parallelize the code, it only adds critical sections (atomic blocks) to the output.

In the related work in Sec. 2, we compare our approach to find critical sections to other approaches. We discuss the basic algorithm of our technique in Sec. 3. Sec. 4 discusses optimizations of the basic algorithm.

When refactoring sequential legacy applications, we cannot judge how well our technique works in practice because we do not know what an optimal refactoring looks like. Hence, in Sec. 5 we evaluate the presented compiler techniques on a standard benchmark suite for which the optimum, i.e., the minimal set of the necessary atomic blocks is known. We use 7 benchmark codes with explicit critical sections. We strip those critical sections from the codes and show that our technique essentially rediscovers the same sections and achieves the same execution times. Annotations are needed in only 2 benchmarks to achieve results similar to the original codes.

The contributions of this paper are (1) a language independent technique that identifies critical sections in the code by analyzing data dependences and a few annotations, and (2) optimizations that further analyze the large number of data dependences in the program in order to detect those dependences that do not indicate correlations between shared variables and therefore do not need to be considered to identify correct critical sections, causing larger sections to be broken up into multiple smaller ones.

2 Related Work

The Abstraction-Guided Synthesis of Synchronization (AGS) algorithm of [19] identifies atomic blocks in programs, by means of programmer supplied specifications that describe which program states may not occur. AGS then uses abstract interpretation of the program to find all interleavings that may lead to invalid states. Provably minimal critical sections are added to the program to avoid such interleavings. An incorrect specification results in incorrect atomic blocks. In contrast, our work identifies (potentially non-minimal) critical sections, but it does not need a specification. AGS' critical sections are also only minimal with regard to the abstraction used in the abstract interpretation. AGS also suffers from the state explosion problem that leads to very long compile times. While we tested our work in an industry-sponsored refactorization project and with a full set of standard benchmark codes, AGS is evaluated with small kernels only that neither use pointers nor dynamically allocated memory which we allow.

An algorithm that detects critical sections must know which variables are correlated and therefore need to be accessed within the same critical section. Unfortunately, our approach cannot detect all correlations. We miss so-called implicit relationships, see below, that the programmer needs to express with annotations. MUVI [13] is an orthogonal approach that searches for such variable correlations that are not explicitly specified in the code. While we rely on the programmer to use additional annotations if necessary, MUVI analyzes a program with critical sections for variables that are accessed together. MUVI uses code metrics (like static distance in the code) to classify whether accesses to different variables belong together. As these are heuristics, the analysis can still miss some correlations. If related variables are updated in an inconsistent manner or if the accesses appear in two different critical regions, MUVI reports this as an error to the programmer. We could integrate MUVI into our compiler and help the programmer in adding explicit annotations for implicit relationships.

In Data-Centric Synchronization (DCS) [18] a programmer declares synchronization constraints that specify up front which data need to be accessed within the same critical section. The programmer annotates a (Java) class and declares which fields in objects need to be accessed together atomically. DCS then automatically infers critical sections. As object fields cannot be accessed through pointers, DCS can find all places that need synchronization and encloses them in atomic blocks. Colorama [4] is hardware support for DCS. The hardware tracks data accesses and decides when to start a critical section, while our algorithm is a purely static approach that does not need hardware support. In contrast to DCS's explicit declarations, our algorithm derives synchronization constraints from data dependences that are present in the code and only relies on annotations where necessary. While it is easy to derive that a critical section must start before the first access to shared data, DCS does not know when a critical section is allowed to end. DCS assumes that a critical section ends at the end of the function that started the critical section. Our algorithm closes a critical section for all affected variables after the last access to any of them. Both approaches can lead to critical sections that are too short. In our case, a critical section may end too early if there are implicit relationships in the program. In practice however, both the heuristics of DCS and our approach lead to correct sections. Nevertheless, in case of implicit relationships, we could also use the type of annotations used in DCS to help the compiler find smaller critical sections. DCS and Colorama use the same benchmarking method that we do. They remove the synchronization from a given parallel program, let the compiler attempt to rediscover critical sections, and compare the results to the original critical sections in given program. While DCS uses the Java Collections package for its evaluation, Colorama was applied to applications such as Firefox and the MySQL database server. In contrast, we use a standard benchmark suite (STAMP) that is available for C.

Race detection tools examine whether two threads may simultaneously access the same shared memory location without holding a lock, either at runtime like FastTrack [6] or statically like Chord [16]. These tools however only show where races may occur, they do not give any hints on how to prevent them. In contrast, our algorithm suggests how to add synchronization to the code to prevent the potential races.

AtomTracker [15] infers atomic blocks from the code and also detects atomicity violations. While we use static analysis, AtomTracker analyzes the memory traces of several concurrent executions of the program to calculate atomic regions.

Using an escape analysis, the work of Bogda and Hölzle [2] removes unnecessary synchronization from Java objects that are used by only one thread. We use an escape analysis as well to decide beforehand which data is thread-private. We can thus avoid to generate unnecessary atomic blocks.

Our paper focuses on the identification of critical sections and their optimization and not on the mechanics to realize the mutual exclusion. We could either use lock inference or transactional memory to implement the critical sections. Lock inference algorithms analyze programs with critical sections and add explicit lock/unlock statements to the code. Current lock inference approaches scale well for large programs. For example, the algorithm of Gudka et al. [7] uses sparse representations of interprocedural dataflow information and various optimizations to speed up dataflow propagations. With transactional memory (TM) [10], critical sections execute atomically inside transactions. A typical implementation records accesses to shared data inside a transaction in a log. At the end of the transaction, the implementation attempts to commit by atomically writing all changes to shared data back to main memory. If another critical section/transaction modified a subset of the shared data that was touched by the committing transaction, the committing transaction aborts and retries to execute the atomic block. For simplicity, we use TM to implement the atomic blocks that our compiler detects to be required for correctness.

3 Identification of Critical Sections

The high-level approach of our technique is to detect correlations between shared variables that are used by multiple threads. These correlations determine where critical sections need to be placed in a program. As we will show, there are two classes of correlations, which we call explicit and implicit relationships. Explicit relationships can be detected by dependence analyses, while implicit relationships need to be annotated by the programmer.

To identify critical sections, the compiler must first analyze which statements of the program may execute concurrently to each other. An Andersen-style interprocedural alias analysis (similar to the algorithm of [9]) extended with an escape analysis [17] that is used to identify thread-local data then conservatively approximates the sets of variables that these statements use. The intersection between read and written variables of those sets gives the sets of shared variables. Of course, thread-local variables do not belong to the set of shared variables. In the remainder of this paper, we call a statement of a thread a *concurrent instruction* if it statically touches shared data. Without synchronization, such statements are prone to race conditions and need to be put into critical sections. Using dependence analyses, our compiler then solves the problem of how concurrent instructions need to be grouped into critical sections, i.e., where critical sections need to start and end.

Sec. 3.1 first discusses correctness issues and limits of our technique to identify critical sections in a program. Sec. 3.2 explains in more detail how the compiler finds concurrent instructions. The algorithm explained in Sec. 3.3 then calculates (large) atomic blocks that guarantee correct synchronization. Sec. 3.4 illustrates the algorithm with an example.

3.1 Correctness Considerations and Limits

Let us briefly consider correctness first. Assume a collection of threads that accesses shared variables a to d. These variables may or may not affect each other. For example, the value of b may be calculated from the value of a, which is a classical flow data dependence [14] that a compiler can detect. An invariant could be that c always has the same value as d. Although invariants often are essential for the correctness of the program, the compiler in general cannot derive them from the code. Data dependences[1] between shared variables are *explicit relationships* between variables. A program can also contain invariants/correlations between variables that have no data dependences between. These are correlations implied due to program logic. We call them *implicit relationships*. Our view of relationships is that once a shared variable a changes its value, all shared variables b related to a must appear to update their values at the same time, i.e., in the same (generated) critical section (instant update policy). If one statement uses several shared variables, we treat the variables as related as well. Hence, all statements that access related variables (explicitly or implicitly) need to execute atomically within the same (generated) critical section.

All the statements X that access related variables are spread over the code. At runtime their critical sections must start before the *first* of all statements X is executed and must end after the *last* of them. To statically generate a critical section in the code, the analysis needs to calculate *first* and *last*. *First* dominates[2] all statements X and *last* post-dominates them. *First* also dominates *last* and *last* post-dominates *first*. All instructions that lie on a path from *first* to *last* also belong to the critical section. *First* and *last* therefore correspond to the entries and exits of single-entry-single-exit regions in the program structure tree by Johnson [12].

```
collection_t a = { x };   void run2() {
collection_t b = {};          l1 = 1;
void run1() {                 r1 = 1;
    int i = 0;              }
    while (true) {
        i++;                void run3() {
        if ((i % 2 == 0)) {   l2 = l1;
            a.remove(x);      r2 = r1;
            b.insert(x);    }
        } else {
            b.remove(x);
            a.insert(x);
} } }
```

Fig. 1. Implicit relationships

For programs with solely explicit relationships, our algorithm guarantees correct synchronization. Implicit relations pose limits to a fully automatic approach. Fig. 1 shows two different examples. Within function *run1*, either one of the two collections a and b holds the element x at all times. To cope with the absence of a data dependence between the collections, the programmer can add an annotation $related(a, b)$ that renders the relationship between a and b explicit. This causes the compiler to put accesses to a and b together into a critical section. In the second example, *run2* and *run3* execute concurrently. In a sequential execution, where *run2* is called before *run3*, the variables *l1*, *l2*, *r1*, and *r3* would have the same value. If *run2* and *run3* executed concurrently, one critical section enclosing the assignments to *l1* and *r1*, as well as another critical section around the assignments to *l2* and *r2* are necessary. However, as there are no explicit data dependences between these variables, the analysis cannot establish the necessary synchronization. If it is necessary for the algorithm that *l1*, *l2*, *r1*, and *r2* store the same value, a *related* annotation is needed to enable the compiler to identify the necessary critical sections. There is also an annotation $unrelated(a, b)$ to explicitly break explicit relationships between variables that are dependent, but that

[1] Unless otherwise noted, a dependence is a flow dependence in this paper.

[2] Statement A dominates statement B if A appears on every control flow path from the entry of the function to B. A post-dominates B if A appears on every control flow path from B to the exit of the function.

do not need to be in the same critical section. Both types of annotations extend the applicability of our approach in an iterative refactoring tool.

Our approach exploits data dependences and the above annotations to determine critical sections. Note that this includes control dependences, because they are converted into data dependences using techniques from Muchnick [14].

Due to aliases and function pointers the conservative analysis cannot avoid false positive data dependences. These cause larger than necessary (but sill correct) critical sections. While our algorithm already works well on nicely structured programs in legacy languages, modern managed languages offer fewer possibilities to introduce aliases and are therefore even better to analyze, enhancing the applicability of our technique. Furthermore, when our technique is used during the development of new parallel applications, early warnings can indicate too coarse-grained atomic blocks, and help the programmer to improve the code by reducing dependences between variables. As asserts and annotations become more common in new languages, relationship annotations are easy to digest.

Other approaches to infer critical sections such as AGS and DCS always need the programmer to provide explicit specifications in order to work correctly. In contrast, our algorithm works with plain code and just needs annotations in presence of implicit relationships.

3.2 Concurrency Analysis

The first step in the identification of critical sections is the analysis that determines which statements may execute concurrently to other statements. Consider the code in Fig. 2 that creates and destroys multiple threads.

```
int a, b, c;

void f() { F: a++;      // M2–M6, G, H }
void g() { G: b++;      // F, M4–MC, H, I, J }
void h() { H: b++;      // M5–MC, F, G, I, J }
void i() { I: j();      // MC, G, H, I, J }
void j() { J: c += b;   // MC, G, H, I, J }

void main() {
M0: pthread_t u, v, w;
M1: a++;                                  //
M2: t = spawn f();                        // F
M3: a++;                                  // F
M4: u = spawn g();                        // F, G
M5: v = spawn h();                        // F, G, H
M6: join t;                               // F, G, H
M7: a++;                                  // G, H
M8: if (random() > 5) { w = u; }          // G, H
M9: else { w = v; }                       // G, H
MA: join w;                               // G, H
MB: parfor (i = 0; i < N; i++) {          // G, H
MC:       i();
// G, H, I, J
} }
```

Fig. 2. Concurrency analysis

To find out which statements may execute concurrently, we use a may-happen-in-parallel analysis [8]. The results of this analysis are given in Fig. 2. The comments show for every statement which other statement may run concurrently to it. For example, after t (function f) is spawned in *main*, statement F may run in parallel to the statements *M2* to *M6* in *main* up to the join on t. On the other hand, *M2* to *M6* as well as G and H may run concurrently to F, because u and v are spawned while t is running. Also note that the parallel for loop spawns multiple instances of i. Therefore, I runs concurrently to itself in i. As either u or v is still running when the parallel for loop is entered, G and H may still execute concurrently to the loop body (*MC*) as well as concurrently to I. As i calls j and multiple instances of i run concurrently to each other, I is concurrent to J.

The compiler also analyzes (with a standard alias and escape analysis) which data a thread may touch. For every thread, the compiler examines all statements of the thread and collects all accessed variables in the thread's a read/write set. Given a spawn statement *x* = *spawn r*, the statements of x are the statements in r plus all the statements in functions that are transitively called by r. In the example, the statements of t, u, and v are F, G, and H. The statements that belong to the threads spawned in the parallel for loop are I and J. Thread t reads/writes a, threads u and v read/write b. The threads spawned in the loop read b and write c. To determine which data is shared by two threads x and y, we intersect the read/write set of x with the write set of y and vice versa. We call the statements that may access shared data *and* that may execute concurrently to other statements *concurrent instructions*.

In Sec. 4.4, we will show that an instruction can be falsely identified as a concurrent instruction that is in fact executed by only one thread. In that section, we also illustrate two optimizations that reduce false positives in the identification of concurrent instructions.

3.3 Basic Algorithm

We explain the basic algorithm that finds atomic blocks with the pseudo code in Fig. 3. The algorithm starts by calling *get_concurrent_insns* to interprocedurally find all concurrent instructions in which two threads may access shared data at the same time. How this is done has been discussed in the previous section.

One aspect worth mentioning is that *get_concurrent_insns* also analyzes barriers in the program. Statements in different computational phases separated by a barrier never execute concurrently, since the next phase only starts as soon as all threads have finished their previous phase. Hence, a critical section can never span two computational phases divided by a barrier. An algorithm for the division of code into barrier-separated disjoint phases is given by Jeremiassen et al. [11]. All further steps explained later take those phases into account when they calculate critical sections, but for simplicity we will not mention them any further.

The algorithm then has to bundle all instructions that access related shared variables and all statements that must be executed between those instructions into a single critical section. We call such a bundle of instructions a *partition*. In the pseudo code, every single concurrent instruction starts a partition of its own that *part_intersect* and *flow* will later combine. We describe the optimization *remove_superfluous_dependences* in Sections 4.1 to 4.3. *Part_intersect* compares all pairs of partitions to combine those partitions that access a common subset of shared data. *Merge* calculates the *first* and *last* instructions for every partition and also adds all dominated and post-dominated statements in between. *Flow* compares all pairs of partitions to check whether they have statements with flow dependences/explicit relationships to the data of the partitions and that hence must also be in the merged partition.[3]

[3] There is a flow dependence between two partitions *P* and *Q* if there is a statement *S1* in *P* and a statement *S2* in *Q* and there is a path in the dependence graph from *S1* to *S2*.

```
void basic_algorithm() {
  s = get_concurrent_insns();
  foreach i in s {
    part[i] = new Partition();
    vars(part[i]) = shared_vars(i);
  }

  part_intersect();
  remove_superfluous_dependences();
  flow();
  flatten();

  // Insert atomic blocks
  foreach p in part {
    wrap_atomic_around(first(p),
                       last(p));
} }

void part_intersect() {
  foreach p in part {
    foreach q in part {
      if (p == q) { continue; }
      set = intersect(vars(p),
                      vars(q));
      if (set != {}) {
        merge(p, q);
} } } }
```

```
void flow() {
  foreach p in part {
    foreach q in part {
      if (p != q) {
        if (dep_chain(p, q)) {
          merge(p, q);
} } } } }

void flatten() {
  foreach p in part {
    // Returns all instructions
    // between first(p) and last(p)
    foreach i in dfs(first(p),
                     last(p)) {
      if (part[i] != p) {
        merge(p, part[i]);
  } } }
  interproc_merge();
}

void merge(Partition p,
           Partition q) {
  foreach i in insns(q) {
    part[i] = p;
    vars(p) += vars(q);
  }
  update_first_and_last(p);
}
```

Fig. 3. Basic algorithm

After *flow*, the resulting partitions are correct critical sections. But they may overlap. Two partitions p and q overlap if on any path from *first*(p) to *last*(p) there is an instruction that belongs to partition q or vice versa. We merge nested or overlapping atomic blocks because most STM implementations do not support them. *Flatten* therefore performs an intraprocedural depth-first search from the *first* to the *last* instruction of a partition. As *last* post-dominates *first*, a depth-first search from *first* will eventually reach *last*. If on any search path there is an instruction belonging to another partition, the two partitions are merged. After the depth-first search, *flatten* calls *interproc_merge* to remove atomic blocks that nest within the call hierarchy. For example, a function f may contain an atomic block that calls g, which itself contains an atomic block. The block in g is nested within the block in f. An interprocedural analysis walks the call-graph to find call-paths where callees always execute within an atomic block of a caller. Then the callee's atomic blocks can be removed. If g is called both from an atomic context and from a non-atomic context, function cloning generates two versions of g. From an atomic context, the version of g without the atomic block is called. Otherwise, the version of g that contains an atomic block is used. Currently, our approach can only handle function pointers if the pointed-to functions provably do not contain atomic blocks. To lift this restriction, the compiler could generate two versions—one version with atomic blocks and one version without—of all functions that have the same signature as the function pointer and use a runtime mechanism to direct calls via function pointers to the proper function in order to avoid nested atomic blocks.

After *flattening*, each remaining critical section is wrapped into an atomic block.

3.4 Sample Execution

We now apply the algorithm to the example in Fig. 4. Its shared variables are the *Nodes* that are held in the *inputs* array, *output*, *next_input*, *counter*, and *double_counter*. The concurrent instructions are marked with *S1* to *S8*. These are concurrent instructions because the *parfor* loop created multiple threads that all execute the same function *process*. Fig. 5 shows the dependence graph of the example's *process* code. Statements that belong to a partition are marked with *P1* to *P4*.

```
struct Node {
  Node *next;
  int value;
} *inputs[N], *output;

int next_input;
int counter, double_counter;

void main() {
  for (int i = 0; i < N; i++) {
    Node *n = malloc(sizeof *n);
    n->next = NULL;
    n->value = random();
    inputs[i] = n;
  }
  parfor (i = 0; i < K; i++) {
    process();
} }
```

```
void process() {
  int index = 0;
  while (index < N) {
S1:  index = next_input;
S2:  next_input = index + 1;

     if (index >= N) { break; }

X:   Node *n = inputs[index];
Y:   int val = n->value;

Z:   int inc = val * index;
S3:  int next = counter + inc;
S4:  counter = next;
S5:  int t = counter;
S6:  double_counter = 2 * t;

S7:  n->next = output;
S8:  output = n;
} }
```

Fig. 4. Example program for atomic block identification

To avoid races, three atomic blocks are needed. First, *next_input* must be incremented atomically (*S1*, *P1*). Second, both partitions *P2* and *P3* must be in a critical section to satisfy the explicit relationship that *double_counter* (*S6*, *P3*) is always twice as big as *counter* (*S5*, *P2*). Third, prepending n to the linked list pointed to by *output* must be atomic. But since all four partitions of Fig. 5 are linked to each other by flow dependences, the basic algorithm combines them into one correct but too large critical section and generates the code in Fig. 6.[4] Sec. 4 presents optimizations that achieve a more fine-grained synchronization. No annotations were needed in this example.

