LNCS 7210

ARCoSS

Michael O'Boyle (Ed.)

Compiler Construction

21st International Conference, CC 2012
Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012
Tallinn, Estonia, March/April 2012, Proceedings

Lecture Notes in Computer Science 7210

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

Editorial Board

David Hutchison, UK
Takeo Kanade, USA
Josef Kittler, UK
Jon M. Kleinberg, USA
Alfred Kobsa, USA
Friedemann Mattern, Switzerland
John C. Mitchell, USA
Moni Naor, Israel
Oscar Nierstrasz, Switzerland
C. Pandu Rangan, India
Bernhard Steffen, Germany
Madhu Sudan, USA
Demetri Terzopoulos, USA
Doug Tygar, USA
Gerhard Weikum, Germany

Advanced Research in Computing and Software Science

Subline of Lectures Notes in Computer Science

Subline Series Editors

Giorgio Ausiello, *University of Rome 'La Sapienza', Italy*
Vladimiro Sassone, *University of Southampton, UK*

Subline Advisory Board

Susanne Albers, *University of Freiburg, Germany*
Benjamin C. Pierce, *University of Pennsylvania, USA*
Bernhard Steffen, *University of Dortmund, Germany*
Madhu Sudan, *Microsoft Research, Cambridge, MA, USA*
Deng Xiaotie, *City University of Hong Kong*
Jeannette M. Wing, *Carnegie Mellon University, Pittsburgh, PA, USA*

Michael O'Boyle (Ed.)

Compiler Construction

21st International Conference, CC 2012
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2012
Tallinn, Estonia, March 24 – April 1, 2012
Proceedings

Springer

Volume Editor

Michael O'Boyle
University of Edinburgh
School for Informatics
10 Crichton Street, Edinburgh, EH8 9AB, UK
E-mail: mob@inf.ed.ac.uk

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-28651-3 e-ISBN 978-3-642-28652-0
DOI 10.1007/978-3-642-28652-0
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012932643

CR Subject Classification (1998): D.2, D.3, D.2.4, C.2, D.4, D.1, F.3.2

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

© Springer-Verlag Berlin Heidelberg 2012
This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.
The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

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

Printed on acid-free paper

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

Foreword

ETAPS 2012 is the fifteenth 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), 21 satellite workshops (ACCAT, AIPA, BX, BYTECODE, CMCS, DICE, FESCA, FICS, FIT, GRAPHITE, GT-VMT, HAS, IWIGP, LDTA, LINEARITY, MBT, MSFP, PLACES, QAPL, VSSE and WRLA), and eight invited lectures (excluding those specific to the satellite events).

The six main conferences received this year 606 submissions (including 21 tool demonstration papers), 159 of which were accepted (6 tool demos), giving an overall acceptance rate just above 26%. 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.

This year, ETAPS welcomes a new main conference, *Principles of Security and Trust*, as a candidate to become a permanent member conference of ETAPS. POST is the first addition to our main programme since 1998, when the original five conferences met in Lisbon for the first ETAPS event. It combines the practically important subject matter of security and trust with strong technical connections to traditional ETAPS areas.

A step towards the consolidation of ETAPS and its institutional activities has been undertaken by the Steering Committee with the establishment of *ETAPS e.V.*, a non-profit association under German law. ETAPS e.V. was founded on April 1st, 2011 in Saarbrücken, and we are currently in the process of defining its structure, scope and strategy.

ETAPS 2012 was organised by the *Institute of Cybernetics at Tallinn University of Technology*, in cooperation with

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

and with support from the following sponsors, which we gratefully thank:

> INSTITUTE OF CYBERNETICS AT TUT; TALLINN UNIVERSITY OF TECHNOLOGY (TUT); ESTONIAN CENTRE OF EXCELLENCE IN COMPUTER SCIENCE (EXCS) FUNDED BY THE EUROPEAN REGIONAL DEVELOPMENT FUND (ERDF); ESTONIAN CONVENTION BUREAU; and MICROSOFT RESEARCH.

The organising team comprised:

General Chair: *Tarmo Uustalu*

Satellite Events: *Keiko Nakata*

Organising Committee: *James Chapman, Juhan Ernits, Tiina Laasma, Monika Perkmann* and their colleagues in the *Logic and Semantics* group and *administration* of the *Institute of Cybernetics*

The ETAPS portal at http://www.etaps.org is maintained by *RWTH Aachen University.*

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

Vladimiro Sassone (Southampton, Chair), Roberto Amadio (Paris 7), Gilles Barthe (IMDEA-Software), David Basin (Zürich), Lars Birkedal (Copenhagen), Michael O'Boyle (Edinburgh), Giuseppe Castagna (CNRS Paris), Vittorio Cortellessa (L'Aquila), Koen De Bosschere (Gent), Pierpaolo Degano (Pisa), Matthias Felleisen (Boston), Bernd Finkbeiner (Saarbrücken), Cormac Flanagan (Santa Cruz), Philippa Gardner (Imperial College London), Andrew D. Gordon (MSR Cambridge and Edinburgh), Daniele Gorla (Rome), Joshua Guttman (Worcester USA), Holger Hermanns (Saarbrücken), Mike Hinchey (Lero, the Irish Software Engineering Research Centre), Ranjit Jhala (San Diego), Joost-Pieter Katoen (Aachen), Paul Klint (Amsterdam), Jens Knoop (Vienna), Barbara König (Duisburg), Juan de Lara (Madrid), Gerald Lüttgen (Bamberg), Tiziana Margaria (Potsdam), Fabio Martinelli (Pisa), John Mitchell (Stanford), Catuscia Palamidessi (INRIA Paris), Frank Pfenning (Pittsburgh), Nir Piterman (Leicester), Don Sannella (Edinburgh), Helmut Seidl (TU Munich),

Scott Smolka (Stony Brook), Gabriele Taentzer (Marburg), Tarmo Uustalu (Tallinn), Dániel Varró (Budapest), Andrea Zisman (London), and Lenore Zuck (Chicago).

I would like to express my sincere gratitude to all of these people and organisations, the Programme Committee Chairs and PC 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.

Finally, I would like to thank the Organising Chair of ETAPS 2012, Tarmo Uustalu, and his Organising Committee, for arranging to have ETAPS in the most beautiful surroundings of Tallinn.

January 2012

Vladimiro Sassone
ETAPS SC Chair

Preface

This volume contains the paper presented at CC 2012, the 21st International Conference on Compiler Construction held on March 28–29 in Tallinn, Estonia as part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2012). Papers were solicited from a wide range of areas including compiler analysis, code generation and optimization, runtime systems, just-in-time compilation, programming tools, techniques for specific domains and the design and implementation of novel language constructs. The submssions and the papers in this volume reflect the variety of our research domain.

There were 51 submissions. Each submission was reviewed by at least three Program Committee members and was subjected to several rounds of thorough discussions. The Program Committee finally decided to accept 13 papers.

Many people contributed to the success of this conference. First of all, I would like to thank the authors for submitting their papers of high quality. I would particularly like to thank the members of the Program Committee for their insightful reviews and keeping to often tight timescales. Also thanks are due to the developers and supporters of the EasyChair conference managment system for making life so much easier for the authors and the Program Committee.

CC 2012 was made possible by the ETAPS Steering Committee and the Local Organizing Committee. Finally, I would like to thank Francois Bodin for his CC 2012 invited talk entitled "Programming Heterogeneous Many-Cores Using Directives."

January 2012 Michael O'Boyle

Organization

Program Committee

Erik Altman	IBM, USA
Rastislav Bodik	UC Berkeley, USA
John Cavazos	University of Delaware, USA
Nathan Clark	Georgia Tech, USA
Murray Cole	University of Edinburgh, UK
Alain Darte	CNRS, France
Bjorn De Sutter	Ghent University, Belgium
Amer Diwan	University of Colorado, USA
Derek Dreyer	Max Planck Institute for Software Systems, Germany
Matthew Flatt	University of Utah, USA
Sumit Gulwani	Microsoft Research, USA
Atsushi Igarashi	Graduate School of Informatics, Kyoto University, Japan
Ranjit Jhala	UC San Diego, USA
Andreas Krall	TU Wien, Austria
Julia Lawall	DIKU/INRIA-Regal, France
Anton Lokhmotov	ARM Ltd., UK
Püschel Markus	ETH Zurich, Switzerland
Michael O'Boyle	University of Edinburgh, UK
Erez Petrank	Microsoft Research and Technion, Israel
David Sands	Chalmers, Sweden
Vivek Sarkar	Rice University, USA
Jan Vitek	Purdue University, USA

Invited Program

Programming Heterogeneous Many-Cores Using Directives

Francois Bodin

CAPS Enterprise, Rennes, France
http://www.caps-entreprise.com/

Abstract. Directive-based programming models are a pragmatic way of adapting legacy codes to heterogeneous many-cores such as CPUs coupled with GPUs. They provide programmers an abstracted and portable interface for developing many-core applications. The directives are used to express parallel computation, data transfers between the CPU and the GPU memories and code tuning hints. The challenge for such environment is to achieve high programming productivity and at the same time provide performance portability across hardware platforms.

In this presentation we give an overview the state of the art of directives based parallel programming environments for many-core accelerators. In particular, we describe OpenACC (http://www.openacc-standard.org/), an initiative from CAPS, CRAY, NVIDIA and PGI that provides a new open parallel programming standard for C, C++ and Fortran languages. We show how tuning can be performed in such programming approach and specifically address numerical library inter-operability issues.

Table of Contents

GPU Optimisation

Improving Performance of OpenCL on CPUs 1
Ralf Karrenberg and Sebastian Hack

Automatic Restructuring of GPU Kernels for Exploiting Inter-thread Data Locality 21
Swapneela Unkule, Christopher Shaltz, and Apan Qasem

Program Analysis

Programming Paradigm Driven Heap Analysis 41
Mark Marron, Ondřej Lhoták, and Anindya Banerjee

Parallel Replication-Based Points-To Analysis 61
Sandeep Putta and Rupesh Nasre

A New Method for Program Inversion 81
Cong Hou, George Vulov, Daniel Quinlan, David Jefferson, Richard Fujimoto, and Richard Vuduc

Analytical Bounds for Optimal Tile Size Selection 101
Jun Shirako, Kamal Sharma, Naznin Fauzia, Louis-Noël Pouchet, J. Ramanujam, P. Sadayappan, and Vivek Sarkar

Objects and Components

Static Detection of Unsafe Component Loadings 122
Taeho Kwon and Zhendong Su

Object Model Construction for Inheritance in C++ and Its Applications to Program Analysis 144
Jing Yang, Gogul Balakrishnan, Naoto Maeda, Franjo Ivančić, Aarti Gupta, Nishant Sinha, Sriram Sankaranarayanan, and Naveen Sharma

GC-Safe Interprocedural Unboxing 165
Leaf Petersen and Neal Glew

Dynamic Analysis and Runtime Support

Compiler Support for Value-Based Indirect Branch Prediction 185
Muhammad Umar Farooq, Lei Chen, and Lizy Kurian John

Compiler Support for Fine-Grain Software-Only Checkpointing 200
Chuck (Chengyan) Zhao, J. Gregory Steffan, Cristiana Amza, and Allan Kielstra

VMAD: An Advanced Dynamic Program Analysis and Instrumentation Framework 220
Alexandra Jimborean, Luis Mastrangelo, Vincent Loechner, and Philippe Clauss

Sambamba: A Runtime System for Online Adaptive Parallelization 240
Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack

Author Index 245

Improving Performance of OpenCL on CPUs

Ralf Karrenberg and Sebastian Hack

Saarland University, Germany
{karrenberg,hack}@cdl.uni-saarland.de

Abstract. Data-parallel languages like OpenCL and CUDA are an important means to exploit the computational power of today's computing devices. In this paper, we deal with two aspects of implementing such languages on CPUs: First, we present a static analysis and an accompanying optimization to exclude code regions from control-flow to data-flow conversion, which is the commonly used technique to leverage vector instruction sets. Second, we present a novel technique to implement barrier synchronization. We evaluate our techniques in a custom OpenCL CPU driver which is compared to itself in different configurations and to proprietary implementations by AMD and Intel. We achieve an average speedup factor of 1.21 compared to naïve vectorization and additional factors of 1.15–2.09 for suited kernels due to the optimizations enabled by our analysis. Our best configuration achieves an average speedup factor of over 2.5 against the Intel driver.

Keywords: OpenCL, SIMD, Vectorization, Data Parallelism, Code Generation, Synchronization, Divergent Control Flow.

1 Introduction

In this paper, we present two techniques to speed up data-parallel programs on machines with explicit SIMD operations (e.g. current CPUs). Although we focus on OpenCL in this paper, the presented techniques are also applicable to similar languages like CUDA. A data-parallel program is written in a scalar style. It is then executed in n *instances* (sometimes called threads, however this is not to be confused with an operating system thread) on a computing device. To a certain extent, the order of execution among all instances of the program is unspecified to allow for parallel or sequential execution as well as a mixture of both. Every instance is identified with a *thread ID* which is called `tid` in the following. Usually, the data-parallel program uses the `tid` to index data. Hence, every instance can process a different data item.

Since data-parallel semantics explicitly do not define an order of the instances, languages like OpenCL lend themselves to vector processing. In this paper, we consider machines that support SIMD instruction sets such as Intel's SSE and AVX, or ARM's NEON. SIMD instructions operate on a vector of W data items where W is the SIMD width (e.g. 4 32 bit values for SSE, 8 for AVX). To implement a data-parallel program on such a processor, one creates a vector

M. O'Boyle (Ed.): CC 2012, LNCS 7210, pp. 1–20, 2012.
© Springer-Verlag Berlin Heidelberg 2012

program that executes W instances of the original program in parallel. The challenge in this setting is divergent control flow: at a conditional jump, it might be the case that instance i takes the branch, while instance j does not. Hence, the vector program must accommodate both situations. Usually this is solved by control-flow to data-flow conversion [1] where branches are replaced by predicate variables which express the control condition. Then, control flow is *linearized,* i.e. the vector program executes every instruction of the program and masks out the inactive computations using the predicates. The latter happens either by explicit *blending* of the values or by instruction predication provided by the hardware.

However, applying control-flow linearization naïvely leaves some potential unexploited: in many data-parallel programs, several branches do not diverge because they depend on values which are the same (*uniform*) for all instances. Consider the example in Figure 1:

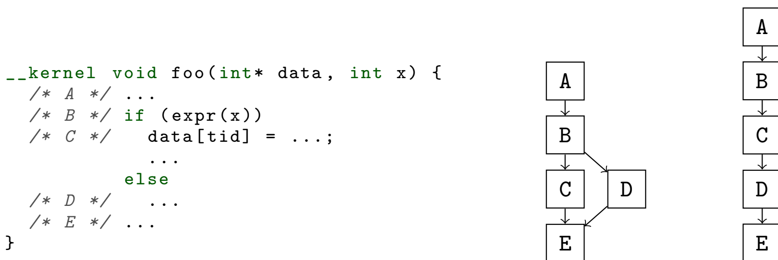


Fig. 1. An example kernel, the control-flow graph of its scalar version, and one possible linearization of the vector version after applying control-flow to data-flow conversion

The condition `expr(x)` only depends on uniform values.[1] Hence, the branch need not be linearized because either all instances take it or none. In addition, the evaluation of the condition can be hoisted outside the kernel. On the other hand, the computations inside the branch cannot be hoisted because they depend on the variable `tid` which is *not* uniform across all instances.

Previous work [22] injects code into the *linearized* vector program to *dynamically* test whether all instances evaluate a condition to the same value. If so, the corresponding code part can be bypassed by a branch, trading a reduction of the amount of executed code for some overhead for the dynamic test. While this reduces the number of instructions executed, it cannot compensate another drawback of control-flow to data-flow conversion: the increase of register pressure on architectures that do not support predication. Reconsider the example in Figure 1. When control flow is linearized, every variable that is live at the entrance of `D` will be live throughout `C`. Correspondingly, all variables live out at `C` will live throughout `D`. Our experiments have shown that the increase in register pressure often causes more spilling and reloading which deteriorates

[1] Note that the arguments passed to the kernel are the same for every instance, hence they are uniform.

performance. Keeping control flow for this code part prevents the increase of register pressure.

In this paper, we present a static analysis to identify branches that will never diverge and a code transformation that exploits these results to avoid control-flow linearization whenever possible. The result is a vector program in which only code parts of which we could not prove non-divergence are linearized.

Another important feature of languages like OpenCL is barrier synchronization. A kernel can use a *barrier* statement to enforce that no instance of a thread group executes the statement following the barrier before all instances of that group have reached the barrier. GPUs have dedicated hardware support to implement barrier synchronization. On CPUs, barriers need to be implemented in software. Simple implementations use the support of the runtime system and the operating system [24] which boils down to saving and restoring the complete state of the machine. More sophisticated techniques use loop fission on the abstract syntax tree to decompose the kernel into separate pieces that are executed in a way such that all barriers are respected [25]. However, this technique potentially introduces more synchronization points than needed. In this paper, we generalize the latter approach to work on control-flow graphs (CFGs, instead of abstract syntax trees) while not increasing the amount of synchronization points.

1.1 Contributions

To summarize, this paper makes three main contributions:

1. We present a static analysis that identifies blocks in the CFG of a data-parallel program that do not *diverge*. For these blocks, control flow can be retained and need not be replaced by data flow.
2. We present an SSA-based linearization algorithm that leverages the divergence analysis by retaining control flow, even in presence of irregular and arbitrarily nested control-flow structures (e.g. loops with multiple exits or jumps that exit multiple loops).
3. We present a novel technique to implement barrier synchronization for data-parallel languages on CPUs that splits the function into continuations. This does not require costly interaction with the OS, prevents introduction of additional synchronization points, and reduces overhead by only saving the live values.

1.2 Structure of This Paper

In the next section, we give an overview of our OpenCL driver and present our implementation of barrier synchronization. Section 3 describes the divergence analysis and how it can be used to increase the efficiency of vectorized kernels. Section 4 discusses related work and Section 5 presents our experimental evaluation.

2 OpenCL Driver Implementation

In this section, we describe code-generation techniques to improve the efficiency of an OpenCL driver. The compilation scheme of our driver looks like this:

1. Perform SIMD vectorization → Section 2.1
2. Implement barriers → Section 2.3
3. Create loops over local instances → Sections 2.2, 2.3
4. Remove API callbacks (such as `get_global_id()`) → Section 2.2
5. Create wrapper for driver interface

The interface wrapper allows the driver to call the kernel with a static signature that receives only a pointer to a structure with all parameters. Pseudo-code for the modified kernel is shown in Figure 2 (before inlining and the callback optimizations described in Section 2.2).

2.1 SIMD Vectorization

We employ a modified algorithm for "Whole-Function Vectorization" (WFV) [12] to exploit SIMD instruction sets by transforming the kernel such that it computes W instances of the original kernel.

Enabling WFV in OpenCL can be summarized as follows: Vectorization is based upon changing the callback functions `get_global_id()` and `get_local_id()` to return a vector of the W IDs whose instances are executed by the vectorized kernel. From there, all uses of these indices are vectorized, mask and blend operations are created as necessary, and control flow is linearized [12]. However, we enhance these phases by additional analyses and optimizations that are described in Section 3.

2.2 Runtime Callbacks

OpenCL allows the user to organize threads in multiple dimensions (each thread is identified by an n-tuple of IDs for n dimensions). Furthermore, it allows to create groups of threads that are executed together and can be synchronized (see Section 2.3).

Given a kernel and a global number of threads $N_x \times N_y$ organized in a two-dimensional grid with groups of size $G_x \times G_y$, the driver is responsible for calling the kernel $N_x \times N_y$ times and for making sure that calls to `get_local_id()` etc. return the appropriate thread ID of the given dimension. The most natural iteration scheme for this employs nested "outer" loops that iterate over the number of groups of each dimension (N_x/G_x and N_y/G_y) and nested "inner" loops that iterate over the size of each group (G_x and G_y). Consider Figure 2 for some pseudo-code.

If the application uses more than one dimension for its input data, the driver has to choose one *SIMD dimension* for vectorization. This means that only

```
clEnqueueNDRangeKernel(Kernel kernelWrapper, TA arg_struct,
        int* globalSizes, int* localSizes) {
    int iter_0 = globalSizes[0] / localSizes[0];
    int iter_1 = globalSizes[1] / localSizes[1];
    for (int i=0; i<iter_0; ++i) {
        for (int j=0; j<iter_1; ++j) {
            int groupIDs[2] = { i, j };
            kernelWrapper(arg_struct, groupIDs, globalSizes, localSizes);
}   }   }

void kernelWrapper(TA arg_struct, int* groupIDs,
        int* globalSizes, int* localSizes) {
    T0      param0 = arg_struct.p0;
    ...
    TN      paramN = arg_struct.pN;
    int     base0  = groupIDs[0] * localSizes[0];
    int     base1  = groupIDs[1] * localSizes[1];
    __m128i base0V = <base0, base0, base0, base0>;
    for (int i=0; i<localSizes[1]; ++i) {
        int lid1 = i;                           // local id   (dim 1)
        int tid1 = base1 + lid1;                // global id  (dim 1)
        for (int j=0; j<localSizes[0]; j+=4) {
            __m128i lid0 = <j, j+1, j+2, j+3>; // local ids  (dim 0)
            __m128i tid0 = base0V + lid0;      // global ids (dim 0)
            simdKernel(param0, ..., paramN, lid0, lid1, tid0, tid1,
                       groupIDs, globalSizes, localSizes);
}   }   }
```

Fig. 2. Pseudo-code implementation of clEnqueueNDRangeKernel and the kernel wrapper before inlining and optimization (2D case, $W = 4$). The outer loops iterate over the number of groups, which can easily be parallelized across multiple threads. The inner loops iterate over all instances of a group (step size 4 for the SIMD dimension 0).

queries for instance IDs of this dimension will return a vector, queries for other dimensions return a single ID. Because it is the natural choice for the kernels we have analyzed so far, our driver currently always uses the first dimension. However, it would be easy to implement a heuristic that chooses the best dimension, e.g. by comparing the number of memory operations that can be vectorized in either case. The inner loop that iterates over the dimension chosen for vectorization is incremented by W in each iteration as depicted in Figure 2.

We automatically generate a wrapper around the original kernel that includes the inner loops while only the outer loops are implemented directly in the driver (to allow multi-threading, e.g. via OpenMP). This allows us to remove all overhead of the callback functions: All these calls query information that is either statically fixed (e.g. `get_global_size()`) or only depends on the state of the inner loop's iteration (e.g. for one dimension, `get_global_id()` is the local size multiplied with the group ID plus the local ID). The static values are supplied as arguments to the wrapper, the others are computed directly in the inner loops. After the original kernel has been inlined into the wrapper, we can remove all overhead of callbacks to the driver by replacing each call by a direct access to a value. Generation of the inner loops "behind" the driver-kernel barrier also exposes additional optimization potential of the kernel code together with the

surrounding loops and the callback values. For example, loop-invariant code motion moves computations that only depend on group IDs out of the innermost loop (reconsider the example in Figure 1).

2.3 Continuation-Based Synchronization

OpenCL provides the `barrier()` statement to implement barrier synchronization of all threads in a group. A barrier enforces all threads of the group to reach it before the threads in the group can continue executing instructions behind the barrier. This means that the current context of a thread needs to be saved when it reaches the barrier and restored when it continues execution. Instead of relying on costly interaction with the operating system, we use the following code transformation to implement barrier synchronization.

Let the set $\{b_1, \ldots, b_n\}$ be the set of all barriers in the kernel. We apply the following recursive scheme: From the start node of the CFG, start a depth-first search (DFS) which does not traverse barriers. All nodes reached by this DFS are by construction barrier free. The search furthermore returns a set of barriers $B = \{b_{i_1}, \ldots, b_{i_m}\}$ which it hit. At each hit barrier, we determine the live variables and generate code to store them into a structure. For every instance in the group, such a structure is allocated by the driver. The last instruction generated is a return with the ID of the hit barrier. Now, the instructions $b_{i_1}, \ldots, b_{i_m}$ are taken as start points for m different kernels. For each one, we apply the same scheme until there are no more kernels containing barriers. Figure 3 gives an example for this transformation.

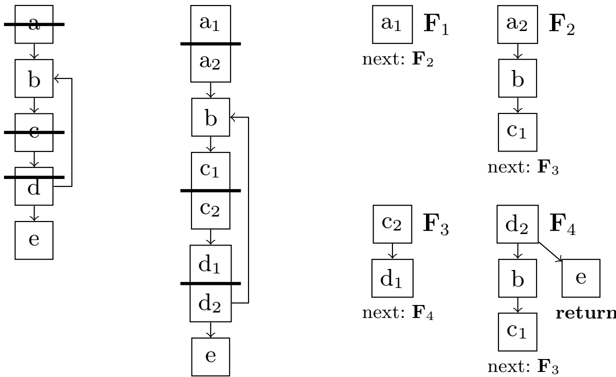


Fig. 3. Example CFG of a kernel which requires synchronization (the barriers are indicated by the bars crossing the blocks), the CFG after splitting blocks with barriers, and the resulting set of new functions $\{F_1, \ldots, F_4\}$

Then, we generate a wrapper that switches over the generated functions dependent on the last returned barrier ID (see Figure 4).

```
void newKernel(T0 param0, ..., TN paramN, int localSize, ...) {
    void* data[localSize/W] = alloc((localSize/W) * liveValSize);
    int next = BARRIER_BEGIN;
    while (true) {
        switch (next) {
            case BARRIER_BEGIN:
                for (int i=0; i<localSize; i+=W)
                    next = F1(param0, ..., paramN, tid, ..., &data[i/W]);
                break;
            case B2:
                for (int i=0; i<localSize; i+=W)
                    next = F2(tid, ..., &data[i/W]);
                break;
            ...
            case B4:
                for (int i=0; i<localSize; i+=W)
                    next = F4(tid, ..., &data[i/W]);
                break;
            case BARRIER_END: return;
}   }   }
```

Fig. 4. Pseudo code for the kernel of Figure 3 after implementation of barriers and before inlining and optimization (1D, computations of `tid` etc. are omitted). The value of `liveValSize` is the maximum size required for any continuation, `data` is the storage space for the live variables of all instances.

Note that the semantics of OpenCL require all instances to hit the *same* barrier, otherwise the program's behavior is undefined. Hence, if not all instances return the same ID, the kernel is in a bad state anyways, so we simply use the ID returned by the last instance.

3 Exploiting Uniform Computations

In this section, we describe our main contribution, a static analysis of control-flow divergence and its application in the context of whole-function vectorization. The analysis is based on a value analysis presented by Karrenberg and Hack [12] which identifies *uniform* computations.

3.1 Uniform Value Analysis

The uniform value analysis determines whether an operation produces the same value for all instances of a kernel. If a value is not uniform, we call it *varying*. If a branch depends on a varying condition, we call it a *varying branch*. Input arguments to OpenCL kernels are always uniform because all instances are called with the same arguments. The OpenCL functions `get_global_id()` and `get_local_id()` produce varying values if called with the SIMD dimension as parameter. If called with another dimension, they produce uniform values because we only vectorize one dimension (see Section 2.2).

3.2 Divergence Analysis

As described in the introduction, our goal is to retain as much control flow during vectorization as possible. To exclude a certain code part from control-flow to data-flow conversion, we must prove that the instances implemented by the vector program never diverge in this code part. In general, this is a dynamic property that can change for different kernel inputs. The following definition describes an overapproximation of *divergence* which can be statically proven.

Definition 1 (Static Divergence). *Let b be a block that can be reached from another block v that ends with a varying branch. b is marked as divergent if*

1. *b is a direct successor of v, or*
2. *there exist two* disjoint *paths from v to b, or*
3. *b is an exit block of a loop which includes v, and there exist two* disjoint *paths from v to b and from v to the loop's latch ℓ.*[2]

Figure 5 illustrates these conditions, Figures 6 and 7 depict some more involved examples.

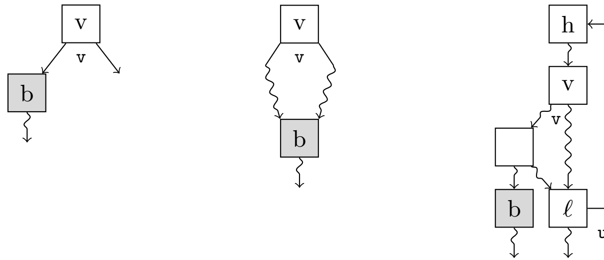


Fig. 5. Illustration of the three possibilities for divergence of a block b (Definition 1). Note that in the second case, b is neither required to post-dominate v, nor is v required to dominate b.

Informally, the second condition means that in the kernel execution under consideration, b can be reached by some instances from *both* edges that leave v. Hence, b is subject to control-flow to data-flow conversion.

The third condition is required because loops behave differently in terms of divergence: If b is a loop exit and v is inside the loop, there might be a path from one edge of v to an exit block and another, disjoint path from the other edge to a back edge. Even if all exit branches are uniform, this would still make it possible that some instances are still active in the loop when an exit edge is taken. Therefore, b is divergent under this condition.

[2] We assume that every loop has a latch which is the only basic block in the loop that has a back edge to the header.

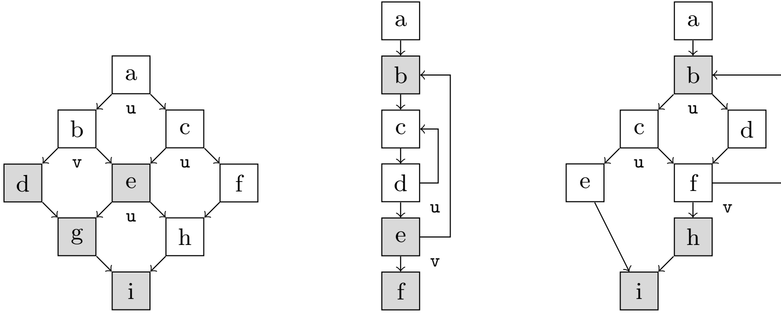


Fig. 6. Example CFGs showing our analysis results. Uniformity of conditional branches is shown with a lowercase letter below the block, divergent blocks are shaded. Our analysis determines that significant parts of these CFGs are non-divergent and therefore do not have to be linearized (see Figure 10).

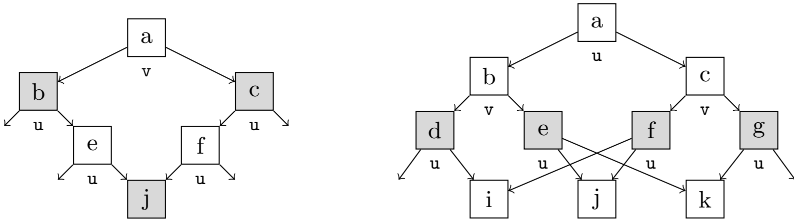


Fig. 7. More complex examples. In the left CFG, j is neither always executed by all instances that were active in a nor is it only executed by instances that took the left *or* right branch in a. Therefore, the linearized CFG has to make sure that j has to be executed regardless of the branch decision in a. In the right CFG, i, j, and k are non-divergent because none can be reached from both edges of the same varying branch. However, linearization requires duplication (see Section 3.3).

We compute uniformity and divergence using a data-flow analysis. Both properties influence each other mutually: First, as can be seen from Definition 1, divergence depends on uniformity. If a branch is labelled varying, control flow diverges. Second, divergence also influences uniformity. Consider a ϕ-function over uniform arguments. If that ϕ-function resides in a divergent block, the ϕ's value is *not* uniform itself, because not all instances enter the ϕ's block from the same predecessor in every execution. However, if we can prove the ϕ's block non-divergent, the ϕ's value is uniform, too. Hence, the data-flow analysis computes divergence and uniformity together.

The analysis uses join (not meet) lattices and employs the common perspective that instructions reside on edges not nodes. Program points thus sit between the instructions (see Figure 8). This has the advantage that the join and the update of the flow facts are cleanly separated.

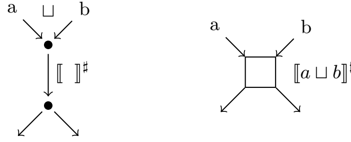


Fig. 8. Left: Our analysis setup with separated join and update of flow facts. Right: Classic setup with mixed join/update.

The analysis lattice uses two simple lattices for uniformity and divergence:

varying	v	d	divergent
uniform	u	n	non-divergent
	$\mathbb{U}$	$\mathbb{D}$	

The analysis lattice itself is defined as

$$\mathbb{L} := \mathcal{P}(V \times V) \times (V \to \mathbb{U}) \times \mathbb{D}$$

We record in every flow fact a set A of control-flow edges, a mapping u from variables to uniformity information, and information about divergence d of that program point. The join is defined component-wise:

$$(A, u, d) \sqcup (A', u', d') = (A \cup A', u \sqcup_{\mathbb{U}} u', \lambda w. \, (d(w) \sqcup_{\mathbb{D}} d'(w)))$$

For a program point x, the update function

$$(A', u', d') = [\![w]\!]^\sharp(A, u, d), \qquad [\![\cdot]\!]^\sharp : \mathbb{L} \to \mathbb{L}$$

is defined as follows. First, consider the divergence update which reflects Definition 1:

$$d' = \begin{cases} \mathtt{d} & \text{if } \exists p \in \mathit{pred}(x). \, u(p) = \mathtt{v} \\ \mathtt{d} & \text{if } \exists p_1, p_2 \in \mathit{pred}(x). \, \mathit{divergedPaths}_A(p_1, p_2) \\ \mathtt{d} & \text{if } x \in E_L \wedge \mathit{divergedPaths}_A(x, \ell_L) \\ \mathtt{n} & \text{otherwise} \end{cases}$$

where E_L is the set of program points behind the exits of a loop L and ℓ_L is the program point at the loop's back edge. Further, *pred*(x) is the set of predecessor program points of x. The Information whether two paths are disjoint and divergent is given by the helper function

$$\mathit{divergedPaths}_A(p_1, p_2) = \exists e_t, e_f \in A(p_1) \cup A(p_2). \, (e_t \notin A(p_1) \vee e_f \notin A(p_2)).$$

divergedPaths uses the set A which captures information about edges leaving blocks with varying branch conditions. The control-flow edge is inserted into A if the branch of a block was detected varying by the analysis. To this end, every

branch instruction gets two extra program points (one for the `true` and one for the `false` program point) on which transfer functions can be placed that add the corresponding edges to A. For each branch node v, let v_t and v_f be these program points. For some $v_\Box$, the update function for A is

$$A' = A \cup \begin{cases} \{v_\Box\} & \text{if } u(v) = \mathtt{v} \\ \emptyset & \text{otherwise,} \end{cases}$$

otherwise we define $A' = A$.

Finally, the uniformity part provides information to the other two components of the analysis. A common operation $s = \oplus(o_1, \ldots, o_n)$ is uniform if all operands o_i are uniform. Thus, using $f \mid x \mapsto y$ as an abbreviation for

$$\lambda w. \begin{cases} y & \text{if } w = x \\ f(w) & \text{otherwise,} \end{cases}$$

the uniformity update for non-ϕ instructions is given by:

$$u' = u \mid s \mapsto \bigsqcup_i u(o_i)$$

For ϕ-functions, this does not hold. Even if all parameters of a ϕ are uniform, the ϕ will *not* produce a uniform value if two instances can enter it via *different* predecessors. Hence, to produce a uniform value, the ϕ-function's program point has to be detected non-divergent.

$$u'_\phi = u \mid s \mapsto \begin{cases} \mathtt{u} & \text{if } d(s) = \mathtt{n} \wedge \bigsqcup_i u(o_i) = \mathtt{u} \\ \mathtt{v} & \text{otherwise} \end{cases}$$

We omit the proof of monotonicity due to space limitations.

3.3 Optimizations

In the following paragraphs we describe techniques to take advantage of the results of the presented uniformity and divergence analyses.

Retaining Scalar Computations. The uniformity analysis allows us to prevent vectorization of uniform values. The benefit of using uniform computations is straightforward: register pressure is taken from the vector units and scalar computations can be executed in parallel to the vector computations. Furthermore, if a uniform value is used by a varying instruction, the value is *broadcast* to a vector beforehand. As a byproduct, our analysis computes the program points where scalar values have to be broadcast to vector values. This is important because we observed that eager broadcasting often causes performance degradation.

Retaining Control Flow. As discussed in the introduction, the divergence analysis opens up the possibility to retain uniform control flow. In the following, we describe an algorithm for CFG linearization which allows to exclude arbitrary, non-divergent control-flow structures.

First, we build regions of divergent blocks using a depth-first search on the CFG. While traversing, we create a new divergent region whenever we see a varying branch and mark this region as *active*. If we encounter a block that post-dominates all blocks in this region, we finish the region and set the active region to the last unfinished one. Divergent blocks are always added to the active region. Regions that overlap or have common entry or exit blocks are merged.

Next, each region is linearized recursively (inner regions before outer regions) as follows: We first determine an order for the contained blocks by sorting them topologically by data dependencies (linearized, inner regions are treated like one block). Next, we schedule the blocks of the region by rewiring all edges that target a divergent block. A block is scheduled after all its edges have been visited. The new target of each edge is the first divergent block of the current region's order that has not yet been scheduled. Edges that target non-divergent blocks remain untouched.

The reason behind creating schedules of the blocks and rewiring edges is that no block of a divergent region must be skipped during execution because this may violate the semantics of the original kernel.

Figure 9 illustrates an example where the non-divergent block e has a neighbor d that is divergent and thus always has to be executed. If we linearize all divergent blocks and retain the incoming and outgoing edges of e, we end up with a graph where blocks b and d can be skipped (Figure 9(d)), although some instances might want to take the path $a \to b \to d \to f \to g$. Dependencies are maintained correctly by rewiring the edge $e \to f$ to $e \to b$ (Figure 9(e)).

Figure 10 shows linearizations of the examples of Figure 6. In the leftmost CFG, only d, e, g, and i have to be linearized due to the varying branch in b. Because only one path from a to h leads through a varying branch, h is non-divergent. In the middle CFG, the inner loop, although being nested in a loop with varying exit, does not require any mask updates or blending because all active instances always leave the loop together. The rightmost CFG shows a case where it is allowed to retain the uniform loop exit branch in c: there are only uniform branches inside the loop, so either all or no active instance will leave the loop at this exit. However, h must not be skipped because of instances that might have left the loop earlier.

Linearization of patterns such as in the second graph of Figure 7 requires additional logic. This is because there is no schedule where all edges leaving i, j, and k can be rewired to a single block that has not yet been scheduled without violating dependencies. One possibility to handle this is duplication of code, another one is inserting conditional branches that depend on the previously executed divergent block. Due to space constraints we omit linearization examples for Figure 7 and leave a more detailed discussion of this issue for future work.

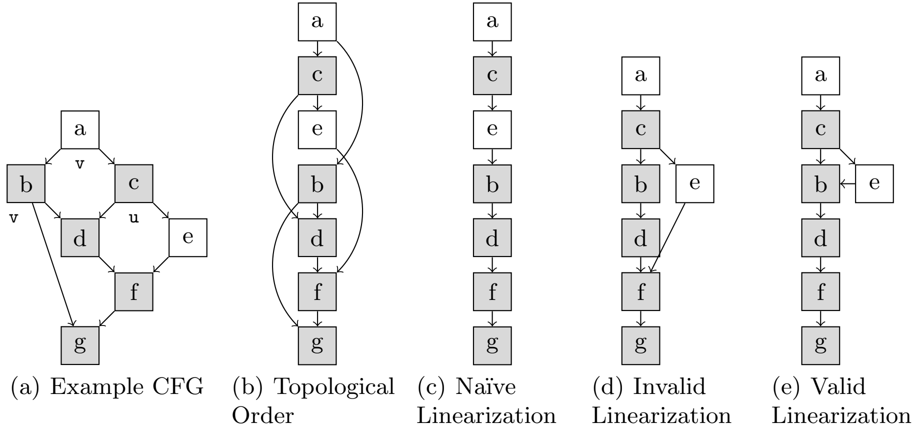


Fig. 9. CFG linearization example. In the original CFG (a), e is non-divergent because it can not be reached through different edges of varying branches. The topological sorting that is used in this linearization is shown in (b). The linearization (d) is invalid because it must not be possible to skip b and d. The graph (e) shows the correct linearization, which is likely to have better runtime than the naïve approach (c).

Reducing Mask Operations. During control-flow to data-flow conversion, each block is assigned a mask that is updated at each varying branch (conjunction and negation) and at each join block (disjunction).

If a conditional branch is uniform however, we use the incoming mask of the block for both outgoing edges instead of updating the mask with the comparison result. This implies that on paths with only non-divergent blocks, all edges have the same mask as the first block.

If our analysis found out that a block is always executed by *all* instances, the mask is set to `true`. At the end of regions with a single entry and exit block, the mask is reset to the one of the entry block. If a non-divergent block has multiple incoming edges, we generate a ϕ-operation instead of mask disjunctions. This is because only one of the incoming paths may have been executed.

In the rightmost CFG of Figure 10, blocks c and d can both use the entry mask of block b instead of performing conjunction-operations with the (negated) branch condition in b, and block f can use the same mask instead of the disjunction of both incoming masks.

Loops require special *loop exit masks* in order to store the information which instances have left the loop through which exits [12]. However, if an exit block is non-divergent, we omit its loop exit mask because it is equal to the active mask. If all exit blocks are non-divergent no loop mask is required because all instances that enter the loop will exit together through one of the exits.

Reducing Blend Operations. If control flow is linearized, ϕ-operations in blocks with multiple predecessors have to be transformed to select-operations that conditionally blend together incoming values based on the active mask.

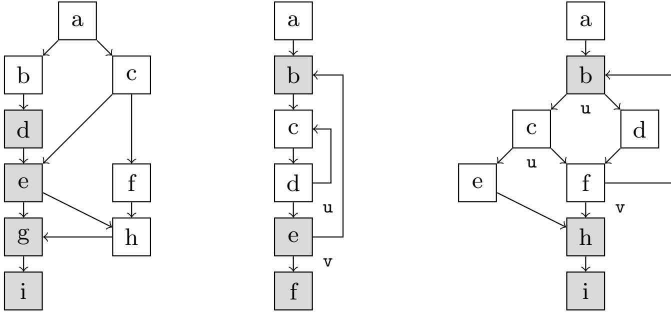


Fig. 10. Valid linearizations of the CFGs shown in Figure 6

However, if a block with multiple incoming edges is non-divergent, we can retain its ϕ-operations. Blending in such a case is not necessary because only one of the incoming paths may have been executed. For example, in the leftmost CFG of Figure 10, the ϕ's in block h remain untouched.

If an edge is rewired, the ϕ's from the old target block have to be modified (one direction will yield only dummy values that are later masked out) and moved to the new target block of the edge. For example, the ϕ's in block f of Figure 9 have to be moved to block b.

Loops require a blend operation for all values *live across loop boundaries* before the back branch of the loop [12]. If our analysis proves the loop to only exit to non-divergent blocks, we also do not require any blending because all instances will iterate equally often.

Optimizing Operations with Side-Effects. Operations with side-effects such as store operations and function calls have to be split up and each scalar operation must not be executed unless the corresponding mask element is `true`. If our analysis proves that a block is always executed by *all* instances, we prevent generation of such expensive code because all mask elements will be `true` when reaching the block.

Optimizing Loop Induction Variables. A more subtle optimization aims at values that are independent of any input of the function: values related to loop induction variables. Consider the Mandelbrot kernel in Figure 11. The exit condition of the main loop is varying because one of the two comparisons depends on varying values ($x2$, $y2$). Therefore, all values that are live across loop boundaries also have to be considered varying because they may differ between instances due to different loop trip counts. This means that `iter`—because it has a use after the loop—and all its uses (the increment and the comparison) have to be vectorized.

```
uint iter;
for(iter=0; (x2+y2 <= scaleSquare) && (iter < maxIter); ++iter) {
    y = 2 * x * y + y0;
    x = x2 - y2   + x0;
    x2 = x*x;
    y2 = y*y;
}
int tid = get_global_id(0);
image[tid] = 255*iter/maxIter;
```

Fig. 11. Main loop of the OpenCL Mandelbrot kernel

However, we can perform the following optimization: We introduce an additional vector variable that holds the "result" of `iter` which is updated after every iteration of the loop and is also used to determine when to exit the loop. The update is performed by a broadcast of `iter` (which remains scalar) followed by blend operation. This allows us to perform all computations inside the same loop iteration that only depend on input-independent or uniform values in scalar registers.

In the Mandelbrot example, this means that the increment of `iter` and the comparison `iter < maxIter` can remain scalar and `maxIter` does not require a broadcast. We observed a speedup of over 400% when applying this optimization (see Section 5). The reason for this is that the optimized operations are inside a frequently executed loop. One required vector register less or more in this critical part can affect performance significantly.

4 Related Work

Our SIMD vectorization technique stems from work of Allen [1] on the conversion of control flow to data flow. In classic loop vectorization [2,5,23,6,17], the innermost loop level is unrolled before combining isomorphic statements to vector operations ("unroll-and-jam"), either for vector machines or SIMD instruction sets. Target loops are usually restricted to either static iteration counts, specific data-dependency schemes, or straight-line code. Ngo [16] was the first to describe "Outer-Loop Vectorization" (OLV) [18]. In OLV, outer loops are unrolled to improve vectorization, e.g. due to longer trip counts of outer loops or better memory access schemes. Whole-Function Vectorization (WFV) [12] applied a scheme similar to OLV to data-parallel languages like OpenCL. By vectorizing an entire function, the new kernel computes W instances of the scalar code in parallel, effectively vectorizing the driver-level loop over the input data. Less generic approaches have been applied to various domain-specific languages [7,20,9,21,15].

The uniform value analysis was introduced as part of WFV. We are not aware of other related work that attempts to classify data-parallel code into uniform and varying instructions automatically. However, some data-parallel languages like ISPC [21] or the RenderMan Shading Language [4] have explicit keywords

to enforce specific behavior where required. In languages like OpenCL [13] and CUDA [19] this is not necessary because the semantics of the kernel are given by the execution model.

The only prior work directly related to our divergence analysis that we are aware of is a technique that employs "branches-on-superword-condition-code" (BOSCC) introduced by Shin [22]. This technique reintroduces control flow into vectorized code to exploit situations where the predicates (masks) for certain parts of the code are entirely `true` or `false` at runtime. While this prevents unnecessary execution of code, it suffers from some overhead for the dynamic test and does not solve the problem of increased register pressure: the code still has to account for situations where the predicate is not entirely `true` or `false`. Our analysis can do better by providing the necessary information statically, which allows to retain the original control flow that does not require any blending. However, it is possible to benefit from both our optimizations *and* BOSCCs. The next section will evaluate both approaches.

An increasing number of OpenCL drivers is being developed by different software vendors for all kinds of platforms from GPUs to mobile devices. For comparison purposes, the x86 CPU drivers by Intel [10] and AMD [3] are most interesting. However, there is not much detail on the underlying implementations. Both drivers have in common that they build on LLVM and exploit all available cores with some multi-threading scheme. The Intel driver also performs SIMD vectorization similar to our implementation[3]. However, to our knowledge, it lacks analyses to retain uniform computations and control flow, an important source of performance (see Section 5).

Recently, The Portland Group released an x86 CPU driver for CUDA [26] that also makes use of both multi-threading and whole-function vectorization, but no implementation details are publicly available. MCUDA [25] is another x86 CPU implementation of CUDA that introduced the first "thread loop"-based synchronization scheme. Their approach uses loop fission of the thread loop to remove barriers, which results in similar code as our approach if no barriers are inside loops. In that scenario however, MCUDA generates additional, artificial synchronization points at the loop header and before the back branch. This can impose significant overhead due to additional loading and storing of live variables. Jääskeläinen et al. [11] implemented a standalone OpenCL compiler that generates customized code for FPGAs and also uses this synchronization scheme. In contrast to our driver, they rely on the instruction-level parallelism of the FPGA design by duplicating kernel code W times instead of performing explicit SIMD vectorization. TwinPeaks [8] is an implementation of the OpenCL API that targets both CPUs and GPUs but does not perform aggressive code transformations. Their synchronization scheme uses custom implementations of `setjmp()/longjmp()` whereas our driver modifies the kernel code directly, storing only the live values instead of blindly saving registers. Clover [24] is an open

[3] We have no information about the AMD driver but suspect that no whole-function vectorization is used due to the inferior performance.

Table 1. Median kernel execution times of our OpenCL driver in different configurations for different applications (no multi-threading, 50 iterations). The row "speedup" shows the effect of our divergence optimizations, comparing "UniCF" to "UniVal" (95% confidence level).

OpenCL Kernel Performance (milliseconds)

Application	Input Size	Scalar	Naïve	UniVal	BOSCC	UniCF	Speedup
BitonicSort	1,048,576	1,649	549	519	518	519	1.00×
BlackScholes	16,777,216	2,743	713	672	672	672	1.00×
DCT	$4{,}000^2$	732	1,100	857	857	411	2.09×
FastWalshTransform	134,217,728	9,852	13,317	13,450	13,451	13,458	1.00×
FloydWarshall	512	444	4,081	3,592	3,420	3,603	1.00×
Histogram	$15{,}000^2$	1,575	1,703	1,454	1,469	1,266	1.15×
Mandelbrot	$8{,}192^2$	4,136	8,114	1,724	1,727	1,725	1.00×
MatrixTranspose	$12{,}000^2$	2,295	2,378	1,599	1,599	1,600	1.00×
NBody	19,968	3,768	2,099	1,410	1,408	1,035	1.36×

source OpenCL driver on top of Gallium3D which implements synchronization with POSIX contexts. Both TwinPeaks and Clover do not employ WFV.

5 Experimental Evaluation

Our OpenCL driver is based on the LLVM compiler framework [14] and the AMD APP SDK [3]. We did not attempt to implement the full OpenCL 1.1 API rather than a sufficiently complete fraction to run benchmarks from the APP SDK. If necessary, the benchmarks were modified to only use scalar values instead of the OpenCL built-in vectors to allow for automatic vectorization. All experiments were conducted on a Core 2 Quad at 2.8 GHz with 4 GB of RAM running Windows 7. The vector instruction set is Intel's SSE 4.1, yielding a SIMD width of four 32 bit values. The machine ran in 64 bit mode, thus 16 vector registers were available.

We report kernel execution times of our driver in different configurations and compare to Intel's [10] and AMD's [3] CPU driver. Each measurement shows the median of 50 individual runs per configuration per benchmark without warm-up. Although the machine was not rebooted after every run, the numbers reported here are as realistic as possible for one cold-started, arbitrary run of the application. In addition, we conducted tests that ensure significance of our results using the "SpeedUp Test" [27].

5.1 Benchmarks

Table 1 shows the runtime performance of a diverse set of applications in different configurations: The first configuration ("scalar") performs no vectorization, so the kernel is executed sequentially. The "naïve" configuration performs vectorization without retaining uniform values and with complete linearization. The

Table 2. Median kernel execution times of our OpenCL driver (VecOCL, vectorized and multi-threaded) compared to the proprietary drivers of Intel and AMD. Values marked with an asterisk are execution times without WFV: for FloydWarshall, the Intel driver does not perform vectorization. We obtain an average speedup factor of the median of over 2.5 against the Intel driver at a confidence level of 95%.

OpenCL Kernel Performance (milliseconds)

Application	VecOCL	Intel	AMD	Speedup vs Intel
BitonicSort	164	1,170	47,271	7.13×
BlackScholes	241	329	717	1.37×
DCT	201	350	693	1.74×
FastWalshTransform	4,944	6,661	8,601	1.35×
FloydWarshall	934(148*)	525*	471	0.56×(3.55×*)
Histogram	387	1,178	527	3.07×
Mandelbrot	632	1,930	29,045	3.05×
MatrixTranspose	1,072	2,933	10,748	2.74×
NBody	343	676	1,253	1.97×

"UniVal" configuration linearizes all control flow, but retains uniform values where possible. The "UniCF" configuration additionally employs our analysis, leaving non-divergent control flow intact. "BOSCC" refers to the "UniVal" configuration with additional insertion of BOSCCs.

The overall observation is that performance increases with the addition of analyses and optimizations (from left to right in Table 1). Retaining uniform values proves to be effective for all of the benchmarks with an average speedup factor of 1.21. Retaining non-divergent control flow helps most in presence of loops with non-divergent exits as in DCT, Histogram, and NBody. These benchmarks profit from the reduced overhead of mask and blend operations and retained control flow, which results in speedup factors of 2.09, 1.15, and 1.36. As expected, there is no effect on benchmarks that do not have any non-divergent control flow, such as BitonicSort, BlackScholes, or MatrixTranspose.

Table 1 also shows numbers of a configuration that does not use our divergence analysis but inserts BOSCCs after linearization. It can be observed that this technique does not impact performance largely (only FloydWarshall runs 5% faster) in contrast to the configuration which makes use of our divergence analysis. This is mostly due to the fact that BOSCCs do not help in presence of loops, which are the hot spots in most of the benchmarks: a loop always has to be executed as long as any instance is still iterating. Introducing BOSCCs does not help here, whereas our optimizations can remove blend and mask operations if all loop exits are non-divergent. When combining all techniques, we match the performance of the "BOSCC"-configuration for FloydWarshall, all other benchmark results remain unchanged from "UniCF".

The Mandelbrot benchmark additionally profits from the special optimization described in Section 3, which resulted in a reduction of the kernel execution time from 8.1 seconds to 1.8.

It is also important to note that naïve vectorization is often inferior to scalar execution (DCT, Histogram, and MatrixTranspose), which highlights the importance of additional optimizations. Despite our efforts, there are still benchmarks that are not suited for vectorization such as FastWalshTransform and Floyd-Warshall, which are dominated by random memory accesses.

For a fair comparison against Intel's and AMD's drivers we implemented a naïve, unoptimized multi-threading scheme that uses OpenMP. Table 2 shows that our custom driver significantly outperforms both drivers in all test-cases (statistically significant with a confidence level of 95%).

6 Conclusion

Whole-function vectorization of kernels is the technique of choice to achieve maximum performance of data-parallel languages on CPUs. However, naïvely vectorizing all code can greatly limit the benefits due to the possibly large overhead of control-flow to data-flow conversion. We presented key techniques to reduce this overhead based on the analysis of divergent control flow.

In addition, we described code generation techniques to reduce the overhead that is inherent to data-parallel languages like OpenCL and CUDA: we integrated parts of the driver code into the kernel and used a novel synchronization-scheme based on continuations to enable aggressive optimizations.

Our techniques have proven to be successful on a variety of different benchmarks, significantly outperforming proprietary drivers by Intel and AMD.

We are aware of the fact that we did not provide a formal proof of our transformations. However, proving correctness requires a formal semantics of a data-parallel language such as OpenCL which has not been developed yet. Such a semantics would also enable a more formal definition of divergence. We leave this for future work.

Acknowledgement. This work is part of the ECOUSS project and has been funded by the German Ministry for Education and Science (BMBF) and the Intel Visual Computing Institute Saarbrücken. The authors would like to thank Christoph Mallon and Daniel Grund for insightful discussions about control-flow divergence. Furthermore, we thank Roland Leißa and the anonymous reviewers for their helpful comments and remarks.

References

1. Allen, J.R., Kennedy, K., Porterfield, C., Warren, J.: Conversion of control dependence to data dependence. In: POPL, pp. 177–189. ACM (1983)
2. Allen, R., Kennedy, K.: Automatic translation of FORTRAN programs to vector form. ACM Trans. Program. Lang. Syst. 9(4), 491–542 (1987)
3. AMD: AMD APP SDK v2.5 (March 2011)
4. Apodaca, A., Mantle, M.: RenderMan: Pursuing the Future of Graphics. IEEE Computer Graphics & Applications 10(4), 44–49 (1990)

5. Cheong, G., Lam, M.: An Optimizer for Multimedia Instruction Sets. In: Second SUIF Compiler Workshop (1997)
6. Darte, A., Robert, Y., Vivien, F.: Scheduling and Automatic Parallelization. Birkhauser, Boston (2000)
7. Fritz, N., Lucas, P., Slusallek, P.: CGiS, a New Language for Data-Parallel GPU Programming. In: VMV, pp. 241–248 (2004)
8. Gummaraju, J., Morichetti, L., Houston, M., Sander, B., Gaster, B.R., Zheng, B.: Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In: PACT, pp. 205–216. ACM, New York (2010)
9. Hormati, A.H., Choi, Y., Woh, M., Kudlur, M., Rabbah, R., Mudge, T., Mahlke, S.: Macross: macro-simdization of streaming applications. In: ASPLOS, pp. 285–296. ACM, New York (2010)
10. Intel: Intel OpenCL SDK 1.1 (June 2011)
11. Jaskelainen, P.O., de La Lama, C.S., Huerta, P., Takala, J.: OpenCL-based design methodology for application-specific processors. In: SAMOS 2010, pp. 223–230 (July 2010)
12. Karrenberg, R., Hack, S.: Whole Function Vectorization. In: CGO, pp. 141–150 (2011)
13. Khronos Group: OpenCL 1.1 Specification (June 2011)
14. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: CGO (March 2004)
15. Newburn, C.J., So, B., Liu, Z., McCool, M.D., Ghuloum, A.M., Toit, S.D., Wang, Z.G., Du, Z., Chen, Y., Wu, G., Guo, P., Liu, Z., Zhang, D.: Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In: CGO, pp. 224–235 (2011)
16. Ngo, V.: Parallel loop transformation techniques for vector-based multiprocessor systems. Ph.D. thesis, University of Minnesota-Twin Cities (May 1994)
17. Nuzman, D., Henderson, R.: Multi-platform auto-vectorization. In: CGO, pp. 281–294 (2006)
18. Nuzman, D., Zaks, A.: Outer-loop vectorization: revisited for short simd architectures. In: PACT, pp. 2–11. ACM (2008)
19. NVIDIA: CUDA Programming Guide (2009)
20. Parker, S., et al.: RTSL: A Ray Tracing Shading Language. In: IEEE Symposium on Interactive Ray Tracing (2007)
21. Pharr, M.: Intel SPMD Program Compiler (June 2011)
22. Shin, J.: Introducing Control Flow into Vectorized Code. In: PACT, pp. 280–291. IEEE Computer Society (2007)
23. Sreraman, N., Govindarajan, R.: A vectorizing compiler for multimedia extensions. Int. J. Parallel Program. 28(4), 363–400 (2000)
24. Steckelmacher, D.: An OpenCL State Tracker for Gallium based on Clover (August 2011), http://people.freedesktop.org/~steckdenis/clover
25. Stratton, J.A., Stone, S.S., Hwu, W.-m.W.: MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 16–30. Springer, Heidelberg (2008)
26. The Portland Group, Inc.: PGI CUDA-x86 (June 2011)
27. Touati, S.A.A., Worms, J., Briais, S.: The Speedup Test. Rapport de recherche (2010), http://hal.inria.fr/inria-00443839/en/

Automatic Restructuring of GPU Kernels for Exploiting Inter-thread Data Locality*

Swapneela Unkule, Christopher Shaltz, and Apan Qasem

Texas State University, San Marcos, TX 78666, USA

Abstract. Hundreds of cores per chip and support for fine-grain multithreading have made GPUs a central player in today's HPC world. For many applications, however, achieving a high fraction of peak on current GPUs, still requires significant programmer effort. A key consideration for optimizing GPU code is determining a suitable amount of work to be performed by each thread. Thread granularity not only has a direct impact on occupancy but can also influence data locality at the register and shared-memory levels. This paper describes a software framework to analyze dependencies in parallel GPU threads and perform source-level restructuring to obtain GPU kernels with varying thread granularity. The framework supports specification of coarsening factors through source-code annotation and also implements a heuristic based on estimated register pressure that automatically recommends coarsening factors for improved memory performance. We present preliminary experimental results on a select set of CUDA kernels. The results show that the proposed strategy is generally able to select profitable coarsening factors. More importantly, the results demonstrate a clear need for automatic control of thread granularity at the software level for achieving higher performance.