[4] As there is a chain of dependences between the statements $index = next\_input$ (in partition *P1*), $inc = val * index$, and $next = counter + inc$ (in partition *P2*), *flow* merges *P1* and *P2*. There are additional chains of flow dependences from statements in *P2* to instructions in *P4*, as well as from *P3* to *P4*. Both *P4* and *P3* join that partition. As there are no overlapping partitions in the running example, *flatten* shows no effect here.

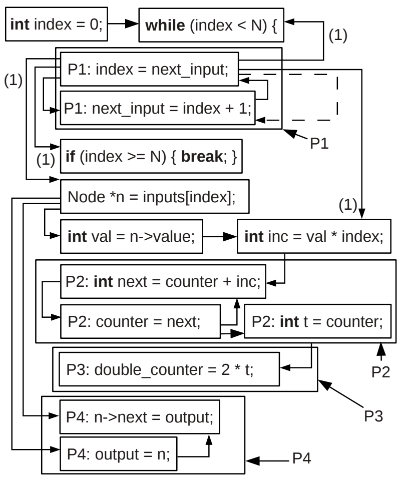


Fig. 5. Example program with flow dependences (full arrows)

```
void process() {
  int index = 0;
  while (index < N) {
    atomic {
S1:   index = next_input;
S2:   next_input = index + 1;

      if (index >= N) { break; }

A:    Node *n = inputs[index];
B:    int val = n->value;

C:    int inc = val * index;
S3:   int next = counter + inc;
S4:   counter = next;
S5:   int cur = counter;
S6:   double_counter = 2 * cur;

S7:   n->next = output;
S8:   output = n;
} } }
```

Fig. 6. Synchronized code for Fig. 4

4 Optimizations

The basic algorithm calculates a correct synchronization, but it is sometimes too conservative, as *flow* from Fig. 3 merges any partitions with dependences between them. If we can remove unnecessary dependences between partitions, i.e., dependences that need not be considered to ensure a proper synchronization, large critical sections break up into smaller ones. This allows more parallelism.

For the discussion in this section, always assume that multiple threads execute the code being discussed.

4.1 Removal of Non-critical Dependences

Function *plain* in Fig. 7 writes the shared variable *shared1* and copies it to thread-local variables x and y. If executed sequentially, all four variables have the same value after the final assignment to *shared2*. This invariant can be discovered by analyzing flow dependences. It is necessary for parallel execution and preserves the invariant to put *all* four flow dependent statements of *plain* into a single critical section. Assume *flow* would not have merged the two partitions for the two shared variables. With the resulting two critical sections (around the first two statements and around the final assignment) one thread could modify *shared1* while the other assigns the old value of *shared1* that it keeps in y to *shared2*. This would break the invariant.

```
void plain() {      void uncritical() {
  shared1++;          int x = shared1;
  int x = shared1;    shared1++;
  int y = x;          int y = x;
  shared2 = y;        shared2 = y;
}                   }
```

Fig. 7. Motivation for removal of non-critical dependences

Of course it depends on the application whether it is necessary to satisfy the invariant in a parallel execution. But to conservatively ensure correctness, our technique must make sure that the identified atomic sections satisfy the invariants. If an invariant is not necessary, an annotation could be used. In the example, marking *shared1* and *shared2* as unrelated would lead to different atomic sections.

The code in *uncritical* is slightly different and demonstrates optimization potential. (Assume that the code is free of implicit relationships as there are no annotations). The code first stores a copy of *shared1* in x. Then it increments *shared1*. Because the statements work on the same shared variables and since they are flow dependent, the basic algorithm would again create an embracing critical section. But now the discoverable invariant is that the three variables x, y, and *shared2* have the same value. There is no explicit relationship between *shared1* and *shared2* anymore. Hence it is sufficient to put the first two assignments into one critical section and to put the assignment to *shared2* in a second critical section.[5] The assignment to y operates only on thread-local data and thus does not need to be in a critical section at all. Even if a concurrent thread changes *shared1* between those two atomic blocks, x, y, and *shared2* will remain unaffected. To generate a correct synchronization, *flow* therefore can ignore the dependence between the assignment to x and the assignment to y.

More formally, the *remove_superfluous_dependences* optimization (Fig. 3) looks for statements s and a that meet the following criteria: s reads some shared variable, i.e., it belongs to some partition P. Statement a postdominates s and is must-antidependent on s, i.e., it is guaranteed to overwrite the variable that s has read (*shared1*++ in *uncritical*). Notice that as a writes the same shared variable, it belongs to P as well. If there are such statements s and a, then *flow* can ignore all outgoing dependence edges of s (the one from x to y in the example).

To see why the optimization is correct, assume that there is a flow dependence between s and another statement t. In case *part_intersect* has already put both s and t into P, they belong into the same critical section, no matter what dependences exist between the two statements. In the other case, i.e., if t is outside of P, there is room for optimization. If t is in another partition Q or if s and t are part of a dependence chain that ends in another partition Q, *flow* merges P and Q so that s and t are in the same critical section. In the example t is the assignment to y that is linked to the update of *shared2* which is in the second partition. Now the statement a comes into play. As a

```
void process() {
  int index = 0;
  while (index < N) {
    atomic {
S1:   index = next_input;
S2:   next_input = index + 1;
    }

    if (index >= N) { break; }

A:  Node *n = inputs[index];
B:  int val = n->value;
C:  int inc = val * index;

    atomic {
S3:   int next = counter + inc;
S4:   counter = next;
S5:   int cur = counter;
S6:   double_counter = 2 * cur;
    }

    atomic {
S7:   n->next = output;
S8:   output = n;
} } }
```

Fig. 8. Optimized result for Fig. 4

[5] Of course there is an implicit invariant $x == y == \mathit{shared2} == \mathit{shared1} - 1$. But this invariant cannot be found by flow dependences (i.e., explicit relationships) only.

belongs to *P* and post-dominates s, and as t is outside of *P*, a *always* executes between s and t and it *always* modifies the shared data used by s and t. Because s and t are part of a dependence chain from *P* to *Q*, we cannot construct an invariant from this chain that involves the shared data between *P* and *Q*, as a always overwrites that data. Thus, the flow dependence edge between s and t does not indicate an explicit relationship and can be removed in *flow*. If there is no other dependence chain between *P* and *Q*, these two partitions are not merged and result in separate critical sections.

Remove_superfluous_dependences examines every concurrent statement s in every partition P in some arbitrary order. From s it traverses all outgoing antidependence edges that lead to an antidependent statement a. If a post-dominates s and if the results of the alias analysis indicate that a always overwrites the shared data used in s, then all outgoing flow dependence edges are removed from s

This optimization is applicable to the example of Figs. 4 and 5. After *next_input* is read and copied to *index* by *S1* (statement s), *next_input* is changed by *S2* (antidependent statement a, dashed arrow). Hence, it is possible to increment *next_input* in a critical section of its own, as it cannot affect the behavior of flow dependent statements X and Z. Note that statement X then still loads n from the proper array element. *Flow* can ignore all edges marked with (1) in Fig. 5 and only merges partitions *P2* and *P3* because of data dependences. The resulting code in Fig. 8 has the increment of *next_input*, the counter updates, and the list prepending in separate critical sections.

4.2 Removal of Dependences over Read-Only Variables

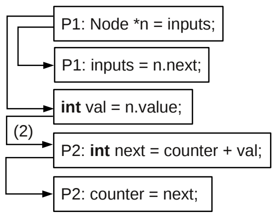


Fig. 9. Dependence over read-only data

We can also safely eliminate dependences over pointer accesses in concurrent instructions that provably only reach read-only data. Assume that a few threads concurrently execute the code with its dependence graph in Fig. 9. The difference to Fig. 4 is that the *Nodes* are no longer read from an *inputs* array. Instead, they are removed from a linked list. *Part_intersect* has identified the partitions *P1* and *P2*. As there is a chain of dependences from *P1* to *P2*, *flow* would merge those partitions. But it is correct to keep *P1* and *P2* in two critical sections, because after n is removed from *inputs*, all accesses to it are read-only. No thread could concurrently write to n. The only dependence entering *P2* from outside comes from a read-only access to n. Once the code has retrieved n from *inputs*, it does not matter how much later the code in *P2* is executed, as the memory reachable from n (that *P2* depends on) will always contain the same values.

In the example, the edge marked with (2) is removed which causes *P1* and *P2* to become individual atomic blocks.

Remove_superfluous_dependences examines all flow dependence edges between any two statements s and s'. If s' dereferences a pointer p and if all memory locations transitively reachable from p are read-only (according to the results of the alias analysis), then no concurrent modification between s and s' can influence the effects of these two statements. Thus, no matter how long the time span between s and s' is, s' always appears to execute instantly after s. Hence, if s and s' are linked in a dependence chain that starts in a partition P and that ends in another partition Q, an update of shared variables in P always appears to happen instantly with the corresponding update in Q, even if P and Q belong to different critical sections. This allows *flow* to safely remove the dependence between s and s'. If there is no other dependence chain between P and Q, these two partitions can execute in separate critical sections.

In the example, all threads executed the same function and the fields of n were never written. Recall from Sec. 3.2 that the compiler analyzes which instructions may execute concurrently. In the above analysis, if the compiler finds an instruction that accesses p, it checks which set T of threads may execute concurrently to this instruction. If all threads in T only read all memory locations transitively reachable from p, then the optimization above is correct. That means, threads that write to memory locations reachable from p are still possible. As long as those threads are not in T, the optimization is applicable.

4.3 Removal of Dependences between Builtin Data Structures

In Fig. 10 a *pop* routine of a hand-written stack is called. There is a chain of dependences from the internal representation of the stack (an array of items) to the access of the returned item after the *pop*. This forces all the statements into a single critical section. In contrast, with opaque thread-safe library data structures with known semantics as they are common in modern languages (Java, Python, etc.), there is no such dependence chain, resulting in two smaller critical sections one for *pop* and one for the access to the popped item.

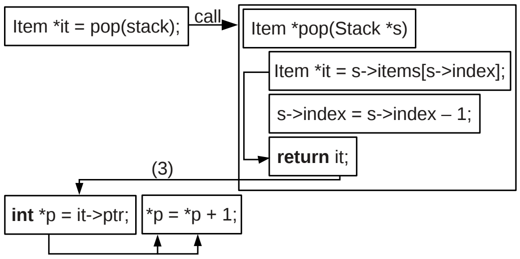


Fig. 10. Dependences over a builtin data structure

Note that compiler analysis is sped up as well because the library codes do not need to be analyzed. For this optimization to be correct, there may be no implicit relationships between a black box builtin container data structure and and other shared data. Otherwise *related* annotations are needed. *Remove-_super-fluous_dependences* finds all calls of functions that are known to be thread-safe and removes outgoing dependences from those calls. In the example, the edge marked with (3) is removed.

4.4 Detection of Concurrent Instructions That Execute in a Single Thread

All threads that execute the code in Fig. 11 share V and A. Although the statements *S1* to *S3* appear to be concurrent instructions, *S1* is *not* because only the thread with ID 0 executes it. *S3* is *neither* a concurrent instruction as all threads access different elements of A (*threadId()* is unique for each of them).

```
void foo() {
        if (threadId() == 0) {
S1:         V = 6;
        }
        barrier();
S2:     int t = ++V;
S3:     A[threadId()] = t;
}
```

Fig. 11. Concurrent instructions executing in one thread

Hence, to eliminate concurrent instructions that run in only one thread, we first check if there is a computational phase that matches the construct *if* $(threadId() == x)\{...\}$. Then only one thread executes the body which is thus free of concurrent instructions. Second, two statements accessing the same array are considered concurrent instructions unless their subscript expressions have the form $a * threadId() + b$, with constant values for $a \neq 0$ and b. Is this case we know that every thread accesses a different array element. Dependence analysis as used in loop parallelization [1] can further improve subscript analysis.

5 Evaluation

Methodology. Although we have developed our techniques for a refactoring tool, we discuss an evaluation that uses a parallel benchmark suite. The evaluation of our approach is designed to show that (a) the identified critical sections are correct and small, and (b) the execution times of the code generated by our compiler match the execution times of code written by an expert programmer. We let our compiler analyze programs from STAMP [3], a benchmark suite for testing transactional memory implementations. The code contains atomic sections that permit as much concurrency as possible. For our evaluation, we remove the atomic start and atomic end statements and use the resulting unsynchronized thread-parallel codes as input.

We use this method of evaluation because it is hard to evaluate how good a refactoring tool is. For a given sequential legacy application it is simply unknown what the best parallel refactorization is. Thus there is no way to compare the results of a tool to some unknown optimal result. We could therefore make no sound statements about the quality of our approach.

Therefore, we compare the number and sizes of the atomic blocks in STAMP with the atomic blocks generated by our compiler. If they match, our algorithm works. In case of variations, the runtime effects are interesting. Hence, we also compare execution times with SwissTM [5] as target platform for the measurements. We also use STM instead of lock inference because STAMP is an STM benchmark that does not have lock/unlock statements in the code either. Because the critical sections in STAMP are (almost) minimal, if an automated approach can identify identical critical sections and generate code that is equally fast, the automatic approach is good.

This general evaluation methodology to use existing programs, remove all the synchronization that is present in the codes, and then let the tool try to rediscover them is also used in the evaluation of DCS [18] and Colorama [4].

We ran all benchmarks on a 2.66 GHz, 8 core Xeon (X5550) with 8 MB cache and 24 GB main memory, with Linux 2.6, using one, two, four, and eight cores. We excluded the *Ssca2* benchmark as even the original STAMP version already always crashed in our setup.

Setup and Measurements. Table 1 holds the lines of code, the times t_A for the alias analysis, t_E for escape analysis, t_D for the initial construction of the dependence graph, t_I for the identification of atomic blocks, and t_T for the total compile time. As we ported STAMP to our C dialect first, we report the number of lines of the ported versions. Column *orig* contains the original number of atomic blocks in STAMP, columns *gen* and gen_{opt} give the number of blocks identified by our basic algorithm and with optimizations enabled. Column *opts* lists the applicable optimizations (sub-section numbers).

Table 1. Lines of code, compile times, and atomic block details

Benchmark	Lines	t_A	t_E	t_D	t_I	t_T	*orig*	*gen*	gen_{opt}	*opts*
Vacation	504	0.09s	0.02s	0.15s	0.01s	0.27s	3	3	3	none
Intruder	3266	0.97s	1.66s	0.55s	0.04s	3.28s	3	2	3	4.3, 4.4
Bayes	5856	1.23s	0.33s	0.77s	0.14s	2.47s	2+13=15	2+1=3	2+3+1=6	4.3, 4.4
									2+13+1=16	+ annot.
Yada	6292	0.49s	0.97s	1.09s	0.14s	2.69s	6	3	5	4.3
									7	+ annot.
Genome	1861	0.19s	0.04s	0.33s	0.18s	0.74s	5	7	5	4.4
Labyrinth	2880	0.21s	0.04s	0.46s	0.01s	0.72s	3	3	4	4.2
Kmeans	742	0.08s	4.03ms	0.16s	3.68ms	0.25s	3	2	4	4.1, 4.4

To enable optimization 4.3 in STAMP, we treat its container structures such as lists, heaps, etc. as builtin.

Fig. 12 shows the improvements of the execution times of the optimized over the unoptimized code generated by our compiler and compares the runtimes of the optimized code to the original STAMP versions.

General Results. First, and most importantly, our algorithm generates correctly synchronized programs. Second, the number of atomic blocks found by our algorithm with optimizations enabled (gen_{opt}) is in good accordance with the number of atomic blocks present in the original benchmarks (*orig*). We will discuss differences below. Third, even if the optimizations do not lead to the same blocks, the execution times of our code are similar to the original benchmarks. Fourth, the optimizations are essential because the execution times of the programs generated without them are significantly slower. These results show that our approach is helpful for programmers.

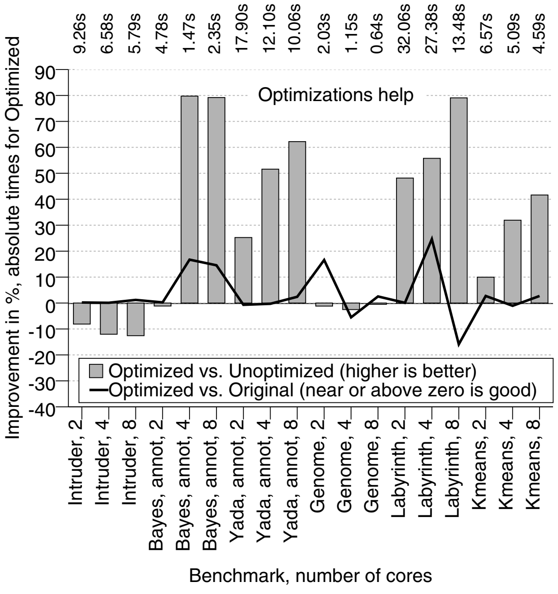


Fig. 12. Execution times

Compile Times. With at most 3.3 seconds they are good and an improvement over the related AGS algorithm (see Sec. 2), which reported compile times to be less than 10 minutes for algorithmic kernels instead of full benchmarks. The time for the identification of atomic blocks generally is between 1.2% and 24.3% of the overall compile time. The alias, escape and dependence analyses are slowest but improvable, as we use an unoptimized $O(n^3)$ Andersen style alias analysis, a reaching definitions analysis to calculate dependences, and an unoptimized implementation of an escape analysis.[6] We did not invest in more advanced state-of-the art analyses as the basic ones were sufficient and let us focus on the identification of critical sections.

We now examine the results of our algorithm and the execution times for the individual benchmarks. We skip single core results as atomic blocks do not matter in absence of parallelism.

Perfect Matches. Our compiler finds *exactly* the original atomic blocks for *Intruder* and *Vacation*. For *Vacation* no optimizations are needed. So we skip the runtime numbers as there is no effect to be seen. For *Intruder*, the version with optimization has the same speed as the original code. Surprisingly, *without* optimizations (2 atomic blocks), the code with 3 atomic blocks is between 7.5% and 11.2% faster. Fewer and larger atomic blocks are beneficial for STM performance if transaction setup can be saved for tiny and adjacent atomic blocks. This result shows potential for STM optimizations and improvements of the atomic block annotations in STAMP.

Good Matches; Annotations Needed. For *Bayes* and *Yada* we discover all atomic blocks as they are in STAMP, which then leads to the same execution times. But we need some annotations. *Bayes* turns out to be the hardest benchmark. It contains two computational phases. Let us first discuss the results without annotations. Phase (1) creates a list of tasks that are then used by phase (2). Phase (1) has two atomic blocks which our compiler detects. The first block updates a global variable, the second block adds a task to the list. The difference to the original STAMP code is that our compiler puts the task creation into the atomic block, as it falsely assumes that the task is shared at that point. But this does not cause any slowdown. Phase (2) contains 13 atomic blocks in STAMP. Our compiler generates 3 larger blocks (plus one unnecessary block) because it has to conservatively respect some dependences. With added pseudo-barrier annotations, the 3 blocks are split into 13 separate critical sections. A pseudo-barrier is a no-op at runtime but like for an ordinary barrier, the analysis exploits that atomic sections cannot span barriers. The superfluous block is small. It holds an access to the task data structure that is thread-private at that point. This is harmless at runtime as the logging overhead of the STM is negligible and as the task structure at that point is thread-private. The optimizations are important, as they improve runtime by 79.8% to 79.2% over the unoptimized code. But the additional transaction setup overhead of more atomic blocks can only be amortized at higher thread counts.