1 Introduction

Increased programmability and inclusion of higher precision arithmetic hardware has opened the doors for general-purpose computing on the GPU. With hundreds of cores and extremely fine-grain multithreading, GPUs offer massive amounts of on-chip parallelism. However, much of the raw computational power of GPUs today, remain unrealized. Consider the recent release of the top500 list, where three of the top five systems are configured with GPUs. Yet, none of these three systems achieve more than 55% of peak floating-point computation, falling significantly short of the 67% average for the entire list [4]. Furthermore, with the exception of very regular streaming algorithms, successful porting of HPC applications to GPUs has required significant amount of manual tuning [22,28,17]. Thus, harnessing the computation power of these emerging platforms merits the need for software tools that can reason about program behavior, automatically restructure code to yield better performance and reduce programmer effort.

* This research was supported by IBM through a Faculty Award and by Nvidia Corportation through an equipment grant.

M. O'Boyle (Ed.): CC 2012, LNCS 7210, pp. 21–40, 2012.
© Springer-Verlag Berlin Heidelberg 2012

A key consideration in producing high-performing GPU code is effectively managing both the thread and memory hierarchies. In the CUDA programming model, threads are grouped together in *blocks* and thread blocks are grouped into *grids*. Thus, the programmer needs to deal with multi-level parallelism by ensuring that not only the individual threads but the thread blocks can execute in parallel. On the other hand, effectively utilizing the memory hierarchy on the GPU is challenging because it is structured differently than its CPU counterpart and is divided into various subspaces including global, local, constant, cache, shared and texture memory. The element that further complicates the issue is the sharing of memory resources at various levels in the thread hierarchy. For example, unlike CPUs, the total number of registers is divided among threads in the same thread block. Similarly, although each thread block has access to its own shared memory, global memory is shared by all thread blocks in a grid. Thus, the placement and reuse of data (both intra and inter thread reuse) at various levels of the memory hierarchy and the decomposition of tasks are interrelated and needs to be carefully orchestrated by the programmer to obtain the desired performance.

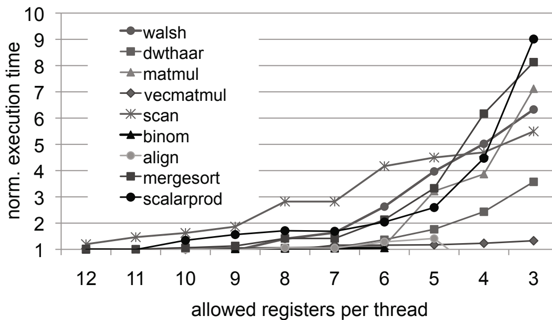


Fig. 1. Performance sensitivity to register pressure of selected CUDA SDK kernels

This work focuses on the register and shared-memory subspaces of the memory hierarchy and proposes a software-based strategy that enables automatic adjustment of thread granularity for exploiting inter-thread data locality at the shared memory level. Although current GPU platforms provide a large number of registers per block [5], it has been shown that for some kernels, ineffective use of the register space can cause significant loss in performance [24]. To understand how *register pressure* (the ratio between required and available registers) can affect performance, we conducted a simple experiment with a select set of kernels from the CUDA SDK [2]. Since the number of required registers in a thread cannot be modified arbitrarily, we used the *maxregcount* flag in NVIDIA's `nvcc` compiler to control the number of allocated registers, and thereby the register pressure, in each kernel. Fig. 1 shows normalized execution times for nine kernels as the register pressure is increased progressively by decreasing the available registers. We observe that except for matrix-vector multiply (`vecmatmul`), the

performance of all other kernels are significantly affected by changes in register pressure. Closer inspection revealed that most of this performance loss was to due additional accesses to local or shared memory which could have been curbed through better register reuse. One of the main factors that determines register pressure is thread granularity. Generally, a coarser granularity implies more work is done per thread with a higher demand for registers, while a finer granularity indicates lower demands. However, there are two factors that complicate this relationship between register pressure and thread granularity. First, if sibling threads in a thread block exhibit data locality either at the shared or global memory levels, then this locality can potentially be exploited at the register level by fusing multiple threads and increasing granularity. Although this will increase register pressure, the saved memory references can lead to improved performance. The second complicating issue is that thread granularity is also intimately tied with occupancy. Since GPU codes generally tend to perform better with higher occupancy, when increasing granularity for improved register reuse, care must be taken to not reduce occupancy to the point where it hurts performance. Therefore, automatic thread coarsening requires careful consideration of these two factors, and striking the right balance between occupancy and register reuse. Our proposed approach aims to achieve this goal.

Our strategy supports both automatic and semi-automatic methods of thread coarsening. In the semi-automatic mode, the programmer specifies the coarsening factor as a source-level pragma, whereas in the automatic method thread granularity is determined using a compiler heuristic. In both cases, we verify the legality of the coarsening transformation. To this end, we develop a generalized dependence analysis framework that allows fast analysis of CUDA programs. The analysis is based on the observation that the CUDA programming model enforces certain restrictions on thread execution, which are not enforced on mutlithreaded code for CPUs. For example, only barrier synchronization is supported among threads within a given thread block while no synchronization is allowed *between* thread blocks. These constraints allow for the construction of a simpler dependence framework and enables us to verify the legality of certain code transformations that would not be feasible on CPUs. Although, applied only to thread coarsening in this paper, the dependence framework has wider applicability and can be used to determine legality and profitability of a range of transformations including loop fusion and fission, unroll-and-jam and loop interchange. To facilitate fully automatic thread coarsening, we analyze the threads to detect the presence of inter-thread data locality at the shared memory level and then apply a heuristic that uses estimates of thread occupancy and register pressure as constraints to select a profitable thread granularity. Thus, the main contributions of this paper are

- a framework to provide compile-time support for both automatic and semi-automatic thread coarsening of CUDA kernels
- a generalized dependence analyzer for fast implementation of compiler optimizations of GPU codes

- an analytical model to estimate register pressure in kernels, on a per-thread basis

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 provides an example, illustrating the coarsening transformation. Section 4 describes our dependence analysis framework, the thread coarsening algorithm and the analytical model for estimating register pressure. Section 5 presents an overview of our optimization framework. Preliminary experimental results appear in Section 6 and finally we conclude in Section 7.

2 Related Work

Because general-purpose computing on GPUs is a fairly new idea and the technology is still maturing, much of the software-based performance improvement strategies have been limited to manual optimization. Ryoo et al. [23] present a general framework for optimizing applications on GPUs. Their proposed strategies include utilizing many threads to hide latency, and using local memories to alleviate pressure on global memory bandwidth. Govindaraju et al. develop new FFT algorithms for the GPUs and hand optimize the kernels to achieve impressive performance gains over the CPU-based implementation [17]. The key transformation used in their work was the combining of transpose operations with FFT computation. Demmel and Volkov [24] manually optimize the matrix multiplication kernel and produce a variant that is 60% faster than the autotuned version in CUBLAS 1.1. Among the optimization strategies discussed in this work are the use of shorter vectors at program level and the utilization of the register file as the primary on chip storage space.

There has been some work in combining automatic and semi-automatic tuning approaches with GPU code optimization. Murthy et al. have developed a semi-automatic, compile time approach for identifying suitable unroll factors for selected loops in GPU programs [15]. The framework statically estimates execution cycle count of a given CUDA loop, and uses the information to select optimal unroll factors. Liu et al. [27] propose a GPU adaptive optimization framework (GADAPT) for automatic prediction of near-optimal configuration of parameters that affect GPU performance. They take unoptimized CUDA code as input and traverse an optimization search space to determine optimal parameters to transform the unoptimized input CUDA code into optimized CUDA code. Choi et al. present a model-driven framework for automated performance tuning of sparse matrix-vector multiply (SpMV) on systems accelerated by GPU [12]. Their framework yields huge speedups for SpMV for the class of matrices with dense block substructure, such as those arising in finite element method applications. Williams et al. have also applied model-based autotuning techniques to sparse matrix computation that have yielded significant performance gains over CPU-based autotuned kernels [25]. Nukada and Matsuoko also provide a highly-optimized 3D-FFT kernel [21]. Work on autotuning general applications on the GPU is somewhat limited. Govindaraju et al. propose autotuning techniques for improving memory performance for some scientific applications [16]

and Datta et al. apply autotuning to optimize stencil kernels for the GPU [14]. The MAGMA project has focused on autotuning dense linear algebra kernels for the GPU, successfully transcending the ATLAS model to achieve as much as a factor of 20 speedup on some kernels [20]. Grauer-Gray and Cavazos present an autotuning strategy for utilizing the register and shared memory space for belief propagation algorithms [18].

Automatic approaches to code transformation has been mainly focused on automatically translating C code to efficient parallel CUDA kernels. Baskaran et al. present an automatic code transformation system (PLUTO) that generates parallel CUDA code from sequential C code, for programs with affine references [8]. The performance of the automatically generated CUDA code is close to hand-optimized CUDA code and considerably better than the benchmarks' performance on a multicore CPU. Lee et al. [19] take a similar approach and develop a compiler framework for automatic translation from OpenMP to CUDA. The system handles both regular and irregular programs, parallelized using OpenMP primitives. Work sharing constructs in OpenMP are translated into distribution of work across threads in CUDA. However the system does not optimize data access costs for access in global memory and also does not make use of on-chip shared memory.

The work presented in this paper distinguishes itself from earlier work in two ways. First the focus here is on automatic compiler methods rather than manual optimization techniques. Second, the approach supports direct optimization of CUDA source rather than C or OpenMP variants. We do not claim that this approach is superior to the approaches proposed previously but rather that framework can be used in conjunction with many of the strategies mentioned in this section.

3 An Example

In this section, we use a simple example to illustrate the interaction between data locality and thread granularity and discuss its performance implications. Consider the CUDA kernel shown in Fig. 2(a). In this kernel, all threads are organized in *one* single-dimensional thread block and each thread computes one element of array A, based on elements in B and C. Arrays A and B are allocated in global memory whereas C is allocated in shared memory. Because the computation is based on neighboring elements in B and C, the kernel exhibits temporal reuse in both shared and global memory. Values in B and C, accessed in thread i are reused by thread $i + 1$. Although the register pool on a multiprocessor is shared among threads in a warp, the distribution of registers occurs before thread execution and hence, during execution a thread cannot access registers that belong to a co-running thread. This implies that inter-thread data reuse will remain unexploited in the version of code shown in Fig. 2(a).

In Fig. 2(b), the kernel is transformed to perform two updates per thread and is invoked with half as many threads as the original version, thereby increasing thread granularity. This variant converts inter-thread reuse of data elements in

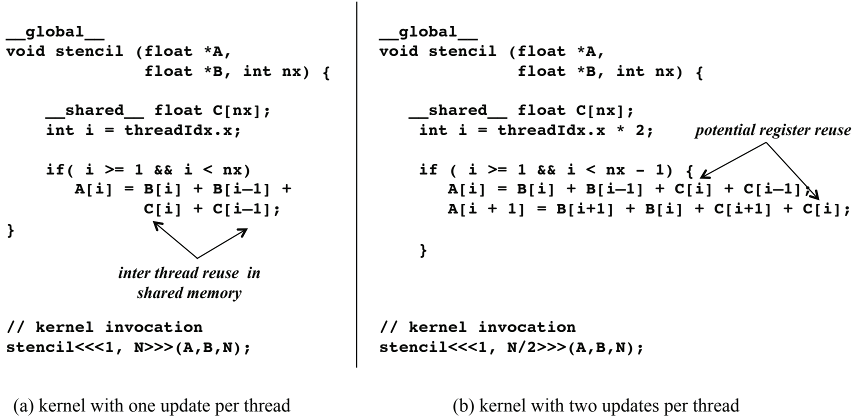


Fig. 2. Inter thread locality in stencil computation

B and C, into *intra*-thread reuse, allowing the compiler to allocate the values $B[i]$ and $C[i]$ into registers of thread i, leading to better register reuse[1]. Thus, the coarsening transformation shown in Fig. 2(b) can potentially reduce shared memory traffic by 25%. In the CUDA programming model, however, employing a fewer number of threads for the same computation generally implies a lower warp occupancy. For the kernel in Fig. 2, if we assume 8 registers per thread and 512 threads per block, then for a GPU with CC 2.0, we would have a 100% occupancy. The occupancy remains at a 100% when we reduce thread count to 256. However, it falls to just 67% when the thread count is reduced to 128. Thus, for this kernel, executing two updates per thread will almost inevitably lead to performance gains but any further coarsening will have to be weighed against the cost of the reduced occupancy.

The other performance consideration in this context is register pressure. Increasing thread granularity can potentially increase the number of required registers. This not only can lead to spills and cause more accesses to global memory but also have an impact on occupancy. If we assume a register count increase of two per coarsening factor for the example kernel then for a factor of 12, the number of required registers is 32, which drops the occupancy down to 33%. Therefore, all three factors : inter-thread data locality, register pressure and occupancy, need to be considered for profitable thread coarsening.

4 Automatic Thread Coarsening

In this section, we describe the main code transformation in our proposed approach and discuss the analyses needed to apply this transformation safely and profitably.

[1] Our experience showed that `nvcc` does not always perform scalar replacement and hence we included this transformation in our framework

4.1 Notation and Terminology

We introduce the following notation and terminology to describe our transformation framework

N	number of *simple* high-level statements in kernel
T	number of threads in thread block
s_i	i^{th} statement in kernel
$synch_i$	`syncthreads` primitive, appearing as i^{th} statement in kernel
$s_i \succ s_j$	data dependence from s_i to s_j
$S_{(i,p)}$	a simple statement *instance*: i^{th} statement in kernel, executed by p^{th} thread
$S_{(i,p)} \succ S_{(j,q)}$	a dependence between statement instances $S_{(i,p)}$ and $S_{(j,q)}$, where $S_{(i,p)}$ is the *source* and $S_{(j,q)}$ is the *sink* of the dependence
C_x	coarsening factor along dimension x

4.2 Dependence Analysis

The goal of our dependence analysis framework is to determine if there is a dependence between two statement *instances*. To achieve this, we augment a *data dependence graph* (DDG) to represent thread and statement instances and maintain data access information across execution of multiple threads. We assume that the input to our framework is a CUDA program that is legally parallelized for execution on a GPU. This assumption simplifies some of the analysis since it implies, that in the absence of `syncthreads()` primitives (i.e., barrier synchronization), statement instances that belong to different threads (or threads blocks) are dependence-free. Thus, given this framework we can make the following claim:

$$\textit{if } \nexists \textit{ syncthreads() primitives in the kernel body then}$$
$$\nexists S_{(i,p)} \succ S_{(j,q)},\ \forall i,j \in 1,...,N \textit{ and } \forall p,q \in 1,...,T$$

To detect and estimate inter-thread data locality, the analyzer needs to consider *read-read* reuse of data, which may or may not occur between statement instances, regardless of parallel configuration. For this reason, we extend our dependence framework to handle input dependences. Our framework handles the following two cases of dependence between two statement instances:

(i) $\exists\ S_{(i,p)} \succ S_{(j,q)}$, *iff* $S_{(i,p)}$ *and* $S_{(j,q)}$ *access the same memory location*
(ii) $\exists\ S_{(i,p)} \succ S_{(j,q)}$, *iff* $\exists\ synch_k$ *such that* $i < k < j$

Conventional dependence analysis [7] can be applied to statements within the body of a kernel to determine if statements access the same memory location. For CUDA kernels, one issue that complicates the analysis is that memory accesses

can be dependent on the value of *thread ID*. Hence, although a subscript analysis of the source may show two statements as accessing the same memory location, in reality they would access different memory locations. To handle this situation we take the following strategy : we first identify all statements in the kernel that are dependent on thread ID values; expand index expressions to replace subscripts with thread ID values (using scalar renaming [13]) and then apply the subscript test on the expanded expressions. Once all data dependences have been identified, our dependence analyzer makes another pass through the DDG to identify dependences that arise from the presence of `syncthreads()`. This final pass mainly involves checking for the existence of condition (ii), mentioned above.

4.3 Safety Analysis

For simplicity, we only describe the analysis necessary to safely apply thread coarsening along the innermost dimension, x. The same principles can be adopted, in a relatively straightforward manner, for coarsening along the y dimension and also for increasing thread block granularity (i.e., fusing two thread blocks).

Two factors determine the legality of the coarsening transformation. One is the relationship between the coarsening factor C_x and the number of threads in the original kernel T and the other is the presence of coarsening preventing dependences. For coarsening to be legal, there have to be enough threads available in the original configuration to satisfy the coarsening factor. If the coarsening factor is larger than the original thread count then extra work will be performed in each thread, violating the semantics. Also, when C_x does not evenly divide T, special handling of the remaining threads is necessary, which complicates the transformation and is likely to have a negative impact on overall performance. For this reason, we enforce the constraint that C_x evenly divides T, for coarsening to be legal. Thus, the first legality constraint for coarsening is as follows:

$$T \; mod \; C_x = 0 \tag{1}$$

A dependence between two statement instances will cause coarsening to be illegal, if as a result of coarsening, the direction of the dependence is reversed. We refer to such a dependence, as a *coarsening preventing dependence (cpd)* and derive the following conditions under which a *cpd* will *not* occur when the coarsening factor is C_x.

$$\nexists \; S_{(i,p)} \succ S_{(j,p-q)}, \; where \; i, j \in 1, ..., N, \; p \in \{C_x + 1, \ldots, T\}, \; q \in \{1, \ldots, C_x\} \tag{2}$$

or

$$\forall S_{(i,p)} \succ S_{(j,p-q)}, \; where \; i, j \in 1, ..., N, \;\; p \in \{C_x + 1, \ldots, T\}, \; q \in \{1, \ldots, C_x\}$$

$$\nexists \; S_{(k,p)} \succ S_{(j,p-q)}, \; where \; k \in \{j + 1, \ldots, N\} \tag{3}$$

Constraint (2) describes the situation where we have no dependence between statement instances within the coarsening range. Note, we are only concerned

about dependences that emanate from a *higher* numbered thread. For the coarsening transformation, dependences that emanate from a *lower* numbered thread is irrelevant, since, by default, all statement instances in p get executed after the last statement in q, where $p > q$. Thus, all such dependences will be preserved automatically. Constraint (3) considers the case where there is a dependence within the coarsening range but we can avoid violating this dependence if, in the merged thread body, we can move the source statement instance above the sink of the dependence. In our framework, we use a variant of the partial redundancy elimination (PRE) algorithm [9] that performs this task, once threads have been coarsened.

4.4 Profitability Analysis

There are two main profitability considerations for the thread coarsening transformation. First, we need to ensure that there is enough inter-thread data locality in the kernel to make coarsening worthwhile. Second, we need to determine if coarsening will cause an excessive increase in register pressure.

Detecting Inter-thread Locality. A CUDA kernel exhibits inter-thread data locality if two threads in the same thread block access the same location, either in shared memory or global memory. Generally, scalars are allocated to registers within each thread and hence coarsening does not help with reuse of such values. Thus, we focus on array references which are typically *not* allocated to registers by the compiler. Also, on the Fermi chip, it is possible for two threads to access two memory locations that map to the same cache line. However, we do not explicitly consider the case of spatial reuse in this paper.

Given this framework, an array reference in the kernel can only exhibit either *self-temporal* or *group-temporal* inter-thread data reuse. Self-temporal reuse can only occur if no subscript in the array reference depends on any of the *thread ID* variables. If the subscripts are not dependent on thread ID variables it implies that for that reference, all threads in the thread block will access the same memory location. Thus, identifying self-temporal reuse is simply a matter of inspecting each array reference and determining if the subscript values are independent of thread ID values.

To compute group-temporal reuse we introduce the notion of thread independent dependence. There is a thread independent dependence between two references if it can be established that there is a dependence between the two references when the entire kernel executes sequentially. The advantage of using thread independent dependences is that their existence can be determined by using conventional dependence tests. Once group-temporal reuse has been established between two references M_1 and M_2 in a thread independent way, we determine if the locality translates to inter-thread locality when the task is decomposed into threads. For inter-thread reuse to exist, at least one subscript in either reference has to be dependent on the thread ID value. This implies that

Algorithm 1. Estimating Register Pressure

```
repeat
  read each node in AST
  if node is of type variable declaration then
    add the variable name to the global array declared for storing all distinct variables in kernel

    if variable name and data type match any of the kernel input variable then
      set the scope as global
    end if
    if variable name is preceded by __shared__ keyword then
      set scope as shared else set the scope as register
    end if
  else if node type is assignment then
    if RHS has only one term and that term is a variable then
      save the index of RHS variable with LHS variable
    end if
  else if statement type is comparison then
    for all variables present in the statement increment read count
  end if
until end of AST
```

although M_1 and M_2 access the same memory location, the access may occur from two different threads. We formally, define the presence of inter-thread reuse as follows

There is inter-thread data reuse in kernel K if

(i) *there exists an array reference A with subscripts $i_0, \ldots, i_n$ in K such that no $i \in \{i_0, \ldots, i_n\}$ is a function of the thread ID value*
(ii) *there exists thread independent dependence between array reference M_1 and M_2, and at least one subscript in M_1 or M_2 is an affine function of the thread ID*

Estimating Register Pressure. Both the PTX analyzer [1] and the CUDA profiler [5] can provide fairly accurate per-thread register utilization information. Nevertheless, because we apply the code transformations on CUDA source, we require a method to estimate register pressure at the source level. To this end, we developed a register pressure estimation algorithm based on the strategy proposed by Carr and Kennedy [11]. An outline of our algorithm is given in Algorithm 1. Our strategy operates on the source code AST and the DDG built by our framework. The basic idea is to identify references that exhibit register level reuse. If there are multiple references with register level reuse then we predict that the compiler will coalesce them into a single register. One limitation of this approach is that it does not consider the possibility of any other transformation being applied to the kernel other than thread coarsening. We plan to extend this model in future to address this issue.

4.5 Code Transformation

The goal of the coarsening transformation is to restructure the kernel to perform more *work* in each thread. In essence, for a coarsening factor CF ($CF > 1$), we

```
__global__ void kernel1(int *in) {

   __shared__ float as[];
   . . . .
   sum = as[threadIdx.x] + 1;
   . . . .
}
```

(a) before

```
__global__ void kernel1(int *in){
 __shared__ float as[];
 int __i = threadIdx.x;
 for(int _k=0;_k<CF;
     _k++,__i+=(BS/CF)){
    sum = as[__i] + 1;
    . . . .
  }
}
```

(b) after

Fig. 3. Simple Thread Coarsening

want thread i to execute statements in thread $(i+1)$ through $(i+CF-1)$. This can be achieved by introducing a loop in the kernel body that iterates CF times. Of course, the main challenge is in determining what statements are included in the body of the loop and how the memory references need to be adjusted to affect the change. Fig. 3 shows parts of a CUDA kernel in its original and thread coarsened form. Here, BS represents block size, while CF represents the coarsening factor. In the coarsened version, a loop is added around the core computation to execute CF times. The variable $__i$ is used to store the value of the current thread id. This variable is incremented by BS/CF during each iteration of the loop, ensuring that the correct location in array *as* is accessed in each iteration.

As mentioned in Section 4.3, the presence of `__syncthreads()`, which acts as a barrier synchronization, complicates the coarsening transformation. We identify three separate cases related to `__syncthreads()` that need to be handled by our algorithm:

(i) `__syncthreads()` *is not present in the kernel:* This is the simple case that corresponds to constraint (2), derived in Section 4.3. This implies there are no dependences between statement in different threads. Therefore, in this case, we only need to insert the loop and adjust the memory references (as shown in Fig. 3).

(ii) `__syncthreads()` *is present but is not control-dependent on any loop:* Fig. 4 depicts the scenario where a `__syncthreads()` primitive is present in the kernel but the primitive does not appear inside a loop. In this case, to increase thread granularity, we can insert the loop and then *distribute* it around the `__syncthreads()` statement. The distribution ensures that the barrier synchronization is preserved, as it forces all statements controlled by the synchronization directive to execute *before* the `__syncthreads()` statement. The value of $__i$ needs to be reinitialized at the point of distribution to ensure correct memory reference by statements in the second loop.

The case where there is a dependence from the part of the code above `__syncthreads()` to the part following it and the value in the dependence

```
__global__ void kernel2(int *in)
{
as[threadIdx.x] = in[index];
__syncthreads ();
sum = as[threadIdx.x] +
      as[threadIdx.x+2 ];
. . . .
}
```

(a) before

```
__global__ void kernel2(int *in){
 int __i = threadIdx.x;
 for(int _k = 0; _k < CF;
    _k++,__i+=(BS/CF)){
    as[__i] = in[index];
 }
 __syncthreads ();
 __i = threadIdx.x;
 for(int _k = 0; _k < CF;
      _k++,__i+=(BS/CF)){
    sum = as[__i] + as[__i+2]
    . . . .
  }
}
```

(b) after

Fig. 4. Thread Coarsening in the presence of `syncthreads()`

is a function of thread ID, needs special handling. In this situation, while distributing the loop we need to change the scalar variable to an array with size equal to CF. This allows for all values computed in the top loop to be saved and thus preserves the dependence across the two loops.

(iii) `__syncthreads()` *is present and is control-dependent on some loop:* If a synchronization primitive appears inside a loop in the kernel then loop distribution results in an illegal transformation. In such a case, we perform (implicitly) an unroll of the loop by the coarsening factor. Fig. 5 illustrates this transformation.

```
__global__ void kernel3 (float* A) {

   __shared__ float as[] , sum;
   . . . .
   for (int i = 0; i <size; i++) {
       int v1 = as[threadIdx.x] ;
       __syncthreads();
       sum += as[i];
   }
  . . . .
}
```

(a) before

```
__global__ void kernel3 (float* A){

__shared__ float as[],sum;
. . . .
for (int i = 0; i <size; i++){
 int v1 = as[threadIdx.x] ;
 int v2 = as[threadIdx.x+(BS/CF)];
 __syncthreads();
 sum += as[i];
 }
. . . .
}
```

(b) after

Fig. 5. Thread Coarsening in the presence of `syncthreads()` in loops

Our coarsening algorithm accounts for all three cases mentioned above. The current implementation will detect the third case but the unrolling of the coarsening loop has to be performed by hand. The algorithm for thread coarsening is shown in Algorithm 2.

Algorithm 2. Thread Coarsening Transformation

```
get the first reference of the thread Id
declare a new variable (i) and assign thread Id value to it
add the variable declaration before the first reference of thread Id
replace all the occurrences of thread Id with this new variable (i)
from block size and coarsening factor calculate the increment value (inc) for thread Id
create a for loop with iterations equal to coarsening factor and increment the value of i by inc
with each iteration
add this for loop after the variable declaration i
find all the occurrence of __syncthreads() in the kernel and store in a list
if list is not empty then
   if none of the occurrence of __syncthreads() are inside a loop then
      repeat
         take reference of next occurrence of __syncthreads()
         limit the for loop body before this reference
         after the reference of __syncthreads() reinitialize variable i to thread Id value
         start a new for loop after reinitialization of i variable
         check for occurrences of __syncthreads() in rest of the code
      until end of list
   else
      return from function with error signal
   end if
else
   expand loop body till last statement of kernel and return
end if
```

5 CREST : CUDA Kernel Restructuring and AutoTuning

Fig. 6 gives an overview of CREST, our code restructuring and tuning framework for CUDA kernels. The framework leverages several existing tools including `nvcc` for code generation, HPCToolkit [6] and PAPI [10] for collection of performance metrics, and PSEAT [26] for online search algorithms. The rest of the components have been developed from scratch.

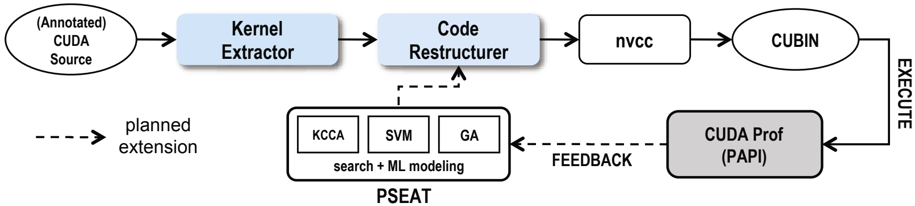


Fig. 6. Overview of CREST

5.1 Kernel Extraction

To facilitate analysis, a stand alone Perl script is used to extract the kernel from the CUDA source file before parsing. This simplifies the parser by setting aside everything external to the kernel being analysed. The extraction is performed on a line by line basis, using Perl regular expressions to detect the kernel specific portion of the source file. The kernel is extracted into a separate file for further processing and everything else is held in temporary files for later reassembly.

The kernel extraction is designed to be independent of the succeeding phases, and could be used in any application where CUDA kernels are to be examined.

5.2 Code Restructurer

At the heart of the framework is a source-to-source code transformation tool that analyzes dependences in CUDA kernels and implements a range of optimizations, some of which, to our knowledge, are currently not supported by any source-to-source transformer or by `nvcc`. These transformations include unroll-and-jam, scalar replacement, multi-level loop fusion and distribution, loop interchange, and thread coarsening. More specialized transformations such as array padding for eliminating conflicts in cache (on Fermi) and shared memory banks, array contraction for orchestrating placement of data, and iteration space splicing for reducing conditionals in kernels are currently being included in the tool. In addition to providing support for high-level code transformations, the code restructurer also implements several GPU specific heuristics for profitable application of these transformations. For example, when applying tiling on the GPU, the framework selects a tile size (and shape) that aims to improve intra-thread locality and reduce the number of synchronizations.

Another useful feature of the code restructurer is its support for fine-grain control over optimizations through source code annotation. Fig. 7 shows example directives for thread coarsening and unroll-and-jam, embedded in the *3D Jacobi* kernel. A directive is simply a comment line that specifies a particular transformation and one or more optional parameter values. In the matrix transpose code, the directive specifies coarsening of threads within a block, along the x dimension by a factor of two, and unroll-and-jam of the outer loop by a factor of 4. These directives can be associated with any loop, data structure, or kernel, providing explicit control over the scope of the optimization. This level of fine-grain control over transformations is generally not available in compilers. `nvcc` exposes a few control knobs to the user (e.g., –maxregcount). For comprehensive tuning of CUDA kernels, however, this merely scratches the surface.

5.3 Autotuning Support

Autotuning of kernels and applications has emerged as a dominant strategy in the HPC domain. As architectures and applications grow in scale and complexity, the need for autotuning is likely to be even more pronounced. With this picture in mind, we have designed our framework with support for autotuning. Currently, `cudaprof` is used to collect performance feedback from CUDA kernel execution. This feedback is sent to a search engine, PSEAT, that implements a variety of online (genetic algorithm, simulated annealing), and offline (support vector machines, kernel component analysis) search methods. Our future plans include developing an interface with PAPI 4.2 to collect better diagnostic feedback to make the tuning process more efficient.

```
// pragma specifies action, dimension and factor

# coarsen X 2
__global__ void jacobi(float *un, float  *u, ) {

  unsigned int j = blockIdx.x * BLOCK_DIM + threadIdx.x;
  unsigned int k = blockIdx.y * BLOCK_DIM  + threadIdx.y;

# uj 4
  for (unsigned int i = 1; i <= N; i++)
     for (; j <= N; k++)
         for (; k <= N; k++)
            Un[i, j, k] = (U[i-1, j, k] + U[i+1, j, k]
                     + U[i, j-1, k] + U[i, j+1, k]
                     + U[i, j, k-1] + U[i, j, k+1] ) * c
}
```

Fig. 7. Source code directives in CREST

Table 1. Description of CUDA kernels used in experimental evaluation

Kernel	Description	Source
`stencil`	computes sum of neighbouring elements within a block	Hand coded
`reduce`	computes min, max, sum	SC CUDA Tutorial [3]
`lintext`	demonstrates use of textures bound to pitch linear memory	CUDA SDK 3.2
`surfacewrite`	demonstrates use of texture fetches in CUDA	CUDA SDK 3.2
`transpose`	performs transpose of a single-precision matrix	CUDA SDK 3.2

6 Experimental Results

Experimental Setup. All experiments are performed on a Tesla C2050 NVIDIA Fermi GPU. The card has a compute capability of 2.0 and consists of 448 cores divided among 14 multiprocessors. Number of 32-bit registers allocated to each multiprocessor is 32K, while the amount of shared memory available per block is 48K. All CUDA programs are compiled with `nvcc` version 3.2, and all C programs are compiled with GCC 4.1.2. CUDA kernels used in our experiments are described in Table 1.

Performance Potential. We first conduct an experiment to gauge the effectiveness of the proposed strategy under *ideal* circumstances. To this end, we construct a synthetic benchmark `synth` with a high degree of inter-thread reuse. `synth`, uses an array with $512K$ elements which is divided into 1024 blocks of size 512. Each thread running on a block, computes the sum of all 512 elements residing in the block which are stored in shared memory. Threads running on different blocks all access different elements.

We ran `synth` with varying coarsening factors from 2 to 16. Fig. 8 shows the overall performance, occupancy, and shared and global memory access for `synth` as the coarsening factor is varied. The best speedup of 8.5 is obtained at a

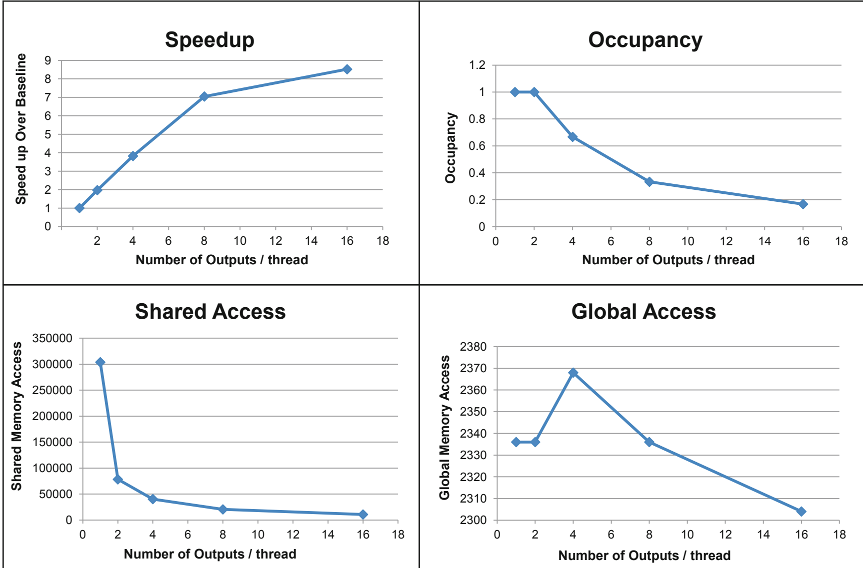


Fig. 8. Performance characteristics of `synth` for varying coarsening factors

coarsening factor of 16. This performance gain is directly attributed to reduced shared and global memory traffic. The shared memory traffic never increases as a result of increasing the thread granularity, however, there is a spike in global memory access when going from factor 2 to 4. This spike may be explained by increased register pressure. Our model estimates an increase of 8 registers per thread for each coarsening factor. Thus, the spike indicates a situation where the combination of register pressure and thread block size causes spills. Interestingly, the maximum performance is achieved at the lowest occupancy levels (16%), which emphasizes the need for considering factors other than occupancy when optimizing code for GPUs.

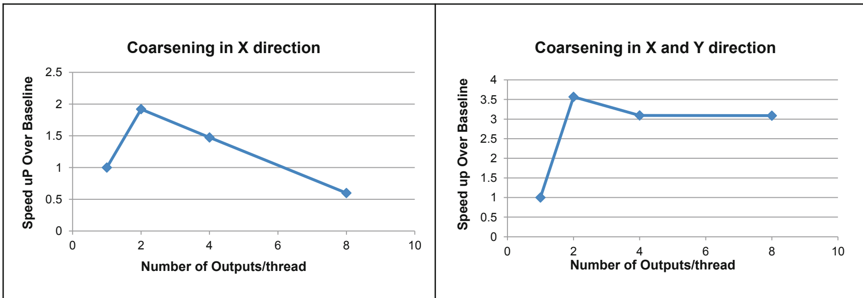


Fig. 9. Multi-dimensional coarsening with `transpose`

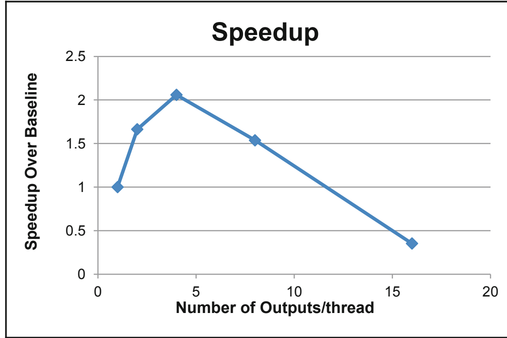


Fig. 10. Performance sensitivity to coarsening factors for `reduce`

Multi-dimensional Coarsening. We evaluate the effects of coarsening along multiple dimensions with `transpose`. The chart on the left in Fig. 9 shows performance of `transpose` for different coarsening factors along the X dimension, while the chart on the right depicts the performance when coarsening is done along both X and Y dimensions. Clearly, for `transpose`, it is more profitable to coarsen along both X and Y dimensions, which obtains a speedup of 3.5 over the baseline version. We also note that performance degrades (below the baseline) for larger coarsening factors when coarsening along the X dimension. This indicates that there is not enough inter-thread locality along this dimension to outweigh the costs of lower occupancy. Therefore, it is important to consider data locality in other dimensions when coarsening.

Performance Sensitivity. Although thread coarsening helps in improving performance, it is not guaranteed that performance will always increase. Fig. 10 shows the speedup observed for `reduce` with different coarsening factors. We observe that performance increases when the kernel is coarsened by factors of 2 and 4 but beyond that it starts decreasing rapidly, with factor 16 more than doubling the execution time. On further inspection, we found that this decrease in performance is mainly due to increase in global memory and shared memory accesses. This indicates that perhaps not enough registers were available to exploit the exposed data reuse. The other factor that contributed to the loss is the lower occupancy. For `reduce`, our model was able to rightly predict that the optimal coarsening factor is 4.

Overall Performance. Fig. 11 shows the speedup obtained for each kernel in our test suite, using our strategy. We only selected kernels that had some amount of inter-thread locality. Therefore, it is not too surprising that we observe performance improvement on all five kernels. The more interesting aspect of these results is that not all coarsening factors yielded good performance for all kernels. In fact, for some coarsening factors the performance degraded significantly. However, our proposed strategy was able to weed out the bad values and balance the cost of lower occupancy against the benefits of reduced memory traffic to pick suitable coarsening factors that resulted in significant performance gains.

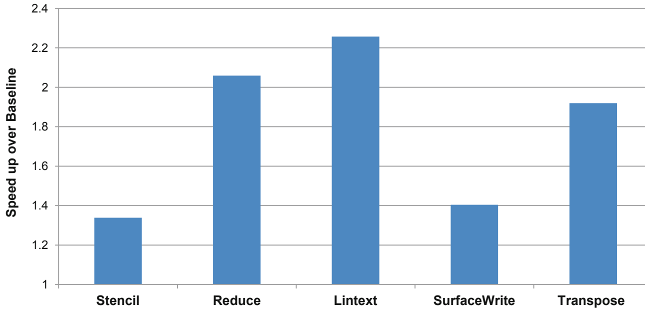


Fig. 11. Performance improvement from thread coarsening

7 Conclusions

This paper presented an automatic approach for controlling thread granularity in GPU kernels. We described the analysis required to apply the coarsening transformation, safely and profitably. The dependence analyzer presented in this paper can serve as a framework for implementing a range of memory hierarchy transformations on the GPU. The model for register pressure estimation can be used in developing compiler heuristics and also in the realm of autotuning. The experimental results suggest that the proposed approach can select suitable coarsening factors to improve register reuse, reduce shared memory traffic and increase overall performance for kernels that exhibit inter-thread data locality. These results are preliminary and more extensive experimentation is needed to evaluate the true effectiveness of the proposed method. Nevertheless, these results reiterate the need for an automatic strategy for controlling thread granularity for improved performance of GPU code.

Acknowledgement. We would like to thank the reviewers for helping us improve the quality of the final version of this paper. We also thank Dr. Martin Burtscher for allowing us compute time on his GPUs.

References

1. CUDA PTX ISA, http://www.nvidia.com/content/CUDAptxisa1.4.pdf
2. GPU Computing SDK, http://developer.nvidia.com
3. Kernel for min-max and reduction, http://supercomputingblog.com/cuda/cuda-tutorial-3-thread-communication/
4. Top 500 Supercomputer Sites, http://www.top500.org
5. CUDA Programming Guide, Version 3.0. NVIDIA (2010)
6. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: Hpctoolkit: tools for performance analysis of optimized parallel programs. Concurrency and Computation: Practice and Experience 22(6), 685–701 (2010)

7. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann (2002)
8. Baskaran, M.M., Ramanujam, J., Sadayappan, P.: Automatic C-to-CUDA Code Generation for Affine Programs. In: Gupta, R. (ed.) CC 2010. LNCS, vol. 6011, pp. 244–263. Springer, Heidelberg (2010)
9. Briggs, P., Cooper, K.D.: Effective partial redundancy elimination. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI 1994 (1994)
10. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A scalable cross-platform infrastructure for application performance tuning using hardware counters. In: ACM/IEEE 2000 Conference, Supercomputing (November 2000)
11. Carr, S., Kennedy, K.: Improving the ratio of memory operations to floating-point operations in loops. ACM Transactions on Programming Languages and Systems 16(6), 1768–1810 (1994)
12. Choi, J.W., Singh, A., Vuduc, R.W.: Model-driven autotuning of sparse matrix-vector multiply on GPUs. In: PPoPP 2010: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 115–126. ACM, New York (2010)
13. Cytron, R., Ferrante, J.: What's in a name? -or- the value of renaming for parallelism detection and storage allocation. In: ICPP 1987, pp. 19–27 (1987)
14. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: SC 2008: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, pp. 1–12. IEEE Press, Piscataway (2008)
15. Murthy, G., Ravishankar, M., Sadayappan, M.B., Optimal, P.: loop unrolling for gpgpu programs. In: IEEE International Symposium on Parallel Distributed Processing (2010)
16. Govindaraju, N.K., Larsen, S., Gray, J., Manocha, D.: A memory model for scientific algorithms on graphics processors. In: SC 2006: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, p. 89. ACM, New York (2006)
17. Govindaraju, N.K., Lloyd, B., Dotsenko, Y., Smith, B., Manferdelli, J.: High performance discrete fourier transforms on graphics processors. In: SC 2008: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, pp. 1–12. IEEE Press, Piscataway (2008)
18. Grauer-Gray, S., Cavazos, J.: Optimizing and Auto-tuning Belief Propagation on the GPU. In: Cooper, K., Mellor-Crummey, J., Sarkar, V. (eds.) LCPC 2010. LNCS, vol. 6548, pp. 121–135. Springer, Heidelberg (2011)
19. Lee, S., Min, S.J., Eigenmann, R.: OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2009)
20. Nath, R., Tomov, S., Dongarra, J.: Accelerating GPU Kernels for Dense Linear Algebra. In: Palma, J.M.L.M., Daydé, M., Marques, O., Lopes, J.C. (eds.) VECPAR 2010. LNCS, vol. 6449, pp. 83–92. Springer, Heidelberg (2011)
21. Nukada, A., Matsuoka, S.: Auto-tuning 3-d FFT library for CUDA GPUs. In: SC 2009: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pp. 1–10. ACM, New York (2009)

22. Rahimian, A., Lashuk, I., Veerapaneni, S., Chandramowlishwaran, A., Malhotra, D., Moon, L., Sampath, R., Shringarpure, A., Vetter, J., Vuduc, R., Zorin, D., Biros, G.: Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (2010)
23. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.M.W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2008)
24. Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune dense linear algebra. In: SC 2008: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (2008)
25. Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. Parallel Comput. 35(3), 178–194 (2009)
26. Yi, Q., Qasem, A.: Exploring the Optimization Space of Dense Linear Algebra Kernels. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 343–355. Springer, Heidelberg (2008)
27. Yixun, L., Zhang, E.Z., Shen, X.: A cross-input adaptive framework for GPU program optimizations. In: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing (2009)
28. Zhuo, Y., Wu, X.L., Haldar, J.P., Hwu, W.M., Liang, Z.P., Sutton, B.P.: Accelerating iterative field-compensated mr image reconstruction on GPUs. In: Proceedings of the 2010 IEEE International Conference on Biomedical Imaging: From Nano to Macro, ISBI 2010 (2010)

Programming Paradigm Driven Heap Analysis

Mark Marron[1], Ondřej Lhoták[2], and Anindya Banerjee[1]

[1] IMDEA Software Institute
{mark.marron,anindya.banerjee}@imdea.org
[2] University of Waterloo
olhotak@uwaterloo.ca

Abstract. The computational cost and precision of a shape style heap analysis is highly dependent on the way method calls are handled. This paper introduces a new approach to analyzing method calls that leverages the fundamental object-oriented programming concepts of encapsulation and invariants. The analysis consists of a novel partial context-sensitivity heuristic and a new take on *cutpoints* that, in practice, provide large improvements in interprocedural analysis performance while having minimal impacts on the precision of the results.

The interprocedural analysis has been implemented for .Net bytecode and an existing abstract heap model. Using this implementation we evaluate both the runtime cost and the precision of the results on a number of well known benchmarks and real-world programs. Our experimental evaluations show that, despite the use of partial context sensitivity heuristics, the static analysis is able to precisely approximate the ideal analysis results. Further, the results show that the interprocedural analysis heuristics and the approach to cutpoints used in this work are critical in enabling the analysis of large real-world programs, over 30K bytecodes in less than 65 seconds and using less than 130 MB of memory, and which could not be analyzed with previous approaches.

1 Introduction

Understanding the structure and evolution of the program heap is a critical part of understanding the semantics of a program, and thus is a central issue to optimizing, refactoring, or checking a program for errors. A central challenge when analyzing the memory behavior of a program is that memory structures are often long lived and can be passed through many method calls which may both use and modify them. Thus, using context-sensitivity when analyzing method calls is critical to the efficiency and accuracy of a static analysis of the program heap ([12, 17, 27, 30]). Context insensitive analysis approaches provide good computational performance. However they often result in overly approximate analysis results. Conversely fully context-sensitive analyses provide precise heap information but are computationally expensive and often intractable for all but trivial programs. Thus, there is interest in the development of heuristics that trade off small amounts of precision for large increases in analysis performance ([12, 17]). The general approach to analyzing a program in a context-sensitive manner involves building a memo table of the analysis input and outputs $(\widehat{h}_{in}, \widehat{h}_{out})$ for each method in the program. Whenever a method call is encountered the analysis checks if there is an appropriate match in the memo table. If there is a match it is used, otherwise a new entry

M. O'Boyle (Ed.): CC 2012, LNCS 7210, pp. 41–60, 2012.
© Springer-Verlag Berlin Heidelberg 2012

is added and the needed output model is computed. When analyzing a program in this manner it is critical to minimize the number of entries in these memo tables.

Approaches to minimizing the number of states in the memo tables include defining normal form constructions ([16], [31]) for the heap models, the introduction of cutpoints ([17], [26]), and partially context-sensitive call analysis heuristics ([12], [27]). However, the introduction of these heuristics (and simplifications) are often done in an ad-hoc manner and can be improved upon by explicitly considering how encapsulation, invariants, and pre/post conditions are used in the construction of object-oriented programs. While very few programs actually contain explicit (much less machine readable) annotations for these properties, empirical results ([1], [14]) suggest that in general object-oriented programs make heavy implicit use of these design concepts. This paper proceeds under the assumption that the program under analysis has been constructed using basic encapsulation and class invariant concepts but that these invariants (or pre/post conditions) are not made explicit in the code. If this assumption does not hold then the precision of the results will be decreased although they will still be sound.

Under the assumption that a program generally follows good object-oriented design principles there are a number of inferences that can be made about the state of the heap at the entry and exit of method calls. In particular at the entry of a method call every object that is visible to the callee should satisfy its class invariant and the reachable heap state should satisfy the pre-condition of the method. Similarly we know that at the return from the method all objects should again satisfy their class invariants and the caller should not depend on anything that is not ensured by the method post-condition. In an ideal world this would imply that context insensitive analyses would work well but in practice developers do not adhere to this strict use of isolation and uniformity in their programs. To allow for this practical reality we need to take a more relaxed approach to merging and simplifying the input/output models. One way to do this is to use the hypothesis that if a method has a simple implementation then the programmer is more likely to break encapsulation by *peering* into the implementation while for complex methods the programmer will make few assumptions beyond what is implied by the class invariants and pre/post conditions. We can use the call graph structure as a measure of how complex the called method is based on the number of subcalls and recursion.

Contributions. To address the challenges in analyzing real world object-oriented programs with a shape style heap domain this paper makes the following contributions:

- This paper introduces a hybrid call graph structure and abstract domain based context-sensitivity heuristic. This is motivated by features of object-oriented programming paradigms, class invariants, and encapsulation. It uses the structure of the call graph to heuristically estimate which calls are good candidates to apply partially sensitive merging operations and how aggressive the merging should be. This approach provides a framework for understanding why, and to what extent, we can expect the use of partial context-sensitivity to impact the precision of the analysis.
- To support the context-sensitive analysis we propose *unique fresh cutpoints* as an alternative definition for *cutpoints* ([17], [26]). These cutpoints allow us to project out the part of the abstract heap model that cannot be accessed by the callee method. In

contrast to previous work these cutpoints are always created with fresh names. This eliminates issues with spurious name conflicts and the creation of multiple memo table entries that differ due to naming of cutpoints, while still ensuring termination.

In order to quantify the performance and precision impacts of the interprocedural analysis technique we present an extensive experimental evaluation (Section 4) of well known benchmarks from SPEC JVM98 and DaCapo. The evaluation shows that the interprocedural analysis techniques proposed in this paper have a negligible impact on the precision of the analysis. The base domain used in this paper is capable of precisely expressing the majority of the connectivity, shape, and sharing properties that occur in practice and, despite the use of partial context-sensitivity, the static analysis is able to precisely (with a rate of 80-90%) approximate the ideal results. The approach in this paper enables the analysis of real world programs with an expressive shape style heap domain, requiring less than 65 seconds and 130 MB of memory for programs up to 30K bytecodes. Thus, this work provides a new framework, based on the principles of object-oriented program design, for thinking about interprocedural analysis techniques.

2 Abstract Heap Domain

For concreteness and to enable empirical evaluation it is useful to have a fixed abstract heap model. Thus, for this work we use the *Structural Analysis* domain from ([16]) which is briefly summarized in this section.

2.1 Concrete Heaps

The state of a concrete program is modeled in a standard way where there is an environment, mapping variables to addresses, and a store, mapping addresses to objects. We refer to an instance of an environment together with a store and a set of objects as a *concrete heap*. Given a program that defines a set of concrete types, Types, and a set of fields (and array indices), Labels, a concrete heap is a tuple $(\mathsf{Env}, \sigma, \mathsf{Ob})$ where:

$$\begin{aligned}
\mathsf{Env} \in \mathsf{Environment} &= \mathsf{Vars} \rightharpoonup \mathsf{Addresses} \\
\sigma \in \mathsf{Store} &= \mathsf{Addresses} \to \mathsf{Objects} \cup \{\mathsf{null}\} \\
\mathsf{Ob} &\in 2^{\mathsf{Objects}} \\
\mathsf{Objects} &= \mathsf{OID} \times \mathsf{Types} \times (\mathsf{Labels} \rightharpoonup \mathsf{Addresses}) \\
&\text{where the object identifier set } \mathsf{OID} = \mathbb{N}
\end{aligned}$$

Each object o in the set Ob is a tuple consisting of a unique identifier for the object, the type of the object, and a map from field labels to concrete addresses for the fields defined in the object. We use the notation $\mathsf{Ty}(o)$ to refer to the type of an object. The notation $o.l$ refers to the value of the field (or array index) l in the object. It is also useful to refer to a *non-null pointer* as a specific structure in a number of definitions. Therefore, we define a *non-null pointer* p associated with an object o and a label as l in a specific concrete heap, $(\mathsf{Env}, \sigma, \mathsf{Ob})$, as $p = (o, l, \sigma(o.l))$ where $\sigma(o.l) \neq \mathsf{null}$. We define a helper function $\mathsf{Fld} : \mathsf{Types} \mapsto 2^{\mathsf{Labels}}$ to get the set of all fields (or array indices) that are defined for a given type.

In the context of a specific concrete heap, $(\mathsf{Env}, \sigma, \mathsf{Ob})$, a *region* of memory is a subset of concrete heap objects $C \subseteq \mathsf{Ob}$. It is useful to define the set $P(C_1, C_2, \sigma)$ of all non-null pointers crossing from region C_1 to region C_2 as:

$$P(C_1, C_2, \sigma) = \{(o_s, l, \sigma(o_s.l)) \mid \exists o_s \in C_1, l \in \mathsf{Fld}(\mathsf{Ty}(o_s)) \,.\, \sigma(o_s.l) \in C_2\}$$

Injectivity. Given two regions C_1 and C_2 in the heap, $(\mathsf{Env}, \sigma, \mathsf{Ob})$, the non-null pointers with the label l from C_1 to C_2 are *injective*, written $\mathsf{inj}(C_1, C_2, l, \sigma)$, if for all pairs of non-null pointers (o_s, l, o_t) and (o'_s, l, o'_t) drawn from $P(C_1, C_2, \sigma)$, $o_s \neq o'_s \Rightarrow o_t \neq o'_t$. As a special case when we have an array object, we say the non-null pointer set $P(C_1, C_2, \sigma)$ is *array injective*, written, $\mathsf{inj}_{[]}(C_1, C_2, \sigma)$, if for all pairs of non-null pointers (o'_s, i, o_t) and (o_s, j, o'_t) drawn from $P(C_1, C_2, \sigma)$ and i, j valid array indices, $i \neq j \Rightarrow o_t \neq o'_t$.

Shape. We characterize the shape of regions of memory using standard graph theoretic notions of trees and general graphs treating the objects as vertices in a graph and the non-null pointers as defining the (labeled) edge set. We note that in this style of definition the set of graphs that are trees is a subset of the set of general graphs. Given a region C in the concrete heap $(\mathsf{Env}, \sigma, \mathsf{Ob})$:

- The predicate $\mathsf{any}(C)$ is true for any graph. We use it as the most general shape that doesn't satisfy a more restrictive property.
- The predicate $\mathsf{tree}(C)$ holds if the subgraph $(C, P(C, C, \sigma))$ is acyclic and does not contain any pointers that create cross edges.
- The predicate $\mathsf{none}(C)$ holds if the edge set in the subgraph is empty, $P(C, C, \sigma) = \emptyset$.

2.2 Abstract Heaps

An abstract heap is an instance of a storage shape graph ([3]). More precisely, an abstract heap graph is a tuple: $(\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}})$ where:

$$\begin{aligned}
\widehat{\mathsf{Env}} \in \text{Environments } &= \mathsf{Vars} \rightharpoonup \widehat{\mathsf{Addresses}} \\
\widehat{\sigma} \in \text{Stores } &= \widehat{\mathsf{Addresses}} \to \mathsf{Inj} \times 2^{\text{Nodes}} \\
&\quad \text{where the injectivity values } \mathsf{Inj} = \{\mathit{true}, \mathit{false}\} \\
\widehat{\mathsf{Ob}} \in \text{Heaps } &= 2^{\text{Nodes}} \\
\text{Nodes } &= \mathsf{NID} \times 2^{\mathsf{Types}} \times \mathsf{Sh} \times (\widehat{\mathsf{Labels}} \rightharpoonup \widehat{\mathsf{Addresses}}) \\
&\quad \text{where the shape values } \mathsf{Sh} = \{\mathsf{none}, \mathsf{tree}, \mathsf{any}\} \\
&\quad \text{and the node identifier set } \mathsf{NID} = \mathbb{N}
\end{aligned}$$

The abstract store $(\widehat{\sigma})$ maps from abstract addresses to tuples consisting of the injectivity associated with the abstract address and a set of target nodes. Each node n in the set $\widehat{\mathsf{Ob}}$ is a tuple consisting of a unique identifier for the node, a set of types, a shape tag, and a map from abstract labels to abstract addresses. The use of an infinite set of node identity tags, NID, allows for an unbounded number of nodes associated with a given type/allocation context allowing the local analysis to precisely represent freshly allocated objects for as long as they appear to be of special interest in the program ([16]). The abstract labels $(\widehat{\mathsf{Labels}})$ are the field labels and the special label $[]$. The special label

[] abstracts the indices of all array elements (i.e., array smashing). Otherwise an abstract label $\widehat{l}$ represents the object field with the given name.

As with the concrete objects we introduce the notation $\widehat{\mathsf{Ty}}(n)$ to refer to the type set associated with a node. The notation $\widehat{\mathsf{Sh}}(n)$ is used to refer to the shape property, and the usual $n.\widehat{l}$ notation to refer to the abstract value associated with the label $\widehat{l}$. Since the abstract store ($\widehat{\sigma}$) maps to tuples of *injectivity* and node target information we use the notation $\widehat{\mathsf{Inj}}(\widehat{\sigma}(\widehat{a}))$ to refer to the *injectivity* and $\widehat{\mathsf{Trgts}}(\widehat{\sigma}(\widehat{a}))$ to refer to the set of possible abstract node targets associated with the abstract address. We define the helper function $\widehat{\mathsf{Fld}} : 2^{\mathsf{Types}} \to 2^{\widehat{\mathsf{Labels}}}$ to refer to the set of all abstract labels that are defined for the types in a given set (including [] if the set contains an array type).

2.3 Abstraction Relation

We are now ready to formally relate the abstract heap graph to its concrete counterparts by specifying which heaps are in the concretization (γ) of an abstract heap:

$$(\mathsf{Env}, \sigma, \mathsf{Ob}) \in \gamma((\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}})) \Leftrightarrow \exists \text{ an embedding } \mu \text{ where}$$
$$\mathsf{Injective}(\mu, \mathsf{Env}, \sigma, \mathsf{Ob}, \widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}) \wedge \mathsf{Shape}(\mu, \mathsf{Env}, \sigma, \mathsf{Ob}, \widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}})$$

A concrete heap is an instance of an abstract heap, if there exists an embedding function $\mu : \mathsf{Ob} \to \widehat{\mathsf{Ob}}$ which respects the structure and labels of the concrete heap and also satisfies the injectivity and shape relations between the structures.

$$\mathsf{Injective}(\mu, \mathsf{Env}, \sigma, \mathsf{Ob}, \widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}) = \forall n_s, n_t \in \widehat{\mathsf{Ob}}, \widehat{l} \in \widehat{\mathsf{Fld}}(\widehat{\mathsf{Ty}}(n_s))\,.\,\widehat{\mathsf{Inj}}(\widehat{\sigma}(n_s.\widehat{l})) \Rightarrow$$
$$(\widehat{l} \neq [] \Rightarrow \mathsf{inj}(\mu^{-1}(n_s), \mu^{-1}(n_t), l, \sigma)) \wedge (\widehat{l} = [] \Rightarrow \mathsf{inj}_{[]}(\mu^{-1}(n_s), \mu^{-1}(n_t), \sigma))$$

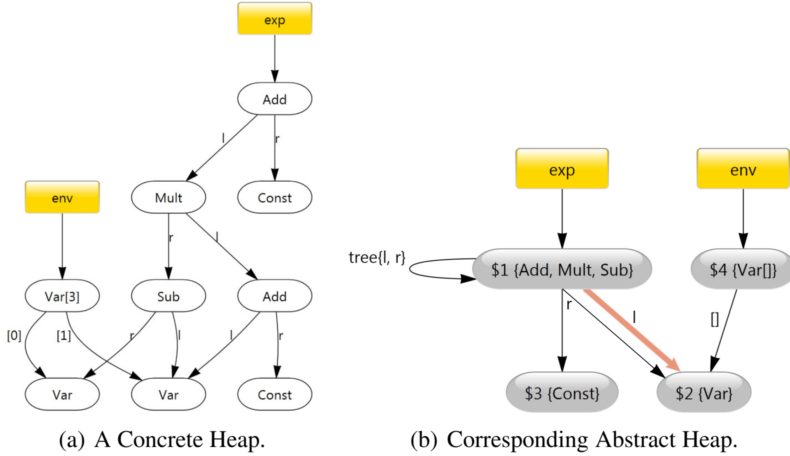
The injectivity relation guarantees that every pointer set marked as injective corresponds to injective (and array injective as needed) pointers between the concrete source and target regions of the heap.

$$\mathsf{Shape}(\mu, \mathsf{Env}, \sigma, \mathsf{Ob}, \widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}) = \forall n \in \widehat{\mathsf{Ob}}$$
$$\widehat{\mathsf{Sh}}(n) = \mathsf{tree} \Rightarrow \mathsf{tree}(\mu^{-1}(n, \sigma)) \wedge \widehat{\mathsf{Sh}}(n) = \mathsf{none} \Rightarrow \mathsf{none}(\mu^{-1}(n, \sigma))$$

The shape relation guarantees that for every node n, the concrete subgraph $\mu^{-1}(n, \sigma)$ abstracted by node n satisfies the corresponding concrete shape predicates.

2.4 Example Heap

Figure 1(a) shows a snapshot of the concrete heap from a simple program that manipulates expression trees. An expression tree consists of binary nodes for `Add`, `Sub`, and `Mult` expressions, and leaf nodes for `Constants` and `Variables`. The local variable `exp` (rectangular box) points to an expression tree consisting of 4 interior binary expression objects, 2 `Var`, and 2 `Const` objects. The local variable `env` points to an array representing an environment of `Var` objects that are shared with the expression tree.



(a) A Concrete Heap. (b) Corresponding Abstract Heap.

Fig. 1. Concrete and Abstract Heap

Figure 1(b) shows the corresponding abstract heap for this concrete heap. To ease discussion each node in a graph is labeled with a unique node id. The abstraction summarizes the concrete objects into three regions represented by the nodes in the abstract heap graph: (1) a node representing all interior recursive objects in the expression tree (`Add`, `Mult`, `Sub`), (2) a node representing the two `Var` objects, and (3) a node representing the two `Const` objects. The edges represent possible sets of non-null cross region pointers associated with the given abstract labels. Details about the order and branching structure of expression nodes are absent but other more general properties are still present. The label `tree{l,r}` on the self-edge expresses that pointers stored in the `l` and `r` fields of the objects represented by node 1 form a tree.

The abstract graph also captures the fact that no `Const` object is referenced from multiple expression objects but that several expression objects might point to the same `Var` object. The abstract graph shows this possible non-injectivity using wide orange colored edges (if color is available), whereas normal edges indicate injective pointers. Similarly the edge from node 4 (the `env` array) to the set of `Var` objects represented by node 2 is injective, not shaded and wide. This implies that there is no aliasing between the pointers stored in the array (a fact which could not be obtained via points-to analysis).

2.5 Normal Form

Given the definitions for the abstract heap it is clear that the domain is infinite. Thus, we define a normal form that ensures the number of distinct normal form graphs is finite and use this set during the fixpoint computation (see [16]) for additional information).

Definition 1 (Normal Form). *We say that the abstract heap is in normal form iff:*

1. *All nodes are reachable from a variable or static field.*
2. *All recursive structures are summarized (Definition 2).*
3. *All equivalent successors are summarized (Definition 4).*
4. *All variable/global equivalent targets are summarized (Definition 5).*

This normal form definition possesses three key properties that ensure finiteness: (1) the resulting abstract heap graph has a bounded depth, (2) each node has a bounded out degree, and (3) for each node the possible targets of the abstract addresses associated with it are unique wrt. the label and the types in the target nodes.

As each of the properties (*recursive structures*, *ambiguous successors*, and *ambiguous targets*) are defined in terms of, congruence between abstract nodes the transformation of an abstract heap into the corresponding normal form is fundamentally the computation of a congruence closure over the nodes in the abstract heap followed by merging the resulting equivalence sets. Thus, we build a map from the abstract nodes to equivalence sets (partitions) using a Tarjan union-find structure. Formally $\Pi : \widehat{\mathsf{Ob}} \to \{\pi_1, \ldots, \pi_k\}$ where $\pi_i \in 2^{\widehat{\mathsf{Ob}}}$ and $\{\pi_1, \ldots, \pi_k\}$ are a *partition* of $\widehat{\mathsf{Ob}}$.

Recursive Structures. The first step in computing the normal form is to identify any nodes that may be parts of unbounded depth structures. This is accomplished by examining the type system for the program that is under analysis and identifying all the types, τ_1 and τ_2, that have mutually recursive type definitions denoted: $\tau_1 \sim \tau_2$.

Definition 2 (Recursive Structure). *Given two partitions* π_1 *and* π_2 *we define the* recursive structure *congruence relation as*[1]*:*

$$\pi_1 \equiv_r^{\Pi} \pi_2 \Leftrightarrow \exists \tau_1 \in \bigcup_{n_1 \in \pi_1} \widehat{\mathsf{Ty}}(n_1), \tau_2 \in \bigcup_{n_2 \in \pi_2} \widehat{\mathsf{Ty}}(n_2) \,.\, \tau_1 \sim \tau_2$$
$$\wedge \exists n \in \pi_1, \widehat{l} \in \widehat{\mathsf{Fld}}(\widehat{\mathsf{Ty}}(n)) \,.\, \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n.\widehat{l})) \cap \pi_2 \neq \emptyset$$

Equivalent Successors and Targets. The other part of the normal form computation is to identify any partitions that have *equivalent successors* and variables that have *equivalent targets*. Both of these operations depend on the notion of a successor partition which is based on the underlying structure of the abstract heap graph and a general notion of node compatibility: π_1 a successor of π_2 and $\widehat{l} \Leftrightarrow \exists n_2 \in \pi_2 \,.\, \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n_2.\widehat{l})) \cap \pi_1 \neq \emptyset$.

Definition 3 (Partition Compatibility). *We define the relation Compatible(*π_1*,* π_2*) as:* $\mathrm{Compatible}(\pi_1, \pi_2) \Leftrightarrow \bigcup_{n' \in \pi_1} \widehat{\mathsf{Ty}}(n') \cap \bigcup_{n' \in \pi_2} \widehat{\mathsf{Ty}}(n') \neq \emptyset$.

Definition 4 (Equivalent Successors). *Given* π_1*,* π_2 *which are successors of* π *on labels* $\widehat{l_1}$*,* $\widehat{l_2}$ *we define the relation* π_1*,* π_2 equivalent successors *as:* $\pi_1 \equiv_s^{\Pi} \pi_2 \Leftrightarrow \widehat{l_1} = \widehat{l_2} \wedge \mathrm{Compatible}(\pi_1, \pi_2)$.

Definition 5 (Equivalent on Targets). *Given a root r (a variable or a static field) and two target partitions* π_1*,* π_2 *we define the* equivalent targets *relation as:* $\pi_1 \equiv_t^{\Pi} \pi_2 \Leftrightarrow \mathrm{Compatible}(\pi_1, \pi_2) \wedge (r \textit{ is a static field} \vee \pi_1, \pi_2 \textit{ only have local var predecessors})$.

Using the *recursive structure* relation and the *equivalent successor (target)* relations we can efficiently compute the congruence closure over an abstract heap producing the corresponding normal form abstract heap (Definition 2). This computation can be done via a standard worklist algorithm that merges partitions that contain equivalent nodes and can be done in $O((N+E) * log(N))$ time where N is the number of abstract nodes in the initial abstract heap, and E is the number of abstract addresses in the heap.

[1] The definition is symmetric on the properties of the nodes, the $\tau_1 \sim \tau_2$ equivalence relation on types, but is not symmetric on the structure of the underlying graph.

3 Interprocedural Analysis

In order to understand how aggressively abstract models can be merged during the analysis and on which methods the precision loss due to the use of partial context-sensitivity will be the smallest we look to the principles of object-oriented program design. Fundamentally, object-oriented programming is designed around the concepts of encapsulation and information hiding.

These concepts can be seen in the use of object invariants and pre/post conditions which allow a client to use a class with only a minimal knowledge of the internal functioning and allow an implementor to build a class that hides most of the internals from the outside world. This requires the caller to ensure the pre-condition at each method call site and allowing the method implementor to assume this condition. Conversely at call exit the implementor is responsible for ensuring that the post-condition is satisfied and the caller is then allowed to assume anything implied by the post-condition. These conditions are augmented by the class invariants which ensure/guarantee that at the method boundaries each object satisfies the class required invariant in addition to properties in the pre/post conditions of the method. Thus, in the ideal case from the standpoint of the program implementation and the static analysis we could assume that for any property P in the program:

1. If the callee depends on a property P holding as part of the pre-condition of the method (or a class invariant) then all callers to the method *must* ensure that P holds.
2. If the caller depends on a property P that is part of the post-condition (or is a class invariant) then every path through the method *must* ensure that P holds.
3. Alternatively a property P may hold on the portion of the heap that is only reachable from the caller but not reachable from the callee in which case the property P will remain unchanged across the call and will not affect the behavior of the callee.

In the context of the heap analysis the first condition (along with scope visibility) ensures that for every property that holds on the program state *accessible* to the callee that either it is part of the precondition and holds for every abstract model or that the callee makes no assumption on the property. Assuming the join operator ($\sqcup$, e.g. [16]) for the domain in Section 2) and the domain are sufficiently precise, the merging should have no impact on the accuracy of the analysis. For example if the callee assumes that an argument (say variable `x`) is always *non-null* then all calls to this method must occur in states where `x` is *non-null*. Thus, all abstract models should entail that `x` is *non-null* and the join of all these models should result in a model that also entails that `x` is *non-null*.

From the perspective of the caller the second condition (along with call scope visibility) ensures that for every property P of the program state after the call, either the property P is part of the post-condition and always holds at method exit or it involves parts of the model that were not accessible to the callee and the value of the property is assumed to be unchanged. In the first case, as the property holds for all models at return from the called method, the join at method return loses no information. Alternatively if the property does not involve part of the program state accessible to the callee then the joins will not affect it (since the use of a *project/extend* ensure these parts of the abstract model are not involved in the joins). The project and extend operations can be seen as the addition of a *frame rule* [25, 31] for the interprocedural analysis.

3.1 Unique Fresh Cutpoints

One simple and effective way to accomplish the splitting into a reachable and external heap section is to define a project and an extend operation based on *cutpoints* ([17, 26]). The cutpoints are special names introduced for abstract storage targets (edges) that cross from the caller only reachable portion of the abstract heap into the callee reachable portion of the abstract heap. However, as noted in previous work, using a fixed set of names leads to the creation of many redundant entries in the memo tables (that differ only in the names of the cutpoints) and inevitably results in the spurious reuse of the same cutpoint name for multiple cross abstract pointers ([17, 26]). However, simply creating a fresh name for each cutpoint that is created leads to non-termination in recursive calls due to the constant addition of new names. To avoid these issues we treat the cutpoints as more than syntactic constructs by defining notions of equivalence classes on them and by extending equality on cutpoints beyond a syntactic property.

To accomplish this we extend the abstract model with a mapping from an unbounded set of cutpoint names, $\mathsf{CPNames}$, to the abstract addresses $\widehat{\mathsf{CP}} \in \mathsf{CPEnvironment} = \mathsf{CPNames} \rightharpoonup \widehat{\mathsf{Addresses}}$, and we extend the definition of the abstract store to be $\widehat{\sigma}$: $\widehat{\mathsf{Addresses}} \to \{true, false\} \times (2^{\widehat{\mathsf{Ob}}} \cup \mathsf{CPNames})$. We use *fresh* names for each cutpoint we introduce but to avoid the termination issues we add them such that each abstract heap location has at most one cutpoint associated with it. Finally, we add an additional clause to the normal form computation in Definition 1 to handle the cutpoints. For equality matching we check for the existence of an isomorphism between the cutpoint name sets in the two graphs.

Definition 6 (Cross References and Cutpoints). *Given an abstract heap, $\widehat{h}_{\text{in}}$, to a method call and the set of abstract nodes reachable from the callee scope $C \subseteq \widehat{\mathsf{Ob}}$ then:*

- *An abstract location $\widehat{l}$ in node n is a cross pointer if $n \notin C \wedge \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n.\widehat{l})) \cap C \neq \emptyset$.*
- *An abstract location v is a cross variable if $\widehat{\mathsf{Trgts}}(\widehat{\sigma}(\widehat{\mathsf{Env}}(v))) \cap C \neq \emptyset$.*
- *A cutpoint c_p is redundant if $\widehat{\mathsf{Trgts}}(\widehat{\sigma}(\widehat{\mathsf{CP}}(c_p))) \cap C \neq \emptyset$.*

The project operation works by partitioning the set of nodes in the graph into two sets based on their reachability from the local variables. For each node in the reachable set that has a cross pointer from a location in the unreachable set, a cross variable, or a redundant cutpoint we add a fresh cutpoint which refers to the node. Next we associate this newly created cutpoint with all the cross references to the node (replacing the node target with the name of the freshly created cutpoint). The resulting abstract heap has at most one cutpoint which refers to any node and as each cutpoint name that is added is fresh we are ensured that there are no name collisions. We note that in our heap model there are no relational properties between the incoming pointers (cutpoints) to a node. In domains that do associate relational properties with pointers the same technique can be applied via a generalization to one cutpoint per *equivalence set* of cross pointers.

Additionally, as the particular domain we are using does not perform strong updates ([16]) we can obtain further speedups in the analysis by taking advantage of the fact that the input heap model is always an under-approximation of the output heap model. So in the project operation defined here we want to extract the callee reachable

portion of the heap and also preserve the entire heap structure (extended with the cutpoint information) in the remaining heap model. If the underlying domain performed strong updates to externally visible heap locations then we would need to, slightly, alter the definition to fully partition the heap structure as done in [17, 26].

$\texttt{project} : \overrightarrow{v}, (\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}, \widehat{\mathsf{CP}}) \rightsquigarrow (\widehat{\mathsf{Env}}_e, \widehat{\sigma}_e, \widehat{\mathsf{Ob}}_e, \widehat{\mathsf{CP}}_e), (\widehat{\mathsf{Env}}_r, \widehat{\sigma}_r, \widehat{\mathsf{Ob}}_r, \widehat{\mathsf{CP}}_r)$ where

$\widehat{\mathsf{Ob}}_{reach} = \{n \mid n \in \widehat{\mathsf{Ob}} \wedge n \text{ reachable from a callee variable in } \overrightarrow{v}\}$

$(\widehat{\mathsf{Ob}}_r, \widehat{\mathsf{Ob}}_e) = (\widehat{\mathsf{Ob}}_{reach}, \widehat{\mathsf{Ob}})$

$(\widehat{\mathsf{Env}}_r, \widehat{\mathsf{Env}}_e) = (\{[v \mapsto \widehat{a_v}] \mid v \in \overrightarrow{v}\}, \widehat{\mathsf{Env}})$

$(\widehat{\sigma}_r, \widehat{\sigma}_e) = (\{[\widehat{a} \mapsto \widehat{\sigma}(\widehat{a})] \mid \widehat{a} \text{ a reachable address from a callee variable}\}, \widehat{\sigma})$

$(\widehat{\mathsf{CP}}_r, \widehat{\mathsf{CP}}_e) = (\emptyset, \widehat{\mathsf{CP}})$

$\forall n_r \in \widehat{\mathsf{Ob}}_r$ let c_p be fresh cutpoint name for n

$\quad \forall n_e \in \widehat{\mathsf{Ob}}_e, \widehat{l} \in \widehat{\mathsf{Fld}}(\widehat{\mathsf{Ty}}(n_e))$ if $n_e, \widehat{l}$ is cross pointer then

$\quad\quad \widehat{\sigma}_e(n_e.\widehat{l}) = (\widehat{\mathsf{Inj}}(\widehat{\sigma}_e(n_e.\widehat{l})), \widehat{\mathsf{Trgts}}(\widehat{\sigma}_e(n_e.\widehat{l}) \cup \{c_p\})$

$\quad \forall v \in Dom(\widehat{\mathsf{Env}})$ if v is cross variable then

$\quad\quad \widehat{\sigma}_e(\widehat{\mathsf{Env}}_e(v)) = (true, \widehat{\mathsf{Trgts}}(\widehat{\sigma}_e(\widehat{\mathsf{Env}}_e(v))) \cup \{c_p\})$

$\quad \forall c'_p \in Dom(\widehat{\mathsf{CP}})$ if c'_p is redundant then

$\quad\quad \widehat{\sigma}_e(\widehat{\mathsf{CP}}_e(c'_p)) = (true, \widehat{\mathsf{Trgts}}(\widehat{\sigma}_e(\widehat{\mathsf{CP}}_e(c'_p))) \cup \{c_p\})$

$\quad \widehat{\mathsf{CP}}_r = \widehat{\mathsf{CP}}_r \cup \{[c_p \mapsto \widehat{a_{c_p}}]\}$ where $\widehat{a_{c_p}}$ a fresh address

$\quad \widehat{\sigma}_r = \widehat{\sigma} \cup \{[\widehat{a_{c_p}} \mapsto \{n\}]\}$

$\texttt{extend:}\ (\widehat{\mathsf{Env}}_e, \widehat{\sigma}_e, \widehat{\mathsf{Ob}}_e), (\widehat{\mathsf{Env}}_r, \widehat{\sigma}_r, \widehat{\mathsf{Ob}}_r) \rightsquigarrow (\widehat{\mathsf{Env}}', \widehat{\sigma}', \widehat{\mathsf{Ob}}')$ where

$\widehat{\mathsf{Ob}}' = \widehat{\mathsf{Ob}}_e \sqcup \widehat{\mathsf{Ob}}_r$

$\widehat{\mathsf{Env}}' = \widehat{\mathsf{Env}}_e + [v_{ret} \mapsto \widehat{\mathsf{Env}}_r(v_{ret})]$

$\widehat{\sigma}' = \widehat{\sigma}_e \sqcup \widehat{\sigma}_r$

$\forall c_p \in Dom(\widehat{\mathsf{CP}}_r), \widehat{a} \in Dom(\widehat{\sigma}')$

$\quad$ if $c_p \in \widehat{\mathsf{Trgts}}(\widehat{\sigma}'(\widehat{a}))$ then

$\quad\quad \widehat{\sigma}' = \widehat{\sigma}' + [\widehat{a} \mapsto (\widehat{\mathsf{Inj}}(\widehat{\sigma}'(\widehat{a})), \widehat{\mathsf{Trgts}}(\widehat{\sigma}'(\widehat{a})) \setminus \{c_p\} \cup \widehat{\sigma}_r(\widehat{\mathsf{CP}}_r(c_p)))]$

$\widehat{\mathsf{CP}}' = \widehat{\mathsf{CP}}_e$

After analyzing the callee method body we need to recombine the two partitions of the abstract heap. The extend operation handles this recombination. It proceeds by taking the union of the contents of the two abstract heaps (the callee reachable and only caller reachable partitions) and then replacing the cutpoint names inserted in the project operation with the needed values. Since we preserved the entire original heap structure in the project operation we cannot simply union the abstract nodes and environments (since the same node or address mapping may appear in both). Instead we must perform

a join on these structures by taking the union of any elements that appear in only one of the abstract heaps and performing a pairwise join on the properties ([16]) of any duplicate nodes or environment mappings.

In the normal form computation of the abstract heap graph, Section 2, the normal form is based on the construction of equivalence classes based on predecessor relations and the identification of recursive data structures. As the introduction of cutpoints adds a new source of possible predecessors we need to extend the normal form to deal with these as well. We note that cutpoints can be introduced in both calls to simple acyclic parts of the call-graph as well as to parts which may contain recursive calls.

In order to preserve as much information as possible we want to only merge nodes that have the same sets of cutpoints referring to them but in order to ensure termination we must ensure that this condition does not result the violation of conditions 2 through 4 in Definition 1. For example it would be possible for a recursive call to allocate a new node and add it to a list before making a recursive call with the list as an argument. This would result in a new node with a distinct cutpoint being created at each call and the abstract graph having no upper bound on its size. We note that the only way this can happen is with cutpoints added in recursive calls, any non-recursive path though the call graph has a finite length and thus only creates a finite number of cutpoints. Thus, we can strongly distinguish nodes in the graph based on cutpoints added in non-recursive calls and only weakly distinguish them based on cutpoints from recursive calls. Thus we update the *Compatible* relation Definition 3 to distinguish on cutpoints in addition to types, where $\mathit{NonRecCP}(\pi)$ returns the set of all cutpoints that refer to a node in the given partition and were introduced in a non-recursive call:

$$\mathit{Compatible}(\pi_1,\pi_2) \Leftrightarrow \bigcup_{n'\in\pi_1} \widehat{\mathsf{Ty}}(n') \cap \bigcup_{n'\in\pi_2} \widehat{\mathsf{Ty}}(n') \neq \emptyset \wedge \mathit{NonRecCP}(\pi_1) = \mathit{NonRecCP}(\pi_2)$$

With these definitions we can now split and rejoin the heap into the section that is relevant to a callee method and the part that is guaranteed to be unchanged across a method call as in ([17], [26]). The use of *unique fresh cutpoints* and associated constructions eliminates the problem of creating unneeded memo contexts or spurious name collisions via the use of fresh names. Further, the normal form and uniqueness requirements still ensure termination.

3.2 Context-Sensitivity Heuristics

As most programs deviate from the very strict uses of invariants and pre/post conditions described previously we want to selectively relax how aggressively we merge different call contexts based on how strongly a method adheres to the ideal version of information hiding within the implementation. A simple hypothesis is that the complexity of the called method will have a large impact on the amount of additional assumptions, beyond the invariant and pre/post conditions, that the programmer may depend on. For example simple leaf methods such as getters or setters can often be called in many unrelated states and have very broad pre/post conditions associated with them. Similar cases hold for other methods that are near the leaves in the call graph that can easily have their implementations examined or that may have very broad pre/post conditions. Conversely

methods that are more complex, higher in the call graph and particularly those that are part of large recursive call structures, are much more likely to have strict pre/post conditions and the callers of these methods generally avoid making additional assumptions about the behavior of the called method. This leads to the following heuristic for selecting which methods should be handled with full context-sensitivity and which methods could be handled in a partially context-sensitive manner.

Definition 7 (Context Compatibility). *A given abstract heap* $\widehat{h}_{\text{in}}$ *is* compatible *with a previously memoized input* $\widehat{h}_{\text{memo}}$ *for the call to the method m if:*

- *The method m is in an acyclic part of the call graph and* $\widehat{h}_{\text{in}} \simeq \widehat{h}_{\text{memo}}$.
- *The method m is in a cyclic part of the call graph and* $\widehat{h}_{\text{in}} \approx \widehat{h}_{\text{memo}}$.

The $\simeq$ operations is fundamentally an extension of the basic equality operation on the abstract heaps from [16] to include the cutpoint set names. Given two abstract heaps $(\widehat{\mathsf{Env}}_1, \widehat{\sigma}_1, \widehat{\mathsf{Ob}}_1 \widehat{\mathsf{CP}}_1)$ and $(\widehat{\mathsf{Env}}_2, \widehat{\sigma}_2, \widehat{\mathsf{Ob}}_2, \widehat{\mathsf{CP}}_2)$ we first determine if they are structurally isomorphic (i.e., if there is an isomorphism, ϕ on the graph structures that respects variable and field labels), then we check that all abstract node and store properties in $(\widehat{\mathsf{Env}}_2, \widehat{\sigma}_2, \widehat{\mathsf{Ob}}_2, \widehat{\mathsf{CP}}_2)$ have the same values in $(\widehat{\mathsf{Env}}_1, \widehat{\sigma}_1, \widehat{\mathsf{Ob}}_1, \widehat{\mathsf{CP}}_1)$ under the isomorphism. Finally, we check is there is an isomorphism ϕ_{cp} between the cutpoints that respects the reachability relations on the graphs.

$$
\begin{aligned}
&(\widehat{\mathsf{Env}}_1, \widehat{\sigma}_1, \widehat{\mathsf{Ob}}_1) \simeq (\widehat{\mathsf{Env}}_2, \widehat{\sigma}_2, \widehat{\mathsf{Ob}}_2) \Leftrightarrow \exists \phi, \phi_{cp} \\
&\qquad \forall n \in \widehat{\mathsf{Ob}}_1 \,.\, \widehat{\mathsf{Ty}}(n) = \widehat{\mathsf{Ty}}(\phi(n)) \wedge \widehat{\mathsf{Sh}}(n) = \widehat{\mathsf{Sh}}(\phi(n)) \\
&\qquad\quad \wedge \forall \widehat{l} \in \widehat{\mathsf{Fld}}(\widehat{\mathsf{Ty}}(n)) \,.\, \widehat{\mathsf{Inj}}(\widehat{\sigma}_1(n)) = \widehat{\mathsf{Inj}}(\widehat{\sigma}_2(\phi(n))) \\
&\qquad\quad \wedge \forall \text{ cutpoints } c_p \in \widehat{\mathsf{CP}}_1 \,.\, \widehat{\mathsf{Trgts}}(\widehat{\sigma}_2(\phi_{c_p}(c_p))) = \{\phi(n') \mid n' \in \widehat{\mathsf{Trgts}}(\widehat{\sigma}_1(c_p))\}
\end{aligned}
$$

For the $\approx$ operation, given $\widehat{h}_{mrg} = \widehat{h}_{in} \sqcup \widehat{h}_{memo}$, then $\approx$ is defined as: $\widehat{h}_{in} \approx \widehat{h}_{memo} \Leftrightarrow \forall v \in \widehat{\mathsf{Env}}_{memo}$ the points-to set in $\widehat{h}_{mrg}$ for v is identical to the points-to set for v in $\widehat{h}_{memo}$. Formally, given $T_{memo} = \widehat{\mathsf{Trgts}}(\widehat{\sigma}_{memo}(\widehat{\mathsf{Env}}_{memo}(v)))$ and $T_{mrg} = \widehat{\mathsf{Trgts}}(\widehat{\sigma}_{mrg}(\widehat{\mathsf{Env}}_{mrg}(v)))$ which we assume always are empty or have size zero (Algorithm 1) or one then:

$$
\begin{aligned}
&T_{memo} \equiv T_{mrg} \Leftrightarrow |T_{mrg}| = |T_{memo}| \wedge \forall n_{mrg} \in T_{mrg}, n_{memo} \in T_{memo} \\
&\qquad \widehat{\mathsf{Ty}}(n_{mrg}) \cap \widehat{\mathsf{Ty}}(n_{memo}) \neq \emptyset \\
&\qquad \wedge InDegree(n_{mrg}) = InDegree(n_{memo}) \wedge InVars(n_{mrg}) = InVars(n_{memo})
\end{aligned}
$$

This test is based on the hypothesis that at the entry of complex method calls the programmer makes very few assumptions about the behavior of the method beyond what is provided by the pre/post conditions. Thus, if two abstract heaps have the same aliasing relations on the argument variables then any differences are in the internal heap structures and are unimportant details that are irrelevant to the overall behavior of the program. This matching enables the analysis to remain fully-context sensitive for simple methods that may not follow a strict pre/post condition protocol with while more complex (and recursive methods) that are likely to have more strict pre/post conditions will be treated with less context-sensitivity.

3.3 Complete Partial Context-Sensitive Call Analysis

The overall interprocedural analysis algorithm is based on a simple worklist approach where methods are taken from and added to a pending worklist as new input models are seen or the result of a child call is updated. During the analysis when a call to a method m is encountered with the input described by the abstract heap $\widehat{h}$ the interprocedural analysis looks at the memoized results in the memotable for the given method. If we find an entry that matches with the input model we return the memoized result model otherwise a new model is added to the memotable for later analysis.

Definition 8 (Memo Table Representation). *For each method m in the program we maintain a list of memoized analysis models* $[\lambda_1, \dots \lambda_k]$ *where* $\lambda_i = ((\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}, \widehat{\mathsf{CP}})_i^{\mathrm{in}}, (\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}, \widehat{\mathsf{CP}})_i^{\mathrm{out}})$.

The *AnalyzeCall* method (Algorithm 1) takes a method call and a receiver object. The first step ensures that each argument variable has at most one abstract target. Next we find the set of possible method implementations for the virtual call. For each of these possibilities we analyze the callee method body (or fetch the memoized result) with the given input abstract heap. The possible implementation methods are then analyzed as needed (via the *AnalyzeBody* method) with the argument abstract model and the results are accumulated to produce the result abstract model ($\widehat{h}_f$).

The *EnsureUniqueAbstractTarget* method simply checks each argument variable and does case splitting if there is more than one target. This splitting operation is useful in preventing ambiguity in the caller scope from propagating into the callee scope which improves the effectiveness of the memoization. The *GetImpls* algorithm takes a call signature c_{sig} and an abstract heap $\widehat{h}$. It then looks up every possible method target of that call signature given the possible types of the receiver object (`this`).

Algorithm 1. AnalyzeCall

input : program p, caller m_{from}, call sig. c_{sig}, abstract heap $\widehat{h}$
output: abstract heap $\widehat{h}_f$

$\widehat{h}_f \leftarrow \bot$;
heapset $\leftarrow$ EnsureUniqueAbstractTarget($\widehat{h}$);
for $\widehat{h}_i \in$ *heapset* **do**
  methodset $\leftarrow p$.GetImpls(c_{sig}, $\widehat{h}_i$);
  for $m \in$ *methodset* **do**
    $\widehat{h}_t \leftarrow$ AnalyzeBody(m, $\widehat{h}_i$);
    $\widehat{h}_f \leftarrow \widehat{h}_f \sqcup \widehat{h}_t$;
return $\widehat{h}_f$;

The *AnalyzeBody* method begins by examining an approximate call graph computed by the frontend (*IsComplexCall*) and selects one of the matching strategies, for simple acyclic components or for calls that are part of a cyclic component of the call graph, and selects the appropriate matching algorithm, $\approx$ or $\simeq$, to use in searching for a match.

Algorithm 2. AnalyzeBody

```
input  : method m, abstract heap ĥ
output: abstract heap ĥ_f
if IsComplexCall(m) then
    (cmplx, match) ← (true, ≈);
else
    (cmplx, match) ← (false, ≃);
memotable ← MemoTableFor(m);
(ĥ_e, ĥ_r) ← project(m.Args, ĥ.Copy(), cmplx);
if memotable.HasMatchFor(ĥ_r, match) then
    ĥ_acc ← memotable.GetMatchMergeIfNeeded(ĥ_r, cmplx, match);
else
    ĥ_acc ← memotable.AddNewEntry(ĥ_r);
ĥ_f ← extend(ĥ_e, ĥ_acc);
return ĥ_f;
```

If we are matching via the $\approx$ operation then the analysis also updates memoized input state as the merge of the matched memoized input model and the new argument model. The algorithm for handling the matching and merge if needed, *GetMatchMergeIfNeeded*, is shown in Algorithm 3. The two helper functions *UniqueifyCutpoints* and *RemapCutpoints* are used to ensure that after the join the one cutpoint per node property is maintained and then to ensure that the set of cutpoints used in the input model match the set of cutpoints that appear in the output model.

Algorithm 3. GetMatchMergeIfNeeded

```
input  : memoized values [λ_1, ... λ_k], abstract heap ĥ, bool cmplx, compare
         match
output: abstract heap ĥ_f
for λ ∈ [λ_1, ... λ_k] do
    if match(ĥ, λ^in) then
        if cmplx then
            λ^in ← λ^in ⊔ ĥ;
            cpremap ← UniqueifyCutpoints(λ^in);
            λ^out ← λ^out ⊔ ĥ;
            RemapCutpoints(cpremap, λ^out);
        return λ^out;
```

4 Experimental Evaluation

Our benchmark suite consists primarily of direct C# ports of commonly used Java benchmarks from Jolden (9), the db and raytracer programs from SPEC JVM98 (28), and the luindex and lusearch programs from the DaCapo suite (9). Additionally we have

analyzed the heap abstraction code, runabs, from ([19]). In practice we translate the .Net assemblies to a simplified IR (intermediate representation) which allows us to remove most .Net specific idioms from the core analysis and to perform specialized analyses on the standard libraries ([20]). Our test machine is an Intel i7 class processor at 2.66 GHz with 2 GB of RAM available. We use the standard 32 bit .Net JIT and runtime framework provided by Windows 7. The domain, operations, and data flow analysis algorithms are all implemented in C# and are publicly available.[2]

Client Applications

The analysis in this paper tracks general classes of properties that have shown, in past work, to be both relevant and useful in a wide range of client applications ([5–8], [11], [14], [18]). However, we have performed additional small scale implementations and case studies with the analysis results to ensure that the particulars of the domain defined in this paper are useful for these types of optimization and program understanding applications. These case studies include:

- The introduction of thread-level parallelization, as in ([18]), to obtain a 3× speedup for bh on our quad-core machine.
- Data structure reorganization to improve memory usage and automated leak detection, as in ([19]), to obtain over a 25% reduction in live memory in raytrace.
- The computation of ownership information for object fields, as in ([14]), identifying ownership properties for 22% of the fields and unique memory reference properties for 47% of the fields in lusearch.

However, we want to examine the quantitative precision of the analysis in a way that is free from biases introduced by the selection of a particular client application. Thus, we examine the precision of the static analysis relative to a hypothetical perfect analysis which uses the same abstract domain. This notion of precision is a more general measurement of the possible imprecision due to the partial context-sensitivity heuristics than the use of a specific client application (which may hide precision losses that *happen* not to matter for the particular client).

Quantitative Precision

We define precision relative to a hypothetical *perfect analysis* which uses the same abstract domain from Section 2 but that is able to perfectly predict the effects of every program operation. Since we cannot actually build such an analysis we approximate it by collecting and abstracting the results of concrete executions. By definition this collection of results from the concrete execution is an under-approximation of the universal information we want to compute, and in the limit of execution of all possible inputs is identical. Formally, given a method and a set of concrete heaps $\{h_1, \ldots, h_k\}$ and a set of abstract heaps $\{\widehat{h}_1, \ldots, \widehat{h}_j\}$ we can compute differences between $\bigsqcup_{h \in \{h_1,\ldots,h_k\}} \alpha(h)$ and $\bigsqcup\{\widehat{h}_1, \ldots, \widehat{h}_j\}$. This gives an unbiased measure of how close our results are to the optimal solution, wrt. the abstract domain we are working with in a way that is independent of peculiarities of a client application or other analysis technique.

[2] Source code and benchmarks available at: `http://jackalope.codeplex.com`

One possible concern with this approach is that the base abstract domain may be very coarse, i.e., $\bigsqcup_{h \in \{h_1,\ldots,h_k\}} \alpha(h)$ is always $\top$ or another very imprecise value. To account for this we report the average percentage of properties (shape and injectivity) that the runtime result marks as precise (none or tree and *injective*) in the models (the *Runtime Precise Rate* group in Table 1). This table shows that in practice the domain achieves a very high rate of precise identification of *shape* values, on average over 90% or more of nodes are precisely identified (the *Shape* column), and a similarly high rate of precise *injectivity* values, on average nearly 90% of the edges are identified as being *injective* (the *Injectivity* column). For reference the example abstract heap, Figure 1(b), abstract heap would have a 100% precise *shape* rate and a 75% precise *injectivity* rate. Thus, we can see that in general the base domain is exceptionally effective in representing the heap properties we are interested and is an effective baseline for comparison.

Table 1. Static Match is percentage of each property (at method entries) which is accurately predicted by the static analysis when compared to *perfect analysis*. Runtime Precise is the percentage of properties that the *perfect analysis* captures precisely.

	Static Match Rate			Runtime Precise Rate	
Benchmark	Region	Shape	Injectivity	Shape	Injectivity
power	100%	100%	100%	100%	100%
bh	100%	90%	87%	100%	100%
db	100%	100%	81%	100%	100%
raytracer	80%	85%	83%	89%	98%
luindex	95%	95%	82%	100%	91%
lusearch	93%	90%	84%	96%	89%
runabs	97%	98%	87%	94%	90%

Table 1 shows the results of comparing the results from our *perfect analysis* with the results from the static analysis described in this paper. In this table we report the percentage of properties in the static analysis results that are the same as reported by the runtime analysis for regions, shapes, and injectivity values. The region percentage (the *Region* column) is the number of nodes that can be exactly matched between the statically computed and ideal result structure. Using this matching we then compute the percentage of the *shape* and *injectivity* properties that are precisely identified by the static analysis (the *Shape* and *Injectivity* columns). These results show that the analysis is able to extract a large percentage of the properties that can be expressed via the abstract domain (i.e. the static analysis is able to answer any client query on aliasing or shape as accurately as possible given the underlying domain for 80% to 90% of queries). Running the static analysis without the partial context-sensitivity heuristics provides a slight increase in the accuracy (3% in the best case) and running the analysis without the use of the unique-cutpoints is infeasible (per Table 2). Thus, the use of the fresh cutpoints and context-sensitivity heuristics result in only small losses in precision in practice.

Analysis Performance

We next examine the time and memory requirements of analyzing a program using the method described in this paper. Table 2 shows the cost of running several analysis variations. The cost of analyzing a program with full context-sensitivity and without the use of the fresh cutpoints is generally infeasible (the No Opt column in Table 2) with the heap domain we are using (all but two of the benchmarks time out). The next variation of interprocedural analysis we examine (the Project column) uses full context-sensitivity but applies the project/extend with fresh cutpoints as described in this paper. The Optimized column uses the partial context-sensitivity heuristics described in this paper and the fresh cutpoint project/extend operations. For each benchmark we list the number of bytecode instructions and the number of methods that each program contains after being translated into the internal IR. These numbers exclude much of the code that would normally be part of the runtime system libraries. This is due to the fact that during the translation from .Net bytecode to the internal IR code which is never referenced is excluded. Additionally for the builtin types/methods that are used the implementations are often replaced by simplified versions or specialized domain operations.

Table 2. Statistics and aggregate performance of the *No Opt*, *Project*, and *Optimized* analysis. For the timeout we use a limit of 10 min. or 500MB of memory.

Benchmark Statistics			No Opt		Project		Optimized	
Name	Insts	Methods	Time	Mem	Time	Mem	Time	Mem
power	3298	320	1.21s	≤20MB	0.11s	≤20MB	0.09s	≤20MB
bh	3723	351	5.38s	72MB	0.61s	≤20MB	0.42s	≤20MB
db	2873	315	-	-	0.21s	≤20MB	0.21s	≤20MB
raytrace	9808	476	-	-	12.11s	40MB	6.72s	32MB
luindex	26852	246	-	-	20.15s	70MB	12.1s	53MB
lusearch	33632	272	-	-	197.79s	489MB	64.3s	130MB
runabs	27875	253	-	-	12.84s	68MB	10.4s	60MB

The *no opt* approach is unable to handle most of the benchmarks, so we do not discuss it further. When compared to the *project* approach the *optimized* technique is a factor of 2-4 faster in all of the smaller benchmarks and enables the analysis to complete the larger program where the *project* analysis times out (lusearch). Both the *project* and *optimized* analysis require very little memory to analyze the simpler programs (less than 20MB for most of them). However, as the size and complexity of the program increases we see that the *optimized* analysis improves memory usage as well.

To understand the source of this difference we examine the impact of the *optimized* analysis on the maximum number of memo entries for any method and the average number of memo entries per method (shown in Table 3). Looking at the results in the table the impact on the number of memo entries per method is quite significant. This reduction occurs in the maximum number of entries and in the average number per method. This has a huge impact on the number of comparisons that need to be done to test if a given call model is memoized and in the number of times each method body is analyzed. As the program gets larger this reduction has a large impact in the amount

of time and memory needed to analyze a program. We note that the improvement is small on the simpler benchmarks but become increasingly important as the complexity increases in db and raytrace, and is critical to successfully analyzing luindex, lusearch, and runabs.

Table 3. Comparison of the *Project* and *Optimized* analysis on model set size and memo table entries. *Avg Entries* is the average number of entries for each method memotable, *Max Entries* is the maximum number of entries, and *Max Iters.* is the maximum number of times a memo table entry is analyzed before it reaches a fixpoint.

Benchmark	Project			Optimized		
Name	Avg Entries	Max Entries	Max Iters.	Avg Entries	Max Entries	Max Iters.
power	1.03	2	3	1.02	2	3
bh	1.46	5	3	1.11	5	2
db	1.80	5	1	1.80	5	1
raytrace	2.36	15	4	2.14	15	4
luindex	2.54	21	7	2.46	17	6
lusearch	11.60	100	17	2.08	22	16
runabs	2.15	14	4	2.11	13	4

These results emphasize the robustness of the scaling of the interprocedural analysis with respect to growth in the memo table sizes. In particular the average number of memo entries is nearly constant regardless of the size of the program and thus the memory (time) required by the analysis scales with the number of contexts and number of iterations required to reach a fixpoint.

5 Related Work

There has been a substantial amount of work on techniques for speeding up interprocedural dataflow analysis. These range from methods for efficiently encoding call contexts using BDD's ([13, 29]), to heuristics that alter the level of context sensitivity based on structural properties of the call or flow graph ([12, 21, 24]), to methods for generating partial call context tokens ([10, 22, 30]). The work in this paper draws on the general concepts of context heuristics presented in ([12, 15, 21, 23, 24]) and work on project/extend operations ([17, 26]), to efficiently and precisely manage the challenges present when performing interprocedural heap analysis on object-oriented programs.

This work differs substantially from previous work in a number of respects. One of the key insights into this work is that there exist specific points in the program where the programmer makes generalizing assumptions about the state of the program and thus most of the differences between abstract heaps are not critical to understanding the later behavior of the program. This idea has been exploited implicitly in previous work which often applies some set of heuristics for merging states either at call or local control flow joins ([17, 21, 23, 31]) but there is little empirical or conceptual justification (beyond the experimental results) for why the specific set of join points selected is a good choice.

Recent work on modular shape analysis ([2, 4]) has also shown promise in scaling to large programs. However these approaches place substantial restrictions on the programs that are being analyzed. Techniques in ([2]) do not allow generalized sharing in the heap structures while the techniques in ([4]) do not handle programs that contain recursive data structures. Further, the programs that these approaches have been demonstrated on contain relatively shallow and small data structures and the code does not heavily employ recursion or dynamic dispatch. In contrast the approach in this paper does not place any constraints on the programs under analysis and has been demonstrated on a range of programs that use large and complex data structures that are connected in a variety of ways. The benchmark programs used in this paper also extensively employ dynamic dispatch and contain non-trivial recursive traversals of heap structures.

6 Conclusion

This work introduced and provided justification for a novel partial context-sensitivity heuristic, and a new take on *cutpoints* that produce a computationally efficient analysis which is still able to produce precise results in practice. The impact of these contributions was demonstrated by integrating them, and an existing high precision heap analysis domain, into a complete context-sensitive dataflow analysis. Using this algorithm we analyzed a number of well known object-oriented programs. The results provide empirical evidence that the use of fresh cutpoints and partial-context sensitivity heuristics based on principles of object-oriented program design drastically reduce the time and memory required to analyze a program (in a number of cases it is faster than many state of the art context-sensitive points-to analyses). Further, these performance improvements were obtained without a substantial degradation in the precision of the results. Thus, we believe the interprocedural analysis presented in this paper represents an important technical advancement in the state of the art in precise and scalable heap analysis techniques and also introduces a new way to think about how program development concepts can be leveraged in the design of static analysis techniques.

References

1. Barr, E., Bird, C., Marron, M.: Collecting a Heap of Shapes. Technical Report MSR-TR-2011-135, Microsoft Research (December 2011)
2. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM (2011)
3. Chase, D., Wegman, M., Zadeck, K.: Analysis of pointers and structures. In: PLDI (1990)
4. Dillig, I., Dillig, T., Aiken, A., Sagiv, M.: Precise and compact modular procedure summaries for heap manipulating programs. In: PLDI (2011)
5. Ghiya, R., Hendren, L.J.: Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In: POPL (1996)
6. Gulwani, S., Tiwari, A.: An Abstract Domain for Analyzing Heap-Manipulating Low-Level Software. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 379–392. Springer, Heidelberg (2007)
7. Guyer, S.Z., McKinley, K.S.: Finding your cronies: static analysis for dynamic object colocation. In: OOPSLA (2004)
8. Guyer, S.Z., McKinley, K.S., Frampton, D.: Free-me: a static analysis for automatic individual object reclamation. In: PLDI (2006)

9. Jolden Suite, http://www-ali.cs.umass.edu/DaCapo/
10. Jones, N., Muchnick, S.: Flow analysis and optimization of Lisp-like structures. In: POPL (1979)
11. Lattner, C., Adve, V.: Automatic pool allocation: improving performance by controlling data structure layout in the heap. In: PLDI (2005)
12. Lattner, C., Lenharth, A., Adve, V.: Making context-sensitive points-to analysis with heap cloning practical for the real world. In: PLDI (2007)
13. Lhoták, O., Hendren, L.: Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. ACM Trans. Softw. Eng. Method. 18(1) (2008)
14. Ma, K.-K., Foster, J.: Inferring aliasing and encapsulation properties for Java. In: OOPSLA (2007)
15. Manevich, R., Sagiv, M., Ramalingam, G., Field, J.: Partially Disjunctive Heap Abstraction. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 265–279. Springer, Heidelberg (2004)
16. Marron, M.: Structural analysis: Combining shape analysis information with points-to analysis computation. arXiv:1201.1277v1 [cs.PL] (2012)
17. Marron, M., Hermenegildo, M., Kapur, D., Stefanovic, D.: Efficient Context-Sensitive Shape Analysis with Graph Based Heap Models. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 245–259. Springer, Heidelberg (2008)
18. Marron, M., Méndez-Lojo, M., Hermenegildo, M., Stefanovic, D., Kapur, D.: Sharing analysis of arrays, collections, and recursive structures. In: PASTE (2008)
19. Marron, M., Sanchez, C., Su, Z., Fahndrich, M.: Abstracting runtime heaps for program understanding (2011), http://heapdbg.codeplex.com/
20. Marron, M., Stefanovic, D., Hermenegildo, M., Kapur, D.: Heap analysis in the presence of collection libraries. In: PASTE (2007)
21. Mauborgne, L., Rival, X.: Trace Partitioning in Abstract Interpretation Based Static Analyzers. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 5–20. Springer, Heidelberg (2005)
22. Milanova, A., Rountev, A., Ryder, B.: Parameterized object sensitivity for points-to and side-effect analyses for Java. In: ISSTA (2002)
23. Muthukumar, K., Hermenegildo, M.: Compile-time derivation of variable dependency using abstract interpretation. JLP 13(2/3) (1992)
24. Reps, T., Lal, A., Kidd, N.: Program Analysis Using Weighted Pushdown Systems. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 23–51. Springer, Heidelberg (2007)
25. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: LICS (2002)
26. Rinetzky, N., Bauer, J., Reps, T., Sagiv, S., Wilhelm, R.: A semantics for procedure local heaps and its abstractions. In: POPL (2005)
27. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: understanding object-sensitivity. In: POPL (2011)
28. Standard Performance Evaluation Corporation. JVM98 Version 1.04 (August 1998), http://www.spec.org/jvm98
29. Whaley, J., Lam, M.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PLDI (2004)
30. Wilson, R., Lam, M.: Efficient context-sensitive pointer analysis for C programs. In: PLDI (1995)
31. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.: Scalable Shape Analysis for Systems Code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)

Parallel Replication-Based Points-To Analysis

Sandeep Putta and Rupesh Nasre

Indian Institute of Technology, Bombay, India
Indian Institute of Science, Bangalore, India
sandeep.p@iitb.ac.in, nasre@csa.iisc.ernet.in

Abstract. Pointer analysis is one of the most important static analyses during compilation. While several enhancements have been made to scale pointer analysis, the work on parallelizing the analysis itself is still in infancy. In this article, we propose a parallel version of context-sensitive inclusion-based points-to analysis for C programs. Our analysis makes use of *replication* of points-to sets to improve parallelism. In comparison to the former work on parallel points-to analysis, we extract more parallelism by exploiting a key insight based on monotonicity and unordered nature of flow-insensitive points-to analysis. By taking advantage of the nature of points-to analysis and the structure of constraint graph, we devise several novel optimizations to further improve the overall speed-up. We show the effectiveness of our approach using 16 SPEC 2000 benchmarks and five large open source programs that range from 1.2 KLOC to 0.5 MLOC. Specifically, our context-sensitive analysis achieves an average speed-up of 3.4× on an 8-core machine.

1 Introduction

Points-to analysis [1,28,6,4,17] is a method of statically determining whether two pointers may point to the same location at runtime. The two pointers are then said to be aliases of each other. While pointer analysis is immensely helpful for compiler optimizations and extracting parallelism, the analysis itself can be run in parallel to take advantage of the multiple resources available.

There is very little literature on parallelizing pointer analysis. Kahlon [17] proposed *bootstrapping* that identifies alias sets using Steensgaard's analysis [28] and then Andersen's analysis [1] is *simulated* to run in parallel on each alias set. Lojo et al. [20] proposed speculative parallelization of inclusion-based pointer analysis to expose amorphous data-parallelism in C programs. Recently, Edvinsson et al. [7] proposed parallel points-to analysis for multi-core machines by exploiting the independence of polymorphic calls and control-flow branches of Java programs. Our method finds more fine-grained parallelism compared to the above methods.

A typical method of parallelizing points-to analysis involves two tasks: (i) identifying non-conflicting constraints and (ii) analyzing the non-conflicting constraints in parallel. The parallelism extracted by the existing techniques is limited due to the inherent irregular nature of the applications (e.g., several SPEC 2000

M. O'Boyle (Ed.): CC 2012, LNCS 7210, pp. 61–80, 2012.
© Springer-Verlag Berlin Heidelberg 2012

benchmarks). For instance, speedup in parallel online analysis time in Lojo et al.'s work [20] is maximum 2x using 8-cores over a set of open source C programs analyzed, while in Edvinsson et al.'s work [7] it is maximum 1.76 using 8-cores over a set of Java benchmarks. Thus, we see that irregularity of applications (as defined in [20]) has been a stumbling block while extracting parallelism for performing points-to analysis.

We observe that the monotonicity of a flow-insensitive points-to analysis can help us achieve more parallelism. A key insight is that by allowing the analysis to keep multiple copies of points-to sets, one can reduce dependence across constraints. In fact, with sufficient number of copies, task (i) above gets trivialized since no two constraints conflict! This allows us complete flexibility while scheduling constraints on multiple cores. On the downside, this kind of data replication can affect analysis soundness, as certain data flow facts may not get computed. However, by exploiting monotonicity property of a flow-insensitive analysis, we show that the analysis soundness can be preserved by carefully merging the multiple copies in each iteration of the analysis. This helps us develop a replication-based, yet sound, parallel analysis.

Major contributions of this paper are as below.

- A replication-based data flow analysis to improve parallelism and a novel method based on monotonicity and unordered nature of flow-insensitive algorithm to achieve a sound analysis in the presence of replicated data.
- Instantiating the data-flow analysis to develop a replication-based parallel points-to analysis.
- Several engineering optimizations specific to pointer analysis, like parallel online cycle elimination and limited cycle detection for extracting more parallelism and for a scalable implementation.
- Detailed evaluation of our method using SPEC 2000 benchmarks and five large open source programs (*httpd*, *sendmail*, *ghostscript*, *gdb* and *wine-server*). Our context-sensitive (insensitive) inclusion-based parallel version achieves an average speed-up of 3.4x (3.0x) on an 8-core machine.

2 Motivation and Background

While our approach applies to several data-flow analyses, we illustrate it by parallelizing Andersen's inclusion-based analysis [1]. Thus, we deal with flow-insensitive points-to analysis. For analyzing a general purpose C program, it is sufficient to consider all pointer statements of the following forms: address-of assignment ($\mathtt{p} = \&\mathtt{q}$), copy assignment ($\mathtt{p} = \mathtt{q}$), load assignment ($\mathtt{p} = *\mathtt{q}$) and store assignment ($*\mathtt{p} = \mathtt{q}$) [24]. Address-of constraints can be evaluated only once. Thus, the analysis iterates over the other three kinds of constraints until a fixed-point.

Consider the following example program.

$$\mathtt{p} = \&\mathtt{a}, \mathtt{a} = \&\mathtt{x}, \mathtt{b} = \&\mathtt{y}, \mathtt{c} = \&\mathtt{z}, \mathtt{d} = \&\mathtt{w}, \mathtt{q} = \mathtt{p}, \mathtt{a} = \mathtt{b},$$
$$\mathtt{e} = \mathtt{a}, \mathtt{r} = \mathtt{q}, \mathtt{a} = \mathtt{c}, \mathtt{s} = \mathtt{r}, \mathtt{e} = *\mathtt{a}, \mathtt{t} = \mathtt{s}, \mathtt{a} = \mathtt{d}, *\mathtt{e} = \mathtt{a}$$

Table 1. Running example (13 steps: 10 for Iteration 1 and 3 for Iteration 2)

Statement	Iteration 0	Iteration 1	Iteration 2	Iteration 3
p = &a	p→{a}			
a = &x	a→{x}			
b = &y	b→{y}			
c = &z	c→{z}			
d = &w	d→{w}			
q = p		q→{a}		
a = b		a→{y}		
e = a		e→{x,y}	e→{z,w}	
r = q		r→{a}		
a = c		a→{z}		
s = r		s→{a}		
e = *a				
t = s		t→{a}		
a = d		a→{w}		
*e = a		x,y→{x,y,z,w}	z,w→{x,y,z,w}	

A sequential analysis of the above program is given in Table 1. Iteration 0 shows the initial points-to information after processing address-of constraints. We denote points-to information as a set of variables (e.g., {x,y}) and the points-to relation using an arrow (→). The analysis requires three iterations to reach a fixed-point which contains the following points-to facts.

$$\mathtt{p,q,r,s,t} \rightarrow \{\mathtt{a}\}, \mathtt{a,e,x,y,z,w} \rightarrow \{\mathtt{x,y,z,w}\},$$
$$\mathtt{b} \rightarrow \{\mathtt{y}\}, \mathtt{c} \rightarrow \{\mathtt{z}\}, \mathtt{d} \rightarrow \{\mathtt{w}\}$$

For exposition purpose, we define a *step* as a computation of points-to information of a variable. Intuitively, it is proportional to the amount of time required to analyze a constraint. Thus, each copy and load constraint requires a single step, while each store constraint requires zero or more steps depending upon the points-to information of the dereferenced variable. Thus, the above example requires 13 steps (of non-address-of constraints) to reach a fixed-point for a sequential analysis. It is easy to see that the non-address-of constraints in the example require at least 12 steps to compute the fixed-point.