Annotations are important for *Yada* as well. The original *Yada* code has six atomic blocks. The shared data structures are a mesh, elements of the mesh, a heap, and two global variables. Every thread executes a loop that contains the atomic blocks. Block (1) removes an element from the heap. Block (2) checks if the element is invalid. If it is, the code starts another loop iteration. Block (3) refines the mesh. Block (4) marks the

[6] Our escape analysis interprocedurally propagates the escape information until a fix-point is reached. Propagation starts at the *main* function of the program. If there is a call to a function f in the program, we (repeatedly) visit f to propagate the information, which is costly.

element removed from the heap as unreferenced. Block (5) adds elements recognized as bad during the execution back to the heap. After the loop, block (6) updates two global variables. With optimizations enabled, we exactly rediscover blocks (1) and (2). As blocks (3)-(5) share data dependences and touch the same data, *part_intersect* and *flow* merge them into one large block. Again, pseudo-barrier annotations help to split this block into the original three parts. The generated code runs at the same speed as the original STAMP code. As block (6) updates two different global variables, our compiler inserts two correct atomic blocks here. The variables are independent and they are only used by the main program once the threads exit. Again, the optimizations are important as the code is between 25.3% and 62.2% faster than the unoptimized version.

Close Matches with Irrelevant Variations. For the remaining three benchmarks, our compiler generates essentially the same atomic blocks as the STAMP codes if optimizations are enabled. This results in identical execution times.

For *Genome* our technique generates similar atomic blocks as the original version. There are three minor differences in the generated blocks. First, the benchmark performs a number of atomic insertions to a hash table in one loop. The original version puts the atomic block around the loop, probably because transaction setups are costly. Our compiler puts the atomic block inside the loop, which is functionally equivalent. Second, the original code has two atomic blocks that call a table insertion routine. Our generated code has the atomic blocks in the insertion routine, which is essentially the same. Third, the third block in our version is a a superfluous block similar to *Bayes* where the compiler conservatively assumes that a statement touches shared data. At the statement, the variable has become thread-private however, which the analysis does not recognize. As this atomic block is small, its overhead at runtime is negligible for the same reasons as for the extra block in *Bayes*. The execution times are essentially the same. If the atomic block runs in the loop, the transactional logs per iteration are smaller and lookups to it are faster. These savings outweigh the additional transaction creation setup. Again, on two cores there is not enough parallelism to smooth over different transaction setup costs, so that our code runs faster than the original STAMP code. Without optimizations, there are two additional blocks executed by only one thread. As they are small and transaction conflicts are impossible, the optimized and unoptimized versions have the same execution times.

Our technique generates more and smaller atomic blocks for *Labyrinth* and *Kmeans*. We re-discover all original atomic blocks in *Labyrinth* plus one additional block. This block is added to an insert method of a linked list that also increments the size field of the list. As there are no dependences (i.e., explicit relationships) between the pointer update and the size increment, our compiler puts these two parts of the code in separate atomic blocks. As *Labyrinth* does not use the list's size field, it runs correctly with this change. Otherwise, the programmer would have to mark the list entries and the size field as related. The small overhead of the additional block shows little runtime effects except for occasional jitter. The optimizations improve runtimes by 48.2% to 79.1%.

For *Kmeans* we generate 4 instead of the original 3 atomic blocks. Fig. 13 shows where the difference comes from. Nevertheless, the execution times are identical. The threads share the arrays *len* and *C*. Our finer synchronization is correct as *len* and *C* are independent, and the loop iterations are independent as well. STAMP uses one atomic

```
atomic {
    len[idx] += 1;
    for (j = 0; j < N; j++) {
        C[idx][j] += F[i][j];
} }
```

(a) STAMP

```
atomic { len[idx] = len[idx] + 1; }
for (j = 0; j < N; j++) {
    atomic { C[idx][j] += F[i][j]; }
}
```

(b) Our output

Fig. 13. Atomic blocks in STAMP and found by our technique

block outside of the loop probably because transaction setup is assumed to be costly. Again, optimizations proved to be effective because they make the code run between 10.0% and 41.6% faster than the unoptimized version.

6 Conclusions and Future Work

To aid programmers in adding critical sections to parallel code, (or even take this burden off of the programmer), we presented an algorithm that detects critical sections in parallelized, but unsynchronized code. We used our technique in a refactoring tool for embedded industry automation applications written in a low-level language.

Whereas previous approaches to detect critical sections always needed explicit specifications in order to be able to infer critical sections, our technical contribution is an algorithm that starts from the data dependences in the plain code (and only needs some annotations) and discovers the necessary critical sections. Several important optimizations have been presented that help to keep the atomic blocks small. These optimizations detect edges in the dependence graphs that do not need to be considered by the algorithm to detect correct critical sections.

We used the STAMP benchmarks to show that our approach often detects the same set of atomic blocks that an expert programmer would add to the code. In most cases, no annotations were needed in order to generate a correct synchronization. Implicit relationships between variables seldom are a problem for our approach in practice. The execution times for the codes generated by our compiler almost always match the execution times of the manually synchronized versions.

Future work based on the presented technique can address a tool that helps the programmer to check manual synchronization for correctness.

References

1. Banerjee, U.: Dependence Analysis. Kluwer Academic Publishers, Norwell (1997)
2. Bogda, J., Hölzle, U.: Removing unnecessary synchronization in java. In: OOPSLA 1999: Proc. ACM SIGPLAN Conf. on Object-Priented Programming, Systems, Languages, and Applications. Denver, Co., ACM (November 1999)
3. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC 2008: Proc. IEEE Intl. Symp. Workload Characterization, Seattle, WA, pp. 35–46 (September 2008)

4. Ceze, L., Montesinos, P., von Praun, C., Torrellas, J.: Colorama: Architectural support for data-centric synchronization. In: HPCA 2007: Proc. IEEE Intl. Symp. High Performance Computer Architecture, Phoenix, AZ, pp. 133–144 (February 2007)
5. Dragojević, A., Guerraoui, R., Kapalka, M.: Stretching transactional memory. In: PLDI 2009: Proc. ACM SIGPLAN Conf. Programming Lang. Design and Impl., Dublin, Ireland, pp. 155–165 (June 2009)
6. Flanagan, C., Freund, S.N.: Fasttrack: efficient and precise dynamic race detection. In: PLDI 2009: Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, Dublin, Ireland, pp. 121–133 (June 2009)
7. Gudka, K., Harris, T., Eisenbach, S.: Lock Inference in the Presence of Large Libraries. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 308–332. Springer, Heidelberg (2012)
8. Halpert, R.L., Pickett Christopher, J.F., Clark, V.: Component-based lock allocation. In: PACT 2007: Proc. 16th Intl. Conf. on Parallel Architecture and Compilation Techniques, Brasov, Romania, pp. 353–364 (September 2007)
9. Heintze, N., Tardieu, O.: Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In: PLDI 2001: Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, Snowbird, UT, pp. 254–263 (June 2001)
10. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. ACM SIGARCH Comput. Archit. News 21(2), 289–300 (1993)
11. Jeremiassen, T., Eggers, S.J.: Static analysis of barrier synchronization in explicitly parallel programs. In: PACT 1994: Proc. IFIP WG 10.3 Working Conf. on Parallel Architectures and Compilation Techniques, pp. 171–180 (August 1994)
12. Johnson, R., Pearson, D., Pingali, K.: The program structure tree: Computing control regions in linear time. In: PLDI 1994: Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, Orlando, FL, pp. 171–185 (June 1994)
13. Lu, S., Park, S., Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R.A., Zhou, Y.: MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In: SOSP 2007: Proc. ACM SIGOPS Symp. Operating Systems Principles, Stevenson, WA, pp. 103–116 (October 2007)
14. Muchnick, S.: Advanced Compiler Design And Implementation. Morgan Kaufmann Publishers Inc., San Francisco (1997)
15. Muzahid, A., Otsuki, N., Torrellas, J.: AtomTracker: A comprehensive approach to atomic region inference and violation detection. In: MICRO 43: Proc. 43rd Annual IEEE/ACM Intl. Symp. Microarchitecture, Atlanta, GA, pp. 287–297 (December 2010)
16. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for java. In: PLDI 2006: Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, Ottawa, ON, Canada, pp. 308–319 (June 2006)
17. Salcianu, A., Rinard, M.: Pointer and escape analysis for multithreaded programs. In: PPoPP 2001: Proc. ACM SIGPLAN Symp. Principles and Practices of Parallel Programming, Snowbird, UT, pp. 12–23 (June 2001)
18. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: POPL 2006: Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages, Charleston, SC, pp. 334–345 (January 2006)
19. Vechev, M., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. In: POPL 2010: Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages, Madrid, Spain, pp. 327–338 (January 2010)

Refactoring MATLAB

Soroush Radpour[1,2], Laurie Hendren[2], and Max Schäfer[3]

[1] Google, Inc.
soroush@google.com
[2] School of Computer Science, McGill University, Montreal, Canada
hendren@cs.mcgill.ca
[3] School of Computer Engineering, Nanyang Technological University, Singapore
schaefer@ntu.edu.sg

Abstract. MATLAB is a very popular dynamic "scripting" language for numerical computations used by scientists, engineers and students world-wide. MATLAB programs are often developed incrementally using a mixture of MATLAB scripts and functions, and frequently build upon existing code which may use outdated features. This results in programs that could benefit from refactoring, especially if the code will be reused and/or distributed. Despite the need for refactoring, there appear to be no MATLAB refactoring tools available. Furthermore, correct refactoring of MATLAB is quite challenging because of its non-standard rules for binding identifiers. Even simple refactorings are non-trivial.

This paper presents the important challenges of refactoring MATLAB along with automated techniques to handle a collection of refactorings for MATLAB functions and scripts including: converting scripts to functions, extracting functions, and converting dynamic function calls to static ones. The refactorings have been implemented using the McLAB compiler framework, and an evaluation is given on a large set of MATLAB benchmarks which demonstrates the effectiveness of our approach.

1 Introduction

Refactoring may be defined as the process of transforming a program in order to improve its internal structure without changing its external behavior. The goal can be to improve readability, maintainability, performance or to reduce the complexity of code. Refactoring has developed for the last 20 years, starting with the seminal theses by Opdyke [1] and Griswold [2], and the well known book by Fowler [3]. Many programmers have come to expect refactoring support, and popular IDEs such as Eclipse, Microsoft's Visual Studio, and Oracle's NetBeans have integrated tool support for automating simple refactorings. However, the benefits of refactoring tools have not yet reached the millions of MATLAB programmers. Currently neither Mathworks' proprietary MATLAB IDE, nor open-source tools provide refactoring support.

MATLAB is a popular dynamic ("scripting") programming language that has been in use since the late 1970s, and a commercial product of MathWorks since 1984, with millions of users in the scientific, engineering and research communities.[1] There are

[1] The most recent data from MathWorks shows one million MATLAB users in 2004, with the number doubling every 1.5 to 2 years; see www.mathworks.com/company/-newsletters/news_notes/clevescorner/jan06.pdf

currently over 1200 books based on MATLAB and its companion software, Simulink (`http://www.mathworks.com/support/books`).

As we have collected and studied a large body of MATLAB programs, we have found that the code could benefit from refactoring for several reasons. First, the MATLAB language has evolved over the years, incrementally introducing many valuable high-level features such as (nested) functions, packages and so on. However, MATLAB programmers often build upon code available online or examples from books, which often do not use the modern high-level features. Thus, although code reuse is an essential part of the MATLAB eco-system, code cruft, obsolete syntax and new language features complicates this reuse. Since MATLAB does not currently have refactoring tools, programmers either do not refactor, or they refactor code by hand, which is time-consuming and error-prone. Secondly, the interactive nature of developing MATLAB programs promotes an incremental style of programming that often results in relatively unstructured and non-modular code. When developing small one-off scripts this may not be important, but when developing a complete application or library, refactoring the code to be better structured and more modular is key for reuse and maintenance.

Refactoring MATLAB presents new research challenges in two areas: (1) ensuring proper handling of MATLAB semantics; and (2) developing new MATLAB-specific refactorings. The semantics of MATLAB is quite different from other languages, thus even standard refactorings must be carefully defined. In particular, to ensure behavior preservation, refactoring tools have to verify that identifiers maintain their correct kind [4] (variable or function), and that their binding is not accidentally changed. MATLAB-specific refactorings include those which help programmers eliminate undesirable MATLAB features. For example, MATLAB scripts are a hybrid of macros and functions, and can lead to unstructured code that is hard to analyze and optimize. Thus, an automatic refactoring which can convert scripts to functions is a useful refactoring transformation which helps improve the structure of the code. Dynamic features like `feval` also complicate programs and are often used inappropriately. Thus, MATLAB-specific refactorings, which convert `feval` to more static constructs are also useful.

In this paper we introduce a family of automated refactorings aimed at restructuring functions and scripts, and calls to functions and scripts. We start with a refactoring for converting scripts into functions, which improves their reusability and modularity. Then we introduce the MATLAB version of the well-known EXTRACT FUNCTION refactoring that can be used to break up large functions into smaller parts. Finally, we briefly survey several other useful refactorings for inlining scripts and functions, and a refactoring to replace spurious uses of the dynamic `feval` feature with direct function calls.

We have implemented our refactoring transformations in our McLAB compiler framework [5], and evaluated the refactorings on a collection of 3023 MATLAB programs. We found that the vast majority of refactoring opportunities could be handled with few spurious warnings.

The main contributions of this paper are:

- Identifying a need for refactoring tools for MATLAB and the key static properties that must be checked for such refactorings.
- Introducing a family of refactorings for MATLAB functions and scripts.
- An implementation of these refactorings in McLAB.

– An evaluation of the implementation on a large set of publicly-available MATLAB programs.

The remainder of this paper is structured as follows. In Section 2 we provide some motivating examples and background about determining the kind of identifiers and the semantics of function lookup. Section 3 describes a refactoring for converting scripts to functions, Section 4 presents EXTRACT FUNCTION, and Section 5 briefly introduces several other refactorings. Section 6 evaluates the refactoring implementations on our benchmark set, Section 7 surveys related work, and Section 8 concludes.

2 Background and Motivating Example

In this section we introduce some key features of MATLAB, and we give a motivating example to demonstrate both a useful MATLAB refactoring and the sorts of MATLAB-specific issues that must be considered.

2.1 MATLAB Scripts and Functions

A MATLAB program consists of a collection of scripts and functions. A script is simply a sequence of MATLAB statements. For example, Figure 1(a) defines a script called `sumcos` which computes the sum of the cosine values of the numbers `i` to `n`. Although using scripts is not a good programming practice, they are very easy for MATLAB programmers to create. Typically, a programmer will experiment with a sequence of statements in the read-eval-print loop of the IDE and then copy and paste them into a file, which becomes the script.

A script is executed in the workspace from which it was called, either the main workspace, or the workspace of the calling function.[2] For example, Figure 1(b) shows function `ex1` calling script `sumcos`. When `sumcos` executes it reads the values of variables `i` and `n` from the workspace of function `ex1`, and writes the value of `s` into that same workspace. Clearly, scripts are highly non-modular, and do not have a well-defined interface. A programmer cannot easily determine the inputs and outputs of a script. Thus, a better programming practice would be to use functions.

Figure 1(d) shows the script `sumcos` refactored into an equivalent function. The body of the function is the same as the script, but now the output parameter `s` and the input parameters `i` and `n` are explicitly declared. As shown in Figure 1(c) and (f), in this case the refactored function produces the same result as the original script.[3]

In general, MATLAB functions may have multiple output and input arguments. However, not all input arguments need to be provided at a call, and not all returned values

[2] Workspaces are MATLAB's version of lexical environments. There is an initial "main" workspace which is acted upon by commands entered into the main read-eval-print loop. There is a also a stack of workspaces corresponding to the function call stack. A call to a function creates and pushes a new workspace, which becomes the current workspace.

[3] These results are snippets taken from an interactive session in the MATLAB read-eval-print loop. The ">>" prompt is followed by the expression to be evaluated. In Figure 1(c) this is a call to function `ex1`. The line after the prompt prints the result of the evaluation.

```
s = 0;
while i <= n
    s = s + cos(i);
    i = i + 1;
end
```

(a) script sumcos.m

```
function ex1( )
  i = 1;
  n = 5;
  sumcos;
  s
end
```

(b) calling sumcos

```
>> ex1
s = -1.2358
```

(c) result of call

```
function s = sumcosFN(i, n)
  s = 0;
  while i <= n
    s = s + cos(i);
    i = i + 1;
  end
end
```

(d) function sumcosFN.m

```
function ex1FN( )
  s = sumcosFN(1, 5)
end
```

(e) calling sumcosFN

```
>> ex1FN
s = -1.2358
```

(f) result of call

Fig. 1. Example script and function

need to be used. Parameters obey call-by-value semantics where semantically a copy of each input and output parameter is made.[4]

2.2 Identifier Kinds

MATLAB does not explicitly declare local variables, nor explicitly declare the types of any variables. Input and output arguments are explicitly declared as variables, whereas other variables are implicitly declared upon their first definition. For example, the assignment to `s` in the first line of Figure 1(d) implicitly also declares `s` to be a variable, and allocates space for that variable in the workspace of function `sumcosFN`.

It is important to note that it is not possible to syntactically distinguish between references to array elements and calls to functions. For example, so far we have assumed that the expression `cos(i)` is a call to function `cos`. However, it could equally well be an array reference referring to the ith element of array `cos`.

To illustrate, consider Figure 2(a), where `cos` is defined to be a five-element vector. The call to `sumcos` in this context actually just sums the elements of the vector, returning `15`. This is because the MATLAB semantics give a *kind* of ID (identifier) to most identifiers in scripts. The rule for looking up identifiers with kind ID at runtime is to first look in the current workspace to see if a variable of that name exists, and if so the identifier denotes that variable. If no such variable exists then the identifier is looked up as a function. Since the script `sumcos` is being executed in the workspace of function `ex2`, and there does exist a variable called `cos` in that workspace, the reference to `cos` refers to that variable, and not the library function for computing the cosine.

The identifier lookup semantics within functions is different. In the case of functions, each identifier is given a static kind at JIT compilation time; for details of this process we

[4] Actual implementations of MATLAB optimize this using either lazy copying using reference counts, or static analyses to insert copies only where necessary [6].

```
function ex2( )                              >> ex2
  cos = [1,2,3,4,5];                         s = 15
  i = 1;
  n = 5;
  sumcos;
  s
end
```

(a) calling script `sumcos` (b) result of call

```
function ex2FN( )                            >> ex2FN
  cos = [1,2,3,4,5];                         s = -1.2358
  s = sumcosFN(1, 5)
end
```

(c) calling function `sumcosFN` (d) result of call

Fig. 2. Calling `sumcos` in a context where `cos` is a variable

refer to the literature [4]. In the case of the refactored function `sumcosFN`, identifiers `i`, `n` and `s` would be determined to have kind VAR (variables), and identifier `cos` would be given the kind FN (function). Thus, the reference to `cos` will always be to the function, and our transformed function `sumcosFN` may have a different meaning than the original script `sumcos`, as demonstrated by the different results in Figure 2(b) and (d).

From this example, it is clear that any MATLAB refactoring of scripts must take care not to change the meaning of identifiers, and in order to do this all of the calling contexts of the script must be taken into consideration.