Analyzing a constraint `DST = SRC` involves (i) reading points-to information of SRC and (ii) updating points-to information of DST. Depending upon the type of the constraint, we have different read and write sets as shown in Table 2. The read-set for the load constraint (p = *q) contains not only the pointees of q but also q itself. Similarly, the read-set of the store constraint (*p = q) contains both p and q. Due to weak-typing of C language, read and write sets for a constraint need not be mutually exclusive. For instance, following statement is valid (with type-casting): p = *p, and its read-set as well as write-set contain p.

Two constraints C1 and C2 *conflict* with each other if at least one of the following three conditions holds: (i) ReadSet(C1) ∩ WriteSet(C2) $\neq \phi$ (ii) WriteSet(C1) ∩ ReadSet(C2) $\neq \phi$ (iii) WriteSet(C1) ∩ WriteSet(C2) $\neq \phi$.

Table 2. Read-Write sets for points-to constraints

Constraint	Read Set	Write Set
p = q	{q}	{p}
p = *q	$\{q\} \cup \{x : q \rightarrow \{x\}\}$	{p}
*p = q	{q, p}	$\{x : p \rightarrow \{x\}\}$

The *conflict* relation is reflexive, symmetric and non-transitive. The above conditions can be easily generalized to multiple constraints. Non-conflicting constraints can be analyzed in parallel. Therefore, identifying the constraints that can be evaluated in parallel involves computing read and write (RW) sets of various constraints in some form. As more points-to information gets computed, the read sets of load constraints and the write sets of store constraints also go on increasing in size, potentially reducing opportunities for parallelism as the analysis progresses. Due to monotonic nature of the flow-insensitive analysis, the RW sets never shrink. Note that since the points-to sets are computed dynamically, the RW sets need to be updated dynamically. Using the RW sets, one can parallelize analyzing the example as shown in the parallel schedule below.

Thread T1: q = p, r = q, s = r, t = s
Thread T2: a = b, a = c, a = d, e = *a, e = a, *e = a

The parallel schedule involves analyzing the two sets of constraints using two threads T1 and T2. Compared to 12 (minimum number of) steps of the sequential analysis, the parallel analysis requires at most 9 steps to reach the fixed-point: one step each for the first five copy and load constraints of T2 and four steps for the last store constraint of T2 as pointer e points to four variables {x, y, z, w}. However, due to conflicting constraints, it is not possible to take advantage of more than two cores for this example using a naive parallel points-to analysis. Thus, even if a points-to analysis is provided with four or eight cores, the parallel analysis would still require at least 9 steps. Further, computing and maintaining RW sets, identifying conflicts across constraints and generating a parallel schedule is a time-consuming process. As the RW sets of a constraint change, it needs to be checked for conflict against all other non-conflicting constraints. This checking requires $O(n^2)$ operations *per change in the read-write set* where n is the number of constraints. Conflicting constraints are commonplace [13] and therefore, a naive parallel analysis is going to be prohibitive in terms of not only the amount of parallelism but also the parallel execution time. Our technique illustrates how to extract more parallelism efficiently even when several conflicting constraints exist.

3 Replication-Based Analysis

In this section we first introduce our replication-based approach at a higher level by giving an outline of the algorithm. We prove that a replication-based approach is sound for a monotonic, unordered data-flow analysis (e.g., a flow-insensitive

points-to analysis). We then explain in detail how replication works for our parallel points-to analysis algorithm.

A flow-insensitive analysis ignores the control-flow information and assumes that program statements may be executed in any order. We state a well-known property of flow-insensitive data-flow analysis to prove soundness of our method.

Theorem 1. *A flow-insensitive data-flow analysis computes the same fixed-point irrespective of the order in which the program statements are analyzed.*

3.1 Algorithm Outline

A data-flow analysis is monotonically increasing if it never *kills* a computed data-flow fact. A flow-insensitive points-to analysis is an example of a monotonically increasing (or simply, monotonic) analysis whereas a flow-sensitive points-to analysis is an example of a non-monotonic analysis. Since flow-insensitive points-to analysis is monotonic and its solution does not depend upon the order of evaluation, our method partitions the set of constraints arbitrarily, analyzes them in parallel, merges the individual solution sets and repeats this process until a fixed-point, as shown in Algorithm 1. The merge operation performs a union of points-to information for each pointer. Thus, if thread T1 computes the following points-to information: p $\rightarrow$ {a}, q $\rightarrow$ {b}, and thread T2 computes the following points-to information: p $\rightarrow$ {c}, q $\rightarrow$ {a,b}, r $\rightarrow$ {c}, then the merge operation on Line 5 of Algorithm 1 computes the following points-to information: p $\rightarrow$ {a,c}, q $\rightarrow$ {a,b}, r $\rightarrow$ {c}.

We first prove that the parallel algorithm computes a safe solution. It is sufficient to prove that the algorithm computes the same solution as that computed by a sequential analysis.

Theorem 2. *Algorithm 1 is sound.*

Proof. Let SEQ be the least fixed-point points-to set computed by a flow-insensitive sequential analysis. We prove that Algorithm 1 computes a solution PAR which equals SEQ, i.e., SEQ = PAR. In this proof, we make a simplistic assumption that in every iteration of the parallel analysis, the constraints are executed in the same order. A multithreaded schedule of the input points-to constraints C can then be represented as an interleaving of the constraints, with constraints analyzed in parallel placed in an arbitrary (but fixed) order. This interleaving forms a sequential ordering S over all the constraints and since it is derived from the parallel schedule, its least fixed-point solution SEQ′ must equal PAR, i.e., SEQ′ = PAR. Now, assume a sequential flow-insensitive analysis analyzing C in the order denoted by S. By Theorem 1, the least fixed-point solution SEQ′ computed by this sequential analysis must equal SEQ, i.e., SEQ = SEQ′. Therefore, SEQ = PAR, proving that Algorithm 1 computes the same solution as that computed by a sequential analysis.

We would like to emphasize that both monotonicity and unorderedness are the required properties of the underlying data-flow analysis for our parallel algorithm

Algorithm 1. Outline of our parallel points-to analysis

Require: set `C` of points-to constraints, number of cores N
1: partition C arbitrarily into N threads
2: **repeat**
3: schedule N threads on N cores
4: wait for N threads to complete
5: merge points-to sets of N threads
6: **until** fixed-point

to compute a safe result. Our algorithm is general and is applicable to any analysis that is monotonic and unordered. For instance, it can be applied to 0-CFA [25].

3.2 Replication

Our method handles conflicting constraints by keeping multiple copies of the conflicting variables and their associated points-to sets. For instance, the constraints p = q, r = p and p = s conflict. However, if they are analyzed in separate threads T1, T2 and T3 respectively, our method creates a copy of p's points-to set in T1 and T3 since their write sets contain p. Since T2 only reads p, it continues to read the master (original) copy of p. The newly computed points-to information of T1 and T3 is merged with that of the master copy of p at the end of each iteration. Note that T2 would contain a copy of the points-to set of r. Further note that it is possible to use multiple copies of variables since the analysis is monotonic. If the analysis is non-monotonic (for instance, if it is flow-sensitive) then naively making multiple copies of the points-to sets may not preserve the solution.

The merge operation performs a union of points-to information for each pointer. Merging of the points-to information for multiple pointers is done in parallel. However, merging of local points-to information of a pointer with its master copy is done in a sequential manner.

Table 3 shows the parallel analysis of the example program using our method. Recall from Section 2 that a naive parallel analysis using read-write sets could not take advantage of more than two cores. Therefore, we illustrate our technique using three threads. Column 1 shows the thread number. Column 2 shows the points-to constraints assigned to each thread. Column 3 and 4 show the new points-to information created as a local copy and the merging of the local copies with the master copies for Iteration 1. The local copies of variable v are denoted as v', v", ... Further columns show the analysis for further iterations.

In contrast to three iterations of the sequential analysis, the parallel analysis requires four iterations to reach the fixed-point. This happens because some points-to information computed by the constraints partitioned across different threads requires another iteration to propagate. In general, our multi-copy parallel analysis requires upto 30% more number of iterations over its sequential counterpart. However, by utilizing more cores and extracting more parallelism by making copies, the overall parallel analysis time gets much smaller.

Table 3. Parallel analysis using three threads (12 steps): Iterations 1, 2, 3 require 4, 2, 3 steps respectively and each merge requires 1 step

T	Stmt	Itr 1	Merge 1	Itr 2	Merge 2	Itr 3	Merge 3	Itr 4
1	q = p	q'→{a}						
	a = b	a'→{y}						
	r = q	r'→{a}						
	e = a	e'→{x,y}	a→{y,z,w},	e'→{x,y,z,w}	e→{z,w}		z,w→{x,y,z,w}	
2	a = c	a"→{z}	e→{x,y}		x,y→{x,y,z,w}		t→{a}	
	e = *a		q,r→{a}		s→{a}			
	s = r			s'→{a}				
3	a = d	a"'→{w}						
	t = s					t'→{a}		
	*e = a			x',y'→{x,y,z,w}		z',w'→{x,y,z,w}		

Table 4. Parallel analysis using four threads (9 steps): Iterations 1, 2, 3 require 2, 1, 3 steps respectively and each merge requires 1 step

T	Stmt	Itr 1	Merge 1	Itr 2	Merge 2	Itr 3	Merge 3	Itr 4
1	q = p	q'→{a}						
	e = a	e'→{x}		e'→{y,z,w}				
2	r = q		a→{y,z,w},	r'→{a}	e→{y,z,w}		y,z,w→{x,y,z,w}	
	a = c	a"→{z}	e→{x},		x→{x,y,z,w}		s,t→{a}	
3	e = *a		q→{a}		r→{a}			
	s = r							
	t = s					s',t'→{a}		
	a = d	a"'→{w}						
4	*e = a			x'→{x,y,z,w}		y',z',w'→{x,y,z,w}		
	a = b	a'→{y}						

Observe from Table 3 that our parallel analysis requires 12 steps (or time-units) to reach the fixed-point (4 for Iteration 1, 2 for Iteration 2 and 3 for Iteration 3). Since merge operation for multiple pointers is done in parallel, we add one step for each merge operation. If we increase the number of threads to four, our analysis computes the same fixed-point in 9 steps (Table 4: 2 for Iteration 1, 1 for Iteration 2, 3 for Iteration 3 and 3 for the merge operations). If we further increase the number of threads to five, our analysis can compute the fixed-point in 8 steps (Table 5: 1 for Iteration 1, 1 for Iteration 2, 3 for Iteration 3 and 3 for the merge operations). Recall from Section 2 that the sequential version required at least 12 steps to reach the fixed-point and the naive parallel version required 9 steps. This illustrates the unique ability of our method to extract more and more fine-grained parallelism from a seemingly sequential component of a program and improving the resource usage of multiple cores.

Note also that replication of points-to information is not transparent to the threads. Each thread simply deals with its own copy whenever it modifies data. A thread does not need to know about other threads or the number of copies of the points-to information it accesses in the system. This helps in keeping the multithreaded code simple and greatly eases the code understanding. We would like to emphasize that this property is an artifact of the monotonicity and unorderedness of flow-insensitive analysis.

Table 5. Parallel analysis using five threads (8 steps): Iterations 1, 2, 3 require 1, 1, 3 steps respectively and each merge requires 1 step

T	Stmt	Itr 1	Merge 1	Itr 2	Merge 2	Itr 3	Merge 3	Itr 4
1	e = a	e'→{x}		e'→{y,z,w}				
2	q = p	q'→{a}						
3	r = q a = c	a"→{z}	a→{y,z,w}, e→{x}, q→{a}	r'→{a}	e→{y,z,w} x→{x,y,z,w} r→{a}		y,z,w→{x,y,z,w} s,t→{a}	
4	e = *a s = r t = s a = d	a"'→{w}				s',t'→{a}		
5	*e = a a = b	a'→{y}		x'→{x,y,z,w}		y',z',w'→{x,y,z,w}		

4 Parallel Points-to Analysis Algorithm

In this section we discuss our parallel points-to analysis algorithm in more detail. Next, we discuss key optimizations which improve overall parallelism of the analysis.

Each analyzer thread runs Algorithm 2. The scheduler (parent) thread communicates with the analyzer threads using global variables $mystate_i$. Each thread runs an indefinite loop until fixed-point (Lines 1–33). The fixed-point is determined by the parent thread during merging operations (Line 5). When a thread is scheduled with an input set of constraints, it analyzes each constraint and depending upon the type of the constraint, it creates local copies of the write-sets and updates the points-to information locally. Note that the local copies are not erased at the end of an iteration and therefore get automatically cached for the further iterations by the thread. As more points-to information is computed, additional local copies of variables are created by the thread. Lines 13–15 process a load constraint (p = *q), The for-loop at Lines 22–24 process a store constraint (*p = q) and Line 29 process a copy constraint (p = q). After processing all the input constraint, Thread i updates its state in global variable $mystate_i$.

The parent thread spawns analyzer threads, waits for them to complete an iteration each, merges their local copies of points-to sets with the master copy in parallel and checks if the fixed-point is reached. Although not shown in the Algorithm, accesses to the global variables (*fixed-point* and $mystate_i$) are protected using locks. Since these accesses occur infrequently (once per thread per iteration), these do not affect the analysis performance in any significant manner.

4.1 Load Balancing

Replication allows an arbitrary distribution of constraints to threads. However, to achieve good performance, proper load-balancing of work is necessary. Unfortunately, the amount of points-to information propagated from one pointer to another differs significantly across pointers and across iterations. Therefore, a static constraint partitioning, which assigns a constraint evaluation to a fixed thread throughout the analysis, achieves only a limited success. On the other extreme, re-calculating the partitions for a perfect load-balance makes the analysis

Algorithm 2. Points-to analysis by thread i

```
Require: thread id i, set C_i of points-to constraints
 1: while true do
 2:    while mystate_i = I do not have work do
 3:       ;
 4:    end while
 5:    if fixed-point reached then
 6:       break;
 7:    end if
 8:    for each constraint c ∈ C_i do
 9:       if c is a load constraint p = *q then
10:          if p is not copied locally then
11:             make a local copy p' of p
12:          end if
13:          for each v ∈ points-to set of q do
14:             points-to set(p') ∪= points-to set(v)
15:          end for
16:       else if c is a store constraint *p = q then
17:          for each v ∈ points-to set(p) do
18:             if v is not copied locally then
19:                make a local copy v' of v
20:             end if
21:          end for
22:          for each v ∈ points-to set(p) do
23:             points-to set(v') ∪= points-to set(q)
24:          end for
25:       else if c is a copy constraint p = q then
26:          if p is not copied locally then
27:             make a local copy p' of p
28:          end if
29:          points-to set(p') ∪= points-to set(q)
30:       end if
31:    end for
32:    mystate_i = Another iteration done
33: end while
```

slower than no load-balancing at all! Therefore, we employ a greedy, incremental approach, which achieves an approximately load-balanced threads at a much reduced cost. Our algorithm first distributes copy and load constraints to threads in an even (round-robin) manner, since both kinds of constraints have a singleton write-set (see Table 2). It then makes a single pass over the (costly) store constraints to distribute those to threads again in an even manner. Recall that store constraints may update the points-to sets of multiple pointers. Each thread also maintains a single number indicating its load (amount of work), based on the constraints assigned. In each iteration, as a constraint evaluation results in new points-to information, each thread updates its load-indicator, keeping track of how much information each constraint changed in that iteration. If the

new load-indicator is more than its value in the previous iteration by a threshold (pre-determined based on the number of constraints and threads), the thread *orphans* a few (fixed at 5 in our experiments) constraints that added the maximum points-to information and adds those to a shared worklist. Other threads, at the end of each iteration, check this worklist and *adopt* a few constraints if their load-indicator is less than the threshold. Higher value of the threshold results in less number of accesses to the shared worklist with reduced load-balancing, whereas lowering the threshold results in improved load-balancing but more communication (via the worklist) across threads. We experimented with several values for the threshold and the optimal value differs considerably across applications. We found that a threshold set to approximately $7-10\%$ of the number of variables achieves a good trade-off for our benchmarks. Note that checking for overload is a (thread-)local strategy, which avoids costly thread-communication overhead. The strategy works reasonably well in practice because the amount of new points-to information added by a constraint in each iteration can be predicted based on that in the previous iteration [22].

4.2 Parallel Online Cycle Elimination

Inclusion-based points-to analysis is generally represented using a constraint graph G wherein a node represents a pointer and a directed edge from node n_1 to node n_2 represents the inclusion relationship points-to set$(n_1) \subseteq$ points-to set(n_2). The points-to information is propagated across the edges. Load and store constraints add more and more edges to G as the analysis progresses generating more opportunities for points-to information propagation. Accumulation of more points-to information at the nodes may result in more edges being added to G. This process is repeated until a fixed-point. Cycles may occur in G at any stage during the analysis. A cycle in G happens due to inter-dependent variables. In terms of RW sets, a cycle indicates a chain $c_1, c_2, ..., c_n$ of constraints such that write-set(c_1) $\cap$ read-set$(c_2) \neq \phi$, write-set(c_2) $\cap$ read-set$(c_3) \neq \phi$, ..., write-set(c_{n-1}) $\cap$ read-set$(c_n) \neq \phi$ and write-set(c_n) $\cap$ read-set$(c_1) \neq \phi$. For instance, a = b and b = a indicates a cycle. An important property of the pointers in a cycle is that all of them (eventually) have the same points-to information. Therefore, to reduce unnecessary propagations, cycles are collapsed. Detecting and collapsing cycles is vital for a scalable inclusion-based points-to analysis [9]. We use Tarjan's algorithm to find strongly connected components (SCC) of a directed graph [29] to detect cycles in the constraint graph.

Cycle elimination involves replacing the nodes in the cycle by a representative node with its points-to information as a union of the points-to information from all the replaced nodes. Further, the incoming edges to and the outgoing edges from the replaced nodes need to be updated to be to and from the representative node respectively. Our algorithm collapses disjoint cycles in parallel. We reuse the same threads as for solving constraints to collapse cycles, since cycle detection and collapsing is done when no threads are solving any constraints.

It is possible for two threads collapsing disjoint cycles to update the incoming edges from the same node, potentially resulting in a conflict. However, this

happens infrequently and therefore, we use locking over the nodes to be updated while collapsing cycles.

In order to get maximum benefit out of cycle detection, one needs to carefully tune the cycle detection frequency [12]. We check for cycles once per iteration.

We observed that the advantage of cycle detection is high during only the initial few iterations of the analysis and it gradually reduces as the analysis progresses. Towards the end of the analysis, the cost of cycle detection outweighs its benefits and therefore, we perform cycle detection only upto certain number of initial iterations of the analysis.

4.3 Reducing the Number of Copies

In this section we discuss optimizations that reduce the number of copies of points-to sets across threads. Reducing the number of copies also reduces the number of iterations to reach the fixed-point.

It is unnecessary to make a copy of a variable's points-to set when there is a single writer thread. We detect this situation by maintaining the number of writer threads for each variable and using directly the master copy when no more than one thread writes to the variable. For instance, in the example shown in Table 3, the constraints q = p, r = q, s = r, t = s and *e = a directly update the master copy of the variables in the write-sets.

Our analysis also takes advantage of difference propagation [14] for improving efficiency. Difference propagation involves keeping track of the difference between points-to information of the nodes forming an edge. This helps in propagating only the additional new information across the edge. Our analysis does not initiate a merge operation if the difference between the points-to information of the local copy and the master copy is nil.

While the soundness of our replication-based analysis is oblivious to the way points-to constraints are distributed across threads in each iteration of the analysis, we use a fixed partitioning across all iterations to improve its performance. This helps us cache certain points-to information locally with the thread, updating it using difference propagation with the master only when some other thread has written to it. Using a fixed constraint partitioning also helps each thread make local decisions on the constraint evaluation.

4.4 Limited Scheduling

The amount of new points-to information computed is high in the initial iterations and gradually reduces as the analysis progresses. In fact, towards the end of the analysis, only a few constraints add more points-to information. Therefore, our method restricts parallel analysis to a limited number of iterations. The decision of when to change from parallel to sequential analysis is taken based on the amount of new points-to information P_i computed in an iteration i. As soon as it falls below 10% of P_{i-1} computed in iteration $i-1$, our method starts evaluating constraints sequentially.

Algorithm 3. Context-sensitive analysis

```
Require: Function f, callchain cc, constraints C, variable set V
 1: for all statements s ∈ f do
 2:    if s is of the form p = alloc() then
 3:       if inrecursion == false then
 4:          V = V ∪ (p, cc)
 5:       end if
 6:    else if s is of the form non-recursive call fnr then
 7:       cc.add(fnr)
 8:       add copy constraints to C for actual and formal arguments
 9:       call Algorithm 3 with parameters fnr, cc, C
10:       add copy constraints to C for return value of fnr and ℓ-value in s
11:       cc.remove()
12:    else if s is of the form recursive call fnr then
13:       inrecursion = true
14:       C-cycle = {}
15:       repeat
16:          for all functions fc ∈ cyclic callchain do
17:             call Algorithm 3 with parameters fc, cc, C-cycle
18:          end for
19:       until no new constraints are added to C-cycle
20:       inrecursion = false
21:       C = C ∪ C-cycle
22:    else if s is an address-of, copy, load, store statement then
23:       c = constraint(s, cc)
24:       C = C ∪ c
25:    end if
26: end for
```

5 Context-Sensitive Analysis

We extend Algorithm 2 for context-sensitivity using an invocation graph based approach [8]. The approach readily disallows non-realizable interprocedural execution paths. The context-sensitive algorithm starts from function *main* and maintains a stack of function invocations, similar to the runtime. Thus, a *return* from a function always matches the function invocation at the top of the stack. We handle recursion, which can introduce potentially unbounded number of contexts, by iterating over the cyclic call-chain and computing a fixed-point of the points-to tuples. Our analysis is field-insensitive, i.e., we assume that any reference to a field inside a structure is to the whole structure. We do not model `setjmp-longjmp` instructions. Our algorithm handles function pointers similar to [8] by gradually refining the target functions. The context-sensitive version is outlined in recursive Algorithm 3.

The algorithm takes four parameters: the function `f` to be processed, its calling context `cc`, the set of constraints `C` to be generated and the set of variables `V` to be created. The analysis first adds(`g`, {}) to `V` for each global variable `g` where {} denotes an empty context (not shown in the algorithm). It then makes the

first call to the algorithm with parameters *main*, $\{main\}$, C={}, V. The procedure processes all the statements in the function and generates context-sensitive points-to constraints in `C`. `C` is later evaluated using Algorithm 2. Lines 2–5 in Algorithm 3 process memory allocation and create a new variable on encountering an *alloc* statement outside recursion. Lines 6–11 handle a non-recursive call. It first adds the callee to the callchain and then maps the actual arguments to the formal arguments. The algorithm recursively calls itself in Line 9 to process the invocation graph of the callee. The callee is analyzed the same way and the set of constraints `C` keeps getting updated. On the callee function's return, its return value is mapped to the ℓ-value in the call statement. Finally, the calling context is updated by removing the callee. A recursive call is handled in Lines 12–21 by iterating over the cyclic call chain and computing a fixed-point of constraints in `C-cycle`. Note that the recursive call to Algorithm 3 in Line 17 uses the same callchain. The fixed-point over the constraints `C-cycle` generated in the cyclic call graph is then merged with `C` in Line 21. The corresponding context-sensitive constraints for address-of, copy, load and store statements are added in Lines 22–25. A context-sensitive constraint contains variables in a particular context. The two sets, `C` and `V` are finally passed on to Algorithm 2 for solving. The reason for designing the analysis as a two step process (generating constraints and solving them), rather than interleaving the two tasks, is to have a common constraint solving phase (with minor modifications). Thus, Algorithm 2 is used for both context-insensitive and context-sensitive analysis.

Making the analysis context-sensitive increases the number of (context-wise) variables and reduces the sizes of read-write sets for constraints. Thus, making the analysis context-sensitive reduces the number of conflicts across constraints, and in turn, the cost of merging. This helps a context-sensitive analysis achieve a better speed-up over the context-insensitive version.

6 Experimental Evaluation

We evaluate the effectiveness of our approach using 16 SPEC C/C++ benchmarks and five large open source programs, namely *httpd, sendmail, ghostscript, gdb* and *wine-server*. The benchmark characteristics are given in Table 6. All programs are run on an 8-core Intel Xeon E5440 with 2.83 GHz clock, 16 GB RAM running Debian GNU/Linux 5.0.

6.1 Context-Insensitive Analysis

Analysis Time. Table 7 shows the speedup obtained using different number of threads. Column SEQ indicates the absolute sequential analysis time in seconds. This base analysis is inclusion-based points-to analysis with offline variable substitution [26] and online cycle elimination [9] implemented. Both the sequential and the parallel implementations use sparse bitmaps to store points-to information. Columns titled 1, 2, 4, 6, 8 indicate the speedup obtained over the sequential version using the said number of threads.

Table 6. Benchmark characteristics

Benchmark	KLOC	# Total Inst	# Pointer Inst	# Func
176.gcc	222.185	328,425	119,384	1,829
253.perlbmk	81.442	143,848	52,924	1,067
254.gap	71.367	118,715	39,484	877
255.vortex	67.216	75,458	16,114	963
177.mesa	59.255	96,919	26,076	1,040
186.crafty	20.657	28,743	3,467	136
300.twolf	20.461	49,507	15,820	215
175.vpr	17.731	25,851	6,575	228
252.eon	17.679	126,866	43,617	1,723
188.ammp	13.486	26,199	6,516	211
197.parser	11.394	35,814	11,872	356
164.gzip	8.618	8,434	991	90
256.bzip2	4.650	4,832	759	90
181.mcf	2.414	2,969	1,080	42
183.equake	1.515	3,029	985	40
179.art	1.272	1,977	386	43
httpd	125.877	220,552	104,962	2,339
sendmail	113.264	171,413	57,424	1,005
ghostscript	438.204	906,398	488,998	6,991
gdb	474.591	576,624	362,171	7,127
wine-server	178.592	110,785	66,501	2,105

The average speedup as a geometric mean is 3.001 on 8 cores. The best speedup of 4.119 is obtained for perlbmk. Our parallel version takes 10% more time than SEQ for a single thread. However, all the benchmarks we experimented with perform better than SEQ for two (and more) threads.

We compare our results with the parallel points-to analysis built using Galois system [20][1]. Figure 1 shows the average speedups over the base sequential versions. The Galois system obtains an average speedup of 2.314 on the set of benchmarks using 8 cores. Our implementation performs consistently better for any number of threads. It should be emphasized that the parallel points-to analysis in Galois uses speculative parallelism which can rollback an activity if a conflict is detected. Our method uses multiple copies of points-to sets and results in no conflicts across threads. It does not incur any rollback overheads. Therefore, it is suited even for a non-speculative execution. We believe that it is possible to improve Galois speedup by taking advantage of the monotonicity and unordered nature of flow-insensitive points-to analysis.

Memory. Table 7 shows the memory requirements (in MB) of SEQ, our parallel algorithm PARALLEL with 8 cores and the Galois system with 8 cores for each benchmark. On an average, PARALLEL requires 12% more memory than SEQ. This is due to a few additional copies of points-to sets stored locally by each thread. However, GALOIS requires 53% more memory than PARALLEL and 71% more than SEQ. The authors of GALOIS [20] attribute the high memory consumption to Java implementation. Our C++ implementation performs optimizations to reduce the number of copies of points-to sets across threads.

[1] Downloaded from http://users.ices.utexas.edu/~marioml/hardekopfPointsTo.html.

Table 7. Context-insensitive analysis: Analysis time, Speedup and Memory

Benchmark	SEQ Time(s)	Speedup					Memory (MB)		
		1	2	4	6	8	SEQ	PARALLEL	GALOIS
gcc	6.546	0.80	1.02	2.39	2.96	3.84	83	95	179
perlbmk	2.345	0.92	1.18	2.14	3.03	4.11	100	135	188
vortex	1.445	0.96	1.04	1.97	2.46	3.59	16	24	28
eon	2.446	0.92	1.29	2.41	3.31	4.00	248	314	346
parser	0.889	0.98	1.11	1.74	2.43	3.23	4	7	7
gap	2.777	0.96	1.17	1.69	2.19	2.99	8	13	14
vpr	0.601	0.89	1.18	1.59	2.00	2.46	2	3	3
crafty	0.595	0.99	1.25	1.83	2.38	2.87	1	2	3
mesa	2.045	0.98	1.18	1.77	2.75	3.58	14	16	23
ammp	0.575	0.95	1.03	1.48	1.98	2.68	3	4	5
twolf	0.686	0.97	1.05	1.68	1.99	2.44	4	4	6
gzip	0.456	0.87	1.06	1.72	2.24	2.89	1	1	2
bzip2	0.396	0.90	1.02	1.49	1.80	2.33	1	1	1
mcf	0.382	0.88	1.00	1.36	1.79	2.17	1	1	2
equake	0.436	0.82	1.04	1.47	1.75	2.01	1	1	1
art	0.485	0.84	1.00	1.60	2.10	2.44	1	1	1
httpd	4.447	0.85	1.18	2.13	2.80	3.68	674	705	1028
sendmail	3.311	0.86	1.18	2.07	2.68	3.43	256	279	511
ghostscript	84.497	0.88	1.19	2.22	2.89	3.71	2871	3193	5719
gdb	174.355	0.89	1.09	1.68	2.31	3.01	3556	3976	5362
wine-server	4.452	0.89	1.08	1.78	2.22	2.77	185	210	336
average	14.008	0.91	1.11	1.80	2.35	3.00	382	428	655

6.2 Context-Sensitive Analysis

Analysis Time. The context-sensitive version of our parallel analysis performs similar to its context-insensitive counterpart. The speedup results are detailed in Table 8. In comparison to the context-insensitive results, since variables now have smaller points-to sets, the number of potential conflicts across threads reduces and the analysis requires less number of local copies and merging. Our parallel analysis achieves a speedup of 3.4x on an 8-core machine. Considering that points-to analysis is an *irregular* application with dynamic constraint graph, we believe, this speedup is quite remarkable.

Memory. Memory requirement of our context-sensitive parallel points-to analysis is shown in Table 8. Once again, PARALLEL-CS requires 14% more memory than SEQ-CS.

In summary, our parallel points-to analysis exploits more fine-grained parallelism from programs and promises a scalable approach to parallelizing monotonic, unordered analyses.

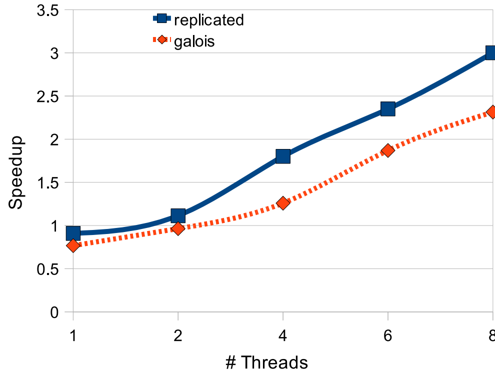


Fig. 1. Speedup comparison between our replication-based approach and Galois system

7 Related Work

Replication-Based Techniques. Replication-based techniques are prevalent in distributed systems, but their main focus is reliability [11,16]. Bal et al. [2] propose partial replication based techniques for speeding up parallelization. Their approach is based on replicating an object based on its read-write access pattern. While being applicable to parallel points-to analysis, their approach does not exploit the monotonicity property of flow-insensitive analysis, which is key to the arbitrary constraint partitioning employed by our algorithm.

Ziegler et al. [32] propose a data-flow analysis algorithm to uncover the parallelization opportunities for array replication and their temporary privatization. Their algorithm does not take advantage of any monotonic computation.

A few optimistic thread executions involve limited forms of data replication. For instance, in the Grace system [3] used for eliminating concurrency errors, threads execute optimistically and write their updates speculatively but locally. Burckhardt et al. [5] propose isolation types which can be used by threads to read and modify local copies of shared data. Our approach exploits the monotonic and unordered nature of flow-insensitive analysis to eliminate contention.

Prabhu et al. [25] develop an algorithm called EigenCFA for accelerating 0-CFA with a GPU. Similar to our analysis, EigenCFA takes advantage of the monotonicity of 0-CFA to allow stale reads. In their analysis, the same row in the representation matrix may have multiple copies of the same lambda term if they try to add the same information at the same time. They claim that this is a "rare inefficiency". In contrast, our analysis heavily depends upon this property and exploits it to improve parallelism. Further, the focus of their work is to run the analysis on a GPU whereas our focus is to run it on a multicore.

Sequential Pointer Analysis. The area of sequential points-to analysis is rich in literature. See [15] for a survey. Most scalable algorithms proposed use unification [28][10]. Steensgaard[28] proposed an almost linear time single-pass

Table 8. Context-sensitive analysis: Analysis time, Speedup and Memory

Benchmark	SEQ-CS Time(s)	Speedup					Memory (MB)	
		1	2	4	6	8	SEQ-CS	PARALLEL-CS
gcc	329.463	0.87	1.12	2.69	3.41	4.28	2859	3419
perlbmk	143.448	0.96	1.29	2.47	3.21	4.03	2133	2628
vortex	91.283	0.98	1.22	2.52	3.22	3.81	1857	2014
eon	93.495	0.96	1.31	2.72	3.41	3.98	1276	1443
parser	35.445	0.99	1.32	2.66	3.33	3.86	478	549
gap	128.478	0.98	1.27	2.50	3.13	3.77	457	514
vpr	29.456	0.93	1.22	2.69	3.08	3.64	735	770
crafty	29.337	0.99	1.28	2.51	2.96	3.46	672	736
mesa	89.388	0.98	1.27	2.01	2.68	3.32	894	949
ammp	34.236	0.96	1.10	1.89	2.77	3.15	427	447
twolf	41.499	0.98	1.10	2.10	2.69	2.90	624	696
gzip	25.234	0.92	1.08	1.74	2.56	2.98	514	641
bzip2	23.322	0.92	1.06	1.85	2.43	2.68	633	686
mcf	22.395	0.91	1.02	1.88	2.60	3.00	403	470
equake	24.306	0.90	1.08	1.96	2.75	3.23	546	610
art	26.459	0.92	1.04	1.92	2.43	2.84	597	656
httpd	224.534	0.89	1.24	2.47	3.01	3.68	991	1131
sendmail	172.743	0.91	1.28	2.58	3.10	3.57	914	1019
ghostscript	4384.238	0.93	1.30	2.66	3.11	3.81	8258	9761
gdb	9338.228	0.93	1.14	2.43	2.99	3.64	5894	6486
wine-server	201.323	0.97	1.16	2.01	2.58	3.10	774	858
average	737.539	0.95	1.19	2.28	2.91	3.44	1521	1737

algorithm that has been shown to scale to millions of lines of programs. However, a unification based approach is very imprecise. Andersen [1] proposed inclusion-based analysis that works on subsumption of points-to sets rather than a bidirectional similarity. An inclusion-based (or subset-based) analysis is more precise than a unification based analysis. However, it is also costly and has a theoretical complexity of $O(n^3)$. Several techniques [4][14][30] have been proposed to improve upon the original work by Andersen. [4] extracts similarity across the points-to sets while [30] exploits similarity across the contexts to make use of the Binary Decision Diagrams (BDD) to store information in a succinct manner. The idea of *bootstrapping* [17] first reduces the problem by partitioning the set of pointers into disjoint alias sets using a fast and less precise algorithm (e.g., [28]) and later running more precise analysis on each of the partitions. To address the analysis cost of a completely context-sensitive analysis, approximate representations were introduced to trade off precision for scalability. Das [6] proposed *one level flow*, Lattner et al. [18] unified contexts, while Nasre et al. [23,21] hashed contexts to alleviate the need to store complete context information.

Parallel Pointer Analysis. In contrast to its sequential counterpart, parallel points-to analysis is still not explored enough. The work on program decomposition identifies various program components on which different analyses can

be executed in parallel [31,27]. Bootstrapping [17] uses partitions of aliases to simulate parallel processing. However, parallelizing is not the main objective of the work and the parallelism extracted is very coarse. Lojo et al. [20,19] proposed the first parallel implementation of inclusion-based points-to analysis by exploiting the constraint graph formulation. Their analysis works on the assumption of speculative parallelism and an activity may be rolled back if a conflict is detected. In contrast, our work is general and does not require the support of speculative execution. By making multiple copies of points-to sets, our analysis strives to obtain more fine-grained parallelism.

A parallel points-to analysis for object oriented programs is proposed by Edvinsson et al. [7] which deals with different target methods of polymorphic function calls and independent control-flow branches. Maximum speedup obtained has been shown to be less than 2.0 for a set of Java benchmarks. Our parallel analysis takes advantage of monotonicity property of a flow-insensitive analysis to create multiple copies of points-to sets achieving better parallelism.

8 Conclusion

Taking advantage of the multi-core processing requires the analyses themselves to be parallel. While several enhancements have been proposed for sequential pointer analysis, enough work is yet to be done for parallel points-to analysis. By exploiting the monotonicity of flow-insensitive points-to analysis, we proposed a replication-based parallel inclusion-based points-to analysis that extracts more fine-grained parallelism from seemingly sequential programs. We showed the effectiveness of our approach over 16 SPEC 2000 benchmarks and five large open source programs. Our parallel context-insensitive (context-sensitive) analysis achieves a speedup of 3.0x (3.4x) on an 8-core machine and illustrates a promising approach to parallelizing a monotonic, unordered data-flow analysis.

Acknowledgments. We thank Mario Mndez-Lojo for helpful comments.

References

1. Andersen, L.O.: Program analysis and specialization for the C programming language, PhD Thesis, DIKU, University of Copenhagen (1994)
2. Bal, H.E., Frans Kaashoek, M., Tanenbaum, A.S., Jansen, J.: Replication techniques for speeding up parallel applications on distributed systems. Concurrency: Pract. Exper. 4, 337–355 (1992)
3. Berger, E.D., Yang, T., Liu, T., Novark, G.: Grace: safe multithreaded programming for c/c++. In: OOPSLA 2009, pp. 81–96. ACM, New York (2009)
4. Berndl, M., Lhoták, O., Qian, F., Hendren, L., Umanee, N.: Points-to analysis using bdds. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI 2003, pp. 103–114. ACM, New York (2003)
5. Burckhardt, S., Baldassin, A., Leijen, D.: Concurrent programming with revisions and isolation types. In: OOPSLA 2010, pp. 691–707. ACM, New York (2010)

6. Das, M.: Unification-based pointer analysis with directional assignments. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2000, pp. 35–46. ACM, New York (2000)
7. Edvinsson, M., Lundberg, J., Löwe, W.: Parallel points-to analysis for multi-core machines. In: Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, HiPEAC 2011, pp. 45–54. ACM, New York (2011)
8. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI 1994, pp. 242–256. ACM, New York (1994)
9. Fähndrich, M., Foster, J.S., Su, Z., Aiken, A.: Partial online cycle elimination in inclusion constraint graphs. In: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, PLDI 1998, pp. 85–96. ACM, New York (1998)
10. Fähndrich, M., Rehof, J., Das, M.: Scalable context-sensitive flow analysis using instantiation constraints. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI 2000, pp. 253–263. ACM, New York (2000)
11. Gifford, D.K.: Weighted voting for replicated data. In: SOSP 1979, pp. 150–162. ACM, New York (1979)
12. Hardekopf, B., Lin, C.: The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2007, pp. 290–299. ACM, New York (2007)
13. Harman, M., Binkley, D., Gallagher, K., Gold, N., Krinke, J.: Dependence clusters in source code. ACM Trans. Program. Lang. Syst. 32, 1:1–1:33 (2009)
14. Heintze, N., Tardieu, O.: Ultra-fast aliasing analysis using cla: a million lines of c code in a second. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI 2001, pp. 254–263. ACM, New York (2001)
15. Hind, M., Pioli, A.: Which pointer analysis should i use? In: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2000, pp. 113–123. ACM, New York (2000)
16. Joseph, T.A., Birman, K.P.: Low cost management of replicated data in fault-tolerant distributed systems. ACM Trans. Comput. Syst. 4, 54–70 (1986)
17. Kahlon, V.: Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, PLDI 2008, pp. 249–259. ACM, New York (2008)
18. Lattner, C., Lenharth, A., Adve, V.: Making context-sensitive points-to analysis with heap cloning practical for the real world. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2007, pp. 278–289. ACM, New York (2007)
19. Méndez-Lojo, M., Burtscher, M., Pingali, K.: A gpu implementation of inclusion-based points-to analysis. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2012. ACM, New York (2012)

20. Méndez-Lojo, M., Mathew, A., Pingali, K.: Parallel inclusion-based points-to analysis. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2010, pp. 428–443. ACM, New York (2010)
21. Nasre, R.: Approximating inclusion-based points-to analysis. In: Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, MSPC 2011, pp. 66–73. ACM, New York (2011)
22. Nasre, R., Govindarajan, R.: Prioritizing constraint evaluation for efficient points-to analysis. In: Proceedings of the 9th IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2011, pp. 267–276 (April 2011)
23. Nasre, R., Rajan, K., Govindarajan, R., Khedker, U.P.: Scalable Context-Sensitive Points-to Analysis Using Multi-dimensional Bloom Filters. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 47–62. Springer, Heidelberg (2009)
24. Pereira, F.M.Q., Berlin, D.: Wave propagation and deep propagation for pointer analysis. In: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2009, pp. 126–135. IEEE Computer Society, Washington, DC (2009)
25. Prabhu, T., Ramalingam, S., Might, M., Hall, M.: Eigencfa: accelerating flow analysis with gpus. In: POPL 2011, pp. 511–522. ACM, New York (2011)
26. Rountev, A., Chandra, S.: Off-line variable substitution for scaling points-to analysis. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI 2000, pp. 47–56. ACM, New York (2000)
27. Ruf, E.: Partitioning dataflow analyses using types. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1997, pp. 15–26. ACM, New York (1997)
28. Steensgaard, B.: Points-to analysis in almost linear time. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1996, pp. 32–41. ACM, New York (1996)
29. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)
30. Whaley, J., Lam, M.S.: An Efficient Inclusion-Based Points-To Analysis for Strictly-Typed Languages. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 180–195. Springer, Heidelberg (2002)
31. Zhang, S., Ryder, B.G., Landi, W.: Program decomposition for pointer aliasing: a step toward practical analyses. In: Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT 1996, pp. 81–92. ACM, New York (1996)
32. Ziegler, H.E., Malusare, P.L., Diniz, P.C.: Array Replication to Increase Parallelism in Applications Mapped to Configurable Architectures. In: Ayguadé, E., Baumgartner, G., Ramanujam, J., Sadayappan, P. (eds.) LCPC 2005. LNCS, vol. 4339, pp. 62–75. Springer, Heidelberg (2006)

A New Method for Program Inversion

Cong Hou[1], George Vulov[1], Daniel Quinlan[2], David Jefferson[2], Richard Fujimoto[1], and Richard Vuduc[1]

[1] Georgia Institute of Technology, Atlanta, GA 30332
[2] Lawrence Livermore National Laboratory, Livermore, CA 94551
{hou_cong,georgevulov}@gatech.edu
{dquinlan,jefferson6}@llnl.gov
{fujimoto,richie}@cc.gatech.edu

Abstract. Program inversion has been successfully applied to several areas such as optimistic parallel discrete event simulation (OPDES) and reverse debugging. This paper introduces a new program inversion algorithm for imperative languages, and focuses on handling arbitrary control flows and basic operations. By building a value search graph that represents recoverability relationships between variable values, we turn the problem of recovering previous values into a graph search one. Forward and reverse code is generated according to the search results. We have implemented our algorithm as part of a compiler framework named Backstroke, a C++ source-to-source translator based on ROSE compiler. Backstroke targets optimistic simulation codes and automatically generates a reverse function to recover values modified by a target function. Experimental results show that our method is effective and produces better performance than previously proposed methods.

Keywords: Program inversion, SSA, SSA graph, reverse computation, state saving, ROSE.

1 Introduction

We consider the problem of how to generate an efficient *inverse* of a program. Informally, suppose a program P begins in state I and ends in state F. Then, an inverse, P^{-1}, reproduces the initial state I when started in state F. State I may not be uniquely reproducible given state F, hence P may have to be instrumented so that the original state I can be restored. The challenge in program inversion is how to instrument P and construct P^{-1} so that, when executed, they incur minimal storage and time overheads. This paper describes novel program analysis and code generation techniques for automatically building the instrumented P and a space- and time-efficient P^{-1}.

Program inversion has numerous applications, but our focus is on *optimistic* parallel discrete event simulations (OPDES). In this context, parallelization is achieved by speculatively executing each event in parallel and using rollback mechanisms to undo any events that have executed out of order [16,14].[1] In

[1] Every event has a *timestamp*, which establishes a total order on events [14].

M. O'Boyle (Ed.): CC 2012, LNCS 7210, pp. 81–100, 2012.
© Springer-Verlag Berlin Heidelberg 2012

a naïve implementation of OPDES, we might instrument P to save the initial values of all variables that it modifies, so that P^{-1} simply restores those initial values. However, for events that manipulate a significant amount of state, this approach could incur significant overheads in both time and storage.

The alternative approach that we consider is *reverse computation* [8]. The idea is to minimally instrument P to store just enough control and data information so that, on rollback, a P^{-1} can computationally reconstruct the initial state through algebraic manipulations. Since on modern systems simple algebraic operations are much cheaper than memory accesses, reverse computation can be much more efficient than naïvely saving state. Figure 1 shows an example of an event, and its instrumented forward and reverse (inverse) versions (`a` and `b` are two state variables); the forward and reverse versions were generated by our algorithm. Rather than save both state variables, we can through reverse computation store just `b` in one branch.

```
int a, b;
void foo() {
  if (a == 0)
    a = 1;
  else {
    b = a + 10;
    a = 0;
  }
}
```

(a)

```
void foo_forward() {
  int trace = 0;
  if (a == 0) {
    trace |= 1;
    a = 1;
  }
  else {
    store(b);
    b = a + 10;
    a = 0;
  }
  store(trace);
}
```

(b)

```
void foo_reverse() {
  int trace;
  restore(trace);
  if ((trace & 1) == 1)
    a = 0;
  else {
    a = b - 10;
    restore(b);
  }
}
```

(c)

Fig. 1. (a) The original event (b) The forward event (c) The reverse event

Contributions. We build a graph that shows explicit relationships between values and allows us to restore values by searching the graph; costs on the edges of this graph correspond to memory overheads in the generated code. This approach allows us to flexibly mix state saving and reverse execution depending on which is more efficient; other approaches focus on either one or the other. Although the search problem is NP-complete, we provide heuristics that work well in practice. We have implemented our approach in the reverse compiler Backstroke, which automatically generates forward and reverse code as in Figure 1.

Limitations. In this paper we only target programs with scalar data types and without function calls. We do not address aliasing, arrays, and structured types; however we think the graph search approach to program inversion can be extended to these scenarios and provide similar benefits to the scalar case. While our cost model accounts only for memory costs, our approach is transparent to the cost model chosen; in the future, a more sophisticated cost model can be built to include other kinds of overhead.

2 Related Work

Most of the work on inverting arbitrary (non-injective) imperative programs has focused an incremental approach: the imperative program is essentially executed in reverse, with each modifying operation in the original execution being undone individually. For example, if statements $s_1 s_2 \dots s_n$ are executed in the forward directions, the reverse function executes statements $s_n^{-1} \dots s_2^{-1} s_1^{-1}$. The incremental approach cannot handle unstructured control flows and is difficult to apply with early returns from functions; the approach presented in this paper suffers from neither of these shortcomings. Furthermore, the incremental inversion restores the initial state by restoring every intermediate program state between the final state and the initial state, even though these states are not needed.

Among the incremental inversion approaches, syntax-directed approaches apply only statement-level analysis. If an assignment statement is lossless, its inverse is used: for example, the inverse of an integer increment is an integer decrement. Otherwise, the variable modified in the assignment has to be saved. An early example of syntax-directed incremental inversion is Brigg's Pascal inverter [7]. This approach was later extended to C and applied both to optimistic discrete event simulation [8] and reversible debugging [6].

Akgul and Mooney introduced a more sophisticated incremental inversion algorithm that uses def-use analysis to invert some assignment statements that are not lossless [2]; we refer to this approach as regenerative incremental inversion. In order to reverse a lossy assignment to the variable a, such as $a \leftarrow 0$, the regenerative algorithm looks for ways to recompute the previous value of a. One technique to obtain the previous value of a is to re-execute its definition; another technique is to examine all the uses of a and see if its value can be retrieved from any of its uses. These two techniques are applied recursively whenever a modifying operation is to be reversed; if they fail to produce a result, the overwritten variable is saved during forward execution. Our approach takes advantage of all the def-use relationships utilized by regenerative inversion, without suffering from the drawbacks of incremental inversion. In addition to def-use information, our approach also derives equality relationships between variables from the outcome of branching statements that test for equality or inequality.

A related line of work is inverting programs that are injective, without using any state saving. Most such work focuses on inverting functional programs [1,15,17]. Approaches to inverting imperative programs include translation to a logic language [20] and template-based generation of inverse candidates using symbolic execution [21].

3 Problem Setup

Let the set of *target variables* be $S = \{s_1 \dots s_n\}$ with *initial values* $V = \{v_1 \dots v_n\}$, where v_i is the initial value of s_i. These variables are modified by a *target function*[2] M, producing $V' = \{v'_1 \dots v'_n\}$, the *final values* of the target

[2] The function here is a C/C++ function, not a function in mathematics.

variables. Our goal is generating two new functions, the *forward function* M^S_{fwd} and the *reverse function* M^S_{rvs}, so that M^S_{fwd} transfers V to V', and M^S_{rvs} transfers V' to V. We define *available values* as values which are ready to use at the beginning of M^S_{rvs}. For example, values in V' and constants are available values. We also call values in V *target values* which are values we want to restore from M^S_{rvs}.

Note that M and M^S_{fwd} have the same input and output, but M^S_{fwd} is instrumented to store control flow information and values that are later used in M^S_{rvs}. This introduces two kinds of cost that must be considered when generating the forward-reverse pair $\{M^S_{fwd}, M^S_{rvs}\}$: extra memory usage and run-time overhead.

4 Reversing Functions without Loops

4.1 Framework Overview

We will first treat the inversion of loop-free code with only scalar data types, without aliasing. When such code is converted to static single assignment (SSA) form [11], each versioned variable is only defined once and thus there is a one-to-one correspondence between each SSA variable and a single value that it holds. We will also take advantage of the fact that loop-free code has a finite number of paths. Loops will be discussed in the next section, and non-scalar data types and aliasing will not be handled in this paper.

Given a cost measurement, for each path in the target function there should exist a best strategy to restore target values. Strategies usually vary among different paths. Therefore, the reversed function we produce should include the best strategy for each path; each path in the original function should have a corresponding path in the reverse function.

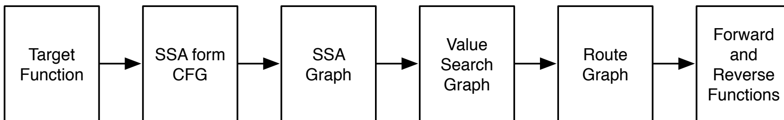


Fig. 2. Overall framework of the inversion algorithm

To restore target values, we will build a graph which shows equality relationships between values. We call this graph the *value search graph*, and it is built based on an SSA graph [3,10]. Then a search is performed on the value search graph to recursively find ways to recover the set of target values given the set of available values. If there is more that one way to restore a value, we choose the one with the smallest cost. The search result is a subgraph of the value search graph which we call a *route graph*. For any path, a route graph shows a specific way to recover each target value from available values. Finally, the forward and reverse functions are built from a route graph. Figure 2 illustrates this process.

4.2 Building the Value Search Graph

We first build an SSA graph for the target function. An SSA graph [3,10], built based on SSA form, consists of vertices representing operators, function symbols, or ϕ functions, and directed edges connecting uses to definitions of values. It shows data dependencies between different variables. The full algorithm for building an SSA graph is presented in [19]. Figure 3(a)(b) show the SSA-transformed CFG and its SSA graph for the function in Figure 1(a). In this example, a and b are two target variables with initial values $\mathtt{a}_0$ and $\mathtt{b}_0$, and final values $\mathtt{a}_3$ and $\mathtt{b}_2$.

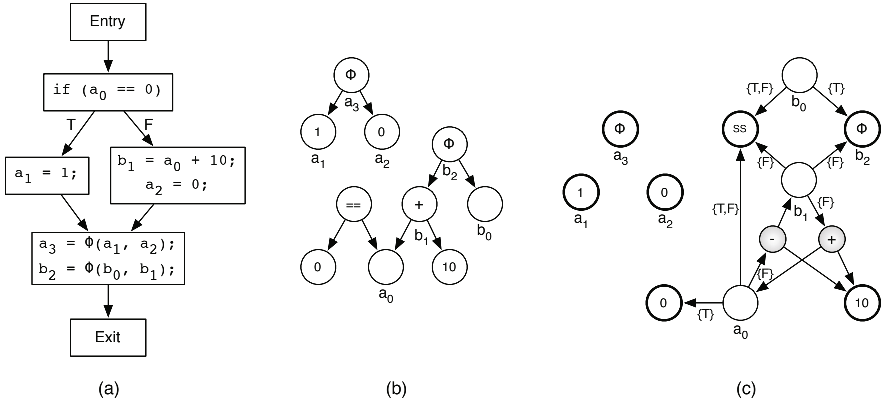


Fig. 3. (a) The SSA-transformed CFG of the function in Figure 1(a) (b) The corresponding SSA graph (c) The corresponding value search graph. Nodes with bold outlines are available nodes; outgoing edges for these nodes are omitted because available nodes need not be recovered. 'SS' is the special state saving node. Edges are annotated with their CFG path set.

A *value search graph* enables efficient recovery of values by explicitly representing equality relationships between values. Unlike an SSA graph, operation nodes are separated from value nodes in the value search graph, since their treatment is different for recovering values. An edge connecting two value nodes u and v implies that u and v have the same value. An edge from value node u to an operation node *op* means that u is equal to the result of evaluating *op* with its operands. To recover the value associated with node v, we can recursively search the graph starting at v.

We attach a set of CFG paths to each edge in a value search graph, meaning the edge is applicable only if one of the CFG paths in that set is selected in the original function. For operation nodes in the SSA graph, let the set of paths attached to each outgoing edge be the CFG paths for which the corresponding operation is executed. Similarly, for ϕ nodes, each reaching-definition edge should be annotated with all CFG paths for which the corresponding reaching definition reaches the ϕ function. We will describe an implementation of the path set representation later.

During the execution of the forward function, once a variable is assigned with a new value, its previous value may be destroyed and cannot be retrieved. To guarantee that a search in the value search graph can always restore a value, we introduce special *state saving edges*. The idea behind these edges is that each value may be recovered by storing it during the forward execution. Whenever a state saving edge appears in the search results, the forward function is instrumented to save the corresponding value. The path set associated with a state saving edge for a value node v is the set of all paths that include v's definition. All state saving edges point to a unique *state saving node*.

We apply the following rules to convert an SSA graph into a value search graph:

- For simple assignment $v = w$, there is a directed edge from v to w in the SSA graph. Since we can retrieve w from v, add another directed edge from w to v with the same path set.
- A ϕ node in the SSA graph has several outgoing edges connecting all its possible definitions. For each of those edges, add an opposite edge with the same path set.
- For each operation node in the SSA graph, split it into an operation node and a value node, with an edge from the value node to the new operation node. The new operation node takes over all outgoing edges, and the value node takes over all incoming edges.
- If an equality operation (`==`) is used as a branching predicate and its outcome is true, we know that the two operands are equal. Therefore, we add edges from each operand to the other, with a path set for the edge equal to the path set of the *true* CFG edge out of the branch. We add the edges analogously for a not-equal operation (`!=`), but with the path set from the *false* side of the branch.
- For every value that is not available, insert a state saving edge from the corresponding value node to the state saving node.

Lossless operations. For certain operations, such as integer addition and exclusive-or, we can recover the value of an operand given the operation result and the other operand. For example, if $a = b + c$, we can recover b given a and c. For each such lossless operation, insert new operation nodes that connect its result to its operands, allowing the operands to be recovered from the result. The new nodes are added according to the following rules:

- Negation (`a = -b`) and bitwise not (`a = ~b`): the new operations are `b = -a` or `b = ~a`, respectively.
- Increment (`++a`) and decrement (`--a`): insert `--a` or `++a`, respectively.
- Integer addition (`a = b + c`) and subtraction: for addition, the new operations are `b = a - c` and `c = a - b`; analogously for subtraction.
- Bitwise exclusive-or (`a = b ^ c`): insert `b = a ^ c` and `c = a ^ b`

There are two special types of nodes in a value search graph: *target nodes* are value nodes containing target values, and *available nodes* are value nodes containing available values plus the state saving node. As an optimization, we never

create any outgoing edges for an available node. Figure 3(c) shows the value search graph built for the code in Figure 1(a). The available nodes are shown with a bold outline. Since the function only has two paths, we use labels 'T' and 'F' to represent the CFG paths passing through the true and false body in the target function, respectively. The '–' operation node connecting a_0 to b_1 and the constant value '10' is generated from the '+' operation. The edge from a_0 to '0' for the path 'T' is added based on the fact that $a_0 = 0$ on that path. The 'SS' node in the graph is the state saving node, and all unavailable nodes are connected to it. From the value search graph, we can find two valid ways to restore b_0 for the path 'T': b_0 to SS node and b_0 to b_2. Obviously the second one is better since it avoids a state saving operation, and this better selection will be produced from the search algorithm described later.