2.3 MATLAB Programs and Function Lookup

MATLAB programs are defined as directories of files. Each file `f.m` contains either: (a) a script, which is simply a sequence of MATLAB statements; or (b) a sequence of function definitions. If the file `f.m` defines functions, then the first function defined in the file should be called `f` (although even if it is not called `f` it is known by that name in MATLAB). The first function is known as the *primary function*. Subsequent functions are *subfunctions*. The primary and subfunctions within `f.m` are visible to each other, but only the primary function is visible to functions defined in other `.m` files. Functions may be nested, following the usual static scoping semantics of nested functions. That is, given some nested function `f'`, all enclosing functions, and all functions declared in the same nested scope are visible within the body of `f'`.

Figure 3(a) shows an example of a file containing two functions. The primary function is `ex3` and will be visible to all functions and scripts defined in other files. This file also has a secondary function `cos`, which is an implementation of the cosine function using a Taylor's approximation. The important question in this example is which `cos` will be called from the script `sumcos`: the library implementation of `cos` or the Taylor's version of `cos` defined as a subfunction for `ex3`? The answer is that the lookup of a function call from within a script is done with respect to the calling function's environment. In

```
function ex3( )
  i = 1;
  n = 5;
  sumcos;
  s
end

function r = cos(x)
  r = 0;
  xsq = x*x;
  term = 1;
  for i = 1:1:10
    r = r + term;
    term = -term*xsq/((2*i-1)*(2*i));
  end
end
```

(a) `ex3.m` with primary and subfunction

```
>> ex3
s = -1.2359
```

(b) result of call

```
function ex3FN( )
  s = sumcosfn(1,5)
end

function r = cos(x)
  % same as above
  ...
end
```

(c) refactored `ex3.m`

```
>> ex3FN
s = -1.2358
```

(d) result of call

Fig. 3. Calling `sumcos` in a context where `cos` is defined as a subfunction

this case the call to `cos` in script `sumcos` refers to the environment of function `ex3`, which was the last called function. Thus, `cos` binds to the subfunction in `ex3`.

The transformed function `sumcosFN`, however, will not call the Taylor's version of `cos` since subfunctions are not visible to functions defined outside of the file. Thus, the results of running the original script and the transformed function are different. Clearly any MATLAB refactoring must take care that it does not change the binding of functions.

In addition to subfunctions, MATLAB also uses the directory structure to organize functions, and this directory structure also impacts on function binding.

MATLAB directories may contain special private, package and type-specialized directories, which are distinguished by the name of the directory. Private directories must be named `private/`, Package directories start with a '+', for example `+mypkg/`. The primary function in each file `f.m` defined inside a package directory `+p` corresponds to a function named `p.f`. To refer to this function one must use the fully qualified name, or an equivalent import declaration. Package directories may be nested. Type-specialized directories have names of the form `@<typename>`, for example `@int32/`. The primary function in a file `f.m` contained in a directory `@typename/` matches calls to `f(a1,...)`, where the run-time type of the primary argument is `typename`.

Overall, the MATLAB lookup of a script/function is performed relative to: *f*, the current function/script being executed; *sourcefile*, the file in which *f* is defined; *fdir*, the directory containing the last called non-private function (calling scripts or private functions does not change *fdir*); *dir*, the current directory; and *path*, a list of other directories. When looking up function/script names, first *f* is searched for a nested function, then *sourcefile* is searched for a subfunction, then the private directory of *fdir* is searched, then *dir* is searched, followed by the directories on *path*. In the case where there is both a non-specialized and type-specialized function matching a call, the non-specialized version will be selected if it is defined as a nested, subfunction or private function, otherwise the specialized function takes precedence.

In summary, a refactoring needs to ensure that identifier kinds do not change unexpectedly, and that function lookup remains the same.

3 Converting Scripts to Functions

In the previous section we have motivated the need for a refactoring that can convert scripts, which are non-modular, into equivalent functions which will help improve the overall structure of MATLAB programs. We also demonstrated that this refactoring is not as straightforward as one might think due to MATLAB's intricate kind assignment and function lookup rules.

In this section we provide an algorithm to refactor a script into a semantically equivalent function. The programmer provides a complete program, and also identifies the script to be converted to a function. If the refactoring can be done in a semantics-preserving manner, the SCRIPT TO FUNCTION refactoring converts the script to a function and replaces all calls to the script with calls to the new function.

This refactoring requires the use of two additional analyses, *Reaching Definitions* and *Liveness*. These are standard analyses which we have implemented in a way that enables our refactoring.

In our implementation of the reaching definition analysis, every identifier is initialized to be have a special reaching definition of "`undef`". This means that if "`undef`" is not in the reaching definition set for an identifier at some program point p, then this identifier is definitely assigned on all the paths to p. Further, if the reaching definition of an identifier only contains "`undef`", the variable is not assigned to on any paths. Calls to scripts can change reaching definition and liveness results so we look into the called scripts' body during the analyses. Global and persistent variables may be defined by function calls, so our analysis handles these conservatively by associating a special "`global_def`" or "`persistent_def`" with each variable declared as global or persistent. These definitions are never killed.

Our liveness analysis is intra-procedural, but also follows calls to scripts. The liveness analysis safely approximates global variables as always being live, and persistent variables as live at the end of the function they are associated with. Variables that are used in nested functions are also kept alive for simplicity.

Recall that the main difference between a script and a function is that a function has its own workspace and communicates with its caller via input and output arguments, while a script executes directly within the caller's workspace. Thus, to convert a script

Notation	Meaning
DA_s	variables definitely assigned on every path through s
PA_s	variables possibly assigned on some path through s
$L_{<s}$	variables live immediately before s
$L_{>s}$	variables live immediately after s
$RD_s(x)$	reaching definitions for x immediately before s
$K_f(x)$	kind assigned to x inside script or function f; $K_f(x) = \lightning$ for inconsistent kind
$\mathit{Lookup}_f(x)$	look up identifier x in function f (subscript omitted where clear from context)

Fig. 4. Notation for auxiliary analysis results; s may be a sequence of statements, or the body of a function or script

s to function f we need to: (1) determine input and output arguments that will work for all calls to s, and (2) ensure that name binding will stay the same after conversion.

To determine arguments, the basic idea is that a variable needs to be made an input argument if it is live within the script and assigned at every call site; conversely, it needs to be an output argument if it is assigned within the script and live at some call site.

This intuition is made more precise in Algorithm 1, which uses the notations defined in Figure 4. To convert script s into a function, we first compute the set L of identifiers that are used before being defined in s, and that may refer to a variable (as opposed to a function); these are candidates for becoming input arguments.

Now we examine every call c to s. If the call occurs in a script s', we abort the refactoring: the lack of structure in scripts makes it all but impossible to determine appropriate sets of input and output arguments; the user can first convert s' into a function, and then attempt the refactoring again.

If c is in a function, we consider the set $DA_{<c}$ of variables definitely assigned at c. As far as call site c is concerned, the set I_c of input arguments should simply be the intersection of this set with L, the set of live variables at the beginning of s. Similarly, the set O_c of output arguments should contain all variables that are possibly assigned in s and that are live immediately after c. We also need to ensure that every output argument is definitely assigned somewhere in the script; otherwise the refactoring cannot go ahead. Finally, we compute a set lookup_c capturing name binding information for functions at c, whose purpose will be explained below.

Next, we need to check that the set of input arguments I is consistent between call sites: if different call sites provide different input arguments, the refactoring cannot go ahead (line 13). For output arguments, on the other hand, no such precaution is required: if an output argument is unused at a particular call site, it can be ignored by binding it to the dummy "$\sim$" identifier. Thus the set of output parameters O is simply the union of output arguments at every call site.

We are now ready to build the function f using input arguments I, output arguments O, and the body of s.

As a final step, we need to check that name resolution and kind assignments have not changed.

The former is easy to do: we simply compute pairs $\langle n, \mathit{Lookup}(n)\rangle$ determining the binding of every identifier n with kind ID or FN in f, and check that these bindings agree with the bindings lookup_c observed at the call sites.

Algorithm 1. SCRIPT TO FUNCTION

Require: script s
Ensure: s converted to function; all calls to s replaced with function calls

```
1: // preliminary definitions
2: $L \leftarrow \{x \mid x \in L_{<s} \wedge K_s(x) \in \{\textsc{Var}, \textsc{Id}\}\}$
3: $C_s \leftarrow$ calls to $s$
4: // compute input and output arguments
5: for all calls $c \in C_s$ do
6:     if $c$ is in another script $s'$ then
7:         abort refactoring
8:     $I_c \leftarrow DA_{<c} \cap L$ // input arguments
9:     $O_c \leftarrow PA_s \cap L_{>c}$ // output arguments
10:    if $O_c \not\subseteq DA_s$ then
11:        abort refactoring
12:    $lookup_c \leftarrow \{\langle n, Lookup(n)\rangle \mid n$ occurs in $s, K_s(n) \in \{\textsc{Id}, \textsc{Fn}\}\}$ // binding information
13: if $\neg\forall c, c' \in C_s.I_c = I_{c'}$ then
14:     abort refactoring
15: else
16:     $I \leftarrow I_c$ for some call $c \in C_s$
17: $O \leftarrow \bigcup_{c \in C_s} O_c$
18: // construct new function
19: construct new function $f$ with input arguments $I$ and output arguments $O$
20: // check name binding and kinds
21: $lookup_f \leftarrow \{\langle n, Lookup(n)\rangle \mid n$ is identifier in $f$ of kind ID or FN$\}$
22: if $\neg\forall c \in C_s.lookup_c = lookup_f$ then
23:     abort refactoring
24: for all identifiers $x$ in $f$ do
25:     if $K_f(x) = \textsc{Id}$ then
26:         abort refactoring
27:     else if $K_s(x) = \textsc{Id}$ and $K_f(x) = \textsc{Fn}$ then
28:         emit warning
29:     else if $K_s(x) = \textsc{Var}$ and $K_f(x) = \lightning$ then
30:         abort refactoring
31: replace calls to $s$ with calls to $f$
```

To check kind preservation, we compare the kind $K_s(x)$ an identifier x had in s, with its kind $K_f(x)$ in the new function f. In general, identifiers of kind ID can remain so or turn into FN, and identifiers with kind VAR can cause a kind conflict.

If $K_f(x) = \textsc{Id}$, x may originally have been referring to a variable created dynamically in the calling function. Since functions do not share their caller's workspace, this cannot be achieved in a function, and the refactoring has to be aborted.

If x's kind changed from ID to FN, we emit a warning informing the user that the refactoring assumes x refers to a function, which is always the case unless a variable of the same name is created dynamically by `eval` or code loading.

Finally, if x was originally of kind VAR, but provokes a kind conflict in f, we need to abort the refactoring, since it is not clear which uses of the identifier were meant to refer to a function, and which to a variable.

If all checks pass, calls to s can be rewritten to function calls, passing in all input arguments in I and extracting output arguments from the result, discarding any output arguments not needed at a particular call site.

4 Extracting Functions

The EXTRACT FUNCTION refactoring makes it possible to split large functions into smaller ones to improve understandability and reusability. Across all our MATLAB benchmarks, we found that the average number of lines of code per function is 22.7; for comparison, this number is 5.4 for Java and 10.5 for C++ [7], which suggests that MATLAB functions tend to be fairly long and could benefit from extraction.

We first introduce the refactoring on an example before giving a precise specification of the extraction algorithm.

```
 1 function printBest(names,
 2                      grades)
 3  bestGrade=-1; bestIdx=-1;
 4  for i=1:length(grades)
 5   if grades(i) > bestGrade
 6    bestGrade=grades(i);
 7    bestIdx=i;
 8   end
 9  end
10  if bestGrade == -1
11   return
12  end
13  disp(names{bestIdx})
14 end
```

(a) Original Function; extract lines 3–12

```
 1 function printBest(names,
 2                      grades)
 3  RET=false;
 4  bestGrade=-1; bestIdx=-1;
 5  for i=1:length(grades)
 6   if grades(i) > bestGrade
 7    bestGrade=grades(i);
 8    bestIdx=i;
 9   end
10  end
11  if bestGrade == -1
12   RET=true;
13  end
14  if (~RET)
15   disp(names{bestIdx})
16  end
17 end
```

(b) After Return Elimination

Fig. 5. An example for EXTRACT FUNCTION

Figure 5(a) shows an example function that takes an array `names` containing the names of students, and an array `grades` containing their grades. On lines 3–12, it searches through `grades` to find the best grade, storing its index in local variable `bestIdx`. If no best grade was found (because `grades` was empty or contained invalid data), the function returns to its caller; otherwise, the name of the student with the best grade is printed.

Assume that we want to extract the code for finding the best grade (lines 3–12) into a new function `findBest`. Note that the extraction region contains the return statement on line 11; if this statement were extracted into `findBest` unchanged, program semantics would change, since it would now only return from `findBest`, not from `printBest` any more. To avoid this, we first eliminate the return as shown in Figure 5(b) by introducing a flag `RET`. In general, return elimination requires a slightly more elaborate transformation than this, but it is still fairly straightforward and will not be described in detail here; the reader is referred to the first author's thesis for details [8].

Next, we need to determine which input and output arguments the extracted function should have. Reasoning similar to the previous section, we determine that `grades` should become an input argument, since it is live at the beginning of the extracted region and definitely assigned beforehand. Conversely, `bestIdx` should become an output argument, since it is assigned in the extracted region and live afterwards.

```
function [RET, bestIdx] =
         findBest(grades)
 bestGrade=-1; bestIdx=-1;
 for i=1:length(grades)
  if grades(i) > bestGrade
   bestGrade=grades(i);
   bestIdx=i;
  end
 end
 if bestGrade == -1
  RET=true;
 end
end

function printBest(names,
                   grades)
 RET=false;
 [RET, bestIdx] = ...
   findBest(grades);
 if (~RET)
  disp(names{bestIdx})
 end
end
```

(a) After extracting function, `RET` may be undefined after the call

```
function [RET, bestIdx] =
         findBest(grades, RET)
 bestGrade=-1; bestIdx=-1;
 for i=1:length(grades)
  if grades(i) > bestGrade
   bestGrade=grades(i);
   bestIdx=i;
  end
 end
 if bestGrade == -1
  RET=true;
 end
end

function printBest(names,
                   grades)
 RET=false;
 [RET, bestIdx] = ...
   findBest(grades, RET);
 if (~RET)
  disp(names{bestIdx})
 end
end
```

(b) Final version of the extracted function and the call

Fig. 6. Example for EXTRACT FUNCTION, continued

Similarly, `RET` should also become an output argument. Figure 6(a) shows the new function with these arguments. Note, however, that `RET` is not assigned on all code paths, so it may be undefined at the point where the extracted function returns, resulting in a runtime error. To avoid this, we have to also add `RET` to the list of input arguments, ensuring that it always has a value. This finally yields the correct extraction result, shown in Figure 6(b).

Algorithm 2. Extract Function

Require: sequence s of contiguous statements in function f, name n
Ensure: s extracted into new function g with name n

```
1: if $s$ contains top-level break or uses vararg syntax then
2:     abort refactoring
3: if $s$ contains return statement then
4:     eliminate return statements in $f$
5: $I \leftarrow \{x \mid x \in L_{<s} \wedge RD_s(x) \not\subseteq \{\mathtt{undef}, \mathtt{global}\}\}$
6: $O \leftarrow PA_s \cap L_{>s}$
7: for all $x \in O \setminus DA_s$ do
8:     if $\mathtt{undef} \notin RD_s(x)$ then
9:         $I \leftarrow I \cup \{x\}$
10:    else
11:        abort refactoring
12: if function with name $n$ exists in same folder as $f$ then
13:    abort refactoring
14: create new function $g$ with name $n$, input arguments $I$, output arguments $O$
15: declare any globals used in $s$ as globals in $g$
16: for all identifier $x$ in $g$ do
17:    if $K_g(x) \neq K_f(x)$ then
18:        abort refactoring
19:    else if $K_g(x) = \textsc{Fn}$ and $\mathit{Lookup}_f(x) \neq \mathit{Lookup}_g(x)$ then
20:        abort refactoring
21:    else if $K_g(x) = \textsc{Id}$ then
22:        abort refactoring
23: replace $s$ by call to $g$
```

Algorithm 2 shows how to extract a sequence s of contiguous statements in a function f into a new function named n, again using the notations from Figure 4.

We first check whether s contains a `break` statement that refers to a surrounding loop that is not part of the extraction region; if so, the refactoring is aborted. Similarly, if s refers to a variable argument list of f using "`varargin`" or "`varargout`", the refactoring is also aborted. Both of these cases would require quite extensive transformations, which we do not believe to be justified.

After eliminating return statements if necessary, we compute the set I of input arguments, and the set O of output arguments for the new function: every variable that is live immediately before the extraction region s and that has a non-trivial reaching definition becomes an input argument; every variable that is potentially assigned in s and is live afterwards becomes an output argument.

Additionally, any output arguments that are not definitely assigned in s but are definitely assigned before (like `RET` in our example above) also become input arguments. We also check for the corner case of an output argument that is neither definitely assigned in s nor before s, which results in the refactoring being aborted.

Having established the sets of input and output arguments, we can now create the extracted function g, but we need to ensure that no function of this name exists already.

```
1 %%% matrix of Fourier coefficients
2 eps1 = feval ('epsgg',r,na,nb,b1,b2,N1,N2);
3 ...
4 for j=1:length(BZx)
5     [kGx, kGy] = feval('kvect2',BZx(j),BZy(j),b1,b2,N1,N2);
6     [P, beta]=feval('oblic_eigs',omega,kGx,kGy,eps1,N);
7     ...
8 end
```

Listing 1.1. Extracts from a script which uses `feval`

We also need to declare any global variables used in s as global in g, as they would otherwise become local variables of g.

Finally, we need to check that name binding and kind assignments work out. First, we check that the kind of all identifiers in g is the same as before the extraction. Additionally, if there is any function reference that refers to a different declaration in the new function g than it did before, we need to abort the refactoring. Lastly, we ensure that no identifier has kind ID, as this may again lead to different name lookup results.

If all these checks pass, we can replace s by a call to the extracted function.

5 Other Refactorings

In addition to the SCRIPT TO FUNCTION and EXTRACT FUNCTION refactorings described in the previous sections, we have implemented several other refactorings that we briefly outline in this section.

Corresponding to EXTRACT FUNCTION, there are two inlining refactorings for inlining scripts and functions. While this does not usually improve code quality, inlining refactorings can play an important role as intermediate steps in larger refactorings.

When inlining a call to script or function g in function f, return statements in g first have to be eliminated in the same way as for EXTRACT FUNCTION. If g is a function, its local variables have to be renamed to avoid name clashes with like-named variables in f. After copying the body of g into f, we then have to verify that name bindings stay the same, and kind assignments either stay the same or at least do not affect name lookup. For details we refer to the first author's thesis [8].

Finally, we briefly discuss a very simple but surprisingly useful MATLAB-specific refactoring, ELIMINATE FEVAL. The MATLAB builtin function `feval` takes a reference to a function (a function handle or a string with the name of the function) as an argument and calls the function. Replacing `feval` by direct function calls where possible leads to cleaner and more efficient code.

Somewhat to our surprise, we found numerous cases where programmers used a constant function name in `feval`. For example, the code in Listing 1.1, which is extracted from one of our benchmarks, uses `feval` for all invocations of user defined functions (lines 2, 5 and 6), even though there is no apparent reason for doing so; all uses of `feval` can be replaced by direct function calls.

Our refactoring tool looks for those calls to `feval` which have a string constant as the first argument, and then uses the results from kind analysis to determine if an identifier with kind VAR with the same name exists. If there is no such identifier in the function, the call to `feval` is replaced with a direct call to the function named inside the string literal. Of course, with more complex string and call graph analyses one could support even more such refactorings. However, it is interesting that such a simple refactoring is useful.

6 Evaluation

We now evaluate our implementation on a large set of MATLAB programs. While it would be desirable to evaluate correctness (i.e., behavior preservation) of our implementation, this is infeasible to do by hand due to the large number of subject programs. Automated testing of refactoring implementations is itself still a topic of research [9] and relies on automated test generation, which is not yet available for MATLAB. Instead, we aim to assess the usefulness of our refactorings and their implementation.

6.1 Evaluation Criteria

We evaluate every refactoring according to the following criteria:

EC1 How many refactoring opportunities are there?
EC2 Among all opportunities, how often can McLAB perform the refactoring without warnings or errors? How often is the user warned of possible behavior changes? How often is McLAB unable to complete the refactoring?
EC3 How invasive are the code changes?

6.2 Experimental Setup and Benchmarks

In order to experiment with our analyses we gathered a large number of MATLAB projects.[5] The benchmarks come from a wide variety of application areas including Computational Physics, Statistics, Computational Biology, Geometry, Linear Algebra, Signal Processing and Image Processing. We analyzed 3023 projects composed of 11698 function files, some with multiple functions, and 2380 scripts. The projects vary in size between 283 files in one project, and a single file in other cases. A summary of the size distribution of the benchmarks is given in Table 1 which shows that the benchmarks tend to be small to medium in size. However, we have also found 9 large and 2 very large benchmarks. The benchmarks presented here are the most downloaded projects among the mentioned categories, which may mean that the average code quality is higher than for less popular projects.

[5] Benchmarks were obtained from individual contributors plus projects from `http://www.mathworks.com/matlabcentral/fileexchange`, `http://people.sc.fsu.edu/~jburkardt/m_src/m_src.html`, `http://www.csse.uwa.edu.au/~pk/Research/MatlabFns/` and `http://www.mathtools.net/MATLAB/`. This is the same set of projects that are used in [4].

Table 1. Distribution of size of the benchmarks

Benchmark Category	Number of Benchmarks
Single (1 file)	2051
Small (2–9 files)	848
Medium (10–49 files)	113
Large (50–99 files)	9
Very Large ($\geq$ 100 files)	2
Total	3023

6.3 Converting Scripts to Functions

We start by evaluating the SCRIPT TO FUNCTION refactoring presented in Section 3.

We consider every script a candidate for the refactoring (**EC1**), thus there are 2380 refactoring opportunities overall (note that some benchmarks only define functions). For criterion **EC2**, Table 2 summarizes the result of using MCLAB to convert all these scripts to functions: in 204 cases, the refactoring completed without warnings or errors; in 1312 cases, the refactoring succeeds with a warning about an identifier changing from kind ID to the more specialized kind FN. This is only a problem if the program defines a variable reflectively through eval or code loading, and the identifier in question is also a function on the path. This is unlikely and should be easy for the programmer to check. Finally, for 864 scripts the refactoring aborted because behavior preservation could not be guaranteed.

Further breaking down the causes of rejection, we see that in most cases the problem is an identifier of kind ID that cannot statically be resolved to a variable or a function. In all these cases, the script is the only script in a single file project; thus it arguably is not a very good target for conversion anyway. In some cases, the script was itself called from a script, which also leads to rejection (as mentioned in Section 3, this could be resolved by first converting the calling script to a function). Finally, in one case different invocations of the script lead to different input argument assignments.

To assess the invasiveness of the code changes (**EC3**), we measured the number of input and output arguments of the newly created functions. A large number of input and output parameters can clutter up the code, so it is important that the refactoring creates no spurious parameters. For those scripts that were called at least once, the number of inputs range between 0 and 5 with an average of 1, and the number of outputs range

Table 2. Results from converting scripts to functions

Refactoring Outcome	Number of Scripts
Success	204
Success with Warning about ID changed to FNs	1312
Unresolved IDs	712
Call from script	151
Input arguments mismatch	1

between 0 and 12 with the average of 1.1. This shows that the algorithm is fairly efficient in choosing a minimal set of parameters.

6.4 Extract Function

For function extraction, the number of refactoring opportunities is hard to measure, since it is not clear how to identify blocks of code for which function extraction makes sense.

In order to nevertheless be able to automatically evaluate a large number of function extraction refactorings, we employ a heuristic for identifying regions that are more or less independent in terms of control and data flow from the rest of a function.

We concentrate on regions starting at the beginning of a function, and comprising a sequence of top-level statements. We only consider functions with at least seven top-level statements; smaller functions are unlikely to benefit from extraction. Since we want the region to contain some reasonable amount of code, we include at least as many statements as it takes for the region to contain 30 AST nodes. We don't want to move all the body of the original function to the new function either, so we never extract the last 30 AST nodes in the function either. In between, we find the choice that will need the minimum number of input and output arguments, but only if that minimum number is less than 15. Figure 7 shows these constraints.

Out of 13438 functions overall, 6469 contain at least seven top-level statements and thus are interesting for extraction (**EC1**). Among these, we can successfully break 6214 functions (i.e., 96%) into smaller ones (**EC2**). The average number of arguments to the newly created functions was 2.8 (**EC3**), with most extracted functions having between one and three arguments. This means that the selection algorithm was effective in selecting regions with minimal inter-dependency. Figure 8 shows the distribution of the number of arguments among these 6214 functions.

In 48 cases the refactoring was rejected because a possibly undefined input argument was detected, and in 21 cases a possibly undefined output argument prevented the refactoring from going ahead.

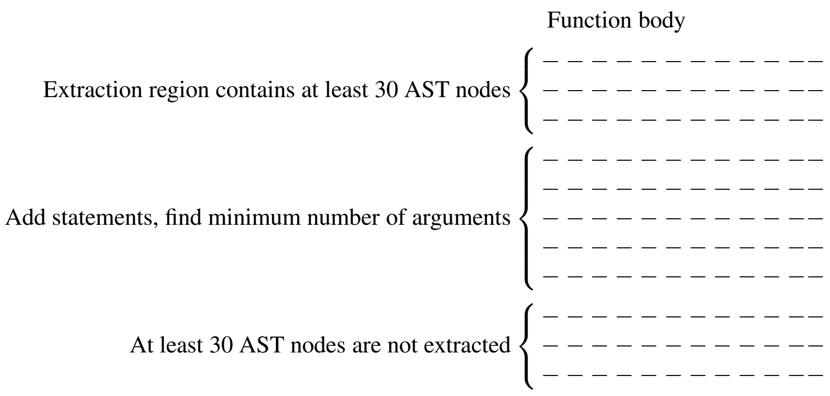

Function body
Extraction region contains at least 30 AST nodes
Add statements, find minimum number of arguments
At least 30 AST nodes are not extracted

Fig. 7. An example showing constraints used to select refactoring region

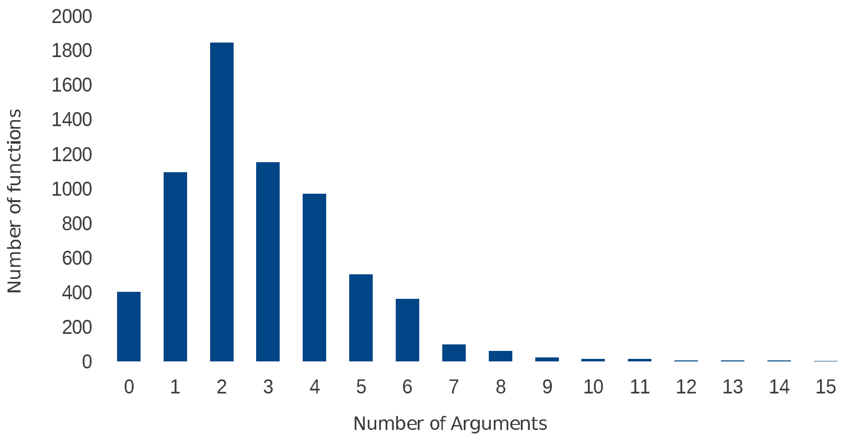


Fig. 8. Distribution of number of arguments for the new functions

6.5 Replacing `feval`

Finally, we evaluated the ELIMINATE FEVAL refactoring for converting trivial uses of `feval` into direct function calls. Of the 200 calls to `feval`, there were 23 uses of it with a string literal argument (**EC1**), and all of them could be eliminated successfully (**EC2**). The transformation performed by this refactoring is very local, and in fact makes the code simpler (**EC3**).

6.6 Threats to Validity

There are several threats to validity for our evaluation.

First, our collection of benchmarks is extensive, but it may not be representative of other real-world MATLAB code. In particular, the percentage of rejected refactorings may be higher on code that makes heavy use of language features that are hard to analyze.

Second, our selection of refactoring opportunities is based on heuristics and may not be representative of actual refactorings that programmers may attempt. This is a general problem with automatically evaluating refactoring implementations. Still, the low number of rejections gives some confidence that the implementation should be able to handle real-world use cases. A more realistic user study will have to wait until our implementation has been integrated with an IDE.

Finally, we have not checked whether the refactorings performed by our implementation are actually behavior preserving in every case; the large number of successful refactorings makes this impossible. We know of two edge cases where behavior may not be preserved: the kind and name analyses do not handle dynamic calls to `cd`, and `eval` is not handled by the liveness or reaching definition analysis. This is similar to how refactorings for Java do not handle reflection. One possibility would be for the refactoring engine to emit a warning to the user if a use of one of these features is detected, but we have not implemented this yet.

7 Related Work

There is a wide variety of work on refactoring covering a large number of programming languages. In particular, there is a considerable body of work on automated refactoring for statically typed languages such as Java with quite well developed and rigorous approaches for specifying correct refactorings[10,11,12]. However, these approaches intrinsically rely on the availability of rich compile-time information in the form of static types and a static name binding structure; thus they are not easily applicable to MATLAB, which provides neither.

Refactoring for dynamically typed languages has, in fact, a long history: the first ever refactoring tool, the Refactoring Browser [13], targetted the dynamically typed object-oriented language Smalltalk. However, the Refactoring Browser mostly concentrated on automating program transformation and performed relatively few static checks to ensure behavior preservation.

More recently, Feldthaus et al. [14] have presented a refactoring tool for JavaScript. They employ a pointer analyis to infer static information about the refactored program, thus making up for the lack of static types and declarations. Most of their refactorings have the goal of improving encapsulation and modularity, thus they are similar in scope to our proposed refactorings for MATLAB.

Even more closely related is recent work on refactoring support for Erlang. Like MATLAB, Erlang has evolved over time, adding new constructs for more modular and concise programming, and refactorings have been proposed that can help with upgrading existing code to make use of these new features. For instance, the Wrangler refactoring tool provides assistance for data and process refactoring [15], clone detection and elimination [16] and modularity maintenance [17].

Most recently, Wrangler has been extended with a scripting language that makes it easy to implement domain specific refactorings [18]. Such scriptable refactorings could be interesting for MATLAB as well, either to implement one-off refactorings to be used for one particular code base, or to provide a refactoring tool with specific information about a program that enables otherwise unsafe transformations.

While Wrangler is an interactive tool, the tidier tool [19] performs fully automatic cleanup operations on Erlang code. The standards for behavior preservation are obviously much higher for a fully automated tool than for an interactive one, so tidier only performs small-scale refactorings, but a similar tool could certainly also be useful for MATLAB.

Refactoring legacy Fortran code has also been the subject of some research. Overbey et. al. [20,21] point out the benefits of refactoring for languages that have evolved over time. Although the specific refactorings are quite different, the motivation and the applicability of our approaches is very similar. Like MATLAB, Fortran is often used for computationally expensive tasks, hence there has been some interest in refactorings for improving program performance [22,23].

In a similar vein, Menon and Pingali have investigated source-level transformations for improving MATLAB performance [24]. The transformations they propose go beyond the typical loop transformations performed by compilers, and capture MATLAB-specific optimizations such as converting entire loops to library calls, and restructuring loops to avoid incremental array growth. Automating these transformations would be an interesting next step, and our foundational analyses and refactorings should aid in that process.

8 Conclusion

In this paper we have identified an important domain for refactoring, MATLAB programs. Millions of scientists, engineers and researchers use MATLAB to develop their applications, but no tools are available to support refactoring their programs. This means that it is difficult for the programmers to improve upon old code which use out-of-date language constructs or to restructure their initial prototype code to a state in which it can be distributed.

To address this new refactoring domain we have developed a set of refactoring transformations for functions and scripts, including function and script inlining, converting scripts to functions, and eliminating simple cases of `feval`. For each refactoring we established a procedure which defined both the transformation and the conditions which must be verified to ensure that the refactoring is semantics-preserving. In particular, we emphasized that both the kinds of identifiers and the function lookup semantics must be considered when deciding if a refactoring can be safely applied or not.

We have implemented all of the refactorings presented in the paper using our McLAB compiler toolkit, and we applied the refactorings to a large number of MATLAB applications. Our results show that, on this benchmark set, the refactorings can be effectively applied. We plan to continue our work, adding more refactorings, including performance enhancing refactorings and refactorings to enable a more effective translation of MATLAB to Fortran.

Acknowledgments. This work has been supported by NSERC (Canada) and the Leverhulme Trust (UK). We would also like to give special acknowledgment to Frank Tip for helping to define the direction of this work.

References

1. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign (1992)
2. Griswold, W.G.: Program Restructuring as an Aid to Software Maintenance. Ph.D. thesis, University of Washington (1991)
3. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
4. Doherty, J., Hendren, L., Radpour, S.: Kind analysis for MATLAB. In: Proceedings of OOPSLA 2011 (2011)
5. McLab, `http://www.sable.mcgill.ca/mclab/`
6. Lameed, N., Hendren, L.: Staged Static Techniques to Efficiently Implement Array Copy Semantics in a MATLAB JIT Compiler. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 22–41. Springer, Heidelberg (2011)
7. English, M., McCreanor, P.: Exploring the Differing Usages of Programming Language Features in Systems Developed in C++ and Java. In: PPIG (2009)
8. Radpour, S.: Understanding and Refactoring the Matlab Language. M.Sc. thesis, McGill University (2012)
9. Soares, G., Gheyi, R., Serey, D., Massoni, T.: Making Program Refactoring Safer. IEEE Software 27(4), 52–57 (2010)
10. Schäfer, M., de Moor, O.: Specifying and Implementing Refactorings. In: OOPSLA (2010)

11. Tip, F., Fuhrer, R.M., Kieżun, A., Ernst, M.D., Balaban, I., Sutter, B.D.: Refactoring Using Type Constraints. TOPLAS 33, 9:1–9:47 (2011)
12. Schäfer, M., Thies, A., Steimann, F., Tip, F.: A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs. TSE (2012) (to appear)
13. Roberts, D., Brant, J., Johnson, R.E.: A Refactoring Tool for Smalltalk. TAPOS 3(4), 253–263 (1997)
14. Feldthaus, A., Millstein, T., Møller, A., Schäfer, M., Tip, F.: Tool-supported Refactoring for JavaScript. In: OOPSLA (2011)
15. Li, H., Thompson, S.J., Orösz, G., Tóth, M.: Refactoring with Wrangler, updated: Data and process refactorings, and integration with Eclipse. In: Erlang Workshop, pp. 61–72 (2008)
16. Li, H., Thompson, S.J.: Clone Detection and Removal for Erlang/OTP within a Refactoring Environment. In: PEPM, pp. 169–178 (2009)
17. Li, H., Thompson, S.J.: Refactoring Support for Modularity Maintenance in Erlang. In: SCAM, pp. 157–166 (2010)
18. Li, H., Thompson, S.: A Domain-Specific Language for Scripting Refactorings in Erlang. In: de Lara, J., Zisman, A. (eds.) Fundamental Approaches to Software Engineering. LNCS, vol. 7212, pp. 501–515. Springer, Heidelberg (2012)
19. Sagonas, K., Avgerinos, T.: Automatic Refactoring of Erlang Programs. In: PPDP, pp. 13–24 (2009)
20. Overbey, J.L., Negara, S., Johnson, R.E.: Refactoring and the Evolution of Fortran. In: SECSE, pp. 28–34 (2009)
21. Overbey, J.L., Johnson, R.E.: Regrowing a Language: Refactoring Tools Allow Programming Languages to Evolve. In: OOPSLA (2009)
22. Overbey, J., Xanthos, S., Johnson, R., Foote, B.: Refactorings for Fortran and High-performance Computing. In: SE-HPCS, pp. 37–39 (2005)
23. Boniati, B.B., Charão, A.S., Stein, B.D.O., Rissetti, G., Piveta, E.K.: Automated Refactorings for High Performance Fortran Programmes. IJHPSA 3(2/3), 98–109 (2011)
24. Menon, V., Pingali, K.: A Case for Source-level Transformations in MATLAB. In: DSL, pp. 53–65 (1999)

On LR Parsing with Selective Delays

Eberhard Bertsch[1], Mark-Jan Nederhof[2,⋆], and Sylvain Schmitz[3]

[1] Ruhr University, Faculty of Mathematics, Bochum, Germany
[2] School of Computer Science, University of St. Andrews, UK
[3] LSV, ENS Cachan & CNRS, Cachan, France

Abstract. The paper investigates an extension of LR parsing that allows the delay of parsing decisions until a sufficient amount of context has been processed. We provide two characterizations for the resulting class of grammars, one based on grammar transformations, the other on the direct construction of a parser. We also report on experiments with a grammar collection.

1 Introduction

From a grammar engineer's standpoint, LR parsing techniques, like the LALR(1) parsers generated by yacc or GNU/bison, suffer from the troublesome existence of *conflicts*, which appear sooner or later in any grammar development. Tracing the source of such conflicts and refactoring the grammar to solve them is a difficult task, for which we refer the reader to the accounts of Malloy et al. [15] on the development of a C# grammar, and of Gosling et al. [10] on that of the official Java grammar.

In the literature, different ways have been considered to solve conflicts automatically while maintaining a *deterministic* parsing algorithm—which, besides efficiency considerations, also has the considerable virtue of ruling out ambiguities—, such as unbounded regular lookaheads [6], noncanonical parsers [25], and delays before reductions [14]. Bertsch and Nederhof [4] have made a rather counter-intuitive observation on the latter technique: increasing delays uniformly throughout the grammar can in some cases introduce new conflicts.

In this paper we propose a parsing technique that *selects* how long a reduction must be delayed depending on the context. More interestingly, and unlike many techniques that extend LR parsing, we provide a *characterization*, using grammar transformations, of the class of grammars that can be parsed in a LR fashion with selective delays. More precisely,

– we motivate in Section 2 the interest of ML(k, m) parsing on an exerpt of the C++ grammar, before stating the first main contribution of the paper: we reformulate the technique of Bertsch and Nederhof [4] as a grammar transformation, and show how selective delays can capture non-ML(k, m) grammars,

[⋆] Part of this research was conducted while the second author was an invited professor visiting ENS Cachan.

K. De Bosschere and R. Jhala (Eds.): CC 2013, LNCS 7791, pp. 244–263, 2013.

- we define the class selML(k, m) accordingly through a nondeterministic grammar transformation, which allows us to investigate its properties (Section 3),
- in Section 4 we propose an algorithm to generate parsers with selective delays, and prove that it defines the same class of grammars.
- We implemented a Java proof of concept for this algorithm (see `http://www.cs.st-andrews.ac.uk/~mjn/code/mlparsing/`), and report in Section 5 on the empirical value of selective delays, by applying the parser on a test suite of small unambiguous grammars [2, 22].
- We conclude with a discussion of related work, in Section 6.

Technical details can be found in the full version of this paper at `http://hal.archives-ouvertes.fr/hal-00769668`.

Preliminaries. We assume the reader to be familiar with LR parsing, but nonetheless recall some definitions and standard notation.