4.3 The Route Graph

A *route graph* is a subgraph of a value search graph connecting all target nodes to available nodes. Each route graph represents one way to restore the target values, and there may exist many valid route graphs for the same set of target values. Edges in the route graph may have different path sets than the corresponding edges in the value search graph. For each edge e in a route graph, let $P(e)$ denote the set of CFG paths that the edge is annotated with. The following properties guarantee that the route graph properly restores all target values:

I) Let $\mathcal{U}$ be the set of all CFG paths. Then, for each target node t,

$$\bigcup_{out \in \text{OutEdges}(t)} P(out) \quad = \quad \mathcal{U}$$

II) For each node n that is neither a target node nor an available node,

$$\bigcup_{out \in \text{OutEdges}(n)} P(out) = \bigcup_{in \in \text{InEdges}(n)} P(in)$$

III) For each value node n, given any two outgoing edges $n \to p$ and $n \to q$, $P(n \to p) \cap P(n \to q) = \emptyset$

IV) If e is a route graph edge and its corresponding edge in the value search graph is e', then $P(e) \subseteq P(e')$

V) For each directed cycle with edges $e_1 \dots e_n$, $\quad \bigcup_{i=1}^{n} P(e_i) = \emptyset$

Property I specifies that each target value is recovered for every CFG path. Property II means that each value is recovered exactly for the paths for which it is needed. Property III requires that for each CFG path, there is at most one way to recover a value. Property IV requires that the set of CFG paths associated with an edge in the route graph is a subset of the CFG paths originally associated with that edge in the value search graph. Finally, property V forbids self-dependence: restoring a value cannot require that value.

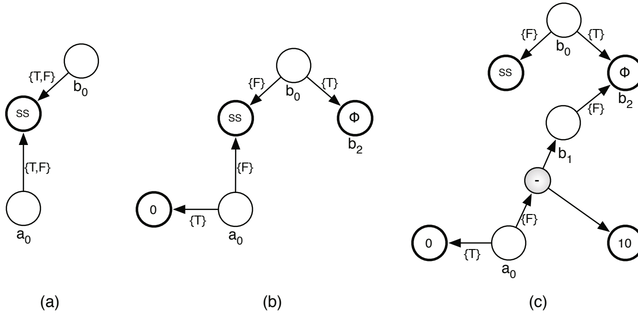


Fig. 4. Three different route graphs for the target values a_0 and b_0 given the the value search graph in Figure 3(c)

Figure 4 shows three valid route graphs for the value search graph in Figure 3. Route graph 4(a) only includes state saving edges. Route graph 4(b) takes advantage of the fact that for the 'T' path the values of both $\texttt{a}_0$ and $\texttt{b}_0$ are known; it only uses staving for the 'F' path. Route graph 4(c) improves upon route graph 4(b) by recomputing $\texttt{a}_0$ as $\texttt{b}_1\texttt{-10}$ for the CFG path 'F'; state saving is only applied to $\texttt{b}_0$ for path 'F'.

4.4 Searching the Value Search Graph

Costs in Route Graphs. As we have seen in Figure 4, there may be multiple valid route graphs that recover the target values, but with different overheads. In order to choose the route graph with the smallest overhead, we must define a cost metric.

Generally, there are two kinds of overhead in forward and reverse functions: execution speed and additional memory usage; we only consider the storage costs. State saving contributes the most to the overhead memory usage and it also significantly affects the running time of both forward and reverse functions. Storing the path taken during forward execution is the other factor that contributes to memory usage; this overhead is bounded and is the same for all route graphs, so we exclude it from our cost estimate. With each state saving edge in the value search graph, we associate a cost equal to the size of the value that must be saved; other edges have cost 0. The cost of a route graph for a specific CFG path is the sum of the cost of those edges whose annotated path sets include that CFG path.

In Figure 4, suppose the cost to store and restore either $\texttt{a}$ or $\texttt{b}$ is c, the following table shows the cost of three route graphs for each CFG path. Obviously the third route graph is the best one.

CFG path	route graph (a)	route graph (b)	route graph (c)
T	$2c$	0	0
F	$2c$	$2c$	c

We have defined the cost of a single CFG path; however, a route graph may have different costs for different CFG paths. When searching the value search

graph, we would like to treat groups of CFG paths that share some edges in the route graph together, rather than performing a full search for each CFG path. For this reason, the search algorithm partitions the CFG paths into disjoint sets of paths that have equal cost and we save the cost for each set of paths independently. In our search algorithm, we denote the costs of a route graph r as r.costSet.

$$r.\text{costSet} = \{\langle P_i, c_i \rangle | P_i \text{ is a set of CFG paths and } c_i \text{ is the cost}\}$$

Search Algorithm. Our search algorithm should aim to find a route graph that has the minimum cost for each path. Theoretically, however, searching for a minimal route graph is an NP-complete problem. To make the problem tractable, we apply the heuristic of finding a route graph for each target value individually; the individual route graphs are then merged into a route graph that restores all the target values. Similarly, in order to recover the value of a binary operation node, we recover each of the two operands independently and then combine the results.

The pseudocode for our heuristic search algorithm is presented in Algorithm 1. The `SearchSubRoute` function returns a route graph given a target node, the paths for which that node must be restored, and the set of value nodes visited so far. The algorithm explores all ways to recover the current node by calling itself recursively on all the nodes that are directly reachable from the current node; available nodes are the base case. Lines 5–10 handle recovering the values of operation nodes. In order to recover the value of an operation node, each of its operands must be recovered. Lines 11–14 return a trivial route graph for available nodes, with a cost of 0. The remaining body of the algorithm (lines 15–27) handles recovering a value node that is not available. Each of the out-edges of the target node may be used to recover its value for the CFG paths associated with that edge; these edges are explored in the `for`-loop in lines 15–23. The variable `newPaths` on line 17 represents the set of paths that we are both interested in and are associated with the current edge. In line 19, we recursively find a route graph that recovers the target value by recovering the target of the current outgoing edge. Lines 21–22 update the cost sets of the new route graph; if it provides a lower cost for some CFG path than the solutions found so far, the partial results are modified so that each CFG path is restored with the cheapest route graph. Finally, the route graph from line 19 is added to the list of partial results (line 23). After all out-edges of the target node have been explored, the partial results are merged into a single route graph and returned (lines 24–27). Note that it is unnecessary to check whether the target node has been successfully recovered, since the state saving edge always provides a valid route graph for the node. Figure 4(c) shows the route graph produced by the algorithm when searching the value search graph from Figure 3(c).

The search algorithm enforces properties I–IV from section 4.3 during its execution. To make sure that the search result does not contain cycles (property V), we record which value nodes are already in the route using a set `visited` in Algorithm 1. This alone is not sufficient to guarantee that the result is acyclic,

Algorithm 1. Searching for a route graph in a value search graph

```
   Initial input: The search start point target, with paths = ∅, visited = ∅
 1 SearchSubRoute(target, paths, visited)
 2 begin
 3     resultRoute ← ∅, subRoutes ← ∅
 4     if target is an operation node then
 5         foreach edge ∈ OutEdges(target) do
 6             if edge.target ∈ visited then return ∅
 7             newRoute ← SearchSubRoute(edge.target, paths, visited)
 8             if newRoute = ∅ then return ∅
 9             add edge and newRoute to resultRoute
10         return resultRoute
11     if target is available then
12         add target to resultRoute
13         add ⟨paths, 0⟩ to resultRoute.costSet
14         return resultRoute
15     foreach edge ∈ OutEdges(target) do
16         if edge.target ∈ visited then continue
17         newPaths ← edge.pathSet ∩ paths
18         if newPaths = ∅ then continue
19         newRoute ← SearchSubRoute(edge.target, newPaths, visited ∪ {target})
20         add edge with paths newPaths to newRoute
21         foreach ⟨paths, cost⟩ in newRoute.costSet do cost += edge.cost
22         foreach route in subRoutes do ChooseMinimalCosts(route, newRoute)
23         add newRoute to subRoutes
24     add target to resultRoute
25     foreach route in subRoutes do
26         if route.pathSet ≠ ∅ then add route to resultRoute
27     return resultRoute

28 ChooseMinimalCosts(route1, route2)
29 begin
30     if route1.pathSet ∩ route2.pathSet = ∅ then return
31     foreach ⟨paths1, cost1⟩ in route1.costSet do
32         foreach ⟨paths2, cost2⟩ in route2.costSet do
33             if paths1 ∩ paths2 = ∅ then continue
34             if cost1 > cost2 then
35                 paths1 ← paths1 − paths2
36                 Remove (paths1 ∩ paths2) from all edges of route1
37             else
38                 paths2 ← paths2 − paths1
39                 Remove (paths1 ∩ paths2) from all edges of route2
40     route1.pathSet = ⋃_{⟨paths,cost⟩∈route1.costSet} paths
41     route2.pathSet = ⋃_{⟨paths,cost⟩∈route2.costSet} paths
```

for there may be two different paths with identical cost to recover a single value node. If one way is chosen to recover a value node v during path of the search, and then later v is recovered differently for the same CFG path, a cycle may form. To prevent this situation from occurring, we always traverse out-edges in the same order of line 19 of Algorithm 1; the first route graph with the smallest cost is chosen. In addition, two paths coming from two different value nodes may also form a cycle when all costs on edges of the cycle are 0. We eliminate this possibility by replacing 0 by a small cost ε.

4.5 Instrumentation and Code Generation

Representing CFG Path Sets. Our search algorithm relies on efficiently computing intersection, union, and complement of CFG path sets, as well as testing whether the set of paths is empty; for this reason we suggest implementing the set representations as bit vectors. Ball and Larus [5] present an path profiling method in which each path is given a number from 0 to $m - 1$, where m is the count of the CFG paths. We use their algorithm to number each path, and for each path we associate exactly one bit in the bit vector used to represent a path set.

Recording CFG Paths. We need to store path information in a way that allows us to efficiently record the CFG path taken (for forward execution), and to efficiently check if the path matches a given set of CFG paths attached to a route graph edge (for reverse execution). However, if we encode each path using its path number, then examining whether a path is a member of a set is inefficient. Instead we use a bit vector to record the CFG path, in which each bit represents the outcome of a branching statement. Since this method is similar to *bit tracing*[4], we call this bit vector a *trace*. Note that two branches may share the same bit if they cannot appear in the same path. Thus, the number of bits required to store the path taken is equal to the largest number of branches that appear on a single CFG path. Algorithm 2 calculates bit-vector position for each branch node accordingly.

In the forward function, we use an integer as the bit vector to record all predicate results[3]. Let `trace` be the variable recording a trace, initialized to zero; then the true edge of each branch node v is instrumented with the statement [4]

`trace = trace | (1 << ` *position*(v)`);`

where *position*(v) is calculated by Algorithm 2. The variable `trace` is stored at the end of the forward function and restored at the beginning of the reverse function. Note that we can further optimize the instrumentation by moving a trace updating operations downward through the CFG and merging them.

[3] Potentially we could omit recording predicates that do not affect the reverse function.

[4] We use several operators in C/C++ syntax here and below, which includes bitwise OR operator `|`, bitwise AND operator `&`, bitwise left shift operator `<<`, equal to operator `==`, and logical OR operator `||` .

Algorithm 2. Generating the bit position for each branch node

```
foreach CFG node u in reverse topological order do
    if u is a leaf node then
        position(u) ← -1
    else if u is a branch node then
        /* u → v and u → w are its two out-going edges                    */
        position(u) ← max(position(v), position(w)) + 1
    else
        /* u → v is its out-going edge                                     */
        position(u) ← position(v)
```

In the reverse function, we must test if `trace` matches the path sets that appear on route graph edges. We start with transforming each path in the set into a trace (the trace for each path can be computed by the same means as recording a trace in the forward function). Then, checking if a path set contains a path represented by `trace` is done by comparing it to each trace. Suppose a path set containing two paths is transformed into two traces `01101` and `01001`. Instead of comparing `trace` to each of them as:

```
if (trace == 01101 || trace == 01001)
```

we can simplify this predicate by using a mask `11011` on `trace`:

```
if ((trace & 11011) == 01001)
```

The combined trace for `01101` and `01001` is `01`×`01`, where × denotes that the bit does not matter. Given a set of traces, we can combine pairs repeatedly to reduce the size of the set. This greatly reduces the complexity of the branching statements in the reverse code.

Algorithm 3 starts out with all traces corresponding to a set of CFG paths and merges them into a minimal set of traces that can be used to test membership in the set. The intuition behind Algorithm 3 is that if the traces are sorted so that bit i is the least significant bit, the traces that are identical to each other except for bit i will be adjacent. However, if we are careful we don't have to pay the full sorting cost for each bit i. If the traces are sorted when their bits are considered in the order $b_1 b_2 \dots b_{i-1} \quad b_k b_{k-1} \dots b_i$ and we want to sort them according to the bit order $b_1 b_2 \dots b_{i-2} \quad b_k b_{k-1} \dots b_{i-1}$, we need only sort each sequence of the trace for which bits 1 through $(i-2)$ are identical. For each such sequence, there are at most three sorted subsequences, indexed by bit b_{i-1}; these can be merged in linear time (similarly to mergesort). If we use a linear-time sort, such as radix sort, for the first iteration, the overall runtime of Algorithm 3 is $O(kn)$, where n is the size of the path set.

After the merge, if we have n traces $t_1, ..., t_n$ for a path set, the resulting predicate would be:

$$\texttt{if ((trace \& } mask_1\texttt{) == } obj_i \texttt{ || ... || (trace \& } mask_n\texttt{) == } obj_n\texttt{)}$$

Algorithm 3. Merging a set of path traces

```
MergePathTraces(traces)
begin
    /* Each trace has k "bits", and each bit is 0, 1, or ×              */
    /* Bits are numbered ascendingly; e.g. m = b1b2...bk                */
    for i ← k down to 1 do
        /* Note: for i = k, the bit ordering is m = b1b2...bk           */
        Sort traces, where trace bits are ordered b1b2...b(i−1)  bkb(k−1)...bi
        for j ← 2 to Length(traces) do
            if traces[j − 1] and traces[j] match except for bit i then
                set bit i to × for traces[j − 1]
                delete traces[j]
```

For each trace t_i, $mask_i$ is obtained by setting all bits which are $\times$ in t_i to 0 and others to 1, and obj_i equals $mask_i$ & t_i.

Inserting State Saving Statements. The other instrumentation in the forward function are state saving statements, which are inserted according to the state saving edges in the route graph. For each state saving edge in the route graph, suppose the variable to store is *var* and the path set on this edge is P. Our task is finding one or several locations to store *var* according to the path set P, ensuring that *var* is only saved once for each CFG path in P.

To find such locations, we first compute the corresponding path traces T of P from Algorithm 3. For each trace in T, we traverse the CFG from the entry. When we reach a branch node, check the corresponding bit in the trace: fall through the true edge if the bit is 1, false edge if the bit is 0. If the bit is $\times$, the traversal forks and that bit is assigned to 0 and 1 respectively forming two new traces; and for each concretized trace the descent continues. The descent stops immediately when all bits which are not checked in the trace are $\times$. After this process, we obtain one or more locations where the descent has stopped. In each location we find a point where the definition of *var* is reachable and a state saving statement is inserted there. However, it is possible that the path set containing the paths passing through this location is larger than the one on which the state saving is needed. In this case, we guard the state saving statement with a branch whose predicate corresponds to the trace at this location.

Building a CFG for the Reverse Function. We build the CFG for the reverse function from a route graph; the reverse CFG is acyclic and each path in it must obey the data dependencies represented in the route graph. Each outgoing edge from a value node in the route graph will be translated to a statement in the reverse function.

There could be a large number of correct reverse CFGs for a route graph, resulting in different control flows and different numbers of branches. We choose

to build a structured CFG to simplify the translation to source code. We also attempt to minimize the number of predicates in the CFG.

There are three kinds of statements that can be generated from a route graph:

- An operator node with its operands and result induces an operation statement, such as `a = b + c`.
- An edge with value nodes as both ends induces an assignment statement.
- An edge pointing to the SS node induces an value restoration statement.

The statements generated from route graph edges retain the path sets attached to the corresponding edges. We build basic blocks of statements that all share the same path sets, and insert branches so that each basic block is executed when the corresponding path is taken in the forward function. While enforcing the path set constraints ensures correct control flow, producing correct data flows depends on the order in which statements are inserted in the CFG. Note that a route graph corresponds to explicit data dependencies, and for each CFG path in the forward function it is acyclic due to property V from section 4.3. Hence, if we order statements in the reverse topological order of the route graph edges, dataflow dependencies are correctly maintained.

Algorithm 4 shows how to build a CFG for the reverse function. We keep a set of basic blocks, `openBlocks`, to which new statements can be appended. We also maintain a set of statements, `pendingStmts`, whose data dependencies have been satisfied, but which have not yet been inserted in the CFG. Each basic block has an associated path set; these are the paths in the forward function for which the corresponding basic block in the reverse function should execute. Similarly, each statement has a set of paths from the forward function. If there is a pending statement and an open basic block whose path sets match, we simply append the statement to the basic block. When a statement is inserted into the CFG, the data dependencies of new statements may now be satisfied; we call the function `BuildReadyStatements` to generate the statements that are now valid for insertion. If there is no pending statement whose path set matches the path set of an open basic block, we must insert or join a branch in the CFG. When a branch is inserted, two new basic blocks are created and the basic block containing the branch is closed. When a branch is joined, the joined basic blocks are closed and a new open basic block is created.

Note that it is possible that the instrumentation to the forward function brings additional implicit data dependencies. For example, if stack is used for state saving the order of values popped in the reverse function should be opposite of the order of pushes in the forward function. In this case, we can order those state saving statements in `pendingStmts` according to the order in which values are pushed.

Generating Code. The forward function is generated by copying the target function and adding state saving and control flow instrumentation (section 4.5). The reverse function is translated from the CFG built by Algorithm 4. Translating a structured CFG to source code is straightforward. Since each variable in the reverse CFG is in SSA form, we can use the versioned name during code

Algorithm 4. Generating a CFG for the reverse function from a route graph

```
GenerateReverseCFG(routeGraph)
begin
    cfg ← ∅, pendingStmts ← ∅, openBlocks ← ∅, pathSetPairs ← ∅
    foreach valNode in routeGraph do valNode.pathSet ← ∅
    foreach available node availNode in routeGraph do
        BuildReadyStatements(availNode, 𝒰, pendingStmts)
    cfg.entry ← BuildBasicBlock(𝒰)
    while pendingStmts ≠ ∅ do
        if ∃s ∈ pendingStmts, b ∈ openBlocks, and s.pathSet = b.pathSet then
            Append s to b
            valNode ← the source node of the edge that generated s
            BuildReadyStatements(valNode, s.pathSet, pendingStmts)
        else if ∃s ∈ pendingStmts, b ∈ openBlocks, and s.pathSet ⊂ b.pathSet
        then
            Append to b a branch, with the predicate generated from s.pathSet
            b1 ← BuildBasicBlock(s.pathSet)
            Append s to b1
            b2 ← BuildBasicBlock(b.pathSet − s.pathSet)
            Insert into cfg edges from b to b1 and b2 with labels true and false
            Add ⟨b1.pathSet, b2.pathSet⟩ to pathSetPairs
            openBlocks ← openBlocks − {b}
            valNode ← the source node of the edge that generated s
            BuildReadyStatements(valNode, s.pathSet, pendingStmts)
        else if ∃b1, b2 ∈ openBlocks, and ⟨b1.pathSet, b2.pathSet⟩ ∈ pathSetPairs
        then
            b ← BuildBasicBlock(b1.pathSet ∪ b2.pathSet)
            Insert into cfg two edges, from b1 and b2 to b
            pathSetPairs ← pathSetPairs − {⟨b1.pathSet, b2.pathSet⟩}
            openBlocks ← openBlocks − {b1, b2}
            if |openBlocks| = 1 then break
    return cfg

BuildReadyStatements(valNode, nodeAvailablePaths, pendingStmts)
begin
    valNode.pathSet ← valNode.pathSet ∪ nodeAvailablePaths
    foreach edge ∈ InEdges(valNode) do
        if edge.pathSet ⊆ valNode.pathSet then
            if edge.source is an operation node then
                Set edge to be a available for edge.source
                if all operands of edge.source are available then
                    Add to pendingStmts the statement for for edge.source, with
                    path set edge.pathSet
            else
                Add to pendingStmts the statement for edge, with path set
                edge.pathSet

BuildBasicBlock(pathSet) begin
    Build an empty basic block b and attach path sets pathSet to it.
    cfg ← cfg ∪ { b },     openBlocks ← openBlocks ∪ { b }
    return b
```

generation. Because our framework generates source code that is later compiled with another compiler, the redundant variables will be optimized away; the only drawback of this approach is readability. If readability is an issue, we can compute data dependencies in the reverse CFG and then remove versions attached to variables where this does not affect data dependencies. After version removal, we would also remove self-assignment statements such `a = a`. Figures 1(b) and 1(c) show the generated forward and reverse functions from the code in Figure 1(a).

5 Handling Loops

Our discussion of loops only considers natural loops with only one entry; loops with more than one entry are quite rare in practice and can be transformed into natural loops [19]. Consider a loop that takes n iterations and modifies a value. There are two approaches to restoring that value. The first one is generating another loop in the reverse function which contains the inverse of the loop body and also executes n times. The other approach is forgoing generating a loop and using other methods such as state saving. We refer to the first approach as the loop solution, and to the second as the non-loop solution.

Although the loop solution may be able to restore a value without state saving, there are two pitfalls with this approach. First, a natural loop only has one entry, but may have several exits (e.g. `break` and `return`). The loop exits may point to anywhere topologically after the loop, so the inverse of a loop body may have several entries with varying reaching definitions. Second, recovering a value through reverse iterations may be less efficient than state saving. Memory storage inside a loop, either for state saving or recording control flow, is multiplied by the number of iterations. Even without memory overheads inside a loop, it may be faster to just save and restore a value than to recompute it through a long iteration.

In this paper we only deal with scalar variables, which seem unlikely to benefit from the loop solution. For loops containing arrays and function calls, the loop solution may be better or even necessary. Space restrictions preclude describing both methods. We will describe the non-loop solution below and show the brief idea of the loop solution.

Non-loop solution. In SSA form each variable has only one definition, but if that definition is in a loop, the versioned variable no longer represents a single value; the algorithm from section 4 is not applicable directly. Our non-loop solution is removing value nodes containing definitions in loops (including the definition from a ϕ function in the loop header) when building the value search graph. Besides, each loop is reduced into a single node, and the loop-free algorithm applies.

Loop solution. The loop solution applies a transformation on the loop to make it only have one exit. This is done by separating the last iteration from others, since it is the only iteration that may exit the loop. Afterwards, the loop

header becomes both the entry and exit of each iteration, and the loop as a region becomes a hammock [12]. We then build the value search graph for this hammock and embed it to the value search graph of the whole method. The search algorithm still determines how to restore each value according to the cost on edges.

6 Experiment Results

We have implemented the framework in our C/C++ source-to-source translator Backstroke based on the ROSE compiler. Since this paper focuses on arbitrary control flows and basic operations with only scalar data types, instead of trying to reverse real-world code, which usually includes function calls, non-scalar data types, aliasing, etc., we employ some representative synthetic benchmarks to illustrate the power of our algorithm. Those benchmarks are listed below.

- **NoBranch**: A variable is modified in the function.
- **Branches1**: There are many CFG paths in the function and only one variable is modified on one path.
- **Branches2**: There are many CFG paths in the function and on each path a distinct variable is modified.
- **Branches3**: There are many CFG paths in the function and a variable is modified up to three times on some paths and is not modified on other paths.
- **Loop1**: A loop in which a variable is modified. The loop is intended to have many iterations at runtime.
- **Loop2**: A loop containing a simple branch and two variables are modified in the true and false body respectively. The loop is intended to have many iterations at runtime.

In addition, each benchmark has two versions in which every variable is modified differently: in the first one, each variable is modified by an assignment; the other one modifies each variable using an increment operation (`++`) so that the assignment can be reversed trivially. We denote those two versions by **Assignment** and **Increment**.

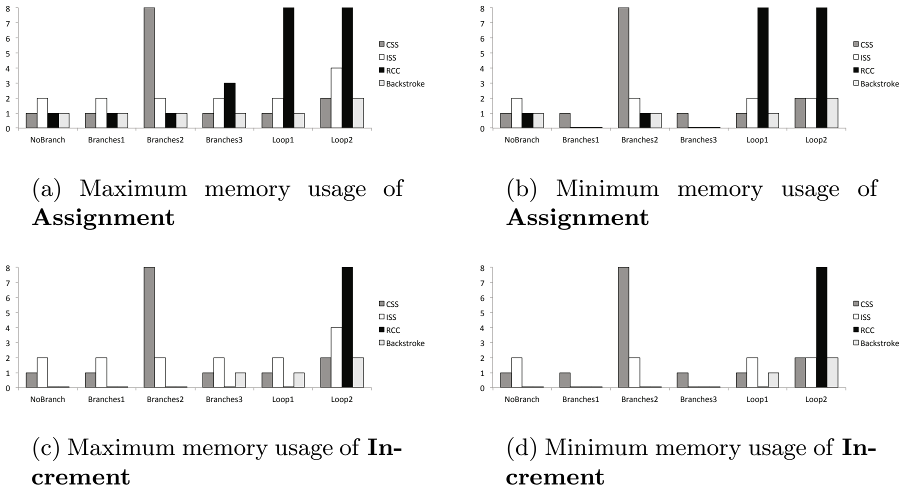
We compare our method[5] to three other approaches commonly employed in the OPDES community to implement rollback:

- **CSS**: Copy state saving. Every target variable is stored at the beginning of the forward function and restored in the reverse function. Here we only store the variables that are potentially modified.
- **ISS**: Incremental state saving. A variable is stored only the first time it is modified. This technique is traditionally implemented by storing the variable's address along with its value, so one can check if the variable is already stored.
- **RCC**: Reverse C compiler [8] is a syntax-directed incremental inversion translator (see section 2).

[5] Note that for loops we use the non-loop solution as defined in section 5.

We count the maximum and minimum memory used for state saving. The memory used to record the control flows outside of loops (including the counter recording the number of iterations in a loop) is ignored because it does not scale with the size of the program state. Figure 5 shows the experiment results, in which (a) and (b) are maximum and minimum memory usage for all benchmarks of the **Assignment** version, and (c) and (d) are of the **Increment** version. The height of each column represents the memory usage.

From the result we can see for most benchmarks Backstroke is the most efficient, which is because our method integrates the advantages from both incremental reverse execution and incremental state saving. **ISS** stores the address of every variable which introduces a large overhead if the address's size is comparable to that of a value's (for scalar data) meanwhile we utilize the CFG path to ensure each variable is stored only once, with much less overhead. **ISS** outperforms **CSS** when there are many variables which are potentially modified but only a small number of them are modified during each execution (see Figure 5 **Branches2**). That is why incremental state saving performs very well when each event only modifies a small portion of the whole state.



(a) Maximum memory usage of **Assignment**

(b) Minimum memory usage of **Assignment**

(c) Maximum memory usage of **Increment**

(d) Minimum memory usage of **Increment**

Fig. 5. Experiment results

From comparing the results from Figure 5 (a)(b) and (c)(d), it is clear that the reverse execution approaches can save much memory. But the amount of benefit from reverse execution is determined by the number of opportunities for reverse computation. For programs that do not have many lossless operations such as ++ and +=, state saving still plays an important role in their inversions.

We must be very cautious when reversing a loop. If the loop solution is applied, we have to determine if storing control flow information is worth it or not. The result of **Loop2** from **RCC** shows that if the number of iterations is large,

storing control flows is not good idea. Saving state inside a loop normally is not a wise choice, as the result of **Loop1** + **Assignment** from **RCC** show.

7 Conclusion and Future Work

We have shown a novel framework for program inversion that combines the advantage of both incremental state saving and incremental reverse execution. Powered by compiler analysis and taking the cost for each operation into account, we gain global optimization through a search algorithm that makes it possible to find the best strategy when reversing a program. The experiment results show that our method is effective comparing to other well-known methods.

Our future work will target reversing functions with structures, arrays, function calls, and aliasing. We believe that our framework is general enough to be reused to solve those problems. For example, a function call can be treated as a special operation. We can extend the value search graph to take advantage of SSA extensions for pointers, arrays, object access, and function calls [9,13,18], allowing us to integrate them into our inversion algorithm. We would also like to run our translator on real-world large simulations and measure the performance gains from parallel execution.

References

1. Abramov, S., Glück, R.: The Universal Resolving Algorithm: Inverse Computation in a Functional Language. Science of Computer Programming 43(2-3), 193–229 (2002)
2. Akgul, T., Mooney III, V.J.: Assembly instruction level reverse execution for debugging. ACM Transactions on Software Engineering and Methodology 13(2), 149–198 (2004)
3. Alpern, B., Wegman, M.N., Kenneth, F.: Detecting Equality of Variables in Programs. In: PPL (January 1988)
4. Ball, T., Larus, J.R.: Optimally profiling and tracing programs. ACM Transactions on Programming Languages and Systems 16(4), 1319–1360 (1994)
5. Ball, T., Larus, J.R.: Efficient Path Profiling. In: MICRO 1996, pp. 46–57 (1996)
6. Biswas, B., Mall, R.: Reverse execution of programs. ACM SIGPLAN Notices 34(4), 61–69 (1999)
7. Briggs, J.S.: Generating reversible programs. Software: Practice and Experience 17(7), 439–453 (1987)
8. Carothers, C.D., Perumalla, K.S., Fujimoto, R.M.: Efficient Optimistic Parallel Simulations Using Reverse Computation. In: PADS (1999)
9. Chow, F., Chan, S., Liu, S.-M., Lo, R., Streich, M.: Effective Representation of Aliases and Indirect Memory Operations in SSA Form. In: Gyimóthy, T. (ed.) CC 1996. LNCS, vol. 1060, pp. 253–267. Springer, Heidelberg (1996)
10. Cooper, K.D., Taylor Simpson, L., Vick, C.A.: Operator strength reduction. ACM Transactions on Programming Languages and Systems 23(5), 603–625 (2001)
11. 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 Transactions on Programming Languages and Systems 13(4), 451–490 (1991)

12. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems 9(3), 319–349 (1987)
13. Fink, S., Knobe, K., Sarkar, V.: Unified Analysis of Array and Object References in Strongly Typed Languages. In: SAS 2000. LNCS, vol. 1824, pp. 155–174. Springer, Heidelberg (2000)
14. Fujimoto, R.M.: Parallel and Distributed Simulation Systems. Wiley (2000)
15. Glück, R., Kawabe, M.: Revisiting an automatic program inverter for Lisp. ACM SIGPLAN Notices 40(5), 8–17 (2005)
16. Jefferson, D.R.: Virtual time. ACM Transactions on Programming Languages and Systems 7(3), 404–425 (1985)
17. Kawabe, M., Glück, R.: The Program Inverter LRinv and Its Structure. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL 2004. LNCS, vol. 3350, pp. 219–234. Springer, Heidelberg (2005)
18. Knobe, K., Sarkar, V.: Array SSA form and its use in parallelization. In: POPL 1998, pp. 107–120. ACM Press, New York (1998)
19. Muchnick, S.S.: Advanced Compiler Design and Implementation (1997)
20. Ross, B.J.: Running programs backwards: The logical inversion of imperative computation. Formal Aspects of Computing 9(3), 331–348 (1997)
21. Srivastava, S., Gulwani, S., Chaudhuri, S., Foster, J.S.: Path-based inductive synthesis for program inversion. In: PLDI 2011, p. 492. ACM Press, New York (2011)

Analytical Bounds for Optimal Tile Size Selection

Jun Shirako[1], Kamal Sharma[1], Naznin Fauzia[2], Louis-Noël Pouchet[2], J. Ramanujam[3], P. Sadayappan[2], and Vivek Sarkar[1]

[1] Rice University
{shirako,kamal.g.sharma,vsarkar}@rice.edu
[2] The Ohio State University
{fauzia,pouchet,saday}@cse.ohio-state.edu
[3] Louisiana State University
jxr@ece.lsu.edu

Abstract. In this paper, we introduce a novel approach to guide tile size selection by employing analytical models to limit empirical search within a subspace of the full search space. Two analytical models are used together: 1) an existing conservative model, based on the data footprint of a tile, which ignores intra-tile cache block replacement, and 2) an aggressive new model that assumes optimal cache block replacement within a tile. Experimental results on multiple platforms demonstrate the practical effectiveness of the approach by reducing the search space for the optimal tile size by 1,307× to 11,879× for an Intel Core-2-Quad system; 358× to 1,978× for an Intel Nehalem system; and 45× to 1,142× for an IBM Power7 system. The execution of rectangularly tiled code tuned by a search of the subspace identified by our model achieves speed-ups of up to 1.40× (Intel Core-2 Quad), 1.28× (Nehalem) and 1.19× (Power 7) relative to the best possible square tile sizes on these different processor architectures. We also demonstrate the integration of the analytical bounds with existing search optimization algorithms. Our approach not only reduces the total search time from Nelder-Mead Simplex and Parallel Rank Ordering methods by factors of up to 4.95× and 4.33×, respectively, but also finds better tile sizes that yield higher performance in tuned tiled code.

1 Introduction

Modern computer systems utilize multi-level memory hierarchies in which the latency of data access from higher levels are orders of magnitude higher than the time required to perform arithmetic operations. *Loop Tiling* [7, 17, 23, 29, 35, 36] is a classical technique to enhance data reuse in memory hierarchy levels close to the processor. Recent advances have made it possible to automatically generate parametrically tiled code, even for imperfectly nested loops [2, 15, 18, 24]. It is well known that the choice of tile sizes has a significant effect on performance, but the *effective selection of optimized tile sizes* remains an open problem that has become ever more challenging as processor memory hierarchies increase in complexity and depth.

Past work has pursued two main types of approaches for tile size selection, *analytical* and *empirical*. In analytical approaches, a compiler selects tile sizes based on static analysis of loop nests and known characteristics of the memory hierarchy. Although several analytical techniques for tile size selection have been proposed in the literature [8, 10, 13, 16, 19, 26, 27, 28], none has been demonstrated to be sufficiently effective

M. O'Boyle (Ed.): CC 2012, LNCS 7210, pp. 101–121, 2012.
© Springer-Verlag Berlin Heidelberg 2012

for use in practice. As a result, the gap between the performance delivered by the best known tile sizes and those selected by an analytical approach has continued to widen, thereby diminishing the utility of past analytical approaches.

Empirical approaches to tile size optimization treat the loop nest as a black box, and perform empirical *auto-tuning* for a given architecture [4, 31, 32, 33] by actually executing the tiled code for a range of different tile sizes. The highly successful ATLAS (Automatically Tuned Linear Algebra Software) system [33] uses empirical tuning at library installation time to find the best tile sizes for different problem sizes on the target machine. One of the most challenging issues in empirical approaches for tile size selection is the enormous search space that must be explored when tiling multiple loops. As shown in many domains (e.g., by Goto and van de Geijn [14] for linear algebra and by Datta et al. [11] for stencil codes), the optimal tile has different tile sizes in different dimensions. The experimental results in this paper support this observation; compared with the best "square" tile, i.e., equal tile sizes along all dimensions, the best non-square tile showed performance speedups of up to 1.40, 1.28, and 1.19 on three different platforms (Xeon, Nehalem, and Power7, respectively). Though many empirical tuning systems attempt to reduce the search space by only examining square tiles, our results reaffirm the importance of including non-square tiles in the search space.

Hybrid approaches to tile size selection that combine analytical models and empirical search have also been pursued [9, 37]. For example, Chen et al. [9] introduced a framework that combines the use of compiler models and search heuristics to perform auto-tuning. However, we are not aware of any hybrid approach that has been demonstrated to be both broadly applicable and effective in practice. In this paper, we develop an analytical approach that is both broadly applicable as well as effectively usable in conjunction with various empirical search strategies for auto tuning.

Since the search spaces for tile size selection increase explosively for multidimensional, non-square and multi-level tiling, an effective approach to prune the search space is critical. Furthermore, while expensive empirical tuning is feasible for libraries such as BLAS that are tuned once per machine and reused across applications, tiled user codes usually require empirical search to be done much more rapidly since the search needs to be performed on all the time-consuming loop nests in the application.

In this paper, we introduce a novel approach using analytical bounds to limit the search space with empirical tuning for square and non-square tiling. As shown in Section 6, the proposed approach to pruning the search space is complementary to, and can be combined with, existing empirical search strategies; e.g., the analytical bounds can be integrated with existing auto-tuning frameworks such as ATLAS [33]. Experimental results show that our approach can reduce the search space by up to four orders of magnitude. Reduction factors of up to 11,879, 1,978, and 1,142 were realized on a Xeon, Nehalem, and Power7, respectively, for the loop nests that we studied.

Our approach employs a pair of analytical models to prune the search space — a conservative model that underestimates the number of iterations in an optimal tile (DL), and an optimistic model that overestimates the number of iterations in an optimal tile (ML). DL (Distinct Lines) [12], a conservative model from past work, models the required cache capacity for a tile as its total data footprint. Under this model, any tiles with a data footprint larger than the cache size are discarded, since they may incur capacity

misses during execution.However, this is a pessimistic assumption for many applications, especially applications with streaming data accesses. We therefore introduce an optimistic analytical model, ML (Minimum working set Lines), that assumes an ideal intra-tile cache block replacement. Because DL and ML respectively provide lower and upper bounds for tile sizes, we can use them to bound tile size search space for empirical tuning. Our experiments show that this bounded search space still contains optimal tile sizes, despite reductions of up to four orders of magnitude in the size of the search space.

The paper is organized as follows. Section 2 reinforces the motivation for this work via a case study that highlights some of the challenges arising from modern memory hierarchies. In Section 3, we provide background on parametric tiling and the DL model from past work. Section 4 introduces the new ML model for single-level tiling. Section 5 elaborates on how the DL and ML models can be used to bound the search space for empirical tuning. Section 6 presents experimental results on three platforms using a number of benchmarks to demonstrate the effectiveness of the approach. Optimal tile sizes were always found within the reduced search space. Related work is discussed in Section 7, and we conclude in Section 8.

2 Motivation and Case Study

Past work on performance models for tile size selection were usually geared towards minimizing capacity and conflict misses for the first level of cache [10, 16, 34]. In this section, we illustrate the impact of higher levels of data cache and Translation Lookaside Buffer (TLB) on tile size selection. As a motivating example, we provide a detailed analysis of the execution of a tiled matrix-multiply kernel. Figure 1 is a sample code from [10] that uses the IKJ loop order.

```
// inter-tile loops
for ii = 1 to N, Ti
  for kk = 1 to N, Tk
    for jj = 1 to N, Tj
      // intra-tile loops
      for i = ii to min(ii+Ti,N)
        for k = kk to min(kk+Tk,N)
          for j = jj to min(jj+Tj,N)
            C[i][j] += A[i][k]*B[k][j];
```

Fig. 1. Tiled Matrix Multiply (IKJ loop order)

Tiling is critical in order to increase data locality in this case: *Ti*, *Tj* and *Tk* must be selected such that the data accessed during the computation of a tile fits entirely within the first level cache in order to avoid capacity misses. Reuse analysis suggests that we set *Ti* to *N*, in order to obtain full temporal reuse of the matrix `B` along the `i` loop [10]. This solution is motivated by the fact that no element of `B` will be used in two different tiles, an apparently ideal solution in terms of L1 cache misses (provided *Tj* and *Tk* are selected adequately). Furthermore, setting $Ti = N$ allows us to explore a two-dimensional search space $Tk \times Tj$ instead of a three-dimensional search space $Ti \times Tk \times Tj$, thereby significantly reducing the search space for optimal tile sizes.

This solution focuses only on minimizing L1 cache misses. To illustrate the deficiency of a Level 1 cache-centric approach, we examine performance variation on an

Intel *Xeon* (E7330 2.40 GHz processor with 32KB L1 cache, 3MB L2 cache, 16 entry TLB1, and 256 entry TLB2 (4KB page size)) for a problem size of 3000×3000. After performing an exhaustive empirical search over tile sizes, we found the optimal tile size on this machine to be $(Ti, Tk, Tj) = (60, 10, 120)$. Note that this optimal point has unequal tile sizes in different dimensions because each dimension has a different data reuse patterns on arrays, and a large tile size is needed along the vectorized dimension (innermost tile size Tj) for effective vectorization. To illustrate the performance impact of the values of Ti, we fix $Tk = 10$ and $Tj = 120$, and plot four metrics — execution time, L1 data cache misses (L1_DCM), L2 data cache misses (L2_DCM), and L2 TLB misses (TLBM_DM) — for different values for Ti in Figure 2. Since the absolute values of these metrics are incomparable, the graph plots use standard min-max normalization to convert each metric to a value in the range $0 \ldots 1$. The normalized value for each metric is computed as the ratio, $(x - \min)/(\max - \min)$, where x, min and max are respectively the absolute, minimum, and maximum value of that metric for different values for Ti.

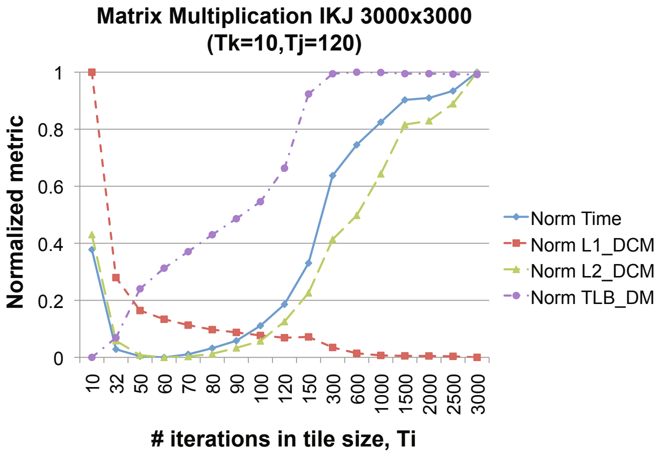


Fig. 2. Normalized Metrics for Matrix Multiplication with an IKJ Loop

First, we observe that L1 misses decrease as Ti increases, as expected. However, the graph clearly shows that the optimal tile size does not occur at $Ti = 3000$. At $Ti > 90$, we see an upward trend in execution time, in contrast to what is suggested by considering just the L1 cache. As the cache footprint of the computation increases, TLB2 misses and L2 misses increase, eventually leading to substantial degradation in execution time.

To better understand this effect, one must also consider the virtual-to-physical address translation for this machine. A lookup for address translation is performed at both the TLB levels. When an entry is not found in eiher TLB, a radix tree page walk is performed, with higher (leftmost) bits fetched from the MMU Cache and lower (rightmost) bits from the L2 data cache. When address translation bits are not found in the L2 cache, it causes DRAM accesses, leading to significant stalls for an application. Thus, increasing an application's footprint causes significant pressure on the data cache

and TLB. This page walk behavior is not just restricted to Intel architectures. AMD's Page Walking Cache and PowerPC's Hashed Table approach exhibit a similar characteristic, where address translation requires traversal of data caches in the case of TLB misses [1, 3].

These observations led us to rethink the tile size selection model. While it is important to optimize for L1 data cache to enable higher reuse, one also needs to consider the reuse effect at higher levels of the memory hierarchy. This leads to considering multiple memory hierarchy levels when searching the $Ti \times Tk \times Tj$ space to find the best tile size for even a single level of tiling. A key contribution of this paper is the development of analytical models to bound the space of candidate tile sizes that take into account multi-level data caches and TLBs. The results in Section 6 demonstrate the following: (1) our analytical models significantly reduced the search space size, while preserving the optimal points in the pruned space; (2) in most cases, the optimal points have non-square tile sizes with performance improvements of up to 40% over square tiles; (3) the developed model also effectively finds optimized tile sizes for parallel tiled code.

3 Background

3.1 DL: Distinct Lines

The DL (Distinct Lines) model was designed to estimate the number of distinct cache lines accessed in a loop-nest [12, 27]. Consider a reference to a contiguously allocated m-dimensional array, A, enclosed in n perfectly nested loops, with index variables $i_1, \cdots, i_n$:

$$A\left(f_1(i_1,\cdots,i_n),\cdots,f_m(i_1,\cdots,i_n)\right) \quad (Fortran)$$
$$A[f_m(i_1,\cdots,i_n)]\cdots[f_1(i_1,\cdots,i_n)] \quad (C),$$

where $f_j(i_1,\cdots,i_n)$ is an affine function. An exact analysis to compute DL is only performed for array references in which all coefficients are compile-time constants (i.e., for affine references). An upper bound for the number of distinct lines accessed by a single array reference [12] with one-dimensional subscript expression $f(i_1,\cdots,i_n)$ is

$$DL(f) \le \min\left(\frac{(f^{hi}-f^{lo})}{g}+1, \left\lceil\frac{(f^{hi}-f^{lo})}{L}\right\rceil+1\right),$$

where g is the greatest common divisor of the coefficients of the enclosing loop indices in f, and L is the cache line size in units of array element size; f^{hi} and f^{lo} are the maximum and minimum values of the subscript expression f across the entire loop nest. In practice, the relative error of this estimation is small when, as is usually the case, the range $(f^{hi}-f^{lo})$ is much larger than the values of the individual coefficients of f. For a multidimensional array reference $A(f_1,\cdots,f_m)$, the upper bound estimate [12] is as follows (with a heuristic assumption that the first dimension of the array has at least L elements).

$$DL(f_1,\cdots,f_m) = DL(f_1) \times \prod_{j=2}^{m}\left(\frac{(f_j^{hi}-f_j^{lo})}{g_j}+1\right).$$

Extensions of this model to account for multiple array accesses in a loop nest have also been developed [12, 27]. These DL definitions for an entire loop nest are also applicable

to a tile whose loop boundaries are expressed using tile sizes. In such a case, the DL definition is a symbolic function of tile sizes $t_1, \cdots, t_n$ denoted by $DL(t_1, \cdots, t_n)$ [27].

The DL definition can also be applied to any level of cache or TLB by selecting its cache line size or page size as L. However, the DL model ignores possible replacement of cache lines within a tile, and therefore provides only conservative upper bounds for the number of cache lines needed.

3.2 Parametric Tiling

Although production compilers today may have limited tiling capability, there have been significant recent advances in automatic source-to-source transformations for tiling and several systems for parametric tiling have been developed and made publicly available such as TLOG [24], HITLOG [18] and PrimeTile [15]. With such tiled-code generators, it is now possible to generate tiled code for compute-intensive inner kernels (including imperfectly nested loops), that can be tuned to the cache characteristics of the target platform. Thus, just as ATLAS [33] is used in auto-tuning dense linear algebra codes, it becomes feasible to use auto-tuning for user kernels such as stencil-based computations. However, unlike library kernel optimization, exhaustive search of the tile parameter space over several hours to days is generally not attractive for tuning user kernels. This motivates the approach developed in this paper to significantly prune the search space.

4 ML: Minimum Working Set Lines

We now introduce ML (Minimum working set Lines), a new analytical cost model based on the cache capacity required for a tile when intra-tile reuse is taken into account. We define the ML model next and then develop an approach to computing ML for a tile by first constructing a special sub-tile based on analysis of reuse characteristics and then computing the DL value for that sub-tile. Although we mainly focus on cache capacity in this section, the model is directly applicable to TLBs by replacing the cache line size with the page size.

4.1 Operational Definition of ML

The essential idea behind the ML model is to develop an estimate of the minimum cache capacity needed to execute a tile without incurring any capacity misses; this minimum cache capacity can be viewed as the minimum working set size for the tile. Consider a memory access trace of the execution of a single tile, run through an idealized simulation of a fully associative cache. The cache is idealized in that its size is "unbounded" (i.e., any access to a data element in the tile will never lead to a capacity miss) and an optimal replacement policy, where a line in the cache is marked for replacement as soon as the last reference to any data on that line has been issued (through an oracle that can determine all future references of the tile). Before each memory access, the simulator fetches the desired line into the idealized cache if needed. After each memory access, the simulator evicts the cache line if it is the last access (according to the oracle). *ML corresponds to the maximum number of lines (high water mark) held in this idealized cache during the execution of the entire trace for the tile.*

4.2 Model of Computation

In this paper, we focus on the class of affine imperfectly nested loops, where loop bounds and array access expressions are affine functions of the surrounding loop iterators and program constants. For this class of program, it is possible to restructure the code automatically [6, 15] to expose rectangular tiles of parametric size. Assuming that a system such as PrimeTile [15] has already been used to generate parametric rectangularly tiled code, we focus on the problem of tile size optimization for such codes.

The outermost loop inside a tile (i.e., the outermost intra-tile loop) is denoted by $loop_1$, and the innermost intra-tile loop is $loop_n$. Since tiles are rectangular by construction, $loop_i$ ($1 \leq i \leq n$) has the same trip count for any of its executions. Note that the case of partial tiles is handled with prolog/epilog code [15]. The iteration domain of a tile is represented with a tuple $[T_1;\ T_2;\ \cdots;\ T_n]$. A tile is surrounded by the loops iterating on all tiles, i.e., the inter-tile loops. In this paper, we assume a single level of tiling; extension to multi-level tiling is a subject for future work.

4.3 Distance in Tiled Iteration Space

A specific instance of the loop body is identified by an iteration vector, that is, a coordinate in the iteration space, noted $p = (p_1, p_2, \cdots, p_n)$. The distance between two iteration vectors p and p' is expressed as the distance vector $d = (d_1, d_2, \cdots, d_n) = p' - p$. The *scalar distance* between two iteration vectors is the number of instances of the inner-most loop body to be executed (in lexicographic ordering) between these two iteration vectors. For instance, in a tiled loop nest the scalar distance between two consecutive iterations of the innermost loop $loop_n$ is 1, representing the shortest possible scalar distance. It corresponds to the distance vector $(0, \cdots, 0, 1) = (p_1, \cdots, p_{n-1}, p_n + 1) - (p_1, \cdots, p_{n-1}, p_n)$, for any values of $p_1, \cdots, p_n$. It is always possible to find a sub-tile tuple that corresponds to this scalar distance. Here it is $[1;\ 1;\ \cdots;\ 1]$, i.e., the tuple describing an n-dimensional rectangle containing exactly one point. We call this form a sub-tile tuple expression of the tile tuple $[T_1;\ T_2;\ \cdots;\ T_n]$.

Let us now consider the distance vector $(1,2,3) = (p_1+1, p_2+2, p_3+3) - (p_1, p_2, p_3)$. In general, to compute the associated scalar distance, we first compute the scalar distance corresponding to two consecutive iterations for each of the intra-tile loops: 1 for $loop_3$, T_3 for $loop_2$, and T_2T_3 for $loop_1$. The scalar distance is computed by the dot product $(1,2,3).(T_2T_3, T_3, 1)^t = T_2T_3 + 2T_3 + 3$. It is always possible to compute an associated sub-tile tuple expression corresponding to the sub-part of the tile iteration domain bounded by the two iteration points p and p', by combining multiple rectangles defined by individual sub-tile tuples. Here, $[1;\ T_2;\ T_3] + [1;\ 2;\ T_3] + [1;\ 1;\ 3]$ is the associated sub-tile tuple expression.

One iteration of $loop_i$ strides over $size_i$, which is the total number of iteration points within the loop body of $loop_i$. $size_i$ is defined in both scalar and sub-tile tuple expression as follows:

$$size_n = 1 = [1;\ 1;\ \cdots;\ 1] \quad (i = n);$$
$$size_i = \prod_{j=i+1}^{n} T_j = [1;\ \cdots;\ 1;\ T_{i+1};\ \cdots, T_n] \quad (i < n).$$

Using size vector $size = (size_1, size_2, \cdots, size_n)$, we define the *scalar distance* of the distance vector $d = (d_1, d_2, \cdots, d_n)$ and its sub-tile tuple expression as follows:

Scalar distance: $size \cdot d = \sum_{i=1}^{n}(d_i \times size_i)$;
Sub-tile tuple: $\sum_{i=1}^{n}([1;\ \cdots;\ 1;\ d_i;\ T_{i+1};\ \cdots;\ T_n])$.

4.4 Temporal and Spatial Reuse Distance

Temporal and spatial data reuse are expressed using widely used definitions of "reuse distance vectors" (often shortened to "reuse vectors") [34]. A number of previous efforts introduced methods to compute spatial reuse vectors [34] while temporal reuse vectors are computed from standard dependence analysis.

```
for (p_1 = [low1 : low1+T_1-1])
  for (p_2 = [low2 : low2+T_2-1])
    for (p_3 = [low3 : low3+T_3-1])
      B[p_1][p_2][p_3] = A[p_1][p_3] + B[p_1-2][p_2-3][p_3];
```

Fig. 3. Sample Code

For array A in Figure 3, the pair $((p_1, p_2, p_3), (p_1, p_2+1, p_3))$ has temporal reuse with a reuse distance vector of $d1 = (0,1,0)$. The pair $((p_1, p_2, p_3), (p_1, p_2, p_3+1))$ has spatial reuse in array A with a reuse distance vector of $d2 = (0,0,1)$. For array B, the pair $((p_1, p_2, p_3), (p_1+2, p_2+3, p_3))$ has a temporal reuse vector $d3 = (2,3,0)$, and the pair $((p_1, p_2, p_3), (p_1, p_2, p_3+1))$ has a spatial reuse vector $d4 = (0,0,1)$.

In Section 4.3, we defined the *scalar distance* for these reuse distance vectors. For Figure 3, the size vector is $size = (T_2T_3, T_3, 1)$. The scalar distance of temporal reuse vector $d1 = (0,1,0)$ is calculated by $size \cdot d1 = T_3$, and the scalar distance of spatial reuse vector $d2 = (0,0,1)$ is $size \cdot d2 = 1$. They are also represented as sub-tile tuple expressions: $[1;\ 1;\ T_3]$ and $[1;\ 1;\ 1]$, respectively. Finally, we call the largest scalar distance Maximum Reuse Distance, or MRD; the MRD is defined on a per-array basis. For example, the MRD for array A in Figure 3 is $[1;\ 1;\ T_3]$, and MRD for array B is $[2;\ T_2;\ T_3] + [1;\ 3;\ T_3]$ due to $d3$.

The approximation of reuse distance vectors for non-uniform reuse patterns is still an open question. As described in Section 5, ML is used to derive the upper bounds of tile sizes and conservative approximation may make the tile size boundaries smaller than optimal points. Therefore, we ignore non-uniform reuse so as to estimate tile sizes optimistically, and leave approximation of non-uniform reuse distance to future work.

4.5 Computation of ML

Using Maximum Reuse Distance for array X, we define ML for array X as follows. First, a pair of iteration instances $(p_1, p_2, \cdots, p_n)$ and $(p_1+d_1, p_2+d_2, \cdots, p_n+d_n)$ has the following Maximum Reuse Distance for array X:

$$MRD_X = \sum_{i=1}^{n}([1;\ \cdots;\ 1;\ d_i;\ T_{i+1};\ \cdots;\ T_n]).$$

In order to exploit all data locality related to array X, the data at $(p_1, p_2, \cdots, p_n)$ must not be removed from the cache memory $(p_1+d_1, p_2+d_2, \cdots, p_n+d_n)$ is accessed. The

cache must keep all the distinct cache lines for array X within the distance of MRD_X. Since the MRD is also represented as a sub-tile tuple, the ML for array X is equivalent to the DL of X for the sub-tile tuple defined by MRD_X. Thus we have

$$ML_X = DL_X(MRD_X).$$

To compute DL of a sum of sub-tile tuples, we compute the sum of DL of each individual sub-tile tuple. As shown above, the expression of Maximum Reuse Distance MRD_X does not include T_1, therefore, ML_X is also independent of T_1. This is true for any reference: the sub-tile tuple never contains T_1 in its components, as shown in Section 4.3. For instance, MRD_B and ML_B for array B of Figure 3 are as follows.

$$MRD_B = [2;\ T_2;\ T_3] + [1;\ 3;\ T_3];$$
$$ML_B = DL_B(MRD_B) = DL_B(2,T_2,T_3) + DL_B(1,3,T_3).$$

ML is defined as the sum of the ML_X values for each array X accessed in the tile:

$$ML = \sum_X (ML_X).$$

In order to leverage all intra-tile data locality, we should select tile sizes so that ML is smaller or equal to the number of cache lines of the target cache memory, which is usually level-1 cache.

4.6 Example

Figure 1 from Section 2 shows a single-level tiling example for Matrix Multiplication. We assume an element of array has 8 Bytes, and the cache line size of L1/L2 is 64 Bytes (a cache line contains eight elements). DL is calculated as follows:

$$DL = DL_C(Ti,Tk,Tj) + DL_A(Ti,Tk,Tj) + DL_B(Ti,Tk,Tj) \ = Ti\left\lceil\frac{Tj}{8}\right\rceil + Ti\left\lceil\frac{Tk}{8}\right\rceil + Tk\left\lceil\frac{Tj}{8}\right\rceil.$$

Also, the MRD for each array is computed from size vector $size = (TkTj,\ Tj,\ 1)$ and reuse distance vectors. $MRD_C = (TkTj,\ Tj,\ 1)\cdot(0,1,0) = [1;\ 1;\ Tj]$, $MRD_A = (TkTj,\ Tj,\ 1)\cdot(0,1,0) = [1;\ 1;\ Tj]$, and $MRD_B = (TkTj,\ Tj,\ 1)\cdot(1,0,0) = [1;\ Tk;\ Tj]$. Assigning each MRD to the corresponding DL expression, the ML for single-level tiling is computed as

$$ML = DL_C(1,1,Tj) + DL_A(1,1,Tj) + DL_B(1,Tk,Tj) = \left\lceil\frac{Tj}{8}\right\rceil + 1 + Tk\left\lceil\frac{Tj}{8}\right\rceil.$$

5 Bounding the Search Space by Using DL and ML

This section presents how our DL/ML model bounds a tiling search space. As discussed in Section 4, ML is used for optimistic cache and TLB capacity constraints for intra-tile data reuse and gives the upper boundaries for estimated tile sizes. In contrast, DL is used for conservative constraints, and gives the lower boundaries. These lower and upper boundaries drastically reduce the search space for single and multi-level tiling. Furthermore, DL, which represents the number of distinct lines within a tile, can be used as a capacity constraint for inter-tile data reuse on higher levels of cache/TLB. The extension of our approach to multi-level tiling is the subject for future work.