A *context-free grammar* (CFG) is a tuple $\mathcal{G} = \langle N, \Sigma, P, S\rangle$ where N is a finite set of *nonterminal* symbols, Σ a finite set of *terminal* symbols with $N \cap \Sigma = \emptyset$—together they define the *vocabulary* $V = N \uplus \Sigma$—, $P \subseteq N \times V^*$ is a finite set of *productions* written as rewrite rules "$A \to \alpha$", and $S \in N$ the *start* symbol. The associated *derivation* relation $\Rightarrow$ over V^* is defined as $\Rightarrow = \{(\delta A\gamma, \delta\alpha\gamma) \mid A \to \alpha \in P\}$; a derivation is *rightmost*, denoted $\Rightarrow_{\text{rm}}$, if γ is restricted to be in Σ^* in the above definition. The *language* of a CFG is $L(\mathcal{G}) = \{w \in \Sigma^* \mid S \Rightarrow^* w\} = \{w \in \Sigma^* \mid S \Rightarrow^*_{\text{rm}} w\}$.

We employ the usual conventions for symbols: nonterminals in N are denoted by the first few upper-case Latin letters A, B, ..., terminals in Σ by the first few lower-case Latin letters a, b, ..., symbols in V by the last few upper-case Latin letters X, Y, Z, sequences of terminals in Σ^* by the last few lower-case Latin letters u, v, w, ..., and mixed sequences in V^* by Greek letters α, β, etc. The empty string is denoted by ε.

Given $\mathcal{G} = \langle N, \Sigma, P, S\rangle$, its *k-extension* is the grammar $\langle N \uplus \{S^\dagger\}, \Sigma \uplus \{\#\}, P \cup \{S^\dagger \to S\#^k\}, S^\dagger\rangle$ where # is a fresh symbol. A grammar is *LR(m)* [13, 23] if it is reduced—i.e. every nonterminal is both accessible and productive—and the following *conflict* situation does not arise in its m-extension:

$$\begin{array}{ll} S^\dagger \Rightarrow^*_{\text{rm}} \delta A u \Rightarrow_{\text{rm}} \delta\alpha u = \gamma u & \quad \delta \neq \delta' \text{ or } A \neq B \text{ or } \alpha \neq \beta \\ S^\dagger \Rightarrow^*_{\text{rm}} \delta' B v \Rightarrow_{\text{rm}} \delta'\beta v = \gamma w v & \quad m : u = m : wv \end{array}$$

where "$m : u$" denotes the prefix of length m of u, or the whole of u if $|u| \leq m$.

2 Marcus-Leermakers Parsing

The starting point of this paper is the formalization proposed by Leermakers [14] of a parsing technique due to Marcus [16], which tries to imitate the way humans parse natural language sentences. Bertsch and Nederhof [4] have given another, equivalent, formulation, and dubbed it "ML" for Marcus-Leermakers.

The idea of uniform ML parsing is that all the reductions are delayed to take place after the recognition of a fixed number k of *right context* symbols, which can contain nonterminal symbols. Bertsch and Nederhof [4] expanded this class by considering m further symbols of terminal lookahead, thereby defining ML(k, m) grammars. In Section 2.2, we provide yet another view on uniform ML(k, m) grammars, before motivating the use of selective delays in Section 2.3. Let us start with a concrete example taken from the C++ grammar from the 1998 standard [11].

2.1 C++ Qualified Identifiers

First designed as a preprocessor for C, the C++ language has evolved into a complex standard. Its rather high level of syntactic ambiguity calls for nondeterministic parsing methods, and therefore the published grammar makes no attempt to fit in the LALR(1) class.

We are interested in one particular issue with the syntax of *identifier expressions*, which describe a full name specifier and identifier, possibly instantiating template variables; for instance, "`A::B<C::D>::E`" denotes an identifier "`E`" with name specifier "`A::B<C::D>`", where the template argument of "`B`" is "`D`" with specifier "`C`".

The syntax of identifier expressions is given in the official C++ grammar by the following (simplified) grammar rules:

$$I \to U \mid Q, \quad U \to i \mid T, \quad Q \to N\,U, \quad N \to U\ \texttt{::}\ N \mid U\ \texttt{::}, \quad T \to i\,\texttt{<}I\texttt{>}.$$

An identifier expression I can derive either an *unqualified identifier* through nonterminal U, or a *qualified identifier* through Q, which is qualified through a *nested name specifier* derived from nonterminal N, i.e. through a sequence of unqualified identifiers separated by double colons "`::`", before the identifier i itself. Moreover, each unqualified identifier can be a *template identifier* T, where the *template argument*, between angle brackets "`<`" and "`>`", can again be any identifier expression.

Example 1. A shift/reduce conflict appears with this set of rules. A parser fed with "`A::`", and seeing an identifier "`B`" in its lookahead window, has a nondeterministic choice between

– *reducing* "`A::`" to a single N, in the hope that "`B`" will be the identifier qualified by "`A::`", as in "`A::B<C::D>`", and
– *shifting* the identifier, in the hope that "`B`" will be a specifier of the identifier actually qualified, for instance "`E`" in "`A::B<C::D>::E`".

An informed decision requires an exploration of the specifier starting with "`B`" in search of a double colon symbol. The need for unbounded lookahead occurs if "`B`" is the start of an arbitrarily long template identifier: this grammar is not LR(k) for any finite k.

Note that the double colon token might also appear inside a template argument. Considering that the conflict could also arise there, as after reading

"`A<B::`" in "`A<B::C<D::E>::F>::G`", we see that it can be arduous to know whether a "`::`" symbol is significant for the resolution of the conflict or not. In fact, this is an example of a conflict that *cannot* be solved by using regular lookahead as proposed in [5, 3, 8], because keeping track of the nesting level of well-balanced brackets is beyond the power of regular languages.[1]

2.2 Uniform ML

Observe that, in our extract of the C++ grammar, if we were to *postpone* the choice between the two possible actions and attempt to parse an N in full, then the issue would disappear. The mechanism Leermakers [14] employs for delaying parsing decisions is to extend a nonterminal with additional terminal and nonterminal symbols from its right context, thus delaying reduction to that nonterminal until the moment when these additional symbols have been parsed in full. This also involves introducing a new end-of-file terminal "#".

We refer the reader to [14, 4] and the full version of this paper for the details of the ML(k, m) parser construction. The automaton obtained by applying this construction on the C++ grammar is too large to be rendered on a single page. In what follows we present an alternative characterization of ML parsing on the basis of a grammar transformation.

Uniform ML as a Transformation. Although Leermakers does not present his technique in these terms, the intuition of extending nonterminals with right context can be realized by a grammar transformation that introduces nonterminals of the form $[A\delta]$ in $N' = N \cdot V^{\leq k}$, which combine a nonterminal A with its immediate right context δ.

This results for $k = 1$ and our C++ example into an LALR(1) grammar with rules:

$$\begin{gathered}
[I\#] \to [U\#] \mid [Q\#], \qquad [I\texttt{>}] \to [U\texttt{>}] \mid [Q\texttt{>}],\\
[U\#] \to i\,\# \mid [T\#], \quad [U\texttt{>}] \to i\,\texttt{>} \mid [T\texttt{>}], \quad [U\,\texttt{::}] \to i\;\texttt{::} \mid [T\,\texttt{::}], \quad [U] \to i \mid [T],\\
[Q\#] \to [NU]\,\#, \qquad [Q\texttt{>}] \to [NU]\,\texttt{>},\\
[NU] \to [U\,\texttt{::}]\,[NU] \mid [U\,\texttt{::}]\,[U],\\
[T\#] \to i\,\texttt{<}\,[I\texttt{>}]\,\#, \quad [T\texttt{>}] \to i\,\texttt{<}\,[I\texttt{>}]\,\texttt{>}, \quad [T\,\texttt{::}] \to i\,\texttt{<}\,[I\texttt{>}]\;\texttt{::}, \quad [T] \to i\,\texttt{<}\,[I\texttt{>}].
\end{gathered}$$

The new grammar demonstrates that our initial grammar for C++ identifier expressions is ML(1, 1): it requires contexts of length $k = 1$, and lookahead of length $m = 1$.

[1] We can amend the rules of N to use left-recursion and solve the conflict: $N \to N\,U\;\texttt{::} \mid U\;\texttt{::}$. This correction was made by the Standards Committee in 2003 (see `http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#125`). The correction was not motivated by this conflict but by an ambiguity issue, and the fact that the change eliminated the conflict seems to have been a fortunate coincidence. The C++ grammar of the Elsa parser [17] employs right recursion.

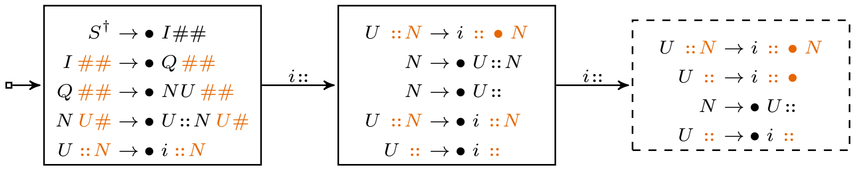

$S^\dagger \to \bullet\, I\#\#$
$I\,\#\# \to \bullet\, Q\,\#\#$
$Q\,\#\# \to \bullet\, NU\,\#\#$
$N\,U\# \to \bullet\, U{::}N\,U\#$
$U\,{::}N \to \bullet\, i\,{::}N$
$i{::}$
$U\,{::}N \to i\,{::}\,\bullet\, N$
$N \to \bullet\, U{::}N$
$N \to \bullet\, U{::}$
$U\,{::}N \to \bullet\, i\,{::}N$
$U\,{::} \to \bullet\, i\,{::}$
$i{::}$
$U\,{::}N \to i\,{::}\,\bullet\, N$
$U\,{::} \to i\,{::}\,\bullet$
$N \to \bullet\, U{::}$
$U\,{::} \to \bullet\, i\,{::}$

Fig. 1. Parts of the uniform ML(2, 0) parser for C++ identifier expressions

Combing Function. Formally, the nonterminals in N' are used in the course of the application of the *uniform k-combing function* comb_k from V^* to $(N' \uplus \Sigma)^*$, defined recursively as:

$$\mathsf{comb}_k(\alpha) = \begin{cases} [A\delta] \cdot \mathsf{comb}_k(\alpha') & \text{if } \alpha = A\delta\alpha', A \in N, \text{ and either } |\delta| = k, \\ & \quad \text{or } |\delta| \le k \text{ and } \alpha' = \varepsilon \\ a \cdot \mathsf{comb}_k(\alpha') & \text{if } \alpha = a\alpha' \text{ and } a \in \Sigma \\ \varepsilon & \text{otherwise, which is if } \alpha = \varepsilon\ . \end{cases}$$

For instance, $\mathsf{comb}_1(ABcDeF) = [AB]c[De][F]$.

The right parts of the rules of $[A\delta]$ are then of the form $\mathsf{comb}_k(\alpha\delta)$ if $A \to \alpha$ was a rule of the original grammar, effectively delaying the reduction of α to A until after δ has been parsed.

Definition 1 (Uniform combing). *Let $\mathcal{G} = \langle N, \Sigma, P, S\rangle$ be a CFG. Its* uniform k-combing *is the CFG $\langle N \cdot V^{\le k}, \Sigma, \{[A\delta] \to \mathsf{comb}_k(\alpha\delta) \mid \delta \in V^{\le k}$ and $A \to \alpha \in P\}, [S]\rangle$.*

Equivalence of the Two Views. Of course we should prove that the two views on ML parsing are equivalent:

Theorem 1. *A grammar is ML(k, m) if and only if the uniform k-combing of its k-extension is LR(m).*

Proof Idea. One can verify that the LR(m) construction on the k-combing of the k-extension of $\mathcal{G}$ and the ML(k, m) construction of Bertsch and Nederhof [4] for the same $\mathcal{G}$ are identical. $\square$

2.3 Selective ML

An issue spotted by Bertsch and Nederhof [4] is that the classes of ML(k, m) grammars and ML(k+1, m) grammars are not comparable: adding further delays can introduce new LR(m) conflicts in the ML($k+1$, m)-transformed grammar.

For instance, the uniform 2-combing of our grammar for C++ identifier expressions is not LR(m) for any m: Fig. 1 shows the path to a conflict similar to that of the original grammar, which is therefore not uniform ML(2, m). Selective ML aims to find the appropriate delay, i.e. the appropriate amount of right context, for each item in the parser, in order to avoid such situations.

Oscillating Behaviour. Bertsch and Nederhof also show that an oscillating behaviour can occur, for instance with the grammar

$$S \to SdA \mid c,\ A \to a \mid ab \tag{$\mathcal{G}_{\mathrm{odd}}$}$$

being ML(k, 0) only for odd values of k, and the grammar

$$S \to SAd \mid c,\ A \to a \mid ab \tag{$\mathcal{G}_{\mathrm{even}}$}$$

being ML(k, 0) only for even values of $k > 0$, from which we can build a union grammar

$$S \to SdA \mid SAd \mid c,\ A \to a \mid ab \tag{$\mathcal{G}_2$}$$

which is not ML(k, 0) for any k.

Observe however that, if we use different context lengths for the different rules of S in $\mathcal{G}_2$, i.e. if we *select* the different delays, we can still obtain an LR(0) grammar $\mathcal{G}_2'$ with rules

$$\begin{aligned}
[S^\dagger] &\to [S\#]\#,\\
[S\#] &\to [Sd][A\#] \mid [SAd]\# \mid c\#,\\
[Sd] &\to [Sd][Ad] \mid [SAd]d \mid cd,\\
[SAd] &\to [Sd][AAd] \mid [SAd][Ad] \mid c[Ad],\\
[A\#] &\to a\# \mid ab\#,\\
[Ad] &\to ad \mid abd,\\
[AAd] &\to a[Ad] \mid ab[Ad]
\end{aligned} \tag{$\mathcal{G}_2'$}$$

As we will see, this means that $\mathcal{G}_2$ is selective ML with a delay of at most 2, denoted selML(2, 0). This example shows that selective ML(k, m) is not just about finding a minimal global $k' \leq k$ such that the grammar is uniform ML(k', m). Because the amount of delay is optimized depending on the context, selective ML captures a larger class of grammars.

3 Selective Delays through Grammar Transformation

We define selML(k, m) through a grammar transformation akin to that of Definition 1, but which employs a *combing relation* instead of the uniform k-combing function. We first introduce these relations (Section 3.1) before defining the selML(k, m) grammar class and establishing its relationships with various classes of grammars in Section 3.2 (more comparisons with related work can be found in Section 6).

3.1 Combing Relations

In the following definitions, we let $\mathcal{G} = \langle N, \Sigma, P, S\rangle$ be a context-free grammar. Combing relations are defined through the application of a particular inverse homomorphism throughout the rules of the grammar.

Definition 2 (Selective Combing). *Grammar $\mathcal{G}' = \langle N', \Sigma, P', S' \rangle$ is a* selective combing *of $\mathcal{G}$, denoted $\mathcal{G}$ comb $\mathcal{G}'$, if there exists a homomorphism μ from V'^* to V^* such that*

1. $\mu(S') = S$,
2. $\forall a \in \Sigma, \mu(a) = a$,
3. $\mu(N') \subseteq N \cdot V^*$, *and*
4. $\{A' \to \mu(\alpha') \mid A' \to \alpha' \in P'\} = \{A' \to \alpha\delta \mid A' \in N', \mu(A') = A\delta,$ *and* $A \to \alpha \in P\}$.

It is a selective k-combing *if furthermore $\mu(N') \subseteq N \cdot V^{\leq k}$.*

We denote the elements of N' by $[A\delta]_i$, such that $\mu([A\delta]_i) = A\delta$, with an i subscript in $\mathbb{N}$ to differentiate nonterminals that share the same image by μ.

Note that, if $\mathcal{G}$ comb $\mathcal{G}'$, then there exists some k such that $\mathcal{G}'$ is a selective k-combing of $\mathcal{G}$, because $\mu(N')$ is a finite subset of $N \cdot V^*$. Another observation is that comb is transitive, and thus we can bypass any intermediate transformation by using the composition of the μ's. In fact, comb is also reflexive (using the identity on N for μ), and is thus a quasi order.

Grammar Cover. It is easy to see that a grammar and all its μ-combings are language equivalent. In fact, we can be more specific, and show that any μ-combing $\mathcal{G}' = \langle N', \Sigma, P', [S]_0 \rangle$ of $\mathcal{G} = \langle N, \Sigma, P, S \rangle$ defines a *right-to-x cover* of $\mathcal{G}$ (see Nijholt [19]), i.e. there exists a homomorphism h from P'^* to P^* such that

1. for all w in $L(\mathcal{G}')$ and right parses π' of w in $\mathcal{G}'$, $h(\pi')$ is a parse of w in $\mathcal{G}$, and
2. for all w in $L(\mathcal{G})$ there is a parse π of w in $\mathcal{G}$, such that there exists a right parse π' of w in $\mathcal{G}'$ with $h(\pi') = \pi$.

Indeed, defining h by

$$h([A\delta]_i \to \alpha) = A \to \mu(\alpha) \cdot \delta^{-1} \tag{1}$$

fits the requirements of a right-to-x cover.

Tree Mapping. Nevertheless, the right-to-x cover characterization is still somewhat unsatisfying, precisely because the exact derivation order x remains unknown. We solve this issue by providing a tree transformation that maps any derivation tree of $\mathcal{G}'$ to a derivation tree of $\mathcal{G}$. Besides allowing us to prove the language equivalence of $\mathcal{G}$ and $\mathcal{G}'$ (see Corollary 1), this transformation also allows us to map any parse tree of $\mathcal{G}'$—the grammar we use for parsing—to its corresponding parse tree of $\mathcal{G}$—the grammar we were interested in in the first place.

We express this transformation as a rewrite system over the set of *unranked forests* $\mathcal{F}(N \cup N' \cup \Sigma)$ over the set of symbols $N \cup N' \cup \Sigma$, defined by the abstract syntax

$$t ::= X(f) \qquad \text{(trees)}$$
$$f ::= \varepsilon \mid f \cdot t \qquad \text{(forests)}$$

where "X" ranges over $N \cup N' \cup \Sigma$ and "$\cdot$" denotes concatenation. Using unranked forests, our tree transformation has a very simple definition, using a rewrite system $\rightarrow_R$ with one rule per nonterminal $[AX_1 \cdots X_r]_i$ in N':

$$[AX_1 \cdots X_r]_i(x_0 \cdot X_1(x_1) \cdots X_r(x_r)) \rightarrow_R A(x_0) \cdot X_1(x_1) \cdots X_r(x_r) \qquad (2)$$

with variables $x_0, x_1, \ldots, x_r$ ranging over $\mathcal{F}(N \cup N' \cup \Sigma)$. Clearly, $\rightarrow_R$ is noetherian and confluent, and we can consider the mapping that associates to a derivation tree t in $\mathcal{G}'$ its normal form $t{\downarrow_R}$ (see full paper for details):

Proposition 1. *Let $\mathcal{G}$ be a CFG and $\mathcal{G}'$ a combing of $\mathcal{G}$.*

1. *If t' is a derivation tree of $\mathcal{G}'$, then $t'{\downarrow_R}$ is a derivation tree of $\mathcal{G}$.*
2. *If t is a derivation tree of $\mathcal{G}$, then there exists a derivation tree t' of $\mathcal{G}'$ such that $t = t'{\downarrow_R}$.*

Since $\rightarrow_R$ preserves tree yields, we obtain the language equivalence of $\mathcal{G}$ and $\mathcal{G}'$ as a direct corollary of Proposition 1:

Corollary 1 (Combings Preserve Languages). *Let $\mathcal{G}$ be a k-extended CFG and $\mathcal{G}'$ a combing of $\mathcal{G}$. Then $L(\mathcal{G}) = L(\mathcal{G}')$.*

3.2 Selective ML Grammars

We define selML(k, m) grammars by analogy with the characterization proved in Thm. 1:

Definition 3 (Selective ML). *A grammar is* selML(k, m) *if there exists a selective k-combing of its k-extension that is LR(m).*

Basic Properties. We now investigate the class of selML(k, m) grammars. As a first comparison, we observe that the uniform k-combing of a grammar is by definition a selective k-combing (by setting μ as the identity on $N \cdot V^{\leq k}$), hence the following lemma:

Lemma 1. *If a grammar is ML(k, m) for some k and m, then it is selML(k, m).*

As shown by $\mathcal{G}_2$, this grammar class inclusion is strict.

A second, more interesting comparison holds between selML(0, m) and LR(m). That a LR(m) grammar is selML(0, m) is immediate since `comb` is reflexive; the converse is not obvious at all, because a 0-combing can involve "duplicated" nonterminals, but holds nevertheless (see full paper for details).

Lemma 2. *A reduced grammar is selML(0, m) if and only if it is LR(m).*

Recall that a context-free language can be generated by some LR(1) grammar if and only if it is deterministic [13], thus selML languages also characterize deterministic languages:

Corollary 2 (Selective ML Languages). *A context-free language has a selML grammar if and only if it is deterministic.*

Proof. Given a selML(k, m) grammar $\mathcal{G}$, we obtain an LR(m) grammar $\mathcal{G}'$ with a deterministic language, and equivalent to $\mathcal{G}$ by Corollary 1. Conversely, given a deterministic language, there exists an LR(1) grammar for it, which is also selML(0,1) by Lemma 2. $\square$

Monotonicity. We should also mention that, unlike uniform ML, increasing k allows strictly more grammars to be captured by selML(k, m). Indeed, if a grammar is a selective k-combing of some grammar $\mathcal{G}$, then it is also a $k+1$-combing using the same μ (with an extra # endmarker), and remains LR(m).

Proposition 2. *If a grammar is selML(k, m) for some k and m, then it is selML(k', m') for all $k' \geq k$ and $m' \geq m$.*

Strictness can be witnessed thanks to the grammar family $(\mathcal{G}_3^k)_{k\geq 0}$ defined by

$$S \to Ac^kA' \mid Bc^kB',\ A \to cA \mid d,\ B \to cB \mid d,\ A' \to cA' \mid a,\ B' \to cB' \mid b \quad (\mathcal{G}_3^k)$$

where each $\mathcal{G}_3^k$ is selML($k+1$, 0), but not selML(k, m) for any m.

Ambiguity. As a further consequence of Proposition 1, we see that no ambiguous grammar can be selML(k, m) for any k and m.

Proposition 3. *If a grammar is selML(k, m) for some k and m, then it is unambiguous.*

Proof. Assume the opposite: an ambiguous grammar $\mathcal{G}$ has a selective k-combing $\mathcal{G}'$ that is LR(m). Being ambiguous, $\mathcal{G}$ has two different derivation trees t_1 and t_2 with the same yield w. As t_1 and t_2 are in normal form for $\to_R$, the sets of derivation trees of $\mathcal{G}'$ that rewrite into t_1 and t_2 are disjoint, and using Proposition 1 we can pick two different derivation trees t'_1 and t'_2 with $t_1 = t'_1{\downarrow_R}$ and $t_2 = t'_2{\downarrow_R}$. As $\to_R$ preserves tree yields, both t'_1 and t'_2 share the same yield w, which shows that $\mathcal{G}'$ is also ambiguous, in contradiction with $\mathcal{G}'$ being LR(m) and thus unambiguous. $\square$

Again, this grammar class inclusion is strict, because the following unambiguous grammar for even palindromes is not selML(k, m) for any k or m, since its language is not deterministic:

$$S \to aSa \mid bSb \mid \varepsilon \quad (\mathcal{G}_4)$$

Undecidability. Let us first refine the connection between selML and LR in the case of linear grammars: recall that a CFG is *linear* if the right-hand side of each one of its productions contains at most one nonterminal symbol. A consequence is that right contexts in linear CFGs are exclusively composed of terminal symbols. In such a case, the selML(k, m) and LR($k+m$) conditions coincide (see full paper for details):

Lemma 3. *Let $\mathcal{G}$ be a reduced linear grammar, and k and m two natural integers. Then $\mathcal{G}$ is selML(k, m) if and only if it is LR($k + m$).*

Note that in the non-linear case, the classes of selML(k, m) and LR(k+m) grammars are incomparable. Nevertheless, we obtain as a consequence of Lemma 3:

Theorem 2. *It is undecidable whether an arbitrary (linear) context-free grammar is selML(k, m) for some k and m, even if we fix either k or m.*

Proof. Knuth [13] has proven that it is undecidable whether an arbitrary linear context-free grammar is LR(n) for some n. □

4 Parser Construction

This section discusses how to directly construct an LR-type parser for a given grammar and fixed k and m values. The algorithm is *incremental*, in that it attempts to use as little right context as possible: this is interesting for efficiency reasons (much as incremental lookaheads in [1, 20]), and actually needed since more context does not necessarily lead to determinism (recall Section 2.3). The class of grammars for which the algorithm terminates successfully (i.e. results in a deterministic parser, without ever reaching a failure state) coincides with the class of selML(k, m) grammars (see Propositions 4 and 5). An extended example of the construction will be given in Section 4.2.

4.1 Algorithm

Algorithm 1 presents the construction of an automaton from the k-extension of a grammar. We will call this the selML(k, m) automaton. In the final stages of the construction, the automaton will resemble an LR(m) automaton for a selective k-combing. Before that, states are initially constructed without right context. Right contexts are extended only where required to solve conflicts.

Items and States. The items manipulated by the algorithm are of form $([A\delta] \to \alpha \bullet \alpha', L)$, where $L \subseteq \Sigma^{\leq m}$ is a set of terminal lookahead strings, and where α and α' might contain nonterminals of the form $[B\beta]$, where $B \in N$ and $\beta \in V^{\leq k}$. Such nonterminals may later become nonterminals in the selective k-combing of the input grammar. To avoid notational clutter, we assume in what follows that B and $[B]$ are represented in the same way, or equivalently, that an occurrence of B in a right-hand side is implicitly converted to $[B]$ wherever necessary.

States are represented as sets of items. Each such set q is associated with three more sets of items. The first is its closure $\mathsf{close}(q)$. The second is $\mathsf{conflict}(q)$, which is the set of closure items that lead to a shift/reduce or reduce/reduce conflict with another item, either immediately in q or in a state reachable from q by a sequence of transitions. A conflict item signals that the closure step that predicted the corresponding rule, in the form of a non-kernel item, must be reapplied, but now from a nonterminal $[B\beta]$ with longer right context β. Lastly, the set $\mathsf{deprecate}(q)$ contains items that are to be ignored for the purpose of computing the Goto function.

$$\frac{([A\delta] \to \alpha \bullet [B\beta_1]\beta_2, L) \in \mathsf{close}(q)}{([B\beta_1] \to \bullet\, \gamma\beta_1, L') \in \mathsf{close}(q)} \begin{cases} B \to \gamma \in P, \\ L' = \mathsf{First}_m(\beta_2 L) \end{cases} \qquad \text{(closure)}$$

$$\frac{\begin{array}{c}([A_1\delta_1] \to \alpha_1 \bullet \beta_1, L_1) \in \mathsf{close}(q) \\ ([A_2\delta_2] \to \alpha_2 \bullet, L_2) \in \mathsf{close}(q)\end{array}}{([A_2\delta_2] \to \alpha_2 \bullet, L_2) \in \mathsf{conflict}(q)} \begin{cases} (A_1\delta_1, \alpha_1, \beta_1) \neq (A_2\delta_2, \alpha_2, \varepsilon), \\ \mathsf{First}_m(\mu(\beta_1)L_1) \cap L_2 \neq \emptyset \end{cases} \qquad \text{(conflict detection)}$$

$$\frac{\begin{array}{c}([A\delta] \to \alpha \bullet [B\beta], L) \in \mathsf{close}(q) \\ ([B\beta] \to \bullet\, \gamma, L) \in \mathsf{conflict}(q)\end{array}}{([A\delta] \to \alpha \bullet [B\beta], L) \in \mathsf{conflict}(q)} \qquad \text{(conflict propagation)}$$

$$\frac{\begin{array}{c}([A\delta] \to \alpha \bullet [B\beta_1]X\beta_2, L) \in \mathsf{close}(q) \\ ([B\beta_1] \to \bullet\, \gamma, L') \in \mathsf{conflict}(q)\end{array}}{([A\delta] \to \alpha \bullet [B\beta_1 X]\beta_2, L) \in \mathsf{close}(q)} \begin{cases} |\beta_1| < k, \\ L' = \mathsf{First}_m(X\beta_2 L), \end{cases} \qquad \text{(extension)}$$

$$\frac{([B\beta] \to \bullet\, \gamma, L) \in \mathsf{conflict}(q)}{\bot} \left\{ |\beta| = k \right. \qquad \text{(failure)}$$

$$\frac{([A\delta] \to \alpha \bullet [B\beta_1 X]\beta_2, L) \in \mathsf{close}(q)}{([A\delta] \to \alpha \bullet [B\beta_1]X\beta_2, L) \in \mathsf{deprecate}(q)} \qquad \text{(deprecation)}$$

$$\frac{([A\delta] \to \alpha \bullet [B\beta_1 X]\beta_2, L) \in \mathsf{close}(q)}{([B\beta_1] \to \bullet\, \gamma', L') \in \mathsf{deprecate}(q)} \begin{cases} B \to \gamma \in P, \mu(\gamma') = \gamma\beta_1, \\ L' = \mathsf{First}_m(X\beta_2 L), \end{cases} \qquad \text{(deprecate closure)}$$

Fig. 2. Closure of set q with local resolution of conflicts

Item Closure. The sets $\mathsf{close}(q)$, $\mathsf{conflict}(q)$ and $\mathsf{deprecate}(q)$ are initially computed from the kernel q alone. However, subsequent visits to states reachable from q may lead to new items being added to $\mathsf{conflict}(q)$ and then to $\mathsf{close}(q)$ and $\mathsf{deprecate}(q)$. How items in these three sets are derived from one another for given q is presented as the deduction system in Figure 2.

The *closure* step is performed as in conventional LR parsing, except that right context is copied to the right-hand side of a predicted rule. The *conflict detection* step introduces a conflict item, after a shift/reduce or reduce/reduce conflict appears among the derived items in the closure. Conflict items solicit additional right context, which

– may be available locally in the current state, as in step *extension*, where nonterminal $[B\beta_1]$ is extended to incorporate the following symbol X—we assume μ here is a generic "uncombing" homomorphism, turning a single nonterminal $[B\beta_1]$ into a string $B\beta_1 \in N \cdot V^{\leq k}$—, or
– if no more right context is available at the closure item from which a conflict item was derived, then the closure item itself becomes a conflict item, by step *conflict propagation*—propagation of conflicts across states is realized by Algorithm 1 and will be discussed further below—, or
– if there is ever a need for right context exceeding length k, then the grammar cannot be selML(k, m) and the algorithm terminates reporting failure by step *failure.*

Algorithm 1. Construction of the selML(k, m) automaton for the k-extension of $\mathcal{G} = \langle N, \Sigma, P, S\rangle$, followed by construction of a selective k-combing

```
 1: $\mathsf{States} \leftarrow \emptyset$
 2: $\mathsf{Transitions} \leftarrow \emptyset$
 3: $\mathsf{Agenda} \leftarrow \emptyset$
 4: $q_{\text{init}} = \{(S^\dagger \to \bullet\ S\#^k, \{\varepsilon\})\}$
 5: $\textsc{NewState}(q_{\text{init}})$
 6: while $\mathsf{Agenda} \neq \emptyset$ do
 7:     $q \leftarrow \mathsf{pop}(\mathsf{Agenda})$
 8:     remove $(q, X, q')$ from $\mathsf{Transitions}$ for any $X$ and $q'$
 9:     apply Figure 2 to add new elements to the three sets associated with $q$
10:     for all $([A\delta] \to \alpha X \bullet \beta, L) \in \mathsf{conflict}(q)$ do
11:         for all $q'$ such that $(q', X, q) \in \mathsf{Transitions}$ do
12:             $\textsc{AddConflict}(([A\delta] \to \alpha \bullet X\mu(\beta), L), q')$
13:         end for
14:     end for
15:     if there are no $([A\delta] \to \alpha X \bullet \beta, L) \in \mathsf{conflict}(q)$ then
16:         $qmax \leftarrow \mathsf{close}(q) \setminus \mathsf{deprecate}(q)$
17:         for all $X$ such that there is $([A\delta] \to \alpha \bullet X\beta, L) \in qmax$ do
18:             $q' \leftarrow \mathsf{Goto}(qmax, X)$
19:             if $q' \notin \mathsf{States}$ then
20:                 $\textsc{NewState}(q')$
21:             else
22:                 for all $([A'\delta'] \to \alpha' X \bullet \beta', L) \in \mathsf{conflict}(q')$ do
23:                     $\textsc{AddConflict}(([A'\delta'] \to \alpha' \bullet X\mu(\beta'), L), q)$
24:                 end for
25:             end if
26:             $\mathsf{Transitions} \leftarrow \mathsf{Transitions} \cup \{(q, X, q')\}$
27:         end for
28:     end if
29: end while
30: construct a selective $k$-combing as explained in the running text
31:
32: function $\textsc{NewState}(q)$
33:     $\mathsf{close}(q) \leftarrow q$
34:     $\mathsf{conflict}(q) \leftarrow \emptyset$
35:     $\mathsf{deprecate}(q) \leftarrow \emptyset$
36:     $\mathsf{States} \leftarrow \mathsf{States} \cup \{q\}$
37:     $\mathsf{Agenda} \leftarrow \mathsf{Agenda} \cup \{q\}$
38: end function
39:
40: function $\textsc{AddConflict}(([A\delta] \to \alpha \bullet X\beta, L), q)$
41:     if $([A\delta] \to \alpha \bullet X\beta, L) \notin \mathsf{conflict}(q)$ then
42:         $\mathsf{conflict}(q) \leftarrow \mathsf{conflict}(q) \cup \{([A\delta] \to \alpha \bullet X\beta, L)\}$
43:         $\mathsf{Agenda} \leftarrow \mathsf{Agenda} \cup \{q\}$
44:     end if
45: end function
```

Step *deprecation* expresses that an item with shorter right context is to be ignored for the purpose of computing the Goto function. The Goto function will be discussed further below. Similarly, step *deprecate closure* expresses that all items predicted from the item with shorter right context are to be ignored.

Main Algorithm. Initially, the agenda contains only the initial state, which is added in line 5. Line 7 of the algorithm removes an arbitrary element from the agenda and assigns it to variable q. At that point, either $\mathsf{close}(q) = q$ and $\mathsf{conflict}(q) = \mathsf{deprecate}(q) = \emptyset$ if q was not considered by line 7 before, or elements may have been added to $\mathsf{conflict}(q)$ since the last such consideration, which also requires updating of $\mathsf{close}(q)$ and $\mathsf{deprecate}(q)$, by Figure 2. By a change of the latter two sets, also the outgoing transitions may change. To keep the presentation simple, we assume that all outgoing transitions are first removed (on line 8) and then recomputed. From line 10, conflicting items are propagated to states immediately preceding the current state, by one transition. Such a preceding state is then put on the agenda so that it will be revisited later.

Outgoing transitions are (re-)computed from line 15 onward. This is only done if no conflicting items had to be propagated to preceding states. Such conflict items would imply that q itself will not be reachable from the initial state in the final automaton, and in that case there would be no benefit in constructing outgoing transitions from q.

For the purpose of applying the Goto function, we are only interested in the closure items that have maximal right context, as all items with shorter context were found to lead to conflicts. This is the reason why we take the set difference $qmax = \mathsf{close}(q) \setminus \mathsf{deprecate}(q)$. The Goto function is defined much as usual:

$$\mathsf{Goto}(qmax, X) = \{([A\delta] \rightarrow \alpha X \bullet \beta, L) \mid ([A\delta] \rightarrow \alpha \bullet X\beta, L) \in qmax\} \ . \qquad (3)$$

The loop from line 22 is very similar to that from line 10. In both cases, conflicting items are propagated from a state q_2 to a state q_1 along a transition (q_1, X, q_2). The difference lies in whether q_1 or q_2 is the currently popped element q in line 7. The propagation must be allowed to happen in both ways, as it cannot be guaranteed that no new transitions are found leading to states at which conflicts have previously been processed.

Combing Construction. After the agenda in Algorithm 1 becomes empty, only those states reachable from the initial state q_{init} via transitions in $\mathsf{Transitions}$ are relevant, and the remaining ones can be removed from States. From the reachable states, we can then construct a selective k-combing, with start symbol $S^\dagger$, as follows.

For each $q_n \in \mathsf{States}$ and $([A\delta] \rightarrow X_1 \cdots X_n \bullet, L) \in \mathsf{close}(q_n) \setminus \mathsf{deprecate}(q_n)$, some $n \geq 0$, find each choice of:

- $q_0, \ldots, q_{n-1}$,
- $\beta_0, \ldots, \beta_n$, with $\beta_n = \varepsilon$,

such that for $0 \leq j < n$,

- $(q_j, X_{j+1}, q_{j+1}) \in \mathsf{Transitions}$,
- $([A\delta] \rightarrow X_1 \cdots X_j \bullet X_{j+1}\beta_{j+1}, L) \in \mathsf{close}(q_j) \setminus \mathsf{deprecate}(q_j)$, and
- $\beta_j = \mu(X_{j+1})\beta_{j+1}$.

It can be easily seen that β_0 must be of the form $\alpha\delta$, for some rule $A \rightarrow \alpha$. For each choice of the above, now create a rule $Y_0 \rightarrow Y_1 \cdots Y_n$, where Y_0 stands for the triple $(q_0, A\delta, L)$, and for $1 \leq j \leq n$:

- if X_j is a terminal then $Y_j = X_j$, and
- if X_j is of the form $[B_j\gamma_j]$ then Y_j stands for the triple $(q_{j-1}, B_j\gamma_j, L_j)$, where $L_j = \mathsf{First}_m(\beta_j L)$.

We assume here that $\mu(Y_0) = A\delta$ and $\mu(Y_j) = B_j\gamma_j$ for $1 \leq j \leq n$.