5.1 Capacity Constraint for Intra-tile Reuse

Section 4.5 shows *ML* for single-level tiling can be dependent on tile sizes T_2, T_3, ..., T_n and is independent on T_1 while DL can depend on all tile sizes T_1, T_2, ..., T_n. CS_1 represents the number of cache lines or TLB entries at level-1 cache or TLB memory. All tile sizes within the lower boundaries due to DL and upper boundaries due to ML satisfy the following constraints.

$$DL(T_1, T_2, \cdots, T_n) \geq CS_1, \quad ML(T_2, T_3, \cdots, T_n) \leq CS_1.$$

We have two bounded regions according to cache and TLB. In our approach, we consider the union of both the regions as candidates for optimal tile sizes, e.g., a point that is within the DL/ML region due to cache but outside the region due to TLB is also a candidate.

5.2 Capacity Constraint for Inter-tile Reuse

Although Section 5.1 shows the boundaries to maximize intra-tile data reuse of level-1 tile, the outermost tile size T_1 is actually not bounded above by the ML constraint. As discussed in Section 2, this corresponds to traditional single-level tiling to fit within single-level cache, where the outermost loop is not tiled [8, 10, 20]. However, the outermost tile size affects inter-tile reuse on higher levels of cache/TLB, and too large tile size would harm the inter-tile data locality and even the overall performance. Using DL definition, we define an additional capacity constraint in order to preserve inter-tile data reuse on level-k ($k > 1$) cache/TLB as follows:

$$DL(T_1, T_2, \cdots, T_n) \leq CS_k.$$

This inequality, which ensures that whole distinct lines within the tile can be kept on level-k cache/TLB and guarantees the inter-tile data reuse, bounds the outermost tile size T_1. It is a subject for future work to select the suitable k according to the target system. In the experiments in Section 6, we select the highest level of cache/TLB as k.

5.3 Empirical Search within Bounded Search Space for Single-Level Tiling

Described in previous section, DL/ML capacity constraints for single-level tile consist of the following three conditions.

$$DL(T_1, T_2, \cdots, T_n) \geq CS_1 \quad \text{(lower boundary for intra-tile reuse);}$$
$$ML(T_2, T_3, \cdots, T_n) \leq CS_1 \quad \text{(upper boundary for intra-tile reuse);}$$
$$DL(T_1, T_2, \cdots, T_n) \leq CS_k \quad \text{(upper boundary for inter-tile reuse).}$$

Empirical search finds the optimal tile sizes for $T_1, T_2, \cdots, T_n$ that minimizes the objective metrics such as execution time.

Let us calculate the search space of Figure 1, which is a single-level tiling example of Matrix Multiplication. We assume the same experimental platform and program size as Section 2 and Section 4.6; L1/L2 cache contains 512/49152 lines respectively.[1] The capacity constraints: $DL = Ti\left\lceil \frac{Tj}{8} \right\rceil + Ti\left\lceil \frac{Tk}{8} \right\rceil + Tk\left\lceil \frac{Tj}{8} \right\rceil \geq 512$, $ML = \left\lceil \frac{Tj}{8} \right\rceil + 1 + Tk\left\lceil \frac{Tj}{8} \right\rceil \leq 512$, and $DL = Ti\left\lceil \frac{Tj}{8} \right\rceil + Ti\left\lceil \frac{Tk}{8} \right\rceil + Tk\left\lceil \frac{Tj}{8} \right\rceil \leq 49152$. Figure 4 shows the bounded 2-D search space for Tk and Tj when Ti is 60, which

[1] We omit details on TLB constraints due to space limitations.

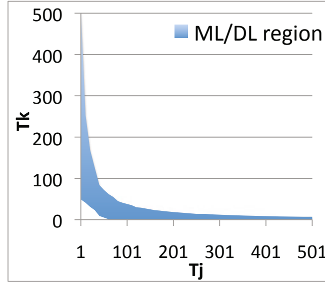


Fig. 4. Search Space for Matrix Multiplication for $Ti = 60$

is much smaller than the original 2-D search space 3000×3000, and the optimal tile size $Ti = 60$, $Tk = 10$, and $Tj = 120$ is found within the bounded region.

5.4 Compiler Pass for Bounded Search Space

Figure 5 shows the compiler framework to implement the DL-ML bounded search space algorithm. This implementation requires standard compiler tools, such as dependence vector computation and array index expression extraction, readily available in most modern compilers. This is the only program-specific data required to compute the DL-ML equations. Plugging the additional machine-specific information about the different cache level sizes and associated line sizes results in a bounded search space of candidate tile sizes, which is drastically smaller than the original set of candidates. Using these bounds, a tile size tuning framework explores only a fraction of points in the original search space, thereby considerably reducing the tuning overhead.

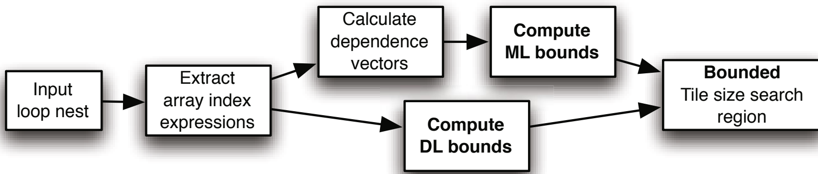


Fig. 5. Compiler Implementation of DL-ML Bounding

6 Experimental Results

An experimental assessment was performed on three Linux-based systems: an Intel Core i7 920 running at 2.66 GHz with shared L3 cache (labeled *Nehalem*), an IBM Power 7 running at 3.55 GHz (*Power7*), and an Intel Xeon E7330 running at 2.40GHz with shared L2 cache (*Xeon*). Previous work has used published cache capacity data from manufacturers in analytical models for cache performance. However, due to factors such as page table entries, OS processes, etc., the full capacity of higher level caches may not be actually available for use by the application. We report in Table 1 the effective capacities for the cache and TLB — the published capacity Spec versus

the effective capacity Effective that was observed using micro-benchmarks for hardware characterization [25]. It may be seen that the effective capacity may be as low as half the documented size, which can affect the DL/ML capacity constraints. In our experiments, we used the effective capacities. The impact of using published capacities instead of effective capacities is studied in Section 6.2.

Table 1. Cache characteristics of the architectures considered

	L1		L2		L3		Line Size	TLB1 Entries	TLB2 Entries	Page Size
	Spec.	Effective.	Spec.	Effective.	Spec.	Effective.				
Nehalem	32kB	32kB	256kB	256kB	8MB	5.2MB	64B	64	512	4kB
Power7	32kB	32kB	256kB	256kB	32MB	18.4MB	128B	64	512	64kB
Xeon	32kB	32kB	3MB	1.5MB	N/A	-	64B	16	256	4kB

We studied five benchmarks using double precision floating point arithmetic. matmult is a standard matrix-multiply kernel: $C = A.B$; dsyrk is a symmetric rank 1 computation: $C = \alpha.A.A^T + \beta.C$; and dtrmm is a triangular in-place matrix multiplication: $B = \alpha.A.B$ (A is triangular). We also considered a representative 9-point two-dimensional stencil computation, 2d-jacobi, and a 2D Finite Difference Time Domain method, 2d-fdtd. Parametrically tiled code for each benchmark was generated using the publicly available PrimeTile code generator [15] after any necessary preprocessing such as skewing [6] to ensure that rectangular tiling of the loops was legal. For all tested versions, including the original code, the same compiler optimization flags were used: for *Nehalem* and *Xeon*, we used Intel ICC 11.0 with option `-O3`; for *Power7*, we used IBM XLC 10.1 with option `-O3`.

6.1 Performance Distribution of Different Tile Sizes

For each benchmark, in the case of single-level tiling, we conducted an extensive set of experiments, for a subset of tile sizes for each loop ranging from 1 to the loop length, in steps of 10 (approximately). Figure 6a plots the data for matmult (size 3000 × 3000) for the three considered architectures. A point (x,y) in this cumulative plot indicates that x% of the tile combinations achieved normalized performance greater than or equal to y, where normalization is with respect to the best performing case among all runs and performance is inversely proportional to execution time.

It may be observed that only a small fraction of the tile combinations achieve very good performance — for example, on the *Nehalem*, only 2% of the tile size configurations achieve more than 90% of the maximal performance. Also, there is a very large variation in performance between the best and worst tile size choices, up to a factor of 10. The performance distribution also varies for different targets — for *Power7*, over 20% of the cases provide good performance. Further, we have also observed that the points with good performance are not uniformly distributed in the search space but are clustered in clouds. This highlights the complexity of the search problem when using a blind random search — convergence towards an optimal point may require sampling of a significant fraction of the search space.

Figure 6b shows a similar analysis for the 2d-jacobi benchmark. For the target machines, we observe quite a different trend compared to matmult: about 55% of the tile

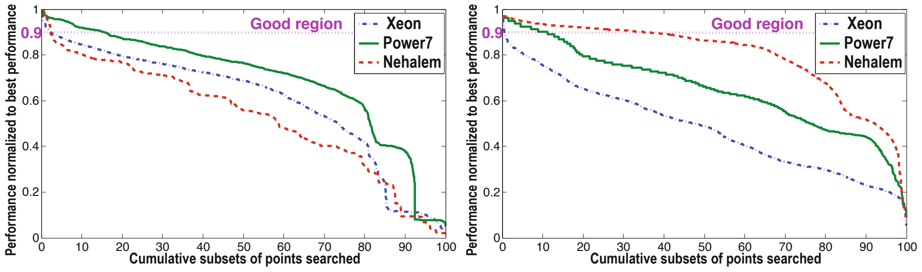


Fig. 6. Performance Distribution for (a) matmult-3000x3000 and (b) 2d-jacobi-50x4000x4000 on *Nehalem*, *Xeon*, and *Power7*

sizes achieve 90% or more of the maximal performance for *Nehalem*, while this ratio significantly decreases for the two other architectures, down to 1% for *Xeon*.

6.2 Search Space Reduction by DL/ML Model

To assess the effectiveness of search space pruning by use of the DL/ML model, Figures 7-11 show the bounded search region superposed with a marking of all tile choices that achieve over 95% of the maximal performance on *Nehalem* and *Power7*.[2] In each 3-D space, the x, y, and z axes show tile size values for the outer loop, middle loop and inner loop respectively. These tile choices are called "best" points in this section. The surface in each 3-dimensional plot represents the DL/ML upper boundary for single-level tiling, considering intra-tile reuse for level-1 cache and TLB, and inter-tile reuse on the highest level of cache and TLB, as described in Section 5. In order to enhance viewability, the figures do not show the lower DL boundaries, since they fall below the best points.

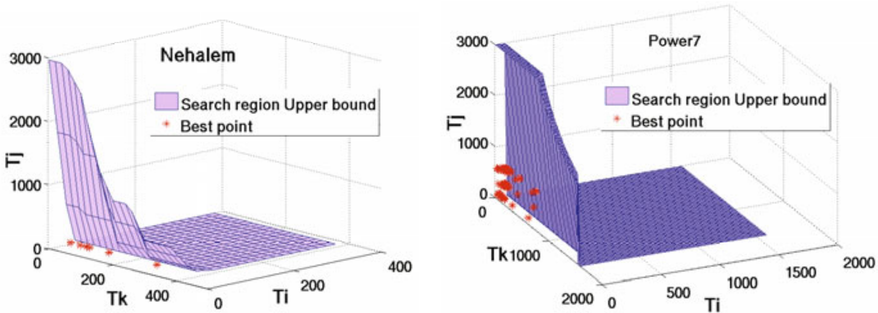


Fig. 7. Best tile sizes (within 95% and more of the optimal) for matmult-3000x3000 with *k*-*i*-*j* loop ordering on 7a *Nehalem* and 7b *Power7*

For all the plots in Figures 7-11, we see that although a small number of points lie outside the bounded search space, the vast majority of best points lie inside it. The density of good solutions in the space is thus very much larger than in the non-pruned

[2] Data for *Xeon* are not included due to space limitations.

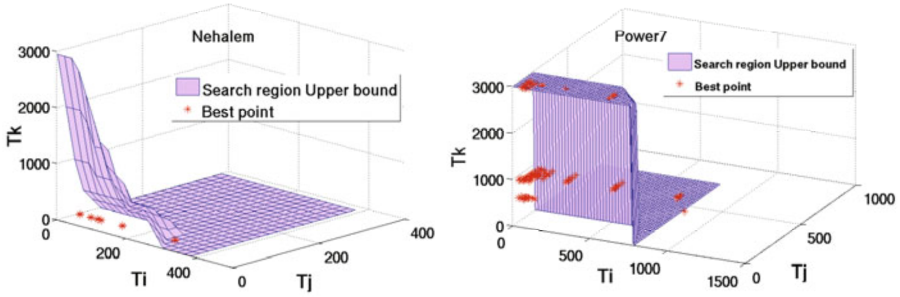


Fig. 8. Best tile sizes (within 95% and more of the optimal) for dsyrk-3000x3000 with *i*-*j*-*k* loop ordering on 8a *Nehalem* and 8b *Power7*

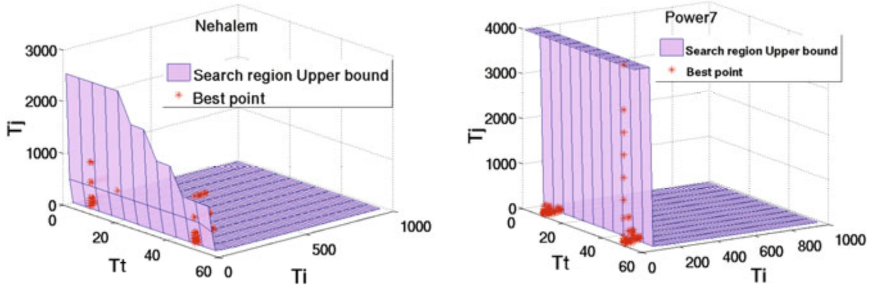


Fig. 9. Best tile sizes (within 95% and more of the optimal) for 2d-jacobi-50x4000x4000 with *t*-*i*-*j* loop ordering on 9a *Nehalem* and 9b *Power7*

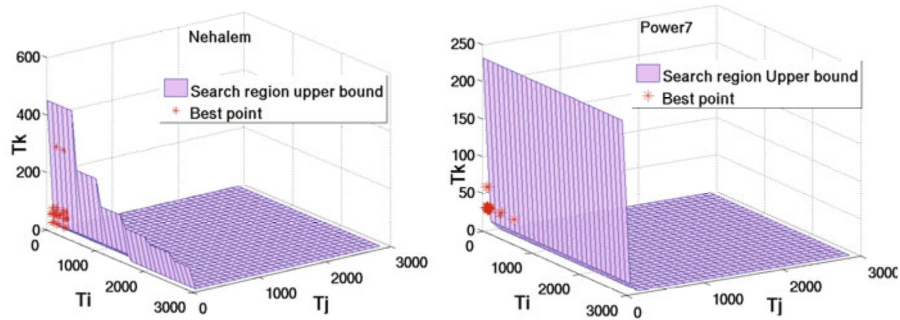


Fig. 10. Best tile sizes (within 95% and more of the optimal) for dtrmm-3000x3000 with *i*-*j*-*k* loop ordering on 10a *Nehalem* and 10b *Power7*

space. For all the benchmarks, we found that an optimal tile was within the bounded search region. Figures 7-11 also show that the best tile sizes are relatively smaller on *Nehalem*, and larger on *Power7*. For example, the best points in dsyrk are within the region of (`Ti` $\leq$ 400, `Tj` $\leq$ 50, `Tk` $\leq$ 100) on both *Nehalem* and *Xeon*. However, the best points on *Power7* are distributed much more broadly, up to the maximum size of 3000.

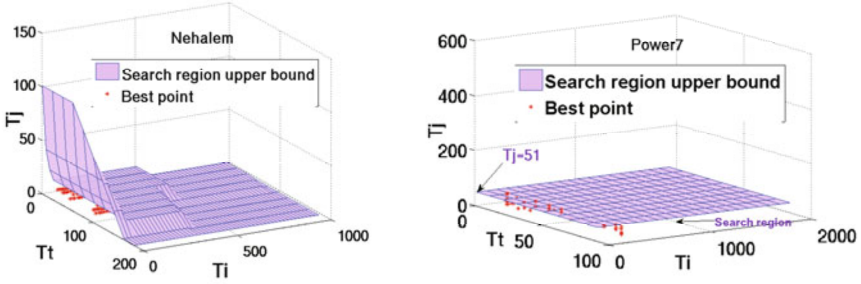


Fig. 11. Best tile sizes (within 95% and more of the optimal) for 2d-fdtd-100x2000x2000 with *t-i-j* loop ordering on 11a *Nehalem* and 11b *Power7*

This trend pertains to the impact of the level-1 TLB size on each system; the small (4KB) page size on *Nehalem* and *Xeon* causes the best tile sizes to be small, while the large (64KB) page size on *Power7* allows much larger tiles without causing severe TLB misses. These differences are directly reflected in the upper boundary of the DL/ML model, which covers a larger region for *Power7* than the other machines. As discussed in Section 5.2, the outermost tile size (x axis) is bounded above only by the inter-tile reuse constraints due to the highest level of cache and TLB. It is obvious that the outermost tile size boundaries also contribute to search space reduction in Figures 7-11.

Table 2 shows the ratio of the space considered in the DL/ML range for the three architectures, compared to the full space of tile sizes. This ratio corresponds to the minimal acceleration factor for an exhaustive or random empirical search compared to using the full space. The factor is much lower for *Power7* due to the larger page size, as explained above. In order to assess the impact of using published versus effective capacities, we repeated our analysis also using the published capacity for the highest level of cache instead of the effective capacity. We found the reduction in search space by use of the ML model was virtually identical with use of published size or effective size (Table 2), with one exception: for matmult with 3000$\times$3000 size on *Power7*, the reduction rate decreased from 93.79 to 84.01. This is because the constraint due to the highest level of cache did not affect the search space boundaries for the evaluated benchmarks and platforms except for Power7/matmult.

Table 2. Search space reduction factor across different architectures

	Problem Size	Xeon	Power 7	Nehalem
matmult	600x600	81.12	1.46	21.90
	3000x3000	8710.81	93.79	1856.49
dsyrk	1000x1000	492.24	2.04	91.62
	3000x3000	11879.67	83.99	1978.26
dtrmm	600x600	41.37	2.31	32.74
	3000x3000	2565.00	1142.23	1238.24
2d-jacobi	50x4000	3102.90	76.43	693.45
2d-fdtd	100x2000x2000	1307.19	45.55	358.74

6.3 Summary of Experiments

1-Level Tiling. We summarize our experiments for 1-level tiling in Table 3. We report, for each benchmark and each architecture, the execution time (in seconds) of the original, untiled code in the Untiled Time column. DL reports the tile sizes and its execution time as obtained by the purely analytical approach using the DL model [12]; Best Square Tile reports the tile sizes and execution time obtained by an exhaustive empirical search only for square tile sizes; and Best DL/ML is obtained by an exhaustive empirical search in the DL/ML range. The Best DL/ML point was also the globally optimal point in the whole search space for all programs/platforms. We observed that the optimal points represent non-square tile sizes for all cases. For efficient vectorization on all three platforms, the vectorized dimension should correspond to a sufficiently large tile size. Furthermore, the different temporal/spatial data reuse pattern along different dimensions contributes to the unequal sizes of tiles in the different dimensions for the optimal choices.

Table 3. 1-level Tiling Results (Time in seconds, N: Nehalem, P: Power7, X: Xeon)

	Untiled	DL		Best Square Tile		Best DL/ML		Impr. by DL/ML	
	Time	Tile Size	Time	Tile size	Time	Tile size	Time	vs. DL	vs. Sq.
matmult-N	33.25	(40, 40, 30)	16.40	(80,80,80)	17.27	(150, 30, 80)	13.48	1.22×	1.28×
matmult-P	25.46	(50, 30, 20)	13.90	(80,80,80)	12.28	(90, 10, 120)	10.60	1.31×	1.15×
matmult-X	153.66	(40, 40, 30)	29.51	(50,50,50)	23.98	(100, 20, 120)	18.35	1.60×	1.31×
dsyrk-N	25.39	(30, 40, 40)	15.47	(80,80,80)	15.54	(30, 30, 90)	12.50	1.23×	1.24×
dsyrk-P	23.32	(40, 30, 30)	15.10	(300,300,300)	10.86	(60, 10, 1000)	9.16	1.64×	1.19×
dsyrk-X	84.89	(30, 40, 40)	26.08	(120,120,120)	25.44	(100, 30, 80)	18.19	1.43×	1.40×
dtrmm-N	142.42	(40, 40, 30)	19.20	(60,60,60)	18.87	(150, 30, 60)	18.20	1.05×	1.04×
dtrmm-P	62.74	(30, 50, 20)	14.60	(60,60,60)	13.06	(600, 30, 32)	11.96	1.22×	1.09×
dtrmm-X	114.70	(40, 40, 30)	28.98	(120,120,120)	29.13	(30, 10, 120)	23.49	1.23×	1.24×
2d-jacobi-N	2.43	(10, 40, 10)	2.60	(50,50,50)	2.24	(10, 8, 150)	2.16	1.20×	1.04×
2d-jacobi-P	2.10	(10, 40, 10)	2.09	(10,50,50)	1.31	(10, 40, 120)	1.19	1.76×	1.10×
2d-jacobi-X	8.75	(10, 40, 10)	2.77	(10,8,8)	2.81	(50, 40, 20)	2.54	1.09×	1.11×
2d-fdtd-N	15.35	(10, 60, 8)	2.41	(50, 8, 8)	2.35	(50, 50, 8)	2.26	1.07×	1.04×
2d-fdtd-P	9.56	(10, 40, 1)	6.90	(50,70,70)	2.11	(40,70,40)	2.09	3.30×	1.01×
2d-fdtd-X	16.42	(10, 60, 8)	4.47	(100,40,40)	4.22	(50,100,8)	4.01	1.11×	1.05×

Empirical Search Using DL/ML Model. This section demonstrates the integration of the analytical bounds with existing search optimization algorithms, the Nelder-Mead Simplex method [22] and the Parallel Rank Ordering (PRO) method [30]. In order to handle boundary constraints due to the DL/ML model, we used the extended version of the PRO algorithm introduced in the Active Harmony framework [32]. The same extension to handle boundaries was employed in our implementation of the Simplex method, and its stopping criteria are based on the work by Luersen [21]. Regarding initial simplex selection for our Simplex search implementation, we used a model-driven approach based on the DL model for square tiling. The square tile size tuple, $T1 = T2 = T3$, which satisfies the DL capacity constraint is selected as one vertex of the initial simplex. Other tree vertices were chosen so as to form a regular triangular pyramid. Note that all the studied kernel loops are triply nested and the simplex always

has four vertices. The initial simplex is bounded by the upper and lower tile sizes in addition to the DL/ML bounds.

Table 4 shows the total execution time for the whole empirical tuning, the best tile size found by each approach, and its execution time. The DL/ML bounds significantly reduced the total tuning time by a factor of 1.02 to 4.95 on *Nehalem*, 1.33 to 2.48 on *Power7*, and 2.95 to 4.66 on *Xeon*. Furthermore, the Simplex and PRO methods using DL/ML boundary constraints found better tile sizes than the cases without DL/ML bounds, except for the simplex method on *Nehalem*. The tile size search space contains various local optimal points, and these empirical search approaches not using the boundary constraints got stuck at local optima far from the global optimal point. Note

Table 4. Empirical Search Results for 1-level Tiling

	Without DL/ML Bounds		With DL/ML Bounds	
	Total [sec]	Best Size / Time [sec]	Total [sec]	Best Size / Time [sec]
matmult-nehalem-simplex	3173.36	(17, 120, 1369) / 13.86	640.98	(36, 56, 64) / 14.71
matmult-nehalem-pro	1294.88	(52, 344, 2270) / 15.64	380.73	(36, 80, 29) / 15.24
matmult-power7-simplex	940.81	(114,1142,858) / 11.4	709.22	(22,82,117) / 11.32
matmult-power7-pro	691.01	(172,1784,2989) / 11.39	442.26	(28,72,126) / 10.58
matmult-xeon-simplex	4268.52	(98,1257,1258) / 21.69	1039.69	(35,56,57) / 19.52
matmult-xeon-pro	2453.03	(97,904,1315) / 21.81	831.73	(31,64,56) / 19.22
2d-jacobi-nehalem-simplex	88.84	(42, 465, 498) / 2.25	26.48	(34,15,64) / 2.32
2d-jacobi-nehalem-pro	51.09	(29, 2001, 2000) / 2.41	50.33	(25,10,627) / 2.2
2d-jacobi-power7-simplex	96.95	(50,37,92) / 1.15	54.94	(50,28,116) / 1.14
2d-jacobi-power7-pro	83.98	(25,8,3495) / 1.61	33.81	(10,53,84) / 1.17
2d-jacobi-xeon-simplex	351.52	(50,40,16) / 2.49	75.49	(50,33,16) / 2.49
2d-jacobi-xeon-pro	248.12	(26,1976,2098) / 8.85	57.34	(10,12,21) / 2.75

Table 5. Parallel 1-level Tiling Results

	Optimal Point (parallel)		Optimal Point (sequential)	
	Tile Size	Speedup vs. Untiled Seq.	Tile Size	Speedup vs. Untiled Seq.
matmult-nehalem (8 Threads)	(80, 10, 120)	9.39×	(150, 30, 80)	2.47×
matmult-power7 (32 Threads)	(100, 1, 300)	15.24×	(90, 10, 120)	2.40×
matmult-xeon (8 Threads)	(150, 32, 80)	57.12×	(100, 20, 120)	8.37×
dsyrk-nehalem (8 Threads)	(150, 30, 120)	0.86×	(30, 30, 90)	2.03×
dsyrk-power7 (32 Threads)	(32, 70, 300)	13.79×	(60, 10, 1000)	2.54×
dsyrk-xeon (8 Threads)	(30, 10, 90)	3.64×	(100, 30, 80)	4.66×
dtrmm-nehalem (8 Threads)	(32, 50, 32)	0.93×	(150, 30, 60)	7.83×
dtrmm-power7 (32 Threads)	(10, 30, 100)	1.90×	(600, 30, 32)	5.25×
dtrmm-xeon (8 Threads)	(1, 1, 30)	1.69×	(30, 10, 120)	4.88×
2d-jacobi-nehalem (8 Threads)	(10, 50, 120)	1.77×	(10, 8, 150)	1.13×
2d-jacobi-power7 (32 Threads)	(10, 32, 120)	2.12×	(10, 40, 120)	1.76×
2d-jacobi-xeon (8 Threads)	(10, 40, 600)	3.06×	(50, 40, 20)	3.44×
2d-fdtd-nehalem (8 Threads)	(30, 80, 8)	14.08×	(50, 50, 8)	6.79×
2d-fdtd-power7 (32 Threads)	(10, 80, 8)	15.17×	(40, 70, 40)	4.57×
2d-fdtd-xeon (8 Threads)	(10, 60, 8)	11.99×	(50, 100, 8)	4.09×

that these search methods can take arbitrary tile sizes in the search space, and hence found slightly better tile sizes in some cases than Table 3, which shows the result of scanning the search space with strided access.

Parallel Execution of Tiled Code. Table 5 reports the best tile sizes found by an exhaustive empirical search using DL/ML bounds when the outer-most tiling loop is parallelized with OpenMP `parallel for` directives. It shows the speedup with respect to the untiled sequential execution when running each program with all cores (parallel) and when running with a single core (sequential, same as Table 3). Although the performance with parallelization is not always better than sequential, the best tile sizes for parallelized benchmarks also lie in the region bounded by the proposed DL/ML model except for dtrmm and jacobi-2d on *Xeon*, whose parallel performance is lower than sequential performance. This performance degradation results from inefficient data distribution, which may also cause unexpected effects on tile size selection.

7 Related Work

Exploiting data locality is a key issue in achieving high levels of performance and tiling has been widely used to improve data locality in loop nests. Nevertheless, the choice of tile sizes greatly influences the realized performance. Wolf and Lam [34] were the first to provide precise definitions of reuse and locality and develop transformations to improve locality. Ferrante el al. [12], Wolf and Lam [34], and Bodin et al. [5] were among the earliest to develop cache estimation techniques designed for data locality optimizations. Several authors proposed techniques for selecting tile sizes aimed at reducing self-interference misses [10, 20]. Ghosh et al. [13] developed cache miss equations to find sizes of the largest tiles that eliminate self-interference, while fitting in cache. Chame and Moon [8] developed techniques to minimize the sum of the capacity and cross-interference misses while avoiding self-interference misses. Rivera and Tseng [26] developed padding techniques to reduce interference misses and studied the effect of multi-level caches on data locality optimizations. Hsu and Kremer [16] presented a comprehensive comparative study of tile size selection algorithms. To the best of our knowledge, all of these techniques find a single tile size for each loop that is being tiled. Recently, Yuki et al. [38] have explored the automatic creation of cubic tile size models. In contrast, we have demonstrated (see Table 3) that the best performance is often realized only for rectangular tiles.

Search-based techniques for finding tile sizes (and unroll factors) have received much attention in performance optimization [4, 19, 31, 32, 33]. The ATLAS system employs extensive empirical tuning to find the best tile sizes for different problem sizes in the BLAS library; tuning is done once at installation. Unfortunately such an approach is not suited for general tiled codes, as the search process is tuned for dense linear algebra codes only. Only square tile sizes are considered, which significantly hampers the performance of a variety of codes (such as stencil codes) that require rectangular tiles for best performance. Furthermore, ATLAS currently includes a simplistic model where tile sizes are searched as to not exceed the square root of the L1 cache size. Our analytical bounds offer a significantly higher accuracy, capturing both intra- and inter-tile reuse at various cache level.

Kisuki et al. [19] have used different techniques such as genetic algorithms and simulated annealing to manage the size of the search space. Tiwari et al. [32] note: "a key challenge that faces auto-tuners, especially as we expand the scope of their capabilities, involves scalable search among alternative implementations." The Active Harmony project [31, 32] uses several different algorithms to reduce the size of the search space such as the Nelder-Mead simplex algorithm. In contrast to these approaches, we use a pair of analytical models — a conservative model that overestimates the number of cache lines by ignoring lifetimes and an aggressive model that underestimates the number of cache lines — each leading to different sets of tile sizes, which are used to bound the search space. With our technique, any of the algorithms from [19, 31, 32] can be used to further reduce the search time.

8 Conclusion

In this paper we developed a novel approach to analytically bound the search space for tile size selection based on two models, a conservative model (DL) that ignores intra-tile cache block replacement and a new aggressive model (ML) that assumes optimal replacement. We described how empirical search can be restricted (pruned) by the two models (DL and ML). Search space reductions ranging from $45\times$ - $11,879\times$ were obtained by using this pruning technique for five benchmarks on three different platforms. Our experimental results for single-level tiling on different benchmarks show that almost all tile sizes that deliver 95% or more of the optimal performance fall between the ML and DL bounds used in our approach. Furthermore, we demonstrated the integration of the analytical bounds with existing search optimization algorithms, and the experimental results show that the total search time was reduced by factors ranging from $1.02\times$ to $4.95\times$. The experiments for parallel execution show that our DL/ML model is also effective for tile size selection of parallelized programs. Taken together, these experimental results make a convincing case of the effectiveness of our new approach to model-driven empirical search for tile sizes.

For future work, we propose to extend our approach to multi-level tiling, and also to leverage correlation studies to identify which levels of the memory hierarchy are most closely tied to performance and compute DL and ML bounds for those hierarchy levels.

Acknowledgments. We thank the reviewers for their feedback and suggestions to improve the presentation of the paper, and we are grateful to Jill Delsigne for her assistance with proof-reading the final version of this paper. This work was supported in part by the Defense Advanced Research Projects Agency through AFRL Contract FA8650-09-C-7915, the Center for Domain-Specific Computing (CDSC) funded by the NSF Expedition in Computing Award CCF-0926127, the U.S. National Science Foundation through awards 0811457, 0811781, 0926687 and 0926688, and by the U.S. Army through contract W911NF-10-1-0004. The opinions and findings in this document do not necessarily reflect the views of the United States Government, Rice University, Ohio State University or Louisiana State University.

References

1. Barr, T.W., Cox, A.L., Rixner, S.: Translation caching: skip, don't walk (the page table). In: ISCA 2010, pp. 48–59. ACM, New York (2010)
2. Baskaran, M., Hartono, A., Tavarageri, S., Henretty, T., Ramanujam, J., Sadayappan, P.: Parameterized tiling revisited. In: CGO, pp. 200–209 (2010)
3. Bhargava, R., Serebrin, B., Spadini, F., Manne, S.: Accelerating two-dimensional page walks for virtualized systems. In: ASPLOS XIII, pp. 26–35 (2008)
4. Bilmes, J., Asanovic, K., Chin, C., Demmel, J.: Optimizing matrix multiply using PHiPAC. In: Proc. ICS, pp. 340–347 (1997)
5. Bodin, F., Jalby, W., Windheiser, D., Eisenbeis, C.: A quantitative algorithm for data locality optimization. In: Code Generation, pp. 119–145 (1991)
6. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral program optimization system. In: PLDI (2008)
7. Boulet, P., Darte, A., Risset, T., Robert, Y. (Pen)-ultimate tiling? Integration, the VLSI Journal 17(1), 33–51 (1994)
8. Chame, J., Moon, S.: A tile selection algorithm for data locality and cache interference. In: ICS, pp. 492–499 (1999)
9. Chen, C., Chame, J., Hall, M.: Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In: CGO 2005 (2005)
10. Coleman, S., McKinley, K.: Tile Size Selection Using Cache Organization and Data Layout. In: PLDI, pp. 279–290 (1995)
11. Datta, K.: Auto-tuning stencil codes for cache-based multicore platforms. Technical report, University of California, Berkeley (December 2009)
12. Ferrante, J., Sarkar, V., Thrash, W.: On Estimating and Enhancing Cache Effectiveness. In: Banerjee, U., Nicolau, A., Gelernter, D., Padua, D.A. (eds.) LCPC 1991. LNCS, vol. 589, pp. 328–343. Springer, Heidelberg (1992)
13. Ghosh, S., Martonosi, M., Malik, S.: Cache miss equations: a compiler framework for analyzing and tuning memory behavior. ACM TOPLAS 21(4), 703–746 (1999)
14. Goto, K., van de Geijn, R.A.: High-performance implementation of the level-3 BLAS. ACM Trans. Math. Softw. 35(1) (July 2008)
15. Hartono, A., Baskaran, M.M., Bastoul, C., Cohen, A., Krishnamoorthy, S., Norris, B., Ramanujam, J., Sadayappan, P.: Parametric multi-level tiling of imperfectly nested loops. In: Proc. ICS (2009)
16. Hsu, C., Kremer, U.: A quantitative analysis of tile size selection algorithms. J. Supercomput. 27(3), 279–294 (2004)
17. Irigoin, F., Triolet, R.: Supernode partitioning. In: ACM POPL, pp. 319–329 (1988)
18. Kim, D., Renganarayanan, L., Strout, M., Rajopadhye, S.: Multi-level tiling: 'm' for the price of one. In: SC (2007)
19. Knijnenburg, P.M.W., Kisuki, T., O'Boyle, M.F.P.: Combined selection of tile sizes and unroll factors using iterative compilation. The Journal of Supercomputing 24(1), 43–67 (2003)
20. Lam, M., Rothberg, E., Wolf, M.: The cache performance and optimizations of blocked algorithms. In: Proc. 4th ACM ASPLOS, pp. 63–74 (1991)
21. Luersen, M., Riche, R.L., Guyon, F.: A constrained, globalized, and bounded nelder-mead method for engineering optimization. Structural and Multidisciplinary Optimization 27(1-2), 43–54 (2004)
22. Nelder, J.A., Mead, R.: A simplex method for function minimization. Computer Journal 7(4), 308–313 (1965)
23. Ramanujam, J., Sadayappan, P.: Tiling multidimensional iteration spaces for multicomputers. JPDC 16(2), 108–230 (1992)

24. Renganarayana, L., Kim, D., Rajopadhye, S., Strout, M.: Parameterized tiled loops for free. In: PLDI, pp. 405–414 (2007)
25. Resource Characterization in the PACE Project, http://www.pace.rice.edu/Content.aspx?id=41
26. Rivera, G., Tseng, C.: Locality optimizations for multi-level caches. In: SC (1999)
27. Sarkar, V.: Automatic Selection of High Order Transformations in the IBM XL Fortran Compilers. IBM J. Res. & Dev. 41(3) (May 1997)
28. Sarkar, V., Megiddo, N.: An analytical model for loop tiling and its solution. In: IEEE ISPASS (2000)
29. Schreiber, R., Dongarra, J.: Automatic blocking of nested loops. Tech. Report 90.38, RIACS, NASA Ames Research Center (1990)
30. Tabatabaee, V., Tiwari, A., Hollingsworth, J.K.: Parallel parameter tuning for applications with performance variability. In: Proc. Supercomputing 2005 (2005)
31. Tapus, C., Chung, I.-H., Hollingsworth, J.K.: Active harmony: towards automated performance tuning. In: SC, pp. 1–11 (2002)
32. Tiwari, A., Chen, C., Chame, J., Hall, M., Hollingsworth, J.: Scalable autotuning framework for compiler optimization. In: IPDPS 2009 (2009)
33. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimization of software and the ATLAS project. Parallel Computing 27(1–2), 3–35 (2001)
34. Wolf, M., Lam, M.S.: A data locality optimizing algorithm. In: PLDI 1991, pp. 30–44 (1991)
35. Wolfe, M.: More iteration space tiling. In: Proc. Supercomputing, pp. 655–664 (1989)
36. Xue, J.: Loop tiling for parallelism. Kluwer Academic Publishers, Norwell (2000)
37. Yotov, K., Pingali, K., Stodghill, P.: Think globally, search locally. In: International Conference on Supercomputing (2005)
38. Yuki, T., Renganarayanan, L., Rajopadhye, S., Anderson, C., Eichenberger, A., O'Brien, K.: Automatic creation of tile size selection models. In: CGO, pp. 190–199 (2010)

Static Detection of Unsafe Component Loadings

Taeho Kwon and Zhendong Su

Department of Computer Science, University of California, Davis
{kwon,su}@cs.ucdavis.edu

Abstract. Dynamic loading of software components is a commonly used mechanism to achieve better flexibility and modularity in software. For an application's runtime safety, it is important for the application to load only its intended components. However, programming mistakes may lead to failures to load a component, or even worse, to load a malicious component. Recent work has shown that these errors are both prevalent and severe, sometimes leading to remote code execution attacks. The work is based on dynamic analysis by monitoring and analyzing runtime component loadings. Although simple and effective in detecting real errors, it suffers from limited code coverage and may miss important vulnerabilities. Thus, it is desirable to develop effective techniques to detect *all possible* unsafe component loadings.

This paper presents the first *static* binary analysis aiming at detecting all possible loading-related errors. The key challenge is how to scalably and precisely compute what components may be loaded at relevant program locations. Our main insight is that this information is often determined locally from the component loading call sites. This motivates us to design a demand-driven analysis, working backward starting from the relevant call sites. In particular, for a given call site c, we first compute its *context-sensitive executable slices*, one for each execution context. Then we emulate the slices to obtain the set of components possibly loaded at c. This novel combination of slicing and emulation achieves good scalability and precision by avoiding expensive symbolic analysis. We implemented our technique and evaluated its effectiveness against the existing dynamic technique on nine popular Windows applications. Results show that our tool has better coverage and is precise—it is able to detect many more unsafe loadings. It is also scalable and finishes analyzing all nine applications within minutes.

1 Introduction

Dynamic component loading is widely used in software development to build flexible and modular software. Operating systems (OSes) typically provide relevant system calls, such as `dlopen`, to load dynamic components. Once a loading system call is invoked, the underlying OS resolves and loads the specified component. Component resolution depends on how the component is specified—either through the intended component's *full path* or its *file name*. Given a full path, the OS simply uses it for resolution. Given only a file name, the OS searches over a sequence of directories to locate a file with the specified name. Which sequence of directories to search is controlled at runtime by the particular directory search order at the time of system call invocation.

M. O'Boyle (Ed.): CC 2012, LNCS 7210, pp. 122–143, 2012.
© Springer-Verlag Berlin Heidelberg 2012

The flexibility of this common style of component loading does come with a price—it introduces an inherent security concern. For runtime safety and security, an application should only load its intended components. However, as the OS resolves a component only through its name, programming mistakes can lead to the loading of an unintended component with the same name.

Although this issue was known, it is not until recently that studies have shown how prevalent and serious the issue is in practice. In particular, it is shown that unsafe loadings on Microsoft Windows are prevalent and can lead to remote code execution attacks [21]. Remote attacks are possible for two main reasons: 1) the OS looks for a component with a given file name and cannot distinguish malicious ones from benign ones with the same file name; and 2) the default directory search order on Microsoft Windows contains the current directory (*i.e.*, "."), where remote attackers can trick a victim user to download files to via social engineering or by exploiting other vulnerabilities.

Here is an example attack scenario on Windows. An attacker sends a victim user via email an archive that contains an arbitrary `.asx` file and a malicious file named `rapi.dll`. The user extracts the archive file and runs `Winamp 5.58` to open the `.asx` file, the `rapi.dll` is loaded, which leads to a remote code execution attack [21]. Besides archive files, the Carpet-Bomb attack [28] and the WebDAV protocol [2] can be exploited for launching remote attacks. This very issue has also received considerable recent media coverage [11,25,27,36,44]. Microsoft released MS10-087, rated "Critical," to patch Microsoft Office [42]. To mitigate the issue, Microsoft also released a fix-it tool to control the directory search order by introducing a new registry key [17,26]. However, it changes the default system-wide setting and leads to backward compatibility issues. Fundamentally, this is a safe programming issue. Microsoft provides programming guidelines for safe dynamic loading [10] and is conducting an ongoing investigation to secure the loading procedure [23].

As the root cause of the issue is unsafe programming, it is important to detect this class of dangerous programming mistakes. Kwon and Su [21] proposed a dynamic technique to detect unsafe component loadings. This technique collects at runtime loading-related information—such as the target component to be loaded, the directory search order, and the actually loaded component—at each of the invocation sites for the loading system call. It then performs an offline analysis to detect two types of unsafe loadings: *resolution failure* and *unsafe resolution*. A resolution failure happens when the target component is not found, while an unsafe resolution happens when other directories are searched before the directory where the loaded component resides. Besides crashing an application, unsafe loadings also make the application vulnerable to component hijacking.

Although the proposed dynamic technique [21] is effective at detecting real unsafe loadings, it may miss errors because of limited code coverage, an inherent weakness of dynamic analysis. We illustrate this issue using *delayed loading*, an optimization to postpone the loading of infrequently used components until their first use. Delayed loading is challenging for dynamic detection because it is difficult to trigger all delayed loadings at runtime. Figure 1 shows a code snippet that uses delayed loading in Microsoft Windows. The code shows two functions `f1` and `f2` that use components registered for delayed loading. In particular, `f1` and `f2` retrieve the addresses of `OpenPrinter` exported by `WINSPOOL.DRV` and `GetSaveFile` exported by `COMDLG32.DLL` respectively.

Although the example only shows two functions `f1` and `f2`, in practice, there are often many more. The infrequent use of the components makes it difficult, if not impossible, to trigger all possible loadings at runtime. Although we have illustrated the problem using delayed loading, poor coverage of dynamic analysis is a general concern for detecting unsafe loadings, as our results also confirm (*cf.* Section 3).

In this paper, we present the first *static* analysis to detect unsafe loadings from program binaries. Two pieces of essential information are needed: 1) all components that may be loaded at each loading call site, and 2) the safety of each possible loading. While the second part is straightforward, the key challenge lies in the first part—how to precisely and scalably compute the possible loadings. Our *key observation* is: for a given invocation of the loading system call, the set of possible loaded components is determined by the system call's parameter values, which are often determined through computations that originate not far from the call site. From these observations, we design a two-phase analysis: *extraction* and *checking*. The extraction phase is *demand-driven*, working backward from each loading call site to compute the set of possible loadings; the checking phase determines the safety of a loading by examining the relevant directory search order at the call site.

Context-Sensitive Emulation. To realize the backward computation of parameter values during the extraction phase, we introduce *context-sensitive emulation*, a novel combination of slicing and emulation. For a given call site, we extract its context-sensitive *executable* slices w.r.t. its parameters, one for each execution context. We then emulate the slices to compute the parameter values.

Incremental and Modular Slicing. One technical obstacle is how to compute backward slices scalably. Standard slicing techniques [1, 5, 13, 30, 35, 38] are based on computing a program's complete system dependence graph (SDG) *a priori* and are thus limited in scalability. Because we only need to consider loading call sites and the execution paths to compute the parameter values to the calls are usually relatively short, only a small fraction of the complete SDG is relevant for our analysis. This motivates the use of an *incremental* and *modular* slicing algorithm (*cf.* Section 2)—incremental because we build the slices lazily when necessary; modular because when we encounter a function call `foo(x,y)`, we use an inferred summary of what dependencies `foo`'s parameters and return value have in analyzing the caller. At the end, we connect the function-level slices in the standard way by linking formal and actual parameters.

```
1  void f1() {
2      ...
3      pDelayDesc1 = &WINSPOOL_DRV_DelayDesc;
4      // WINSPOOL_DRV_DelayDesc.dllname = "WINSPOOL.DRV"
5      func_addr = __delayLoadHelper2(
6         pDelayDesc1, "OpenPrinter"
7      );
8      ...
9  }
10 void f2() {
11     ...
12     pDelayDesc2 = &COMDLG32_DLL_DelayDesc;
13     // COMDLG32_DLL_DelayDesc.dllname = "COMDLG32.DRV"
14     func_addr = __delayLoadHelper2(
15        pDelayDesc2, "GetSaveFile"
16     );
17     ...
18 }
19 int __delayLoadHelper2(pImgDelayDesc, funcName) {
20     hMod = pImgDelayDesc->hMod; // init value = 0
21     if (hMod == 0) {
22        target_dllname = pImgDelayDesc->dllname;
23        hMod = LoadLibrary(target_dllname);
24        pImgDelayDesc->hMod = hMod;
25     }
26     func_addr = GetProcAddress(hMod, funcName);
27     return func_addr;
28 }
```

Fig. 1. Motivating example

Emulation of Context-sensitive Slices. Once we have computed the backward slice s w.r.t. a given loading call site, we need to compute possible values for the relevant parameters. One natural solution is to perform standard symbolic analysis on the slice to compute the values. The main challenge for this approach is the difficulty in reasoning symbolically about system calls because the relevant parameters often depend on complex, low-level system calls. For example, many Windows applications invoke the system call `RegQueryValueExW` to retrieve the fullpath of the target specification stored in the registry key. The system call invokes more than 100 distinct system calls exported by five libraries. To symbolically analyze the system call, it is necessary to symbolically execute its invoked system calls as well, leading to path explosion. Thus, it is difficult in practice to engineer and scale symbolic analysis to compute the possible values of the parameters.

To overcome this difficulty, we use emulation. In particular, we generate, from the backward slice s, a set of *context-sensitive executable sub-slices*, which we then *emulate* to compute the parameter values (*cf.* Section 2). Essentially, we inline callees' function-level slices in each execution context to produce s's sub-slices $s_1, \ldots, s_n$. Instructions in each sub-slice s_i are next emulated topologically, respecting their data- and control-flow dependencies.

For evaluation, we implemented our technique in a prototype tool for Windows applications. We evaluated our tool's effectiveness against the previous dynamic tool [21] in terms of precision, scalability, and coverage. Results on nine popular applications show that our tool is precise and scalable (*cf.* Section 3). For example, it took less than two minutes to analyze each of the nine test subjects, including large applications such as Acrobat Reader, Quicktime, and Safari. The results also show that our proposed context-sensitive emulation achieves orders of magnitude reduction in the size of the code needed to be analyzed and crucially contributes to the scalability of our technique. In terms of coverage, our tool detected many more possible unsafe loadings and nicely complements the dynamic technique.

Main Contributions

- We have developed the first static binary analysis to detect unsafe component loadings. Because of its scalability and higher code coverage, our technique effectively complements the existing dynamic technique.
- We have proposed context-sensitive emulation, an effective approach that combines slicing and emulation for the precise and scalable analysis of runtime values of program variables.
- We have implemented our technique and evaluated its effectiveness by detecting unsafe loadings in nine popular Windows applications.

The rest of this paper is organized as follows. Section 2 presents a detailed description of our static detection algorithm. We describe our implementation and evaluation in Section 3. Finally, Section 4 surveys additional related work, and Section 5 concludes with a discussion of future work.

2 Static Detection Algorithm

In this section, we present background information on unsafe component loadings and details of our analysis.

2.1 Background

Dynamic component loading is commonly supported by operating systems through specific system calls that take as input a full path or file name for the intended component. For example, Microsoft Windows provides component-loading system calls such as `LoadLibraryA`. Once such a system call is invoked, the OS resolves the target component as follows:

- The target component can be specified by its full path or its file name.
- When the full path is used, the OS directly resolves the target using the provided full path.
- Otherwise, if file name is used and is known by the OS, the full path of the specified file is predefined. For example, `KERNEL32.DLL` is known by Microsoft Windows and its full path is predefined as `"C:\WINDOWS\SYSTEM32\KERNEL32.DLL"`.
- If the given file name is unknown to the OS, it iterates through the predefined search directories to locate the first file with the specified file name.

To formalize the component resolution process, it is necessary to model the *file system state*, because even the same component-loading code may result in different resolutions under different file system states. We define a file system state s to be the set of full paths of all files stored on the current file system.

Definition 21 (Component Resolution). *A component resolution function R takes a component specification $f \in \Sigma^*$, a directory search order $d = \langle d_1, \ldots, d_n \rangle \in \Sigma^* \times \ldots \times \Sigma^*$ and a file system state s, and returns a resolved full path $\pi \in \Sigma^*$, where Σ denotes the alphabet used to specify files and directories.*

- *If f is a full path,*

$$R(f, d, s) = \begin{cases} f & \text{if } f \in s; \\ \epsilon & \text{otherwise.} \end{cases}$$

  where ϵ is the empty string.
- *If f is a file name,*

$$R(f, d, s) = \begin{cases} \pi & \text{if } f \text{ is known to the OS as } \pi; \\ d_k + \backslash + f & \text{if } S = \{i \mid d_i + \backslash + f \in s\} \\ & \quad \wedge S \neq \emptyset \wedge k = \min(S); \\ \epsilon & \text{otherwise.} \end{cases}$$

  where "$+$" denotes string concatenation.

We next formalize component loading, for which we need to consider the currently loaded components. The reason is that the OS does not load the same component multiple times. In our formalization, we let Π denote the set of full paths of all the currently loaded components.

Definition 22 (Component Loading). *Given the loaded components Π, a component loading function L takes a component specification $f \in \Sigma^*$, a directory search order $d = \langle d_1, \ldots, d_n \rangle \in \Sigma^* \times \ldots \Sigma^*$, a file system state s, and the set of loaded components Π, and returns a resolution success or failure:*

$$L(f, d, s, \Pi) = \begin{cases} \textit{success} & \text{if } R(f, d, s) \notin \{\epsilon\} \cup \Pi; \\ \textit{failure} & \text{otherwise.} \end{cases}$$

The formalized component loading mechanism in Definition 22 is commonly used on major operating systems. However, as the OS determines a target component only through its name, unsafe programming can make software load an unintended component with the same name. Attackers can exploit this security vulnerability by modifying the file system state. In particular, the loading of a target component can be hijacked if a malicious file with the same name can be created in a directory searched before the directory where the intended component resides. This component hijacking can be misused for local or remote attacks [21].

To formalize unsafe component loading, it is necessary to determine the current file system state as whether or not a component loading is safe is relative to a file system state. We first define a *normal file system state* w.r.t. an application p.

Definition 23 (Normal File System State). *A file system state s is* normal *w.r.t. an application p if no unintended components are loaded while p executes in state s. We use s_p to denote a normal file system state w.r.t. the application p.*

We formalize two types of unsafe loadings: *resolution failure* and *unsafe resolution*. We use R_p and L_p to denote component resolution and component loading performed by an application p, respectively.

Definition 24 (Resolution Failure). *For an application p, a resolution failure occurs at runtime if $R_p(f, d, s_p) = \epsilon$. In this case, with a full path specification f, an arbitrary file with the same full path f can hijack the component loading. If f is file name, an attacker can hijack this loading by placing a file (or tricking the user to place a file) with the specified name f in any directory d_i* writable *by the attacker under the search order $d = \langle d_1, \ldots, d_n \rangle$.*

Definition 25 (Unsafe Resolution). *For an application p, an unsafe resolution occurs at runtime if the following conditions hold: 1) f is the file name of the target component and unknown to the OS; 2) $R_p(f, d, s_p) = d_k + \backslash + f \wedge k > 1$; and 3) $L_p(f, d, s_s, \Pi) = success$. In this case, an attacker can hijack the loading by placing a file (or tricking the user to place a file) with the specified name f in any* writable *directory d_i by the attacker where $i < k$.*

To avoid unsafe loadings, it is necessary for developers to specify the target component in a safe manner. We define safe target component specifications as follows.

Definition 26 (Safe Component Spec). *Under a given threat model, a loading specification for an application p is safe if either of the following holds: 1) if f is a full path, $R(f, d, s_p) \neq \epsilon$ and the attacker cannot overwite f or trick the user to overwrite f; and 2) if f is an unknown file name to the OS, $R(f, d, s_p) = d_i + \backslash + f$ and the attacker cannot place a file or trick the user to place a file named f in any of the d_j for $1 \leq j \leq i$.*

If a loading specification is unsafe, it leads to resolution failure or unsafe resolution. While the first condition checks the resolution failure for the fullpath specification, the second condition checks whether the filename specification leads to resolution failure or unsafe resolution. As many Windows users have the administrator privilege, a realistic

threat model under Windows is that an attacker may be able to trick a (non-malicious) user to place a malicious component in a desired directory to hijack component loading. We have adopted this threat model in our evaluation.

2.2 Detailed Analysis

We now present the details of our analysis. Our technique statically detects unsafe component loadings to achieve high coverage. It first extracts the target component specifications from possible code region executed at runtime and checks their safety based on Definition 26.

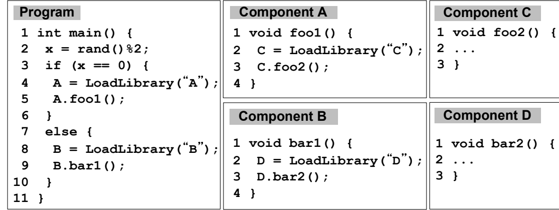


Fig. 2. Component-interoperating code

The executed code region is determined by loaded components. Figure 2 depicts the component loading code whose execution path is controlled by a random variable `x`. If `x` is zero, `foo1` of component `A` and `foo2` of component `C` are executed. Otherwise, `bar1` of component `B` and `bar2` of component `D` are executed. Our observation is that each execution path covers the partial code region of the loaded components. For example, if `x` is zero, the partial code regions of components `Program`, `A`, and `C` are executed. From these observations, we design our static detection as shown in Figure 3: *extraction* and *checking*. From the extraction phase, we obtain a set of the target component specifications from the components that can be loaded at runtime. In the checking phase, we evaluate the safety of each target specification based on Definition 26.

Extraction Phase. A component can load other components at loadtime or runtime. This loading introduces *loadtime* and *runtime* dependencies among components [41]. Based on these dependencies, we determine components that can be loaded during program execution. Specifically, we recursively resolve the components from the program file based on their loadtime and runtime dependencies. To resolve the dependent components, the corresponding target specifications, *i.e.*, full path or file name, are needed. For loadtime dependencies, compilers specify the dependent components in the executable format. For example, the names of the loadtime dependent components are stored in `IMAGE_IMPORT_DIRECTORY` with the PE format [24]. To obtain the specifications of the runtime dependent components, we compute values of parameters to component-loading system calls. This suffices for our setting because the program dynamically loads components via the system calls and their parameters determine the loaded components.

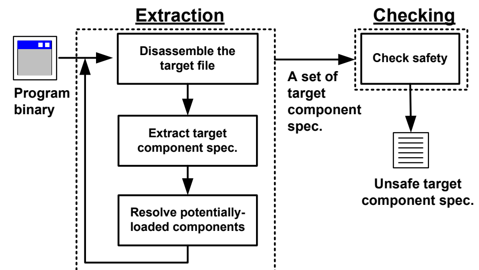


Fig. 3. Detection framework

As an example of recursive resolution, we search the components that can be loaded by `Program` in Figure 2. Suppose that components `E` and `F`, which have no loadtime

```
1   PUSH   EAX
2   PUSH   EAX
3   PUSH   offset 0x7D61AC5C; "xpsp2res.dll"
4   CALL   DWORD PTR DS:[LoadLibraryExA]
```

(a) Memory indirect

```
1   MOV    EBX, DWORD PTR DS:[LoadLibraryW]
2   PUSH   offset 0x65015728; "CABINET.DLL"
3   CALL   EBX
```

(b) Register indirect

Fig. 4. Two types of component-loading call sites

or runtime dependent components, implement the `rand` and `LoadLibrary` functions, respectively. In this case, `Program` loads components `E` and `F` on its startup. Regarding runtime dependencies, `Program` dynamically loads components with the specifications, `"A"` on line 4 and `"B"` on line 8. From this information, we can detect the potentially-loaded components by simulating component resolution. Similarly, we can infer that `C`, `D` and `F`, which are loaded by `A` and `B`. Because `C` and `D` have no loadtime and runtime dependent components, we stop the resolution process. Thus, we detect the seven components potentially loaded at runtime: `Program`, `A`, `B`, `C`, `D`, `E`, and `F`.

The key step of the extraction phase is to obtain the target specification for component loading in a binary. The specification of a loadtime dependent component can be easily obtained from the binary file format. However, extracting the specification of a runtime dependent component is nontrivial because it often requires to locate the code relevant to the value of the specification and analyze its execution. For example, the target component specification for system libraries under Microsoft Windows is sometimes determined by concatenating the system directory path and the file name. To obtain the specification, it is necessary to extract the related code and analyze its execution result.

The concrete value of the parameter to the component-loading system call serves as the specification for the runtime dependent component. From this observation, we extract the specification by searching for the program variable for the specification and then computing its value via *context-sensitive emulation*, a novel combination of backward slicing and emulation. We describe details of the extraction in the following sections.

Searching Program Variable for Specification. In binary code, invoking the component-loading system calls follow the `stdcall` calling convention [45]. When parameters are passed to the call site, they are pushed from right to left. For example, Figure 4(a) represents the binary code corresponding to `LoadLibraryExA(0x7D61AC5C, EAX, EAX)`. Based on the parameter passing mechanism, we locate the program variable, *e.g.*, a register or a memory chunk, which stores the target specification. In particular, we detect the call site for component loading via static taint data analysis and then extract the input operands of the instructions passing the parameter to the call site. We describe details of each step in the rest of this section.

Locating Component-loading Call Sites. In this phase, we aim at finding the call site for component loading in a binary. Our observation is that software stores the address of the system call implementation in its memory space and utilizes it in the call sites for component loading at runtime. Figure 4 shows the two types of component-loading call sites in a binary, which are *memory indirect* and *register indirect*. The main difference between them is what type of program variable stores the address of the

component-loading system call at the call site. While the memory indirect type stores the address in a memory chunk, the register indirect type stores the address in a register, *e.g.*, line 4 in Figure 4(a) and line 3 in Figure 4(b).

Based on this observation, we locate the component-loading call sites through static taint data analysis. In particular, we define the taint sources and the taint sinks as follows:

– *Taint source*: an instruction that references a memory chunk that stores the address of the component-loading system call.
– *Taint sink*: a branch instruction, *e.g.*, `call`, whose target address is tainted. We consider the taint sink instructions as the call sites.

We now present examples on how to detect call sites. In Figure 4(a), line 4 serves as not only the taint source but also the taint sink, *i.e.*, the component-loading call site, because it is the branch instruction, accessing a memory chunk that stores the address of `LoadLibraryExA`. For Figure 4(b), line 1 is the taint source, accessing the address of the `LoadLibraryA`, and line 3 is the taint sink, because it is the `call` instruction whose target is the address, stored in `EBX`.

Extracting Parameter Variables. Once a call site is located, we extract the program variables for the target specification from the predefined number of the instructions to pass the parameters to the call site. In particular, we detect the instructions, *e.g.*, `PUSH`, to initialize the top of stack backward from the call site. Because the number of parameters of a component-loading system call is known, we can precisely extract all the variables to define this target specification. For example, the call site in Figure 4(a) invokes `LoadLibraryExA`, and it has three parameters, *i.e.*, `0x7D61AC5C`, `EAX`, and `EAX`, via the instructions on lines 1–3.

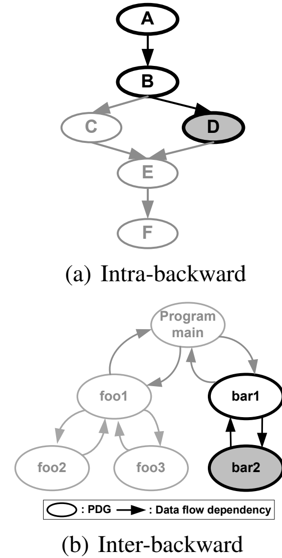
Context-Sensitive Emulation. In this phase, we compute the concrete values of the parameter variables extracted in Section 2.2. The computation may seem trivial at first. For example, the memory chunk at `0x7D61AC5C` in Figure 4(a) contains the target specification, `"xpsp2res.dll"`. However, the computation is in fact challenging because it is necessary to extract the code to compute the variable, requiring interprocedural data flow analyses (*cf.* Figure 1). Also, we need the runtime information of the code to obtain the concrete values of the variable. Symbolic analysis can serve as a potential solution. However, as we mentioned in Section 1, symbolic analysis suffers from poor scalability and is limited in handling system calls, which are often complex.

To address this problem, we introduce *context-sensitive emulation*, which novely combines backward slicing and emulation. Based on this combination, we can scalably and precisely compute the values of the variables of interest. We describe its details in the rest of this section.

Backward Slicing. This phase performs the interprocedural backward slicing w.r.t. the parameter variable, extracting the instructions to compute the variable. This problem has been extensively studied, and many slicing algorithms [1,5,13,30,35,38] have been proposed. These algorithms commonly solve the graph reachability problem over a System Dependence Graph (SDG) [13], a set of Program Dependence Graphs (PDGs) [12]

and edges capturing data flow dependencies among them. In particular, a SDG is constructed beforehand based on an exhaustive data flow analysis over the subject program. Then, the slicing outcome is determined by traversing the SDG from the given slicing criteria. Although the approach has been widely used, it is not appropriate for our problem setting. The reason is that binary files are generally composed of a large number of instructions, and an exhaustive data flow analysis over all the instructions is very expensive, leading to limited scalability.

Our key observation is that the parameter values are often locally determined, that is the execution paths to compute the variables are relatively short. Thus, exhaustive data flow analysis is not be necessary to extract backward slices w.r.t. the given slicing criteria. Figure 5 shows examples of the unnecessary data flow analysis during intraprocedural and interprocedural backward slicing.



(a) Intra-backward

(b) Inter-backward

Fig. 5. Unnecessary data flow analysis

Figure 5(a) shows an example of the CFG for constructing the PDG. Suppose that we perform intraprocedural backward slicing w.r.t. the instruction `D`. In this case, the bold instructions often only affect the instruction `D` in terms of control flow. It is possible that the instruction `D` can be affected by the instructions without control flow dependencies. For example, the instruction `E` initializes a variable and the instruction `B` reads it. However, this case rarely happens in our problem setting in practice, because the parameters for the specification are generally computed by the instructions executed before the component-loading call sites.

Suppose that Figure 5(b) depicts the SDG for the interprocedural backward slicing. If the instructions of the bold PDGs for `bar1` and `bar2` are only traversed during slicing, it is not necessary to perform data flow analysis on the instructions of the grayed PDGs. Because the SDG consists of a large number of PDGs in binary and the target specifications are often locally determined, most of the PDGs are not relevant for interprocedural backward slicing w.r.t. the parameter variables for the target specifications.

Based on this insight, we design our slicing technique as demand-driven, reducing the unnecessary analysis of data flow dependencies. In particular, we perform interprocedural backward slicing by incrementally combining intraprocedural backward slices whose slicing criteria are determined when necessary.

Intraprocedural backward Slicing. For each intraprocedural backward slicing, we analyze only the data flow dependencies among the instructions that are control dependent on the given slicing criteria. To this end, we construct the PDG based on the *predecessor subgraph* w.r.t. the slicing criterion under the CFG. Thus, we can avoid the analysis of the data flow dependency among the instructions not traversed during slicing. Suppose that we perform intraprocedural backward slicing w.r.t. the instruction `D` in the CFG shown in Figure 5(a). If we construct the PDG based on the CFG, the data flow

dependencies among all the instructions in the CFG are analyzed. However, the grayed instructions do not affect the instruction `D` in terms of control flow dependencies. By constructing the PDG based on the subgraph composed of the bold instructions, *i.e.*, the predecessor subgraph w.r.t. the instruction `D`, we can avoid some unnecessary data flow analysis when performing slicing.

One challenge for PDG construction is caused by the call site instructions. Because functions are not generally monolithic, it is necessary to identify which call sites affect the slicing criteria. Although traversing the SDG provides such information, it requires the computation of significant amount of unnecessary data-flow dependencies (*cf.* Figure 5(b)). To address this problem, we utilize the prototypes of the functions invoked at the call sites. Specifically, we consider a call site instruction as a non-branching instruction during our PDG construction, and analyze the data flow dependencies related to the call site in terms of the prototype of the callee function. For example, a call site invokes a function `foo` whose prototype is `int foo(in,inout)`. In this case, the call to `foo` is considered to be an instruction that uses the first and second parameters and defines the second parameter and the return variable. Based on this information, we can effectively determine the data flow dependencies between the call site instructions and the slicing criteria without a whole SDG traversal.

Interprocedural backward Slicing. As aforementioned, an exhaustive SDG construction often leads to significant amount of data flow analysis that is unnecessary for interprocedural backward slicing. To address this problem, we construct the interprocedural backward slices incrementally combining the intraprocedural backward slices whose slicing criteria are chosen in a demand-driven manner.

There are two key challenges for this demand-driven combination. First, it is necessary to determine the new slicing criteria if the interprocedural backward slice consists of multiple intraprocedural backward slices. For example, we construct the interprocedural backward slice in Figure 5(b) by combining the two intra-backward slices extracted from functions `bar1` and `bar2`. In this case, we need to determine the new slicing criteria in the `bar1` function. Second, the composed interprocedural backward slice needs to be easily handled for the later emulation phase.

Our basic idea for building the new slicing criteria is that the interprocedural data flow dependencies are captured by parameter passing. In SDG-based slicing, the PDGs are connected using the edges that model parameter passing, which are traversed to analyze the dependencies. Based on this idea, we choose the slicing criteria as follows. Suppose that an intraprocedural backward slice s is extracted from an instruction whose input operand is initialized through parameter p of the function f. In this case, we determine the new slicing criterion as the parameter

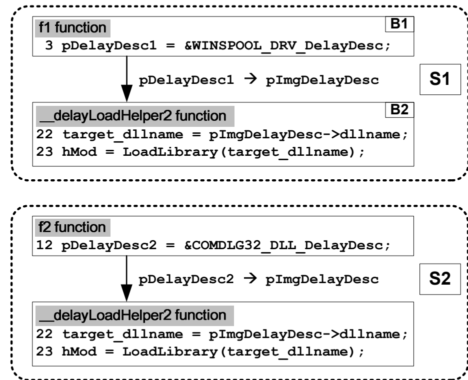


Fig. 6. Example context-sensitive backward slices

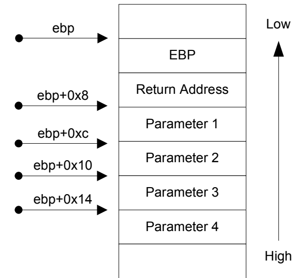
variable corresponding to the parameter p. To locate this parameter variable, we use *caller-callee relationship* and the *callee's function prototype*. In particular, we detect the call site for function f and analyze f's function prototype to obtain the index of the parameter corresponding to p. For example, the intraprocedural backward slice w.r.t. the `target_dllname` in Figure 1 uses the first parameter, *i.e.*, `pImgDelayDesc`, of `__delayLoadHelper2`. As two call sites on lines 5–7 and lines 14–16 invoke `__delayLoadHelper2`, we choose their first parameter variables, *i.e.*, `pDelayDesc1` on line 6 and `pDelayDesc2` on line 15, as the new slicing criterion.

Once the new slicing criterion is determined, we construct the interprocedural backward slice by composing the intraprocedural backward slices and use the composed slice in the emulation phase. One simple method for composing the intraprocedural slices is to collect the instructions of each intraprocedural backward slice. For example, the interprocedural backward slice w.r.t. the `target_dllname` in Figure 1 consists of the instructions of three intraprocedural backward slices w.r.t. the slicing criteria, *i.e.*, `target_dllname`, `pDelayDesc1`, and `pDelayDesc2`. However, this simple method produces *context-insensitive* slices, making the emulation phase complex. In particular, when emulating each instruction of the context-insensitive slice, we have to assume that the values of its operands are determined under all of its calling contexts.

To better support emulation, we combine the intraprocedural backward slices to construct a set of *context-sensitive interprocedural backward slices*. In particular, for a given intraprocedural backward slice s, if multiple new slicing criteria, $p_1 \dots p_n$, are determined, the set of the context-sensitive slices are constructed as $\{s_i \cup s | s_i = \cup_{p_i}$ intraprocedural backward slice w.r.t. p_i where $1 \leq i \leq n\}$. Thus, we can more straightforwardly use the context-sensitive slices to compute possible concrete values of the target component specification. For example, Figure 6 shows the context-sensitive interprocedural backward slices w.r.t. `target_dllname` in Figure 6. We can compute the possible values of `target_dllname` by emulating them. We describe more details of our backward slicing phase in an earlier version of this paper [22].

```
foo:
1  ...
2  mov eax, [ebp+0xc] ; 2nd
3  ...
4  mov [ebp+0x8], eax ; 1st
5  ...
6  mov eax, [ebp+0x10] ; 3rd
7  ...
8  mov [ebp+0x14], eax ; 4th
9  ...
```

(a) Parameter access



(b) Stack layout

Fig. 7. Function prototype analysis

Function Prototype Analysis. The backward slicing phase relies on function prototypes, but such information is often unavailable in binary code. Our solution to this problem is as follows. For a given function f, its parameters are stored in fixed locations during f's execution. Thus, we infer its prototype by analyzing how the instructions of the function access the memory chunks for the parameters, *i.e.*, read or write.

Figure 7 shows an example of our proposed prototype analysis for the `foo` function. Suppose that Figures 7(a) and 7(b) show part of `foo` and the stack layout at the beginning of the function's execution, respectively. In this case, the `idx`-th parameter is stored at the address $\texttt{ebp} + 4 \times (\texttt{idx} + 1)$ where the stack is aligned by four bytes. From this observation, we can infer `foo`'s prototype. It reads data from the memory chunks for its second and third parameters, and initializes the memory chunks for its first and fourth parameters, *i.e.*, its function prototype is `"eax foo(inout,in,in,inout)"`. Here we assume that its result is returned through the `eax` register.

To improve the precision of our prototype inference, we use the following effective heuristic. If the effective address of the memory chunk, obtained by the `lea` instruction, is passed to the function, we consider it as the `inout` parameter. The effective address corresponds to a pointer variable and the memory chunk that it points to is often initialized during function execution. Although this heuristics may increase the size of the computed slice, it is sufficient to compute possible values of the slicing criteria via emulation.

Emulation Phase. In this phase, we compute the possible values of the target component specification by emulating its corresponding context-sensitive slices. There are three challenges for slice emulation. The first challenge is how to schedule the instructions because we do not know their runtime execution sequence. If the instructions are incorrectly scheduled, they may violate the data and control flow dependencies among them, which may lead to imprecise results or emulation failures. The second challenge is how to pass function parameters. Although parameter passing captures useful data flow dependencies, the context-sensitive slices do not explicitly specify the dependencies. The third challenge is how to handle the call site instructions. Because we perform the data flow analysis by considering a call site as an instruction, the backward slice does not contain detailed code of the callee function.

Scheduling Algorithm. To develop a practical scheduling algorithm, we have analyzed all 682 backward slices extracted from nine popular Windows applications (*cf.*, Table 1). We have observed that all the extracted slices form directed acyclic graphs. Therefore, we schedule the basic blocks in their topological order w.r.t. dataflow dependency. We then determine the order of the instructions of each basic block w.r.t. their sequence in the original code. For example, Figure 8 shows the data flow dependency among the basic blocks of the first slice in Figure 6. In this case, we schedule the basic blocks as `B1`, `B2`, and `B3`. For each basic block, the sequence of its instructions is determined as

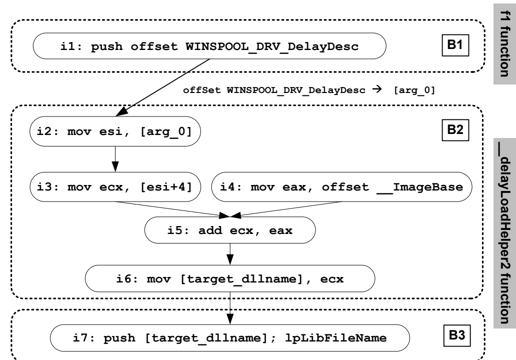


Fig. 8. Data-flow dependency among basic blocks

follows: `i1`, `i4`,`i2`,`i3`,`i5`,`i6`, and `i7`. The scheduled sequence of the instructions does not violate the data- and control-flow dependency among them.

Parameter Passing. To handle parameter passing, we initialize the stack frame before emulating the callee function. In particular, suppose that a parameter p is passed to a function f. In this case, before emulating f's basic blocks, we reserve the stack frame and initialize its memory chunk for the parameter with the concrete value of p. The location of the memory chunk is determined by the index of the passed parameter. For example, the address of the memory chunk for the `idx`-th parameter can be computed by $\texttt{ebp} + 4 \times (\texttt{idx} + 1)$, (*cf.* Figure 7).

For example, Figure 8 shows how we handle the parameter passing from `f1` to `__delayLoadHelper2`. When `B1` is emulated, top of the call stack for `f1` stores `offset WINSPOOL_DRV_DelayDesc`. Assuming that the initial value of `esp` for emulating `B2` is equal to `0x13f258`, the stored value initializes a memory chunk at `arg_0=0x13f258+4×2`, because it corresponds to the first parameter to `__delayLoadHelper2`. The instructions use `arg_0` to reference the first parameter (*e.g.*, `i2`).

Call Site Instruction. To obtain the possible values of the target component specification, it is necessary to emulate the call site instruction. If the code of the invoked function resides in the current file, we can simply emulate the corresponding code. However, if the call site invokes a system call, we may not be able to obtain the code from the current file. Figure 9 shows an example slice with external library calls where each edge represents data flow dependency between two instructions. The slice determines the fullpath of the target component by concatenating the path to the system directory with a string `\kernel32.dll`. In this case, the instructions invoked by `i5` and `i10` are not available in the current file. In particular, `GetSystemDirectoryW` and `wcscat_s` are implemented in `KERNEL32.DLL` and `MSVCRT.DLL`, respectively.

One natural solution is to perform instruction-level emulation over the system call implementations obtained from the corresponding libraries. However, this is not practical because system call implementations typically have a large number of instructions and lead to poor scalability.

Thus, we do not emulate the system call code at the instruction-level. Instead, we use code to model the side effects of system calls and execute the models. For example, Figure 10(a) and Figure 10(b) show the stack layout before and after processing `i5` shown in Figure 9. The example models the side effect of `GetSystemDirectoryW`: 1) retrieve the two parameters from the stack; 2)

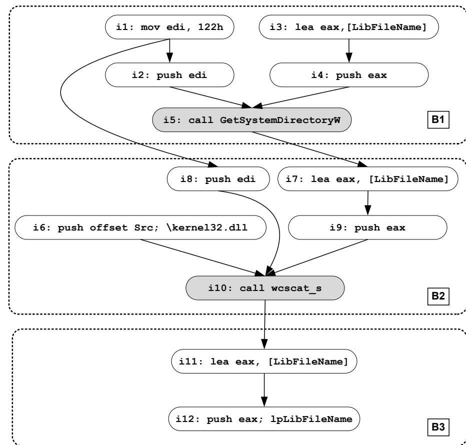


Fig. 9. Backward slice with external library calls

obtain the system directory path by invoking `GetSystemDirectoryW`; 3) write the directory path to the memory chunk pointed to by the first parameter; 4) copy the system call's return value to `eax` register and adjust the `esp` register to clean up the stack frame.

Based on the technique discussed above, we can emulate the context-sensitive slices to compute the possible values of the target component specification. For example, we can compute the value, `"C:\Windows\System32\KERNEL32.DLL"`, of `lpLibFileName` by emulating the backward slice in Figure 9.

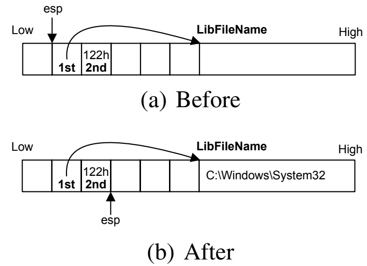


Fig. 10. Side effects of `i5` in Figure 9

Checking Phase. In this phase, we evaluate the safety of the target component specifications obtained from the extraction phase. To this end, for each specification, we check whether or not the safety conditions in Definition 26 are satisfied. In particular, when the fullpath is specified, we check whether or not the specified file exists in the normal file system. For the filename specification, we consider that a specification can lead to unsafe loading if the target component is unknown and the OS cannot resolve it in the directory that is first searched on the normal file system. Note that the names of the known component and the first directory searched by the OS for the resolution are predefined [8,9,21].

As an example of this phase, we check the component loading discussed in the attack scenario in Section 1. When opening the `.asx` file, `Winamp 5.58` tries to load `rapi.dll`. In this case, OS iterates through a list of predefined directories [9] to locate the file named `rapi.dll`. However, no such file is found during the iteration. Thus, this loading is unsafe, because attackers can hijack this loading by placing malicious `rapi.dll` files in the checked directories. In particular, the current working directory, one of the directories, is determined as the same directory as the `.asx` file, leading to the remote code execution attack. Suppose that the file named `rapi.dll` exists in the directory first searched, *i.e.*, the `Winamp` program directory. In this case, this loading is safe, because there is no directory such that attackers can misuse for hijacking.

3 Empirical Evaluation

In this section, we evaluate our static technique in terms of precision, scalability, and code coverage. We show that our technique scales to large real-world applications and is precise. It also has good coverage, substantially better than the existing dynamic approach [21].

3.1 Implementation

We implemented our technique on Windows XP SP3 as a plugin to IDA Pro [15], a state-of-the-art commercial binary disassembler. Our IDA Pro plugin is implemented using IDAPython [16] and three libraries: 1) NetworkX [29] for graph analysis, 2) PyEmu [32] for emulation, and 3) pefile [31] for PE format analysis.

For the precise analysis of binaries, it is important to map between C-like variables and memory regions accessed by instructions. We adapt the concept of an *abstract*

location (*a-loc*) [3], which models a concrete memory address in terms of the base address for a memory region and a relative offset. For example, the a-loc for `&a[4]` is `mem_4` where `mem` is the base address of the array `a` and 4 is the relative offset from the base address. Refer to Balakrishnan and Reps [3] for more details.

Backward slicing in our technique requires function prototypes of system calls. To this end, we analyzed the files in the system directory and collected prototypes for 3,291 system calls.

To emulate the code modeling side effects of system calls, we need to determine what system call is invoked through a given call site instruction. We have extended PyEmu's `set_library_handler` function so that it can register callback functions for external function calls. We implemented the callbacks for 68 system calls used by the extracted slices.

To implement our tool, it is necessary to extract CFGs and call graphs from binaries. We leverage the disassemble result of IDA Pro in our current implementation. It is well-known that indirect jumps can be difficult to resolve for binaries. Although IDA Pro does resolve certain indirect jumps, it may miss control-flow and call dependencies, which is one source of incompleteness in our implementation.

3.2 Evaluation Setup and Results

We aim at detecting unsafe component loadings in applications. Because the detection of unsafe loadings from the system libraries is performed by the operating system, we only resolve the application components in the extraction phase.

The checking phase for a target specification requires the information on the first directory searched by the OS for the resolution and the relevant normal file system state (*cf.*, Definition 26 and Section 2.2). We obtain this information by analyzing the extracted parameters and the applications. For example, suppose that an application p loads an unknown component by invoking `LoadLibrary` with the component's filename. In this case, we can infer the directory where p is installed because Microsoft Windows first checks the directory where p is loaded. Regarding the normal file system state, we installed the applications with the default OS configuration and detected unsafe loadings for each application. In this setting, we assume that 1) the default file system state is normal, and 2) the application itself is benign in that does not cause installed applications to have unintended component loadings.

Detection Results and Scalability. Table 1 shows our analysis results on nine popular Windows applications. We chose these applications as our test subjects because they are important applications in wide-spread use. The results show that our technique can effectively detect, from program binaries, unsafe component loadings potentially loaded at runtime. Note that the results of the extraction phase for `Seamonkey` and `Thunderbird` are identical. This is likely because both applications are part of the Mozilla project and use the same set of program components.

We rely on IDA Pro for disassembling binaries, and Table 1 includes the time that it took IDA Pro to disassemble the nine applications. This time dominates our analysis time as we show later. These are large applications, and also we only need to disassemble the code once for all the subsequent analysis.

Table 1. Analysis of the static detection

	Resolved files			Context-sensitive Emulation					Unsafe loadings / Specifications	
	#	Size (MB)	Disasm. time	Call sites	Slices	Slice inst. (#) mean	Slice inst. (#) max	Failures	Loadtime	Runtime
Acrobat Reader 9.3.2	18	38.2	34m 12s	85	145	5.1	40	34	12 / 109	40 / 111
Firefox 3.0	13	12.5	10m 48s	21	25	2.7	26	3	9 / 77	12 / 22
iTunes 9.0.3	2	25.1	11m 32s	53	128	13.7	187	74	18 / 36	31 / 54
Opera 10.50	3	11.6	12m 46s	28	30	3.0	29	2	8 / 28	11 / 28
Quicktime 7.6.5	17	40.5	9m 15s	70	119	13.5	54	58	19 / 109	19 / 61
Safari 5.31	24	37.5	11m 03s	72	137	5.8	48	33	16 / 158	67 / 104
Seamonkey 2.0.4	15	14.5	20m 44s	34	40	1.7	24	2	9 / 88	20 / 38
Thunderbird 3.0.4	15	15.0	19m 38s	34	40	1.7	24	2	9 / 88	20 / 38
Foxit Reader 3.0	2	10.2	5m 20s	18	18	2.1	13	5	10 / 24	6 / 13

Table 2. Analysis of scalability

Software	Detection time					Relative cost of slice construction							
						# of analyzed functions				# of inst. of analyzed functions			
	Open (s)	Call site (s)	Slicing (s)	Emulation (s)	Total (s)	Demand-driven mean	Demand-driven max	Demand-driven total	Static total	Demand-driven mean	Demand-driven max	Demand-driven total	Static total
Acrobat Reader 9.3.2	95.68	0.03	3.11	6.17	104.93	1.4	3	205	264,551	48.4	220	7,019	9,907,069
Firefox 3.0	41.69	0.03	0.19	0.22	42.13	1.0	1	25	63,550	34.4	158	859	3,071,548
iTunes 9.0.3	15.47	0.03	23.53	16.80	55.83	2.2	5	280	42,689	222.3	7,017	28,460	3,612,724
Opera 10.50	15.35	0.03	0.20	0.57	16.15	1.0	1	30	54,387	28.1	140	843	2,789,126
Quicktime 7.6.5	46.70	0.02	4.65	25.64	77.01	1.9	7	221	63,995	84.4	1,542	10,038	4,885,911
Safari 5.31	48.34	0.02	1.96	3.70	54.02	1.5	7	201	80,899	49.5	500	6,788	5,058,285
Seamonkey 2.0.4	37.51	0.02	0.19	0.52	38.24	1.0	1	40	79,636	30.9	125	1,236	3,840,465
Thunderbird 3.0.4	37.22	0.02	0.22	0.53	37.99	1.0	1	40	78,520	30.9	125	1,236	3,782,799
Foxit Reader 3.0	12.08	0.01	0.17	0.28	12.54	1.2	3	22	56,439	22.8	72	411	2,032,545

According to our analysis of context-sensitive emulation, the number of slices is generally larger than that of the call sites. This indicates that parameters for loading library calls can have multiple values, confirming the need for context-sensitive slices. The average number of instructions for the slices is quite small, which empirically validates our analysis design decisions.

We now discuss the evaluation of our tool's scalability. To this end, we measure its detection time and the efficiency of its backward slicing phase. Table 2 shows the detailed results of detection time and relative cost of slice construction. The results show that our analysis is practical and can analyze all nine large applications within minutes. To further understand its efficiency, we compared cost of our backward slicing with one of standard SDG-based slicing. Although we do expect to explore fewer instructions with a demand-driven approach, we include the comparison in Table 2 to provide concrete, quantitative data. For a standard SDG-based approach, one has to construct the complete SDG before performing slicing. We thus measured how many functions and instructions there are in each application as these numbers indicate the cost of this *a priori* construction (*cf.* the two columns labeled "Static total"). As the table shows, we achieve orders of magnitude reduction in terms of both the number of functions and the number of instructions analyzed.

Comparison with Dynamic Detection. To evaluate our tool's code coverage, we compare unsafe loadings detected by the static and dynamic analyses. In particular, we detected unsafe component loadings with the existing dynamic technique [21] and compared its results with our static detection. To collect the runtime traces, we executed

Table 3. Static detection versus dynamic detection [21]

Software	Component loadings			Unsafe loadings			Static reachability	
	Dynamic	Static	∩	Dynamic	Static	∩	Reachable	Unknown
Acrobat Reader 9.3.2	14	111	11	2	40	1	32	7
Firefox 3.0	16	22	11	6	12	4	1	7
iTunes 9.0.3	5	54	2	3	31	1	29	1
Opera 10.50	20	28	13	9	11	4	7	0
Quicktime 7.6.5	6	61	4	2	19	1	9	9
Safari 5.31	27	104	24	17	67	15	52	0
Seamonkey 2.0.4	24	38	12	9	20	6	0	14
Thunderbird 3.0.4	25	38	11	6	20	5	0	15
Foxit Reader 3.0	6	13	1	0	6	0	6	0

our test subjects one by one with relevant inputs (*e.g.*, PDF files for Acrobat Reader) and collected a single trace per application. Please note that the dynamically detected unsafe loadings are only a subset of all real unsafe loadings.

In this evaluation, we focus on application-level runtime unsafe loadings as loadtime dependent components are loaded by OS-level code. Table 3 shows the detailed results. We see that our static analysis can detect not only most of the dynamically-detected unsafe loadings but also many additional (potential) unsafe loadings, most of which we believe are real and should be fixed. Next we closely examine the results.

Static-only Cases. Our static analysis detects many additional potential unsafe loadings. We carefully studied these additional unsafe loadings manually. In particular, we analyzed whether they are reachable from the entry points of the programs, *i.e.*, whether there exist paths from the entry points to the call sites of the unsafe loadings in the programs' interprocedural CFGss (ICFGs). In this analysis, we consider the main function of an application and the UI callback functions as the entry points of the application's ICFG. Table 3 shows our results on this reachability analysis. Note that those loadings marked as "Unknown" may still be reachable as it is difficult to resolve indirect jumps in binary code, so certain control flow edges may be missing from the ICFGs. All the statically reachable unsafe loadings lead to component-load hijacking if 1) the corresponding call sites are invoked and 2) the target components have not been loaded yet.

Although it is difficult to trigger the detected call sites dynamically (due to the size and complexity of the test subjects), we believe most of the call sites are *dynamically reachable* as dead-code is uncommon in production software. As a concrete example of unsafe loading, Foxit Reader 3.0 has a call site for loading `MAPI32.DLL`, which is invoked when the current PDF file is attached to an email message. This loading can be hijacked by placing a file with the same name `MAPI32.DLL` into the directory where Foxit Reader 3.0 is installed.

Dynamic-only Cases. According to Table 3, our technique misses a few of the dynamically detected unsafe loadings. We manually examined all these cases, and there are two reasons for this: *system hook dependency* and *failed emulation*, which we elaborate next.

First, Microsoft Windows provides a mechanism to hook particular events (*e.g.*, mouse events). If hooking is used, a component can be loaded into the process to handle the hooked event. This component injection introduces a system hook dependency [41]. Such a loading may be unsafe, but since it is performed by the OS at runtime and is not an application error, we do not detect it.

Second, our extraction phase may miss some target component specifications due to failed emulations. If this happens, we may miss some unsafe loadings even if their corresponding call sites are found. Emulation failures can be caused by the following reasons.

External Parameters. A target specification may be defined by a parameter of an exported function that is not invoked. For example, suppose that a function `foo` exported by a component `A` loads a DLL specified by `foo`'s parameter. If `foo` is not invoked by `A`, the parameter's concrete value will be unknown. One may mitigate this issue by analyzing the data flow dependencies among the dependent components. However, such an analysis does not guarantee to obtain all the target specifications, because the exported functions are often not invoked by the dependent components.

Uninitialized Memory Variables. The slices may have instructions referencing memory variables initialized at runtime. In this case, our slice emulation may be imprecise or fail. To address this problem, it is necessary to extract the sequence of instructions from the dependent components that initialize these memory variables and emulate the instructions before slice emulation. Although it is possible to analyze memory values, such as the Value Set Analysis (VSA) [34], it is difficult to scale such analysis to large applications.

Imprecise Inferred Function Prototypes. Our technique infers function prototypes by analyzing parameters passed via the stack. However, function parameters may be passed via other means such as registers. For example, the `__fastcall` convention uses `ECX` and `EDX` to pass the first two parameters. Therefore, when function parameters are passed through unsupported calling conventions, the inferred function prototypes may omit parameters that determine the new slicing criteria. For example, suppose that we extract a context-sensitive sub-slice s from a function `foo`, and `ECX` is used as a parameter variable of s. In this case, we do not continue the backward slicing phase, because the inferred prototype does not contain `ECX`. Although imprecisely inferred function prototypes may lead to emulation failure, our results show that this rarely happens in practice—we observed only 14 cases out of a total of 213.

Unknown Semantics of System Calls. Detailed semantics of system calls is often undocumented, and sometimes even their names are not revealed. When we encounter such system calls, we cannot analyze nor emulate them. When information of such system calls becomes available, we can easily add analysis support for them.

Disassemble Errors. Our implementation relies on IDA Pro to disassemble binaries, and sometimes the disassemble results are incorrect. For example, IDA Pro sometimes is not able to disassemble instructions passing parameters to call sites for delayed loading. Such errors can lead to imprecise slices and emulation failures.

4 Related Work

We survey additional related work besides the one on dynamic detection of unsafe loadings [21], which we have already discussed.

Our technique performs static analysis of binaries. Compared to the analysis of source code, much less work exists [1, 3, 4, 6, 7, 19, 20, 34, 39]. In this setting, Value Set Analysis (VSA) [3, 34] is perhaps the most closely related to ours. It combines numeric and pointer analyses to compute an over-approximation of numerical values of program variables. Compared to VSA, our technique focuses on the computation of

string variables. It is also demand-driven and uses context-sensitive emulation to scale to real-world large applications.

Starting with Weiser's seminal work [43], program slicing has been extensively studied [40, 46]. Our work is related to the large body of work on static slicing, in particular the SDG-based interprocedural techniques. Standard SDG-based static slicing techniques [1, 5, 13, 30, 35, 38] build the complete SDGs beforehand. In contrast, we build control- and data-flow dependence information in a demand-driven manner, starting from the given slicing criteria. Our slicing technique is also modular because we model each call site using its callee's inferred summary that abstracts away the internal dependencies of the callee. In particular, we treat a call as a non-branching instruction and approximate its dependencies with the callee's summary information. This optimization allows us to abstract away detailed data flow dependencies of a function using its corresponding call instruction. We make an effective trade-off between precision and scalability. As shown by our evaluation results, function prototype information can be efficiently computed and yield precise results for our setting.

Our slicing algorithm is demand-driven, and is thus also related to demand-driven dataflow analyses [14, 33], which have been proposed to improve analysis performance when complete dataflow facts are not needed. These approaches are similar to ours in that they also leverage caller-callee relationship to rule out infeasible dataflow paths. The main difference is that we use a simple prototype analysis to construct concise function summaries instead of directly traversing the functions' intraprocedural dependence graphs, *i.e.*, their PDGs. Another difference is that we generate context-sensitive executable slices for emulation to avoid the difficulties in reasoning about system calls.

As we discussed earlier, instead of emulation, symbolic analysis [18, 37] could be used to compute concrete values of the program variables. However, symbolic techniques generally suffer from poor scalability, and more importantly, it is not practical to symbolically reason about system calls, which are often very complex. The missing implementation for undocumented system calls is the challenge for emulation, while for symbolic analysis, complex system call implementation is an additional challenge. We introduce the combination of slicing and emulation to address this additional challenge. Our novel use of context-sensitive emulation provides a practical solution for computing the values of program variables.

5 Conclusion and Future Work

We have presented a practical static binary analysis to detect unsafe loadings. The core of our analysis is a technique to precisely and scalably extract which components are loaded at a particular loading call site. We have introduced context-sensitive emulation, which combines incremental and modular slice construction with the emulation of context-sensitive slices. Our evaluation on nine popular Windows application demonstrates the effectiveness of our technique. Because of its good scalability, precision, and coverage, our technique serves as an effective complement to dynamic detection [21]. For future work, we would like to consider two interesting directions. First, because unsafe loading is a general concern and also relevant for other operating systems, we plan to extend our technique and analyze unsafe component loadings on Unix-like systems. Second, we plan to investigate how our technique can be improved to reduce emulation failures.

Acknowledgments. We thank the anonymous reviewers for their feedback on an earlier version of this paper. This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, NSF TC Grant No. 0917392, and the US Air Force under grant FA9550-07-1-0532. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

1. Kiss, Á., Jász, J., Lehotai, G., Gyimóthy, T.: Interprocedural static slicing of binary executables. In: Proc. SCAM Workshop (2003)
2. An update on the DLL-preloading remote attack vector, http://blogs.technet.com/b/srd/archive/2010/08/31/an-update-on-the-dll-preloading-remote-attack-vector.aspx
3. Balakrishnan, G., Reps, T.: Analyzing Memory Accesses in x86 Executables. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, pp. 5–23. Springer, Heidelberg (2004)
4. Balakrishnan, G., Reps, T.: Analyzing Stripped Device-Driver Executables. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 124–140. Springer, Heidelberg (2008)
5. Binkley, D.: Precise executable interprocedural slices. ACM Lett. Program. Lang. Syst. 2(1-4), 31–45 (1993)
6. Cifuentes, C., Fraboulet, A.: Intraprocedural static slicing of binary executables. In: Proc. ICSM (1997)
7. Comparetti, P.M., Salvaneschi, G., Kirda, E., Kolbitsch, C., Kruegel, C., Zanero, S.: Identifying dormant functionality in malware programs. In: Proc. SSP (2010)
8. dlopen man page, http://linux.die.net/man/3/dlopen
9. Dynamic-Link Library Search Order, http://msdn.microsoft.com/en-us/library/ms682586(VS.85.)aspx
10. Dynamic-Link Library Security, http://msdn.microsoft.com/en-us/library/ff919712(VS.85.)aspx
11. Exploiting DLL Hijacking Flaws, http://blog.metasploit.com/2010/08/exploiting-dll-hijacking-flaws.html.
12. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Prog. Lang. Syst. 9(3), 319–349 (1987)
13. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. Prog. Lang. Syst. 12(1), 26–60 (1990)
14. Horwitz, S., Reps, T., Sagiv, M.: Demand interprocedural dataflow analysis. In: Proc. FSE (1995)
15. IDA Pro Disassmelber, http://www.hex-rays.com/idapro/
16. IDAPython, http://code.google.com/p/idapython/
17. Insecure Library Loading Could Allow Remote Code Execution, http://www.microsoft.com/technet/security/advisory/2269637.mspx
18. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
19. Kruegel, C., Robertson, W., Valeur, F., Vigna, G.: Static disassembly of obfuscated binaries. In: Proc. USENIX Security (2004)
20. Kruegel, C., Robertson, W., Vigna, G.: Detecting Kernel-Level Rootkits Through Binary Analysis. In: Proc. ACSAC (2004)
21. Kwon, T., Su, Z.: Automatic detection of unsafe component loadings. In: Proc. ISSTA (2010)
22. Kwon, T., Su, Z.: Static detection of unsafe component loadings. UC Davis techical report CSE-2010-17 (2010)

23. Microsoft Cooking Up Baker's Dozen of Fixes for Patch Tuesday, http://www.esecurityplanet.com/patches/article.php/3902856/Microsoft-Cooking-Up-Bakers-Dozen-of-Fixes-for-Patch-Tuesday.htm
24. Microsoft Portable Executable and Common Object File Format Specification, http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx
25. Microsoft releases tool to block DLL load hijacking attacks, http://www.computerworld.com/s/article/print/9181518/Microsoft_releases_tool_to_block_DLL_load_hijacking_attacks
26. Microsoft releases tool to block DLL load hijacking attacks, http://www.computerworld.com/s/article/9181518/Microsoft_releases_tool_to_block_DLL_load_hijacking_attacks
27. Microsoft Was Warned of DLL Vulnerability a Year Ago, http://www.esecurityplanet.com/features/article.php/3900186/Microsoft-Was-Warned-of-DLL-Vulnerability-a-Year-Ago.htm
28. MS09-014: Addressing the Safari Carpet Bomb vulnerability, http://blogs.technet.com/srd/archive/2009/04/14/ms09-014-addressing-the-safari-carpet-bomb-vulnerability.aspx
29. NetworkX, http://networkx.lanl.gov/
30. Orso, A., Sinha, S., Harrold, M.J.: Incremental slicing based on data-dependence types. In: Proc. ICSM (2001)
31. pefile, http://code.google.com/p/pefile/
32. PyEmu, http://code.google.com/p/pyemu/
33. Reps, T.: Solving Demand Versions of Interprocedural Analysis Problems. In: Adsul, B. (ed.) CC 1994. LNCS, vol. 786, pp. 389–403. Springer, Heidelberg (1994)
34. Reps, T., Balakrishnan, G.: Improved Memory-Access Analysis for x86 Executables. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 16–35. Springer, Heidelberg (2008)
35. Reps, T., Horwitz, S., Sagiv, M., Rosay, G.: Speeding up slicing. In: Proc. FSE (1994)
36. Researcher told Microsoft of Windows apps zero-day bugs 6 months ago, http://www.computerworld.com/s/article/print/9181358/Researcher_told_Microsoft_of_Windows_apps_zero_day_bugs_6_months_ago
37. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Proc. SSP (2010)
38. Sinha, S., Harrold, M.J., Rothermel, G.: System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In: Proc. ICSE (1999)
39. 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)
40. Tip, F.: A survey of program slicing techniques. Technical report, CWI, Amsterdam, The Netherlands (1994)
41. Types of Dependencies, http://dependencywalker.com/help/html/dependency_types.htm
42. Vulnerabilities in Microsoft Office Could Allow Remote Code Execution, http://www.microsoft.com/technet/security/bulletin/ms10-087.mspx
43. Weiser, M.: Program slicing. In: Proc. ICSE (1981)
44. Windows DLL Exploits Boom; Hackers Post Attacks for 40-plus Apps, http://www.computerworld.com/s/article/9181918/Windows_DLL_exploits_boom_hackers_post_attacks_for_40_plus_apps
45. X86 Calling Conventions, http://en.wikipedia.org/wiki/X86_calling_conventions
46. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. SIGSOFT Softw. Eng. Notes 30(2), 1–36 (2005)

Object Model Construction for Inheritance in C++ and Its Applications to Program Analysis

Jing Yang[1,2], Gogul Balakrishnan[1], Naoto Maeda[3], Franjo Ivančić[1], Aarti Gupta[1], Nishant Sinha[4,⋆], Sriram Sankaranarayanan[5], and Naveen Sharma[6]

[1] NEC Labs America
[2] University of Virginia
[3] NEC Corporation, Japan
[4] IBM Research, India
[5] University of Colorado, Boulder
[6] NEC-HCL Tech., India

Abstract. Modern object-oriented programming languages such as C++ provide convenient abstractions and data encapsulation mechanisms for software developers. However, these features also complicate testing and static analysis of programs that utilize object-oriented programming concepts. In particular, the C++ language exhibits features such as multiple inheritance, static and dynamic typecasting that make static analyzers for C++ quite hard to implement. In this paper, we present an approach where static analysis is performed by lowering the original C++ program into a semantically equivalent C program. However, unlike existing translation mechanisms that utilize complex pointer arithmetic operations, virtual-base offsets, virtual-function pointer tables, and calls to run-time libraries to model C++ features, our translation is targeted towards making static program analyzers for C++ easier to write and provide more precise results. We have implemented our ideas in a framework for C++ called CILpp that is analogous to the popular C Intermediate Language (CIL) framework. We evaluate the effectiveness of our translation in a bug finding tool that uses abstract interpretation and model checking. The bug finding tool uncovered several previously unknown bugs in C++ open source projects.

1 Introduction

Modern object-oriented programming languages provide convenient abstraction and data encapsulation mechanisms for software developers. Such mechanisms include function and operator overloading, constructors and destructors, multiple class inheritance, dynamic virtual-function dispatch, templates, exceptions, functors, standard libraries such as STL and BOOST. However, on the flip side, these features complicate the static analysis of programs that use such features. In the past decade, there have been numerous approaches for static program analysis techniques based on source code. These tools rely on abstract interpretation [12] or software model checking [9], such as ASTREÉ [13], Saturn [40], SLAM [3], CBMC [8], Java PathFinder [16], and FindBugs [17]. However, in practice, these tools have largely been developed and optimized

⋆ Work done while at NEC Labs America.

M. O'Boyle (Ed.): CC 2012, LNCS 7210, pp. 144–164, 2012.
© Springer-Verlag Berlin Heidelberg 2012

to either address C or Java. With respect to static program analysis techniques for object-oriented source code, most work has addressed Java partly because of its less intricate class hierarchy concept. However, in industry, C++ is one of the predominant development languages. Due to its intrinsic complexity mixing object-oriented programming on top of full-fledged C code, there is a stark need for static program analysis for C++ supporting all of its many features.

There are several popular front-ends, such as `Comeau C/C++` [10], `EDG` [15], LLVM [19], and ROSE [27], that support all the complex features of C++. In spite of the availability of C++ front-ends, it is still hard to perform static analysis of C++ programs as observed by the developers of Clang [6,7]:

> *" ... Support in the frontend for C++ language features, however, does not automatically translate into support for those features in the static analyzer. Language features need to be specifically modeled in the static analyzer so their semantics can be properly analyzed. Support for analyzing C++ and Objective-C++ files is currently extremely limited, ... "*

Modeling the C++ features directly in static analysis is a non-trivial task because the C++ semantics is quite complicated [32]. Moreover, every static analyzer may need to encode the semantics differently depending upon the requirements of the analysis. For complex analysis, this process can easily get out of hand. One particular issue in handling of C++ programs compared to other object-oriented programming languages such as Java is the complexity of allowed class hierarchies. C++ allows multiple inheritance, where a class may inherit from more than one class. Presence of multiple inheritance gives rise to complex class hierarchies which must be handled directly (and precisely) by any static analysis. Furthermore, multiple inheritance complicates the semantics of otherwise simple operations such as casts and field accesses [33]. Therefore, techniques developed for Java are not readily applicable to C++ programs. It is important to emphasize that multiple inheritance is used quite frequently by developers even in large C++ projects. Nokia's cross-platform UI framework Qt [25], Apache's Xerces project [1], and `g++` standard IO stream library are good examples.

An alternative approach to analyzing C++ programs with complex hierarchies is to utilize compiler front-ends that compile C++ programs into equivalent C programs (often referred to as "lowering of C++ programs"). A number of such approaches exist, starting from the earliest C++ compilers, such as `Cfront` [34]. Today, there are commercial C++ front-ends available that provide similar features, for example `Comeau C/C++` [10] or `EDG` [15]. Such translations are generally geared towards runtime performance and small memory footprint. Therefore, these front-ends heavily utilize pointer arithmetic operations, virtual-base offsets, virtual-function pointer tables, and rely on runtime libraries, to achieve those goals. Such translations are hardly amenable to precise program analysis.

Consider, for example, the translation using one such generic lowering mechanism shown in the middle column of the table in Fig. 1. Note how a `static_cast` operation in row (a) of Fig. 1 is translated into an adjustment of the source pointer `pr` by 8 bytes in the lowered C program. Similarly, the `dynamic_cast` operation is translated into a call to an opaque runtime function `_dynamic_cast`, which may change the source pointer as in the case of `static_cast`. Such pointer adjustment operations are crucial

	Standard C++-to-C lowering	CHROME-based lowering

(a)	`pb =`
	`((struct B *) ((pr != ((struct R *) 0))`
	`? (((char *)pr) - 8UL) : ((char *) 0)));`
		`assert(pr->soid == B_R);`
		`pb = pr ? pr->dervB : 0;`

(b)	`pl = ((pr != ((struct R *)0))`
	` ? ((struct L *)(_dynamic_cast(...)))`
	` : ((struct L *)0));`
		`assert(pr->soid == B_R);`
		`pl = pr? pr->dervB->baseL : 0;`

(c)	`// Access virtual function table`
	`_T31 = ((((pl->_b_R)._vptr)) + 1);`
	
	`// Call virtual function`
	`(((void (*)(struct R * const))`
	`((_T31->f))) (((struct R *)`
	`(((char *) (&pl->_b_R)) +`
	`((_T31->d))))));`
		`switch(pl->soid) {`
		`case B_L:`
		`case L:`
		`{ T* prt = pl->baseT;`
		`prt->T::vfunc(prt);`
		`break;`
		`}`
		`default: assert(false);`
		`}`

(d)	`((*((struct T *) (((char *) pb) +`
	`((((pb->_b_1L)._vptr))[(-3)]))))).tp)`
	`= ((int *) 0);`
		`pb->baseL->baseT->tp`
		`= (int *)0;`

Fig. 1. Comparison of the standard C++-to-C lowering and CHROME-based lowering mechanisms: (a) `pb = static_cast<B*>(pr)`, (b) `pl = dynamic_cast<L*>(pr)`, (c) `pl->vfunc()`, `vfunc` is a virtual function), and (d) `pb->L::tp = 0`, `L::tp` is a field of a shared base class)

for preserving the semantics of class member accesses in the lowered C program. However, static analysis algorithms typically assume that such pointer adjustments (which are akin to pointer arithmetic operations) do not change the behavior of the program, and therefore, may ignore them completely. Consequently, the analysis of the lowered C program is unsound. Alternatively, a conservative treatment of pointer arithmetic operations results in a large number of false positives, thereby reducing the usefulness of the analysis.

Further, note how a virtual-function call in Fig. 1(c) is translated into a complex combination of virtual-function pointer table lookups (via *vptr* variable) and unintelligible field accesses. Furthermore, in row (d) of Fig. 1, even a simple access to a field of a shared base class (see Sect. 2) is translated into an access through the virtual-function table [33]. Most conventional static analysis are not well equipped to precisely reason about such code. In particular, they are generally imprecise in the presence of arrays of function pointers or pointer offsets (such as the virtual-function pointer table). At a virtual-function call, a naive analysis may enumerate all potential callees, thus causing blowup in the computed call graph due to redundant function invocations. Alternatively, many techniques safely approximate arrays using summary variables, which leads to a severe imprecision in the resolution of field accesses and virtual-function calls.

Our Approach. In this paper, instead of encoding complex C++ semantics in static analyzers or simulating the behavior of the compiler, we adopt a more pragmatic approach to analyzing C++ programs. We believe that in order to allow for a scalable, yet precise, static analysis of C++, such techniques need not have to reason about the low-level physical memory layout of objects. Rather, we require a higher level of abstraction of the memory that is closer to the developer's understanding of the program.

Towards this goal, we propose a representation for modeling C++ objects that is different from the object layout representations used by a compiler. The alternative representation is referred to as CHROME (Class Hierarchy Representation Object Model Extension) and uses the algebraic theory of sub-objects proposed by Rossie and Friedman [30]. Using the CHROME object model, we employ a sequence of source-to-source transformations that translate the given C++ program *with* inheritance into a semantically equivalent C++ program *without* inheritance. Our source-to-source transformations comprises two main steps. The first step employs a *clarifier* module which makes implicit C++ features explicit. An example of such an implicit feature is the invocation of constructors, destructors, and overloaded operators. The second step involves the elimination of inheritance-related features using the proposed CHROME model. The translations are semantics-preserving and *static program analysis friendly*. Our aim is to allow state-of-the-art static program analyzers, that are currently oblivious to inheritance and multiple inheritance in programs, to naturally handle such transformed programs while maintaining their efficiency and precision.

The last column in Fig. 1 shows the CHROME-based transformations. The key idea underlying the CHROME model is to treat sub-objects due to inheritance as separate memory regions that are linked to each other via additional base class and derived class pointer fields. Instead of utilizing virtual-function pointer table lookups and address offset computations to resolve issues related to dynamic dispatch and casts, we instead follow a path in a sub-object graph utilizing these additional pointers to find the required sub-object of interest. Note that this sub-object graph walk may require multiple pointer indirections and would thus be inefficient for runtime performance as well as less memory efficient. However, static program analyzers routinely reason about heaps and pointer indirections. Hence, we build upon the strengths of these tools to allow for a efficient and precise analysis of complex C++ programs. The analysis of the resulting C++ program is also simplified because program analysis tools can treat casts and accesses to fields of inherited classes and virtual-function calls in the same way as regulars casts and accesses to fields of regular classes and normal function calls, respectively.

Our approach has multiple advantages. Various analyses can now focus on the reduced subset of C++ in a uniform manner without being burdened with having to deal with inheritance. Further, we may adopt simpler object-oriented analyses, e.g., those developed for Java programs, to analyze C++ programs. Finally, because the reduced subset is quite close to C, the given C++ program can also be lowered to C. This enables reuse of standard static analyses for C programs without necessitating the loss of high-level data representation.

The transformed source code can be potentially used not only for static analysis but also for dynamic program analysis, code understanding, re-engineering, runtime monitoring, and so on. We believe that our approach provides a uniform way to resolve the principal barrier to analyzing C++ programs, i.e., handling of inheritance precisely in a scalable fashion.

To illustrate the practical utility of our approach, we experiment with an in-house bug-finding tool called F-SOFT [18] that uses abstract interpretation [12] and model checking [9]. Our experiments show that our method provides significant benefits as compared to a straightforward C++-to-C lowering-based approach. We found a total of

ten previously unknown bugs on some C++ open source projects, of which *five* were only found due to the precise modeling of objects using the CHROME object model. We also experimented with a publicly available bug finding tool called CBMC [8], which showed similar improvements when using the CHROME object model.

Contributions. The key contributions of our paper are as follows:

- The CHROME (Class Hierarchy Representation Object Model Extension) object model representation for modeling objects of derived classes.
- An algorithm for translating a C++ program *with* inheritance into a semantically equivalent C++ program *without* inheritance using the CHROME object model.
- Performing the translation without the use of pointer arithmetic operations, virtual-base offsets, virtual-function tables, and runtime functions. Our source-to-source translation is designed with program analysis in mind and not for runtime performance or a small memory footprint.
- Illustrating the practicality of our approach by evaluating the effectiveness of the lowered C++ programs for abstract interpretation and model checking.
- A framework for C++ called `CILpp` that can be used to build analysis and verification tools for C++ programs. `CILpp` is analogous to the popular C Intermediate Language (CIL) framework for C programs [24].

Our implementation is based on the `EDG` frontend [15]. This allows us to focus on the analysis while being able to handle arbitrary C/C++ dialects. Our current lowering mechanism using the CHROME object model relies on the assumption that the tool chain is aware of the complete class hierarchy. This restriction simplifies the object model generation substantially since our generated object models do not have to address dynamic class loading, for example.

The rest of the paper is organized as follows. Sect. 2 describes the algebraic theory of Rossie-Friedman sub-objects. Sect. 3 presents the clarifier module that makes implicit C++ features explicit. Sect. 4 presents the CHROME object model and the source-to-source transformations that eliminate inheritance. Sect. 5 describes the results of our experiments. Sect. 6 discusses the related work. Sect. 7 concludes the paper.

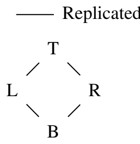
2 Rossie-Friedman Sub-objects

Informally, the Rossie-Friedman sub-object model [30] is an abstract representation of the object layout. When a class inherits from another class, conceptually the base class is embedded into the derived class. Therefore, an object of a derived class consists of different (possibly overlapping) components that correspond to the direct and transitive base classes of the derived class. Intuitively, a *sub-object* refers to a component of a direct or transitive base class that is embedded in a derived class object. The complete derived class object is also considered to be a sub-object.

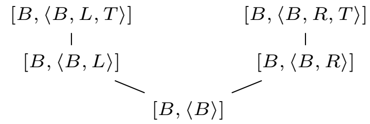
Example 1. Consider a class `L` that inherits from another class `T`. An object of type `L` consists of two sub-objects: (1) a sub-object of type `T` corresponding to the base class `T`, and (2) a sub-object of type `L` that corresponds to the complete object itself. Fig. 4(I)(a) shows the sub-objects of `L`. ■

```
class T  {int tf;}
class L: public T
         {int lf;}
class R: public T
         {int rf;}
class B: public L,
public R {int bf;}
```

(a)



(b)



(c)

Fig. 2. Replicated multiple inheritance: (a) C++ program, (b) class-hierarchy graph, and (c) sub-object poset for class B

C++ supports multiple inheritance through which a class inherits from more than one base class. In case of single inheritance, there is only one copy of every (direct or transitive) base class in a derived class object. However, with multiple inheritance, the number of sub-objects corresponding to a direct or transitive base class depends upon the number of paths between the base class and the derived class in the class hierarchy.

Example 2. Consider the program shown in Fig. 2(a). Fig. 4(I)(b) shows the object layout for `B`. Because `B` inherits from `L` and `R`, a `B`-object contains sub-objects of type `L` and `R`. Further, the sub-objects of type `L` and `R` each contain a distinct sub-object of type `T`. Therefore, a `B`-object has two distinct sub-objects of type `T`: one inherited from class `L` and the other inherited from class `R`. ■

Virtual base classes. It is often not desirable to have multiple sub-objects of a base class in a derived class. Therefore, to prevent replication of base class sub-objects in a derived class, C++ provides *virtual base classes*. Unlike a non-virtual base class, a sub-object of a virtual base class type is shared among the sub-objects of all its direct and transitive derived classes.

Example 3. Consider the class hierarchy in Fig. 3(a). The keyword *virtual* indicates that class `T` is a virtual base class. Fig. 4(I)(c) shows the object layout for `B`. A `B`-object contains sub-objects of type `L` and `R` as usual. However, because `T` is a virtual base class, it is shared among the direct and transitive derived classes `L`, `R`, and `B`. Therefore, a `B`-object contains only one sub-object of type `T`.

When a class inherits from a non-virtual base class, it is referred to as *replicated* inheritance. When a class inherits from a virtual base class, it is referred to as *shared* inheritance. The class hierarchy graph captures the shared and replicated inheritance relationships among the classes.