4.2 Example

Example 2. Let us apply Algorithm 1 to the construction of a selML(2, 0) parser for $\mathcal{G}_{\text{odd}}$. The initial state is $q_{\text{init}} = \{S^\dagger \rightarrow \bullet S\#\#\}$ (there is no lookahead set since we set $m = 0$) and produces through the rules of Fig. 2

$$\mathsf{close}(q_{\text{init}}) = \{S^\dagger \rightarrow \bullet S\#\#, S \rightarrow \bullet SdA, S \rightarrow \bullet c\} \ . \tag{4}$$

Fast-forwarding a little, the construction eventually reaches state $q_{Sd} = \{S \rightarrow Sd \bullet A\}$ with

$$\mathsf{close}(q_{Sd}) = \{S \rightarrow Sd \bullet A, A \rightarrow \bullet a, A \rightarrow \bullet ab\} \ , \tag{5}$$

which in turn reaches state $q_{Sda} = \{A \rightarrow a\bullet, A \rightarrow a \bullet b\}$ with

$$\mathsf{close}(q_{Sda}) = q_{Sda} \ , \tag{6}$$

$$\mathsf{conflict}(q_{Sda}) = \{A \rightarrow a\bullet\} \ . \tag{7}$$

As this item is marked as a conflict item, line 10 of Algorithm 1 sets

$$\mathsf{conflict}(q_{Sd}) = \{A \rightarrow \bullet a\} \ , \tag{8}$$

and puts q_{Sd} back in the agenda. Then, the *conflict propagation* rule is fired to set

$$\mathsf{conflict}(q_{Sd}) = \{A \rightarrow \bullet a, S \rightarrow Sd \bullet A\} \ , \tag{9}$$

and by successive backward propagation steps we get

$$\mathsf{conflict}(q_{\text{init}}) = \{S \rightarrow \bullet SdA\} \ . \tag{10}$$

The *extension* rule then yields

$$\mathsf{close}(q_{\mathrm{init}}) = \{S^\dagger \to \bullet S\#\#, S \to \bullet SdA, S \to \bullet c, S^\dagger \to \bullet[S\#]\#, S \to \bullet[Sd]A\}\ , \tag{11}$$

which is closed to obtain

$$\begin{aligned}\mathsf{close}(q_{\mathrm{init}}) = \{&S^\dagger \to \bullet S\#\#, S \to \bullet SdA, S \to \bullet c, S^\dagger \to \bullet[S\#]\#, S \to \bullet[Sd]A,\\ &[S\#] \to \bullet SdA\#, [S\#] \to \bullet c\#, [Sd] \to \bullet SdAd, [Sd] \to \bullet cd\}\ ,\end{aligned} \tag{12}$$

and we can apply again the *extension* rule with the conflicting item $S \to \bullet SdA$:

$$\begin{aligned}\mathsf{close}(q_{\mathrm{init}}) = \{&S^\dagger \to \bullet S\#\#, S \to \bullet SdA, S \to \bullet c, S^\dagger \to \bullet[S\#]\#, S \to \bullet[Sd]A,\\ &[S\#] \to \bullet SdA\#, [S\#] \to \bullet c\#, [Sd] \to \bullet SdAd, [Sd] \to \bullet cd,\\ &[S\#] \to \bullet[Sd]A\#, [Sd] \to \bullet[Sd]Ad\}\ .\end{aligned} \tag{13}$$

The *deprecate* and *deprecate closure* rules then yield

$$\begin{aligned}\mathsf{deprecate}(q) = \{&S^\dagger \to \bullet S\#\#, S \to \bullet SdA, S \to \bullet c, [S\#] \to \bullet SdA\#,\\ &[Sd] \to \bullet SdAd, \ldots\}\ .\end{aligned} \tag{14}$$

We leave the following steps to the reader; the resulting parser is displayed in Fig. 3 (showing only items in $\mathsf{close}(q) \setminus \mathsf{deprecate}(q)$ in states).

4.3 Correctness

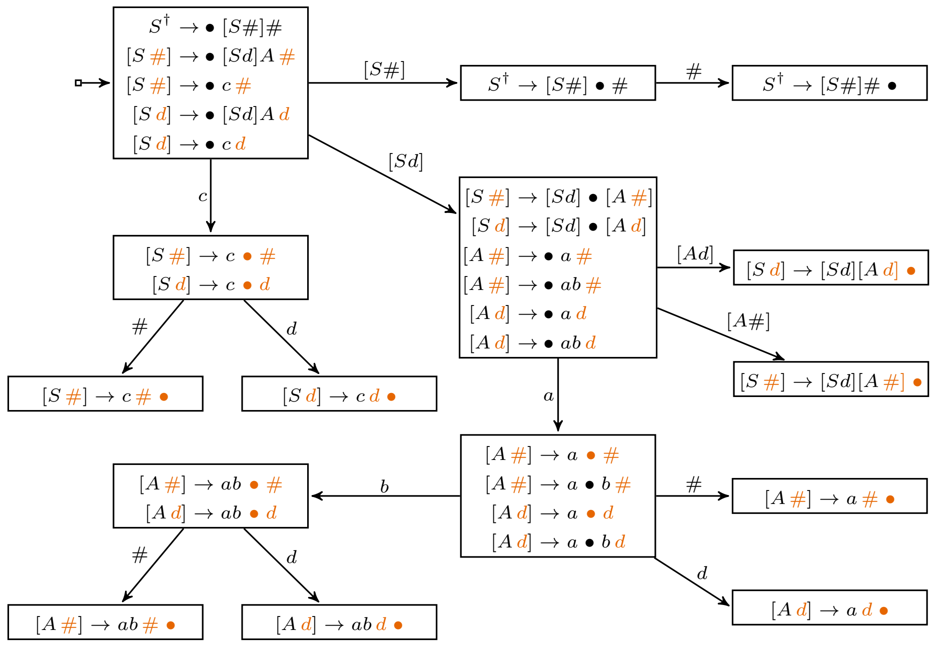
First observe that Algorithm 1 always terminates: the number of possible sets q, along with the growing sets $\mathsf{close}(q)$, $\mathsf{conflict}(q)$ and $\mathsf{deprecate}(q)$, is bounded.

Termination by the *failure* step of Fig. 2 occurs only when we know that the resulting parser cannot be deterministic; conversely, successful termination means that a deterministic parser has been constructed. One could easily modify the construction to keep running in case of failure and output a nondeterministic parser instead, for instance to use a generalized LR parsing algorithm on the obtained parser.

The correctness of the construction follows from Propositions 4 and 5 (see full paper for details).

Proposition 4. *If Algorithm 1 terminates successfully, then the constructed grammar is a selective k-combing. Furthermore, this combing is LR(m).*

Proof Idea. The structure of the selML(k, m) automaton and the item sets ensure that the constructed grammar satisfies all the requirements of a selective k-combing. Had this been non-LR(m), then there would have been further steps or failure. □



$S^\dagger \to \bullet\ [S\#]\#$
$[S\ \#] \to \bullet\ [Sd]A\ \#$
$[S\ \#] \to \bullet\ c\ \#$
$[S\ d] \to \bullet\ [Sd]A\ d$
$[S\ d] \to \bullet\ c\ d$
$[S\#]$
$S^\dagger \to [S\#] \bullet \#$
$\#$
$S^\dagger \to [S\#]\# \bullet$
$[Sd]$
c
$[S\ \#] \to [Sd] \bullet [A\ \#]$
$[S\ d] \to [Sd] \bullet [A\ d]$
$[A\ \#] \to \bullet\ a\ \#$
$[A\ \#] \to \bullet\ ab\ \#$
$[A\ d] \to \bullet\ a\ d$
$[A\ d] \to \bullet\ ab\ d$
$[Ad]$
$[S\ d] \to [Sd][A\ d] \bullet$
$[A\#]$
$[S\ \#] \to [Sd][A\ \#] \bullet$
$[S\ \#] \to c \bullet \#$
$[S\ d] \to c \bullet d$
$\#$
d
$[S\ \#] \to c\ \# \bullet$
$[S\ d] \to c\ d \bullet$
a
$[A\ \#] \to a \bullet \#$
$[A\ \#] \to a \bullet b\ \#$
$[A\ d] \to a \bullet d$
$[A\ d] \to a \bullet b\ d$
b
$[A\ \#] \to ab \bullet \#$
$[A\ d] \to ab \bullet d$
$\#$
$[A\ \#] \to a\ \# \bullet$
d
$[A\ d] \to a\ d \bullet$
$\#$
d
$[A\ \#] \to ab\ \# \bullet$
$[A\ d] \to ab\ d \bullet$

Fig. 3. The selML(2, 0) parser for $\mathcal{G}_{\text{odd}}$

Proposition 5. *If the grammar is selML(k, m), then the algorithm terminates successfully.*

Proof Idea. The selML(k, m) automaton under construction reflects minimum right context for nonterminal occurrences in any selective k-combing with the LR(m) property. Furthermore, failure would imply that right context of length exceeding k is needed. □

As a consequence, we can refine the statement of Theorem 2 with

Corollary 3. *It is decidable whether an arbitrary context-free grammar is selML(k, m), for given k and m.*

5 Experimental Results

We have implemented a proof of concept of Algorithm 1, which can be downloaded from `http://www.cs.st-andrews.ac.uk/~mjn/code/mlparsing/`. Its purpose is not to build actual parsers for programming languages, but merely to check the feasibility of the approach and compare selML with uniform ML and more classical parsers.

Table 2. Results on example grammars

	$\lvert N\rvert$	$\lvert P\rvert$	LR classes: #states	ML classes: #states
Example 3	2	6	non-LR(m)	ML(3,1): 357, selML(3,1): 41, ML(2,2): 351, selML(2,2): 77
Example 4	5	8	LR(2): 16	(sel)ML(1,0): 17
Example 5	3	5	LALR(1): 11	ML(1,0): 17, selML(1,0): 15

Grammar Collection. We investigated a set of *small* grammars that exhibit well-identified syntactic difficulties, to see whether they are treated correctly by a given parsing technique, or lie beyond its grasp. This set of grammars was compiled by Basten [2] and extended in [22], containing mostly grammars for programming languages from the parsing literature and the comp.compilers archive, but also a few RNA grammars used by the bioinformatics community [21].

Conflicts. As expected, we identified a few grammars that were not LALR(1) but were selML(k, m) for small values of k and m. Results are summarized in Table 2.

Example 3 (Tiger). One such example is an excerpt from the Tiger syntax found at `http://compilers.iecc.com/comparch/article/98-05-030`. The grammar describes assignment expressions E, which are typically of the form "$L := E$" for L an lvalue.

$$E \to L \mid L := E \mid i[E] \text{ of } E \qquad\qquad L \to i \mid L[E] \mid L.i$$

The grammar is not LR(m) for any m, but is ML(3, 1) and ML(2, 2): a conflict arises between inputs of the form "$i[E]$ of E" and "$i[E] := E$", where the initial i should be kept as such and the parser should shift in the first case, and reduce to L in the second case. An ML(3, 1) or ML(2, 2) parser scans up to the "of" or ":=" token that resolves this conflict, across the infinite language generated by E.

Example 4 (Typed Pascal Declarations). Another example is a version of Pascal identifier declarations with type checking performed at the syntax level, which was proposed by Tai [26]. The grammar is LR(2) and ML(1,0):

$$\begin{aligned} D &\to \text{var}\, IL\, IT\, ; \mid \text{var}\, RL\, RT\, ; \\ IL &\to i\, ,\, IL \mid i & IT &\to :\ \text{integer} \\ RL &\to i\, ,\, RL \mid i & RT &\to :\ \text{real} \end{aligned}$$

On an input like "var i , i , i : real ;", a conflict arises between the reductions of the last identifier i to either an integer list IL or a real list RL with ":" as terminal lookahead. By delaying these reductions, one can identify either an integer type IT or a real type RT.

Non-Monotonicity. We found that non-monotonic behaviour with uniform ML parsers occurs more often than expected. Here is one example in addition to the C++ example given in Section 2.1; more could be found in particular with the RNA grammars of Reeder et al. [21].

Example 5 (Pascal Compound Statements). The following is an excerpt from ISO Pascal and defines compound statements C in terms of ";"-separated lists of statements S:

$$C \to \text{begin } L \text{ end} \qquad L \to L \text{ ; } S \mid S \qquad S \to \varepsilon \mid C$$

This is an LALR(1) and ML(1, 0) grammar, but it is not ML(2, 0): the nonterminal $[L;S]$ has a rule $[L;S] \to [L;S];[S]$, giving rise to a nonterminal $[S]$ with rules $[S] \to \varepsilon \mid [C]$ and a shift/reduce conflict—in fact, this argument shows more generally that the grammar is not ML(k, 0) for even k.

Parser Size. Because selML parsers introduce new context symbols only when required, they can be smaller than the corresponding LR or uniform ML parsers, which carry full lookahead lengths in their items—this issue has been investigated for instance by Ancona et al. [1] and Parr and Quong [20] for LR and LL parsers. Our results are inconclusive as to the difference of parser size (in terms of numbers of states) between selML and LR. However, selML parsers tend to be considerably smaller than uniform ML parsers. Compare, for example, the numbers of states in the case of ML and selML for Example 3, in Table 2.

In fact, we can make the argument more formal: consider the family of grammars $(G_4^j)_{j>0}$, each with rules:

$$\begin{aligned} &S \to A \mid D, \quad A \to a \mid Ab \mid Ac, \quad D \to EF^{j-1}F \mid E'F^{j-1}F', \\ &F \to a \mid bF, \quad F' \to f \mid bF', \qquad E \to e, \quad E' \to e\ . \end{aligned} \tag{G_4^j}$$

The uniform ML(j, 0) parser for G_4^j has exponentially many states in j, caused by the rules $[Aw] \to aw$ for all w in $\{b, c\}^j$, while the selective ML(j, 0) parser has only a linear number of states, as there is no need for delays in that part of the grammar.

6 Related Work

Grammar Transformations and Coverings. The idea of using grammar transformations to obtain LR(1) or even simpler grammars has been thoroughly investigated in the framework of *grammar covers* [19]. Among the most notable applications, Mickunas et al. [18] provide transformations from LR(k) grammars into much simpler classes such as *simple LR(1)* or *(1,1)-bounded right context*; Soisalon-Soininen and Ukkonen [24] transform *predictive LR(k)* grammars into LL(k) ones by generalizing the notion of left-corner parsing. Such techniques were often limited however to *right-to-right* or *left-to-right* covers, whereas our transformation is not confined to such a strict framework.

Parsing with Delays. A different notion of delayed reductions was already suggested by Knuth [13] and later formalized by Szymanski and Williams [25] as *LR(k, t) parsing*, where one of the t leftmost phrases in any rightmost derivation can be reduced using a lookahead of k symbols. The difference between the two notions of delay can be witnessed with linear grammars, which are LR(k, t) if and only if they are LR(k)—because there is always at most one phrase in a derivation—but selML(k, m) if and only if they are LR($k + m$)—as shown in Lemma 3.

Like selML languages, and unlike more powerful noncanonical classes, the class of LR(k, t) grammars characterizes deterministic context-free languages. The associated parsing algorithm is quite different however from that of selML parsing: it uses the two-stacks model of noncanonical parsing, where reduced nonterminals are pushed back at the beginning of the input to serve as lookahead in reductions deeper in the stack. Comparatively, selML parsing uses the conventional LR parsing tables with a single stack.

Selectivity. Several parser construction methods attempt to use as little "information" as possible before committing to a parsing action: Ancona et al. [1] and Parr and Quong [20] try to use as little lookahead as possible in LR(k) or LL(k) parsing, Demers [7] generalizes left-corner parsing to delay decisions possibly as late as an LR parser, and Fortes Gálvez et al. [9] propose a noncanonical parsing algorithm that explores as little right context as possible.

7 Concluding Remarks

Selective ML parsing offers an original balance between

– enlarging the class of admissible grammars, compared to LR parsing, while
– remaining a deterministic parsing technique, with linear-time parsing and exclusion of ambiguities,
– having a simple description as a grammar transformation, and
– allowing the concrete construction of LR parse tables.

This last point is also of interest to practitioners who have embraced general, nondeterministic parsing techniques [12]: unlike noncanonical or regular-lookahead extensions, selML parsers can be used for nondeterministic parsing exactly like LR parsers. Having fewer conflicts than conventional LR parsers, they will resort less often to nondeterminism, and might be more efficient.

References

1. Ancona, M., Dodero, G., Gianuzzi, V., Morgavi, M.: Efficient construction of LR(k) states and tables. ACM Trans. Progr. Lang. Syst. 13(1), 150–178 (1991)
2. Basten, H.J.S.: The usability of ambiguity detection methods for context-free grammars. In: Electronic Notes in Theoretical Computer Science, LDTA 2008, vol. 238, pp. 35–46. Elsevier (2008)
3. Bermudez, M.E., Schimpf, K.M.: Practical arbitrary lookahead LR parsing. J. Comput. Syst. Sci. 41(2), 230–250 (1990)

4. Bertsch, E., Nederhof, M.J.: Some observations on LR-like parsing with delayed reduction. Inf. Process. Lett. 104(6), 195–199 (2007)
5. Boullier, P.: Contribution à la construction automatique d'analyseurs lexicographiques et syntaxiques. Thèse d'État, Université d'Orléans (1984)
6. Čulik, K., Cohen, R.: LR-Regular grammars—an extension of LR(k) grammars. J. Comput. Syst. Sci. 7(1), 66–96 (1973)
7. Demers, A.J.: Generalized left corner parsing. In: POPL 1977, pp. 170–182. ACM (1977)
8. Farré, J., Fortes Gálvez, J.: A Bounded Graph-Connect Construction for LR-regular Parsers. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 244–258. Springer, Heidelberg (2001)
9. Gálvez, J.F., Schmitz, S., Farré, J.: Shift-Resolve Parsing: Simple, Unbounded Lookahead, Linear Time. In: Ibarra, O.H., Yen, H.-C. (eds.) CIAA 2006. LNCS, vol. 4094, pp. 253–264. Springer, Heidelberg (2006)
10. Gosling, J., Joy, B., Steele, G.: The Java$^{\mathrm{TM}}$ Language Specification, 1st edn. Addison-Wesley (1996)
11. ISO: ISO/IEC 14882:1998: Programming Languages — C++. International Organization for Standardization, Geneva, Switzerland (1998)
12. Kats, L.C., Visser, E., Wachsmuth, G.: Pure and declarative syntax definition: Paradise lost and regained. In: OOPSLA 2010, pp. 918–932. ACM (2010)
13. Knuth, D.E.: On the translation of languages from left to right. Inform. and Cont. 8(6), 607–639 (1965)
14. Leermakers, R.: Recursive ascent parsing: from Earley to Marcus. Theor. Comput. Sci. 104(2), 299–312 (1992)
15. Malloy, B.A., Power, J.F., Waldron, J.T.: Applying software engineering techniques to parser design: the development of a C# parser. In: SAICSIT 2002, pp. 75–82. SAICSIT (2002)
16. Marcus, M.P.: A Theory of Syntactic Recognition for Natural Language. Series in Artificial Intelligence. MIT Press (1980)
17. McPeak, S., Necula, G.C.: Elkhound: A Fast, Practical GLR Parser Generator. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, pp. 73–88. Springer, Heidelberg (2004)
18. Mickunas, M.D., Lancaster, R.L., Schneider, V.B.: Transforming LR(k) grammars to LR(1), SLR(1), and (1,1) Bounded Right-Context grammars. J. ACM 23(3), 511–533 (1976)
19. Nijholt, A.: Context-free grammars: Covers, normal forms, and parsing. LNCS, vol. 93. Springer (1980)
20. Parr, T.J., Quong, R.W.: LL and LR translators need $k > 1$ lookahead. ACM Sigplan. Not. 31(2), 27–34 (1996)
21. Reeder, J., Steffen, P., Giegerich, R.: Effective ambiguity checking in biosequence analysis. BMC Bioinformatics 6, 153 (2005)
22. Schmitz, S.: An experimental ambiguity detection tool. Science of Computer Programming 75(1-2), 71–84 (2010)
23. Sippu, S., Soisalon-Soininen, E.: Parsing Theory, vol. II: LR(k) and LL(k) Parsing. EATCS Monographs on Theoretical Computer Science, vol. 20. Springer (1990)
24. Soisalon-Soininen, E., Ukkonen, E.: A method for transforming grammars into LL(k) form. Acta Inf. 12(4), 339–369 (1979)
25. Szymanski, T.G., Williams, J.H.: Noncanonical extensions of bottom-up parsing techniques. SIAM J. Comput. 5(2), 231–250 (1976)
26. Tai, K.C.: Noncanonical SLR(1) grammars. ACM Trans. Progr. Lang. Syst. 1(2), 295–320 (1979)

Author Index