Definition 1 (Class Hierarchy Graph (CHG)**).** *A class hierarchy graph $\mathcal{G}$ is a tuple $\langle \mathcal{C}, \prec_s, \prec_r \rangle$, where $\mathcal{C}$ is the set of class names, $\prec_s \subseteq \mathcal{C} \times \mathcal{C}$ are shared inheritance edges, and $\prec_r \subseteq \mathcal{C} \times \mathcal{C}$ are replicated inheritance edges. Let $\prec_{sr} = (\prec_s \cup \prec_r)$.*

The formal description relies on the following operators to model transitive applications of $\prec_s$ and $\prec_r$. $<_s = (\prec_s)^+$, $\leq_s = (\prec_s)^*$, $<_r = (\prec_r)^+$, and $\leq_r = (\prec_r)^*$. Similarly, $<_{sr} = (\prec_{sr})^+$, and $\leq_{sr} = (\prec_{sr})^*$.

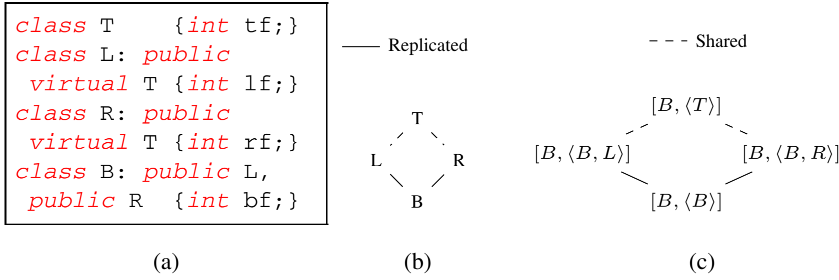


Fig. 3. Shared multiple inheritance: (a) C++ program, (b) class hierarchy graph, and (c) sub-object poset for class B

We require that the reflexive and transitive closure $\leq_{sr}$ of $\prec_{sr}$ is antisymmetric. This ensures that a CHG *is acyclic. Note that $(\mathcal{C}, \leq_{sr})$ is a poset.*

The Rossie-Friedman sub-object model formalizes the notion that, given a derived class `D`, a sub-object of a `D`-object is either the complete `D`-object or a component of a base class type `B` that is embedded in the `D`-object. Because C++ allows multiple inheritance, a `D`-object may have multiple sub-objects of a base class type `B`. Therefore, it is not sufficient to represent a sub-object of type `B` in a `D`-object simply as a pair $\langle \mathtt{D}, \mathtt{B} \rangle$. Rossie and Friedman distinguish different sub-objects of the same base class in a derived class using the path from the base class to the derived class in the CHG.

Definition 2 (Sub-object). *Given a* CHG $\langle \mathcal{C}, \prec_s, \prec_r \rangle$*, a sub-object σ is a pair $[C, \langle X, Y_1, Y_2, \ldots, Y_n \rangle]$, where*

1. $C, X, Y_1, Y_2, \ldots, Y_n \in \mathcal{C}$
2. $X \prec_r Y_1 \prec_r \ldots \prec_r Y_n$
3. $(C = X) \vee \exists (Z \in \mathcal{C})[C <_{sr} Z <_s X]$

C is the derived class to which σ belongs, and Y_n is the type of the sub-object. The path $X, Y_1, Y_2, \ldots, Y_n$ represents the path in the CHG through which class C inherits Y_n. For a repeated sub-object, $X = C$. For a shared sub-object, X is the least derived virtual base class that contains Y_n.

Example 4. Fig. 2(c) show the sub-objects of class `B` with replicated multiple inheritance. The sub-object $[B, \langle B, L, T \rangle]$ in class `B` corresponds to class `T` that is inherited transitively by class `B` through class `L`. The sub-object path $\langle B, L, T \rangle$ represents the corresponding path in the CHG. An instance of class `B` has two copies of `T`, one inherited from `L` and the other inherited from `R`. Therefore, there are two sub-objects $[B, \langle B, L, T \rangle]$ and $[B, \langle B, R, T \rangle]$ that correspond to class `T`. The sub-object path of `T` determines whether it is inherited from `L` or `R`.

Similarly, Fig. 3(c) shows the sub-objects of class `B` with shared inheritance. Note that an instance of class `B` has only one copy of class `T`. Therefore, there is only one sub-object that has an effective class type `T`, namely $[B, \langle T \rangle]$. The sub-object path $\langle T \rangle$ represents the fact that class `B` shared inherits `T` because the first class in the sub-object path is not B. ■

3 Clarifier

C++ provides convenient abstractions, such as constructors and destructors, that simplify the life of a software developer. However, such abstractions also introduce additional operations that are implicit in the control flow of the program. For example, the destructors of the objects allocated on the stack are implicitly invoked whenever the objects go out of scope. Examples of other such implicit operations are calls to constructors and overloaded operators, the `this` parameter in member functions, and implicit casts. The clarifier module exposes such implicit operations in the C++ program.

Example 5. Consider the following C++ program:

```
int cutLen(const string &s,size_t i,size_t n){
  const char *str = s.substr(i,n).c_str();
  return strlen(str);
}
```

The output of the clarifier is shown below:

```
int cutLen(const string &s,size_t i,size_t n){
  const string tmp; // Only declaration, no call to constructor.
  // Copy constructor call for 'tmp'.
  tmp.string(s.substr(i,n));
  const char *str = tmp.c_str();
  // Destruction of temporary 'tmp'
  tmp.~string();
  // Use of invalid pointer 'str'
  return strlen(str);
}
```

In the output, the copy constructor for the temporary object that gets created at the call to `s.substr(...)` is made explicit. Similarly, the destructor for the temporary object is invoked when the temporary goes out of scope, which happens immediately after the initialization of `str`. Method call `tmp.c_str()` returns a pointer to an internal buffer, which is deallocated when the `tmp` object is destroyed. Therefore, `strlen` uses a deallocated string `str`, which can cause a segmentation fault.[1] The clarifier maintains the correct C++ semantics by preserving the order in which the constructors and destructors are invoked, and the order in which the initializations are performed.

The output of the clarifier is an intermediate representation called `CILpp` that is largely inspired by the CIL front-end [24] with relevant extensions for C++ specific features. The `CILpp` representation consists of a mixture of C and C++ constructs. Specifically, the inheritance-related constructs are still present. The inheritance-related features are eliminated by performing source-to-source transformations using the CHROME model as described next.

[1] This example is modeled on some bugs that our framework discovered in the *gold* project, which provides a faster linker as part of the GNU binutils package [2].

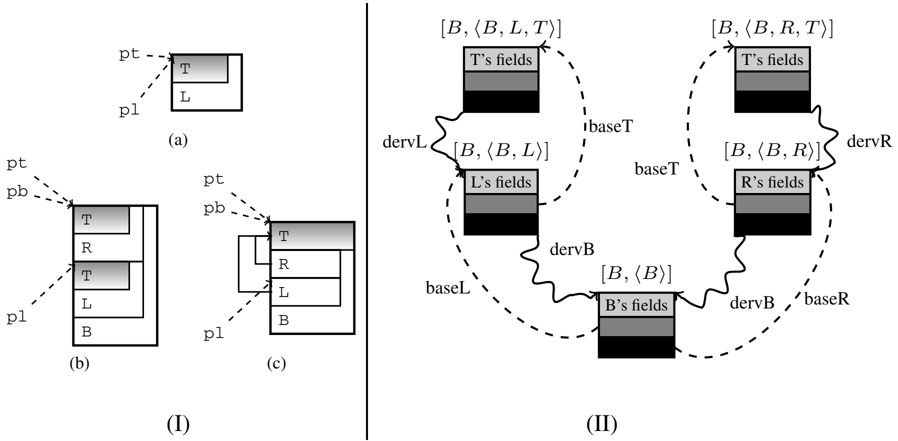


Fig. 4. (I) Object layout used by a standard C++-to-C lowering algorithm: (a) class `L` in Figs. 2 and 3, (b) class `B` with replicated inheritance in Fig. 2, and (c) class `B` with shared inheritance in Fig. 3. (II) CHROME object model for the class `B` in Fig. 2. (`pt`, `pl`, and `pb` are pointers.)

4 CHROME Model

Standard C++-to-C lowering algorithms use a *physical sub-object model* for representing objects of derived classes, which is similar to how compilers layout objects at runtime. In the physical sub-object model, the base classes are embedded into the derived class objects. Fig. 4(I) shows the physical sub-object model for some classes that use replicated and shared multiple inheritance.

There are several problems with using the physical sub-object model for analysis. Because sub-objects are embedded inside the derived classes, a cast between a base and derived class pointer has to be modeled as an offset adjustment to the source pointer. For example, for the objects in Fig. 4(I), the cast statement “`pl = (L*) pb`”, where `pb` is a pointer of type `B*`, requires moving pointer `pb` to the start of the corresponding sub-object of type `L` in the `B`-object as follows: “`pl = ((char*)pb) + 8;`”.

In certain cases, the required offsets cannot be determined statically. For example, consider a cast from a pointer `pl` of type `L*` to a pointer of type `T*`. If `pl` points to a sub-object of type `L` as in Fig. 4(I)(a), no adjustment is necessary. On the other hand, if `pl` points to a sub-object of type `L` in an object of type `B` as in Fig. 4(I)(c), pointer `pl` has to be adjusted by 8 bytes. For such cases, the offsets are stored in the virtual-function pointer table and are consulted at runtime. Consequently, the code generated by a standard lowering algorithm includes a lookup of a virtual-function pointer table even for simple casts. As mentioned in Sect. 1, static analysis algorithms are not precise in the presence of such low-level pointer offset adjustments and arrays of pointers.

To avoid such problems, we propose the CHROME object model for representing the objects of a derived class. In the CHROME object model, an object is viewed as a collection of its sub-objects and no assumptions are made about the layout of the fields in each class. Whenever an object is created, the sub-objects that belong to the class are created independently, and are linked to each other via *additional* pointer fields.

Example 6. Consider the replicated inheritance hierarchy in Fig. 2. The CHROME object model representation for a B-object is shown in Fig. 4(II). The different sub-objects of the class are constructed separately and are connected to each other through additional pointer fields. For example, the `dervL` and `baseT` pointers connect the sub-objects $[B, \langle B, L\rangle]$ and $[B, \langle B, L, T\rangle]$. ■

These auxiliary object hierarchy edges are utilized by CHROME to walk the object when arbitration of inheritance related features is needed. As an example, consider casts, where, instead of computing pointer offset adjustments, we follow the additional pointers in the representation of the B-object. Next, we present the source-to-source transformations for various C++ constructs through examples.

Example 7 (Class Declarations). To facilitate the construction of the CHROME object model, we add the following fields to every class C in the CHG: (1) a `soid` field that is used to identify the sub-object that C represents, (2) a base pointer field and a derived pointer for every immediate base and derived class of C, respectively. For the program in Fig. 2, the classes are modified as follows (replicated multiple inheritance):

```
class T {int soid; L* dervL; R* dervR; int tf;}
class L {int soid; T* baseT; B* dervB; int lf;}
class R {int soid; T* baseT; B* dervB; int rf;}
class B {int soid; L* baseL; R* baseR; int bf;}
```

For the program in Fig. 3, classes L, and R are modified as shown above, and classes T and B are modified as follows (shared multiple inheritance):

```
class T {int soid; L* dervL; R* dervR; B* dervB; int tf;}
class B {int soid; L* baseL; R* baseR; T* baseT; int bf;}
```

Note that pointers `dervB` and `baseT` are added because T is a shared base class of B.

Example 8 (Object Construction). Consider the statement "`B* pb = new B()`", where B is the class from Fig. 2. As a first step, all the sub-objects of the given class B are allocated[2] and the `soid` and base pointer fields are initialized:

```
// Create sub-objects
// (see Fig. 2)
B* pb       = allocnew B();
L* pb_B_L   = allocnew L();
R* pb_B_R   = allocnew R();
T* pb_B_L_T = allocnew T();
T* pb_B_R_T = allocnew T();

// Set base pointer fields
pb->baseL        = p_B_L;
pb->baseR        = p_B_R;
pb->baseL->baseT = p_B_L_T;
pb->baseR->baseT = p_B_R_T;

// Set soid fields
pb->soid       = SOID([B,⟨B⟩]);
pb_B_L->soid   = SOID([B,⟨B,L⟩]);
pb_B_R->soid   = SOID([B,⟨B,R⟩]);
pb_B_L_T->soid=SOID([B,⟨B,L,T⟩]);
pb_B_R_T->soid=SOID([B,⟨B,R,T⟩]);

// Invoke the constructor for B
pb->B();
```

[2] `allocnew C()` allocates memory for an object of type C on the heap. It should be noted that we require that related calls to `allocnew()` either all succeed or the first one itself fails.

In addition to the above steps, all the constructors are modified to initialize the derived pointer fields of every immediate base class and shared base class, and subsequently, invoke their constructors. For example, the constructor for B is modified as follows:

```
B::B(B* this) {
  this->baseL->dervB = this; this->baseL->L();
  this->baseR->dervB = this; this->baseR->R();
  ...
}
```

Similarly, the constructors of T, L, and R are also modified. The invocation of the constructors of L and R (which in turn would invoke the constructor of T) ensures that the sub-objects of B are initialized properly. ■

Note that while the number of sub-object *types* grows exponentially with the size of the class inheritance graph, the actual number of sub-object *instances* only grows linearly with the size of the class inheritance graph. Therefore, we have not seen the increase in the number of sub-objects affect the scalability of the analysis (see Sect. 5).

Example 9 (Cast and Field Accesses). Consider a cast statement "tgt =(T*)src", where tgt is of type T and src is of type S. First, all the sub-objects that src may legally point-to at runtime are determined. For every such sub-object σ, the access path ρ starting from src consisting of a sequence of derived and base pointer fields to reach the required T-sub-object is computed. Finally, a switch..case statement is generated with a case for every sub-object and access path pair $\langle\sigma,\rho\rangle$ that updates tgt.

Fig. 5 shows a few examples. Consider the translation for the cast statement "pb = (B*) pt". The sub-objects that pt may point-to at runtime are $[B,\langle B,L,T\rangle]$ and $[B,\langle B,R,T\rangle]$ (see Fig. 2(c)). The corresponding access paths are pt->dervL->dervB and pt->dervR->dervB, which are assigned to the target pointer pb in the respective cases. Note that if pt does not point to either of the two sub-objects, the target pointer pb is set to NULL, which mimics the semantics of a dynamic_cast.

The cast statement "pr = (R*)pt" in Fig. 5 demonstrates the case where the access path consists of derived pointer fields followed by base pointer fields. The translation is shown in Fig. 5 (after grouping common cases). Note that there is no case for $[L,\langle L,T\rangle]$ because (L $\not<_{sr}$ R), and therefore, $[L,\langle L,T\rangle]$ will not be in the set of sub-objects that pt may legally point to at runtime.

Consider a field access tgt = src->S::m. The statement is transformed in CHROME by considering it as tgt = ((S*)src)->S::m. The idea is to treat a field access as equivalent to the code sequence consisting of a cast of src to (S*) followed by the field access. Basically, the access path from src is generated and then the member is accessed. Fig. 5 shows the translation for z = pl->tf. (A similar strategy is used for member calls without dynamic dispatch.) ■

Example 10 (Virtual-function calls). For p->C::foo(p,...), where C::foo is a virtual function, all possible sub-objects that p could be pointing to at runtime is determined. Finally, a switch..case statement is generated with a case for each

```
// Downcast: pb = (B*)pt
// (B* pb, T* pt)
switch(pt->soid) {
  case SOID([B,⟨B,L,T⟩]):
    pb = pt->dervL->dervB;
    break;
  case SOID([B,⟨B,R,T⟩]):
    pb = pt->dervR->dervB;
    break;

  default:
    pb = NULL;
}
```

```
// Cast: pr = (R*)pt
// (R* pr, T* pt)
switch(pt->soid) {
  case SOID([R,⟨R,T⟩]):
  case SOID([B,⟨B,R,T⟩]):
    pr = pt->dervR;
    break;
  case SOID([B,⟨B,L,T⟩]):
    pr =
     pt->dervL->dervB->baseR;
    break;
  default:  pr = NULL;
}
```

```
// Upcast: pt = (T*)pl
// (T* pt, L* pl)
switch(pl->soid) {
  case SOID([B,⟨B,L⟩]):
  case SOID([L,⟨L⟩]):
    pt = pl->baseT;
    break;
 default: pt = NULL;
}
```

```
// Field access: z = pl->tf
// (int z, L* pl)
switch(pl->soid) {
  case SOID([B,⟨B,L⟩]):
  case SOID([L,⟨L⟩]):
    z = pl->baseT->tf;
    break;
  default: assert(false);
}
```

Fig. 5. CHROME translation for casts and field accesses with the class hierarchy in Fig. 2

sub-object. The actual member function that would be invoked at run-time is called in each case of the switch statement. Note that, in each case, the access paths need to be adjusted accordingly for `p`. Consider the class hierarchy in Fig. 2. Suppose that class `T` defines a virtual function `vfunc` and class `L` overrides it. The virtual-function call `pt->vfunc()` is translated as shown in Fig. 6. Note that if `pt` points to a sub-object inherited from L, `L::vfunc` is invoked. ■

```
switch(pt->soid) {
  case SOID([B,⟨B,L,T⟩]):
  case SOID([L,⟨L,T⟩]):
    pt->dervL->
      L::vfunc(pt->dervL);
    break;
  case SOID([B,⟨B,R,T⟩]):
  case SOID([T,⟨T⟩]):
    pt->T::vfunc(pt);
    break;
  default: assert(false);
}
```

Fig. 6. CHROME translation for a virtual-function call

Special Handling. It is easy to adapt the transformations described so far to the other constructs in the C++0x standard except for the following cases that require special attention. (1) For virtual-base classes, the CHROME transformations ensure that the body of the constructor (or destructor) of a virtual base class is only invoked once and it is always invoked from the constructor (or destructor) of the most derived object. (2) For virtual-function calls on an object that is being constructed, the CHROME lowering follows C++ semantics by

treating the partially constructed object as though it is an object of the type to which the constructor belongs. (3) At object assignments, the CHROME lowering generates additional assignments that copies the sub-objects associated with the source into the sub-objects associated with the target. (4) Template classes and functions are instantiated. (5) Exceptional control-flow is made explicit for sound static analysis. We have presented an algorithm for transforming a C++ program with exceptions into a semantically equivalent C++ program without exceptions elsewhere [26]. The exception-elimination transformations can be performed in conjunction with the inheritance-elimination transformations described here.

Lowering without the `soid` *field.* For ease of presentation, the CHROME transformations presented so far used the `soid` field to determine valid paths in the sub-object graph. However, we can often omit the `soid` field because the auxiliary base class and derived class pointers themselves encode valid paths in the sub-object model; invalid paths are indicated by NULL values for the base class and derived class pointer fields. For example, the downcast in Fig. 5 can be translated without the `soid` field as follows:

```
// Downcast: pb = (B*)pt
// (B* pb, T* pt)
if (pt->dervR && pt->dervR->dervB) {
  pb = pt->dervR->dervB;
} else if (pt->dervL && pt->dervL->dervB) {
  pb = pt->dervL->dervB;
} else {
  pb = NULL;
}
```

The advantage of using the auxiliary pointers directly is that subsequent static analysis algorithms need not maintain the relationship between the value of the `soid` field and the auxiliary base and derived class pointers.

Correctness of the transformations. Here we provide the intuition as to why the transformations are semantics preserving. At object construction, it is easy to see that all the sub-objects are allocated and the fields are initialized by the chaining of constructor calls. After a cast statement, the target of a cast operation has to point to the relevant sub-object. To achieve this, compilers generate code that adjusts the pointer appropriately at runtime. The CHROME transformations mimic this behavior by generating access paths consisting of derived and base pointer fields. Because the derived and base pointer fields are set up to point to the correct sub-objects at object construction, the lowered code mimics the behavior of casts correctly. Similarly, at a virtual-function call, the appropriate member function is invoked based on the runtime type stored in the `soid` field, which is the expected behavior.

Because CHROME uses a different object layout, the memory behavior of a CHROME-lowered program is different from the original program. This is not an issue for the correctness of the transformations unless the program modifies the objects using low-level primitives like `memset`. However, programmers typically do not perform low-level operations like `memset` on objects (like they sometimes do in C) that use inheritance because it is highly compiler-dependent and can mess up the virtual-function pointer tables and other compiler-level data structures. Because we only change the

Table 1. Characteristics of the C++ benchmarks. LOC: the number of non-empty lines after preprocessing. **#class**: the number of classes from the standard libraries (lib) and from the application (app). **#mult**: number of classes with multiple inheritance in standard libraries (lib) and the actual application (app). **#VPTR**: number of accesses to the virtual-function pointer table. **#FPTR**: number of function-pointer calls. **#T**: time taken for CHROME-lowering in seconds.

	C++ program						Lowered C Program				
							COMPILER			CHROME	
			#class		#mult						
	LOC	#T	lib	app	lib	app	LOC	#VPTR	#FPTR	LOC	#FPTR
coldet (1.2)	5.1K	1.7s	32	61	0	4	7.0K	48	31	14.4K	0
mailutils (2.1)	8.3K	2.1s	14	106	0	0	8.7K	2	0	17.1K	0
tinyxml (2.5.3)	4.9K	2.0s	0	59	0	0	12.5K	110	79	21.6K	0
id3lib (3.8.3)	14.5K	8.0s	75	106	6	0	35.7K	632	499	77.7K	0
cppcheck (1.4.3)	30.9K	30s	148	104	7	5	99.9K	217	71	165.0K	0

objects that use inheritance, such low-level operations do not pose a problem for the semantic correctness of our transformations assuming compiler-independent code.

5 Implementation and Experiments

We have implemented the ideas described in the paper in an in-house extension of CIL [24] called `CILpp`. The C++ front-end for `CILpp` is based on `EDG` [15], and therefore, handles all aspects of the C++0x standard. For the experiments, we translated the given C++ program into an equivalent C program using (1) a standard compiler-based lowering mechanism, and (2) the lowering mechanism based on the CHROME object model. Henceforth, we refer to the C program obtained from compiler-based lowering as COMPILER-lowered C program, and the one obtained from CHROME-based lowering as CHROME-lowered C program.

Tab. 1 shows the characteristics of the open source benchmarks used for our experiments. The open source library `coldet` (v1.2) implements collision detection algorithms and is often used in game programming. GNU `mailutils` (v1.2) is a collection of mail utilities, servers, and clients. `TinyXML` (v2.5.3) is a light-weight XML parser which is widely used in open source and commercial products. The open source library `id3lib` (v3.8.3) is used for reading, writing, and manipulating ID3v1 and ID3v2 tags, which is the metadata format for MP3 audio files. `cppcheck` (v1.4.3) is a tool for static C/C++ code analysis[3] that uses a library of problematic code patterns to detect common errors. For the experiments, the sources relevant to the project were merged into a single C++ file and preprocessed. The preprocessed file was lowered using the compiler-based and CHROME-based lowering mechanisms.

[3] We also ran `cppcheck` on our set of benchmarks, but it did not find any of bugs reported in Sect. 5.2. The patterns used by `cppcheck` are not sufficient enough to find bugs that deep static analyzers are capable of detecting.

5.1 Complexity of the Lowered C Programs

The column labeled "LOC" in Tab. 1 refers to the number of non-empty lines of code in the merged file after preprocessing. The COMPILER-lowered C program is up to 3 times larger than the original C++. For the mailutils example, which has no virtual-function calls, the size of the COMPILER-lowered program is roughly the same as the original C++ program. This suggests that the extra statements generated by compiler-based lowering mostly relate to the setup and access of virtual-function pointer tables. The CHROME-lowered C program is roughly *three to five* times the size of the original C++ program, and is roughly *twice* that of the COMPILER-lowered C program. The difference in the sizes between COMPILER-based and CHROME-based lowering is mostly due to `switch..case` statements generated by the CHROME transformations. Even though the COMPILER-lowered programs are smaller than the CHROME-lowered programs, COMPILER-lowered programs contain operations that are hard for a static analyzer to reason about, such as calls via function pointers and accesses to virtual-function pointer tables. The column labeled "#FPTR" shows the number of calls through function pointers, and the column labeled "#VPTR" shows the number of accesses to virtual-function pointer tables. The function pointer calls in COMPILER-lowered program correspond to the virtual-function calls in the original C++ program. The CHROME-lowered programs do not have calls via function pointers because CHROME-based transformations do not use function pointers for virtual-function calls. Note that the number of accesses to a virtual-function pointer table is generally more than the number of calls via function pointers, which indicates that virtual-function pointer tables are also used for purposes other than dispatching virtual-function calls.

5.2 Effectiveness of Lowering for Software Verification

For this experiment, we analyzed the COMPILER-lowered and CHROME-lowered programs using F-SOFT [18]. F-SOFT is a tool for finding bugs in C programs, and uses a combination of abstract interpretation and model checking to find common programming mistakes, such as NULL-pointer dereferences, memory leaks, buffer overruns, and so on. Given a C program, F-SOFT systematically instruments the program in such a way that an assertion is triggered whenever a safety property is violated. For example, at every dereference of a pointer, the program is instrumented to trigger an assertion if the pointer is NULL. An abstract interpreter [12] is used as a proof engine in F-SOFT. The abstract interpreter computes invariants that can be used to prove that certain assertions can never be reached. Our abstract interpreter is inter-procedural, flow and context sensitive. It is built in a domain-independent and extensible fashion, allowing for various abstract domains such as constants, intervals [11], octagons [22], symbolic ranges [31] and polyhedra [14]. These domains

Table 2. Results of memory leak and pointer validity checker using F-SOFT on the lowered programs. **#N**: the number of NULL-pointer dereferences. **#M**: the number of memory leaks. The number in parenthesis shows the number of real bugs. The time limit was set to 20 minutes for each function.

	COMPILER		CHROME	
	#N	#M	#N	#M
coldet	0	0	5(2)	0
mailutils	0	0	0	0
tinyxml	3(0)	0	0	2(0)
id3lib	4(3)	1(1)	6(5)	6(1)
cppcheck	1(1)	0	3(2)	1(0)

```
const uchar* ID3_FieldImpl::GetRawBinary() const
{
  const uchar* data = NULL;
  if (this->GetType() == ID3FTY_BINARY) {
    data = _binary.data();
  }
  return data;
}
void ID3_FieldImpl::RenderBinary(ID3_Writer& writer)
{
  writer.writeChars(this->GetRawBinary(),
                    this->Size());
}
```

Fig. 7. NULL pointer dereference in `id3lib`

are applied in increasing order of complexity. After each analysis is run, the assertions that are proved to be unreachable are removed and the program is simplified by constant propagation and slicing. After abstract interpretation, the remaining properties are checked by a SAT-based bounded model checker. If the model checker finds any violations, they are reported to the user along with a witness trace. For the sake of usability, F-SOFT does not report sound analysis results, and uses heuristics to identify patterns that commonly result in false warnings and eliminates them.

Tab. 2 summarizes the results of a pointer-validity checker and a memory-leak checker in F-SOFT, where the number of witnesses reported by the model checker is presented along with the number of real bugs reported in parenthesis. The time limit was set to 20 minutes for each function. It should be noted that the lowering techniques produce entirely different (but semantically equivalent) programs. Therefore, the number of properties in the CHROME-lowered program is different from the number of properties in the COMPILER-lowered program. In summary, F-SOFT found a total of ten real bugs, of which five were found only when F-SOFT analyzed the CHROME-lowered program. Note, that all of these bugs were previously unknown. Finally, we note that all witnesses found using the COMPILER-lowered were also found by F-SOFT when analyzing CHROME-lowered program.

In the following, we highlight a few bugs that were found using the CHROME-lowered C program. For the CHROME-lowered programs, F-SOFT reported a total of 14 NULL-pointer dereference witnesses of which 9 were found to be real bugs.

NULL-pointer dereference in virtual-function calls. Consider the code snippet from `id3lib` shown in Fig. 7. It is possible to dereference a NULL pointer as follows: the body of `writeChars` method (not shown) assumes that the first argument to `writeChars` is never NULL, but `GetRawBinary` method may return NULL. The class hierarchy in `id3lib` is not trivial: `ID3_FieldImpl` is derived from `ID_Field` and `ID3_Writer` is a base class for 9 derived classes of which 7 are immediate derived classes. Further, all the methods invocations in Fig. 7 are virtual. When analyzing the CHROME-lowered C program, F-SOFT presents a witness that invokes `RenderBinary` with an object of type `ID3_FieldImpl` for the `this` pointer and an object of type

Table 3. Effectiveness of CHROME in CBMC. C++ features are as follows: inheritance (INH), multiple inheritance (MI), dynamic cast (DC), and virtual functions (VF). Results are as follows: false positives (FP), false negative (FN), front-end failure (FF), and OK (√).

	C++ Features				CBMC	CBMC		F-SOFT	
	INH	MI	DC	VF	goto-cc	COMPILER	CHROME	COMPILER	CHROME
P1	√				FP	√	√	FN	√
P2	√	√			FF	√	√	FN	√
P3	√		√		FN	FN	√	FN	√
P4	√			√	√	FN	√	FN	√
P5	√		√	√	FN	FN	√	FN	√
P6	√	√	√	√	FF	FN	√	FN	√

`UnsyncedWriter` for the reference parameter `writer`, followed by a call to the method `UnsyncedWriter::writeChars`, where the NULL pointer dereference occurs. This bug was not found when the COMPILER-lowered code is analyzed by F-SOFT because of the use of a virtual-function pointer table at virtual-function calls.

Interprocedural NULL pointer dereferences. The analysis also discovered four more scenarios where a NULL pointer dereference may occur in `id3lib`. These are related to methods that return NULL pointers instead of valid C-strings under certain error conditions. However, at certain call-sites to these methods, the returned C-string is passed on to string manipulation functions such as `strcmp` without checking if the returned string is NULL, thereby potentially causing segmentation faults. In addition, F-SOFT found four bugs of a similar kind in `coldet` and `cppcheck`.

Finally, in the `TiXmlComment::Parse` method of the `TinyXML` project, F-SOFT reported a NULL-pointer dereference. The NULL pointer dereference occurs when the input string to `TiXmlComment::Parse` is empty. However, from further investigation , it seems unlikely that this function will be called using an empty string.

Memory Leaks. When analyzing the CHROME-lowered program for memory leaks, it reported 9 warnings, of which 1 was a real memory leak. The memory leak happens in a method named `convert_i` in file `utils.cc` of `id3lib`. F-SOFT reported this leak for the COMPILER-lowered program also.

5.3 Applicability in Other Verification Tools

In addition to using F-SOFT, we also wanted to investigate the applicability of our lowering in other verification tools. For this experiment, we created a collection of microbenchmarks [23] that exercises various aspects of C++. Each program has two assertions of which one always fails and the other always succeeds at runtime. For each benchmark, we generated the COMPILER-lowered and CHROME-lowered programs and analyzed them using the CBMC verfication tool [8]. The CBMC suite uses the `goto-cc` C++ front-end to generate an intermediate representation for analysis.

Tab. 3 shows the results of our experiments. The column labeled `goto-cc` shows the results of running CBMC directly on the C++ program. A false negative refers to the case where the tool does not report the failing assertion. A false positive refers to the case where the tool reports an error for the succeeding assertion. As the table shows CBMC with `goto-cc` does not perform well when complex C++ features such

as multiple inheritance and dynamic casts are used. It generates a false positive or a false negative in many cases. The results highlight how difficult and error prone it is to encode C++ semantics in program analyzers. When CBMC is used to analyze the COMPILER-lowered program, it generates false negatives in many cases.

On the other hand, when CBMC is used to analyze the CHROME-lowered program, it does not generate any false reports. (F-SOFT also performs well when a CHROME-lowered program is analyzed.) This points to the effectiveness of using the CHROME transformations even in other verification tools.

6 Related Work

The sub-object formalism presented by Rossie and Friedman forms the basis of our CHROME object model transformations [30]. Ramalingam and Srinivasan present a member lookup algorithm for C++ [28] which operates directly on the class hierarchy graph (CHG) instead of the sub-object graph, which may be exponential in the size of the CHG. The issue of member lookup is orthogonal, but complementary, to the problem of translating a C++ program with inheritance into a semantically equivalent program without inheritance. Based on the Rossie-Friedman model, a number of class hierarchy transformations have also been proposed which preserve the program semantics, e.g., by slicing class hierarchies in libraries according to their clients for producing optimized code [38].

Various approaches formalize the object layout in C++ to study space overhead and performance aspects of object layouts [35,36], and perform formal verification of object layouts [5,29,39]. These formalisms are geared towards devising memory efficient layouts and the correctness of the layout algorithms. In contrast, the goal of CHROME is to embed the object model into the program to make it more amenable to static analysis. However, such formalisms are complementary to our approach and may be used to establish the correctness of CHROME transformations.

Another work that closely relates to this paper is the LLVM compiler framework [19]. The LLVM framework translates a given C++ program into a low-level three address code. Unlike our lowering algorithm, LLVM algorithm uses virtual function tables and runtime libraries during lowering, and therefore, produces code that is not very amenable for precise program analysis.

The ROSE compiler front-end [27] and Clang static analyzer [6] are other popular front-ends that generate an Intermediate Representation (IR) for C++ programs. These front-ends support all C++ language features. However, the IR produced by these front-ends still contain complex C++ features such as inheritance and casts. Therefore, every analysis that uses their IR has to deal with inheritance and casts. On the other hand, the source-to-source transformations presented in this paper eliminate complex C++ features, thereby making the implementation of subsequent analysis easier. The techniques presented here may be used to simplify the IRs generated by ROSE and Clang.

A combination of the notions of delegation [21] (instantiating additional object fields and forwarding method calls to them) and interfaces has been used [37] to simulate multiple inheritance in languages. However, these methods do not delve into the

complexities of C++ object model, e.g., handling shared and replicated inheritance, casting, etc. The CHROME transformations, in contrast, handle these features precisely.

Chen [5] proposed a typed intermediate language (IL) and a method to compile multiple inheritance into the language. The IL, however, is quite mathematical in nature and dedicated analyses must be designed for the IL. In contrast, our target language is (a subset of) C++ itself, and hence, the target program is immediately amenable to conventional static analysis.

Finally, there is a vast literature on analysis (static or dynamic) of object-oriented programs, mostly focused on Java programs [20]. In particular, Chandra et al. proposed directed call graph construction [4] for Java programs to handle the explosion in the number of potential virtual method calls, by interleaving call graph construction with backward symbolic analysis.

7 Conclusions

In this paper, we presented an algorithm to translate a C++ program with inheritance into a C++ program without inheritance using a representation of sub-objects called CHROME. We also showed the effectiveness of the CHROME lowering on program analysis applications such as software model checking. The C program obtained using the CHROME-based transformations enabled better results than the C program obtained from a standard compiler-based lowering algorithm. We found a total of ten previously unknown bugs, of which *five* were only found due to the precise modeling of objects using CHROME. The results are quite encouraging and validates that our CHROME-lowered code is better suited for program analysis.

References

1. Apache. Xerces project, http://xerces.apache.org/
2. Balakrishan, G., Maeda, N., Sankaranarayanan, S., Ivančić, F., Gupta, A., Pothengil, R.: Modeling and analyzing the interaction of C and C++ strings. In: Int. Conf. on Formal Verif. of Object-Oriented Software (2011)
3. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: POPL, pp. 1–3 (2002)
4. Chandra, S., Fink, S.J., Sridharan, M.: Snugglebug: A powerful approach to weakest preconditions. In: PLDI, pp. 363–374 (2009)
5. Chen, J.: A typed intermediate language for compiling multiple inheritance. In: POPL (2007)
6. Clang static analyzer, http://clang-analyzer.llvm.org
7. C++ support for Clang, http://clang-analyzer.llvm.org/dev_cxx.html (accessed January 6, 2012)
8. Clarke, E., Kroning, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
9. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge (1999)
10. Comeau C++ compiler, http://www.comeaucomputing.com

11. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proc. 2nd. Int. Symp. on Programming, Paris (April 1976)
12. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In: POPL (1977)
13. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTREÉ Analyzer. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)
14. Cousot, P., Halbwachs, N.: Automatic discovery of linear constraints among variables of a program. In: POPL, pp. 84–96 (1978)
15. C++ frontend. Edison Design Group, NJ
16. Havelund, K., Pressburger, T.: Model checking java programs using java pathfinder. STTT 2(4), 366–381 (2000)
17. Hovemeyer, D., Pugh, W.: Finding more null pointer bugs, but not too many. In: PASTE, pp. 9–14 (2007)
18. Ivančić, F., Shlyakhter, I., Gupta, A., Ganai, M., Kahlon, V., Wang, C., Yang, Z.: Model checking C programs using F-Soft. In: IEEE International Conference on Computer Design, pp. 297–308 (October 2005)
19. Lattner, C.: LLVM: A compilation framework for lifelong program analysis and transformation. In: Int. Symp. on Code Generation and Optimization (2004)
20. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. Formal Asp. Comput. 19(2), 159–189 (2007)
21. Lieberman, H.: Using prototypical objects to implement shared behavior in object oriented systems. In: OOPSLA, pp. 214–223 (1986)
22. Miné, A.: The octagon abstract domain. In: Working Conf. on Rev. Eng. (2001)
23. NECLA verification benchmarks, http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php
24. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
25. Nokia. Qt: a cross-platform application and UI framework, http://qt.nokia.com/
26. Prabhu, P., Maeda, N., Balakrishnan, G., Ivančić, F., Gupta, A.: Interprocedural Exception Analysis for C++. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 583–608. Springer, Heidelberg (2011)
27. Quinlan, D.J.: Rose: Compiler support for object-oriented frameworks. Parallel Processing Letters 10(2/3), 215–226 (2000)
28. Ramalingam, G., Srinivasan, H.: A member lookup algorithm for C++. In: SIGPLAN Conf. on Prog. Lang. Design and Impl., pp. 18–30. ACM, New York (1997)
29. Ramananandro, T., Reis, G.D., Leroy, X.: Formal verification of object layout for C++ multiple inheritance. In: POPL (2011)
30. Rossie Jr., J.G., Friedman, D.P.: An algebraic semantics of subobjects. In: OOPSLA, pp. 187–199. ACM, New York (1995)
31. Sankaranarayanan, S., Ivančić, F., Gupta, A.: Program Analysis Using Symbolic Ranges. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 366–383. Springer, Heidelberg (2007)
32. C. standards commitee. Working draft, standard for C++, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf (accessed January 6, 2012)
33. Stroustrup, B.: Multiple inheritance for C++. Computing Systems 2(4), 367–395 (1989)

34. Stroustrup, B.: Evolving a language in and for the real world: C++ 1991-2006. In: Proc. of History of Programming Languages III (2007)
35. Sweeney, P.F., Burke, M.: Quantifying and evaluating the space overhead for alternative C++ memory layouts. Softw. Pract. Exper. 33(7), 595–636 (2003)
36. Sweeney, P.F., Gil, J.Y.: Space and time-efficient memory layout for multiple inheritance. In: OOPSLA. ACM, New York (1999)
37. Thirunarayan, K., Kniesel, G., Hampapuram, H.: Simulating multiple inheritance and generics in Java. Comp. Lang. 25(4), 189–210 (1999)
38. Tip, F., Sweeney, P.F.: Class hierarchy specialization. Acta. Inf. 36(12), 927–982 (2000)
39. Wasserrab, D., Nipkow, T., Snelting, G., Tip, F.: An operational semantics and type safety proof for multiple inheritance in C++. In: OOPSLA, pp. 345–362. ACM Press (2006)
40. Xie, Y., Aiken, A.: Saturn: A scalable framework for error detection using boolean satisfiability. Trans. on Prog. Lang. and Syst. 29(3) (2007)

GC-Safe Interprocedural Unboxing

Leaf Petersen and Neal Glew

Intel Labs, Santa Clara CA
{leaf.petersen,neal.glew}@intel.com

Abstract. Modern approaches to garbage collection (*GC*) require information about which variables and fields contain GC-managed pointers. Interprocedural flow analysis can be used to eliminate otherwise unnecessary heap allocated objects (*unboxing*), but must maintain the necessary GC information. We define a core language which models compiler correctness with respect to the GC, and develop a correctness specification for interprocedural unboxing optimizations. We prove that any optimization which satisfies our specification will preserve GC safety properties and program semantics, and give a practical unboxing algorithm satisfying this specification.

1 Introduction

Precise garbage collection (*GC*) for managed languages is usually implemented by requiring the compiler to keep track of meta-data indicating which variables and fields contain GC-managed references and which should be ignored by the garbage collector. We refer to this information as the *traceability* of a field or variable: a field or variable is traceable if it should be treated as a pointer into the heap by the garbage collector.

In order to maintain this information in the presence of polymorphic functions (including subtype polymorphism), many languages and compilers use a *uniform object representation* in which every source level object is represented at least initially by a heap allocated object. All interprocedural use of native (non-heap) data therefore occurs only through fields of objects. This is commonly referred to as *boxing*, objects represented in this way are referred to as *boxed*, and projecting out of the uniform representation is referred to as *unboxing*. Boxing imposes substantial performance penalties for many reasons: the additional overhead of the allocation and projection is substantial, arrays of boxed objects exhibit poor locality, and the additional memory pressure can cause bottlenecks in the hardware. In this paper, we show how to use the results of interprocedural flow analyses (reaching definitions) to implement an interprocedural unboxing optimization while preserving the meta-data necessary for precise garbage collection.

In the following sections, we define a high-level core language that captures the essential aspects of GC meta-data and GC safety. We then give a high-level specification of what a reasonable flow analysis on this language must compute,

M. O'Boyle (Ed.): CC 2012, LNCS 7210, pp. 165–184, 2012.
© Springer-Verlag Berlin Heidelberg 2012

and define a notion of a general unboxing optimization for this language. We give a specification for when such an optimization is acceptable for a given flow analysis. We show that the optimization induced by an acceptable unboxing preserves the semantics of the original program, including GC safety. Finally, we construct an algorithm closely based on one in use in our compiler and prove that it produces an unboxing that satisfies our correctness specification, and hence that it preserves the semantics of the program (including GC safety).

All the lemmas and theorems in the paper have been proven. Proofs are available in an extended technical report available from the first author's website[7].

2 GC Safety

Consider the following program (using informal notation), where `box` denotes a boxing operation that wraps its argument in a heap-allocated structure, and `unbox` denotes its elimination form that projects out the boxed item from the box:

$$\texttt{let}\ f\ =\ \lambda x.(\texttt{box}\, x)\ \texttt{in}\, \texttt{unbox}(\texttt{unbox}(f\ (\texttt{box}\, 3)))$$

The only definition reaching the variable x is the boxed machine integer 3. Information from an interprocedural analysis can be used to rewrite this program to eliminate the boxing as follows:

$$\texttt{let}\ f\ =\ \lambda x.x\ \texttt{in}\, f\ 3$$

In the second version of this program, the traceability of the values reaching x has changed: whereas in the original program all values reaching x are represented as heap allocated pointers, in the second program all values reaching x are represented as machine integers. From the standpoint of a garbage collector, a garbage collection occuring while x is live must treat x as a root in the first program, and must ignore x in the second program. There are numerous approaches to communicating this information to the garbage collector. For example, some implementations choose to dynamically tag values in such a way as to allow the garbage collector to distinguish pointers from non-pointers by inspection. Such an implementation might steal a low bit from the machine integer representation to allow the machine integer 3 to be distinguished from a heap pointer.

Another very commonly used approach (particularly in more recent systems) is to require the compiler to statically annotate the program with garbage collection meta-data such that at any garbage collection point the garbage collector can reconstruct exactly which live variables are roots. Typically, this takes the form of annotations on variables and temporaries indicating which contain heap-pointers (the roots) and which do not (the non-roots), along with information at every allocation site indicating which fields of the allocated object contain traceable data. It is this approach that we target in this paper.

The requirement that the compiler be able to annotate program variables with a single static traceability constrains the compiler's ability to rewrite programs

in that it must do so in a way that preserves the correctness of the GC meta-data of the program. Consider an extension of the previous example.

$$\texttt{let}\ f\ =\ \lambda x.x\ \texttt{in}\,\texttt{unbox}((f\ f)\ (\texttt{box}\,3))$$

Assuming that functions are represented as heap-allocated objects then each variable in this program can be assigned a traceability, since all objects passed to f are heap references. However, an attempt to unbox this program as with the previous example results in f being applied to both heap references (f) and non-heap references (3).

$$\texttt{let}\ f\ =\ \lambda x.x\ \texttt{in}(f\ f)\ 3$$

As this example shows, the concerns of maintaining garbage-collector meta-data constrain optimization[1] in ways not apparent in a GC-ignorant semantic model.

2.1 A Core Language for GC Safety

In order to give a precise account of interprocedural unboxing, we begin by defining a core language capturing the essential features of GC safety. The motivation for the idiosyncracies of this language lies in the requirements of the underlying model of garbage collection. We assume that pointers cannot be intrinsically distinguished from non-pointers, and hence must be tracked by the compiler. In our implementation, the compiler intermediate language under consideration is substantially more low-level: a control-flow graph based, static single assignment intermediate representation. We believe however that all of the key issues are captured faithfully in this higher-level representation.

Traceabilities t ::=	$\texttt{b} \mid \texttt{r}$	Terms m	::=	$x \mid \lambda x^i{:}t.e \mid e_1\ e_2 \mid \texttt{box}_t\, e \mid$
Term variables	x, y, z			$\texttt{unbox}\, e \mid \rho(e)$
Constants	c	Values v	::=	$c \mid \langle \rho, \lambda x^i{:}t.e\rangle \mid \langle v^i{:}t\rangle$
Labels i ::=	$0, 1, \ldots$	Environments ρ	::=	$x_1^{i_1}{:}t_1 = {v_1}^{j_1}, \ldots, x_n^{i_n}{:}t_n = {v_n}^{j_n}$
Labeled Terms e ::=	$m^i \mid v^i$	States M	::=	(ρ, e)

Fig. 1. Syntax

Figure 1 defines the syntax of our core language. The essence of the language is largely that of the standard untyped lambda calculus with an explicit environment semantics, extended with a form of degenerate type information we call *traceabilities*. Traceabilities describe the GC status of variables: the traceability b (for bits) indicates something that should be ignored by the garbage collector, while the traceability r (for reference) indicates a GC-managed pointer. The traceability b is inhabited by an unspecified set of constants c while the

[1] It is worth noting that a serious compiler might be expected to duplicate the body of f in this simple example thereby eliminating this constraint and allowing the unboxing optimization to be more effective.

traceability $\mathtt{r}$ is inhabited by functions (anticipating their implementation by heap-allocated closures) and by boxed objects. Anticipating the needs of the flow analysis, we label each term, value, and variable binding site with an integer label. We do not assume that labels or variables are unique within a program.

Expressions e consist of labeled terms m^i and labeled values v^i. The terms m consist of variables, functions, applications, box introductions, box eliminations, and frames. Variable binding sites are decorated with traceability information ($\lambda x^i{:}t.e$), as are box introductions ($\mathtt{box}_t\, e$). We represent heap allocation in the language via the $\mathtt{box}_t\, e$ term, which corresponds to allocating a heap cell containing the value for e. The traceability t gives the meta-data with which the heap-cell will be tagged, allowing the garbage collector to trace the cell. Objects can be projected out of an allocated object by the $\mathtt{unbox}\, e$ operation. Frames $\rho(e)$ are discussed below.

Values consist of either constants, closures, or heap-allocated boxes. We distinguish between the introduction form ($\mathtt{box}_t\, e$) and the value form ($\langle v^i{:}t\rangle$) for allocated objects. The introduction form corresponds to the allocation instruction, whereas the value form corresponds to the allocated heap value. This distinction is key for the formulation of GC safety and the dynamic semantics. For the purposes of the dynamic semantics we also distinguish between functions ($\lambda x^i{:}t.e$) and the heap allocated closures that represent them at runtime ($\langle \rho, \lambda x^i{:}t.e\rangle$).

For notational convenience, we will sometimes use the notation $v_{\mathtt{b}}$ to indicate that a value v is a non-heap-allocated value (i.e. a constant c), and $v_{\mathtt{r}}$ to indicate that a value v is a heap-allocated value (i.e. either a lambda value or a boxed value). If t is a traceability meta-variable, then we use v_t to indicate that v is a value of the same traceability as t. In examples, we use a derived `let` expression, taking it to be syntactic sugar for application in the usual manner. Environments ρ map variables to values. The term $\rho(e)$ executes e in the environment ρ rather than the outer environment – all of the free variables of e are provided by ρ. The nested set of these environments at any point can be thought of as the activation stack frames of the executing program. The traceability annotations on variables in the environments play the role of stack frame GC meta-data, indicating which slots of the frame are roots (traceability $\mathtt{r}$). The environments buried in closures ($\langle \rho, \lambda x^i{:}t.e\rangle$) similarly provide the traceabilities of values reachable from the closure, and hence provide the GC meta-data for tracing through closures. While we do not make the process of garbage collection explicit, it should be clear how to extract the appropriate set of GC roots from the environment and any active frames.

This core language contains the appropriate information to formalize a notion of GC safety consisting of two complementing pieces. First we define a dynamic semantics in which reductions that might lead to undefined garbage-collector behavior are explicitly undefined. Programs that takes steps in this semantics do not introduce ill-formed heap objects. Secondly, we define a notion of a traceable program: one in which all heap values have valid GC meta-data. Reduction steps in the semantics can then be shown to maintain the traceability property. The

GC correctness criteria for a compiler optimization then is that the optimization map traceable programs to traceable programs, and that it not introduce new undefined behavior.

2.2 Operational Semantics

We choose to use an explicit environment semantics rather than a standard substitution semantics since this makes the GC meta-data for stack frames and closures explicit in the semantics. Thus a machine state (ρ, e) supplies an environment ρ for e that provides the values of the free variables of e during execution. Environments contain traceability annotations on each of the variables mapped by the environment.

$$\frac{x^i{:}t = v^j \in \rho}{(\rho, x^k) \longmapsto (\rho, v^j)} \qquad \frac{}{(\rho, (\lambda x^i{:}t.e)^j) \longmapsto (\rho, \langle \rho, \lambda x^i{:}t.e\rangle^j)}$$

$$\frac{t = t'}{(\rho, (\mathtt{box}_t\, v_{t'}{}^i)^j) \longmapsto (\rho, \langle v_{t'}{}^i{:}t\rangle^j)} \qquad \frac{(\rho, e_1) \longmapsto (\rho, e_1')}{(\rho, (e_1\ e_2)^i) \longmapsto (\rho, (e_1'\ e_2)^i)}$$

$$\frac{(\rho, e_2) \longmapsto (\rho, e_2')}{(\rho, (v^i\ e_2)^j) \longmapsto (\rho, (v^i\ e_2')^j)} \qquad \frac{t = t'}{(\rho, (\langle \rho', \lambda x^i{:}t.e\rangle^j\ v_{t'}{}^k)^l) \longmapsto (\rho, (\rho', x^i{:}t = v_{t'}{}^k)(e)^l)}$$

$$\frac{(\rho, e) \longmapsto (\rho, e')}{(\rho, (\mathtt{box}_t\, e)^i) \longmapsto (\rho, (\mathtt{box}_t\, e')^i)}$$

$$\frac{(\rho, e) \longmapsto (\rho, e')}{(\rho, (\mathtt{unbox}\, e)^i) \longmapsto (\rho, (\mathtt{unbox}\, e')^i)} \qquad \frac{}{(\rho, (\mathtt{unbox}\, \langle v^i{:}t\rangle^j)^k) \longmapsto (\rho, v^i)}$$

$$\frac{(\rho', e) \longmapsto (\rho', e')}{(\rho, \rho'(e)^i) \longmapsto (\rho, \rho'(e')^i)} \qquad \frac{}{(\rho, \rho'(v^i)^j) \longmapsto (\rho, v^i)}$$

Fig. 2. Operational Semantics

Reduction in this language is for the most part fairly standard. We deviate somewhat in that we explicitly model the allocation of heap objects as a reduction step—hence there is an explicit reduction mapping a lambda term $\lambda x^i{:}t.e$ to an allocated closure $\langle \rho, \lambda x^i{:}t.e\rangle$, and similarly for boxed objects and values. More notably, beta-reduction is restricted to only permit construction of a stack frame when the meta-data attached to the parameter variable is appropriate for the actual argument value. This captures the requirement that stack frames have correct meta-data for the garbage collector. In actual practice, incorrect meta-data for stack frames leads to undefined behavior (since incorrect meta-data may cause arbitrary memory corruption by the garbage collector)—similarly here in the meta-theory we leave the behavior of such programs undefined. In a similar

fashion, we only define the reduction of the allocation operation to an allocated value ($\mathtt{box}_t\, v_{t'} \longmapsto \langle v_{t'}{:}t\rangle$) when the operation meta-data is appropriate for the value (i.e. $t = t'$).

It is important to note that this semantics does not model a dynamically checked language, in which there is an explicit check of the meta-data associated with these reductions. The point is simply that the semantics only specifies how programs behave when these conditions are met—in all other cases the behavior of the program is undefined.

2.3 Traceability

The operational semantics ensures that no reduction step introduces mis-tagged values. In order to make use of this, we define a judgment for checking that a program does not have a mis-tagged value in the first place. Implicitly this judgement defines what a well-formed heap and activation stack looks like; however, since our heap and stack are implicit in our machine states, it takes the form of a judgement on terms, values, environments, and machine states.

The value judgement $\vdash_{\mathrm{v}} v{:}t$ asserts that a value v is well-formed, and has traceability t. In this simple language, this corresponds to having the meta-data on the environment of each lambda value be consistent and the meta-data on each boxed value be consistent with the traceability of the object nested in the box. An environment is consistent, $\vdash \rho\ \mathbf{tr}$, when the annotation on each variable agrees with the traceability of the value it is bound to. Since we cannot check the consistency of general terms with the first-order information available, the term judgement $\vdash e\ \mathbf{tr}$ and machine state judgement $\vdash M\ \mathbf{tr}$ simply check that all values and environments (and hence stack frames) contained in the term or machine state are well-formed.

Labeled Terms $\boxed{\vdash e\ \mathbf{tr}}$

$$\frac{\vdash m\ \mathbf{tr}}{\vdash m^i\ \mathbf{tr}} \qquad \frac{\vdash_{\mathrm{v}} v{:}t}{\vdash v^i\ \mathbf{tr}}$$

Terms $\boxed{\vdash m\ \mathbf{tr}}$

$$\frac{}{\vdash x\ \mathbf{tr}} \qquad \frac{\vdash e\ \mathbf{tr}}{\vdash \lambda x^i{:}t.e\ \mathbf{tr}} \qquad \frac{\vdash e_1\ \mathbf{tr} \quad \vdash e_2\ \mathbf{tr}}{\vdash e_1\ e_2\ \mathbf{tr}}$$

$$\frac{\vdash e\ \mathbf{tr}}{\vdash \mathtt{box}_t\, e\ \mathbf{tr}} \qquad \frac{\vdash e\ \mathbf{tr}}{\vdash \mathtt{unbox}\, e\ \mathbf{tr}} \qquad \frac{\vdash \rho\ \mathbf{tr} \quad \vdash e\ \mathbf{tr}}{\vdash \rho(e)\ \mathbf{tr}}$$

Values $\boxed{\vdash_{\mathrm{v}} v{:}t}$

$$\frac{}{\vdash_{\mathrm{v}} c{:}b} \qquad \frac{\vdash \rho\ \mathbf{tr} \quad \vdash e\ \mathbf{tr}}{\vdash_{\mathrm{v}} \langle \rho, \lambda x^i{:}t.e\rangle{:}r} \qquad \frac{\vdash_{\mathrm{v}} v{:}t}{\vdash_{\mathrm{v}} \langle v^i{:}t\rangle{:}r}$$

Environments $\boxed{\vdash \rho\ \mathbf{tr}}$

$$\frac{\vdash_{\mathrm{v}} v_1{:}t_1 \quad \cdots \quad \vdash_{\mathrm{v}} v_n{:}t_n}{\vdash x_1^{i_1}{:}t_1 = v_1{}^{j_1}, \ldots, x_n^{i_n}{:}t_n = v_n{}^{j_n}\ \mathbf{tr}}$$

Machine States $\boxed{\vdash M\ \mathbf{tr}}$

$$\frac{\vdash \rho\ \mathbf{tr} \quad \vdash e\ \mathbf{tr}}{\vdash (\rho, e)\ \mathbf{tr}}$$

Fig. 3. Traceability

The key result for traceability is that it is preserved under reduction. That is, if a traceable term takes a well-defined reduction step, then the resulting term will be traceable.

Lemma 1 (Preservation of traceability). *If* $\vdash M$ **tr** *and* $M \longmapsto M'$ *then* $\vdash M'$ **tr**.

There is of course no corresponding progress property for our notion of traceability, since programs can go wrong. Compiler optimizations are simply responsible for ensuring that they do not introduce new ways to go wrong.

3 Flow Analysis

Our original motivation for this work was to apply interprocedural analysis to the problem of eliminating unnecessary boxing in programs. There is a vast body of literature on interprocedural analysis and optimization, and it is generally fairly straightforward to use these approaches to obtain information about what terms flow to what use sites. This paper is not intended to provide any contribution to this body of work, which we will broadly refer to as *flow analysis*. Instead, we focus on how to use the results of such a generic analysis to implement an unboxing optimization that preserves GC safety.

In order to do this, we must provide some framework for describing what information a flow analysis must provide. For the purposes of our unboxing optimization, we are interested in finding (inter-procedurally) for every $(\texttt{unbox}\, v^j)^i$ operation the set of $(\texttt{box}_t\, e)^k$ terms that could possibly reach v. Under appropriate conditions, we can then eliminate both the box introductions and the box elimination, thereby improving the program. The core language defined in Section 2 provides labels serving as proxies for the terms and variables on which they occur – the question above can therefore be re-stated as finding the set of labels k that reach the position labeled with j.

More generally, following previous work we begin by defining an abstract notion of analysis. We say that an analysis is a pair (C, ϱ). Binding environments ϱ simply serve to map variables to the label of their binding sites. The mappings are, as usual, global for the program. Consequently, a given environment may not apply to alpha-variants of a term. We do not require that labels be unique within a program—as usual however, analyses will be more precise if this is the case. Variables are also not required to be unique (since reduction may duplicate terms and hence binding sites). However, duplicate variable bindings in a program must be labeled consistently according to ϱ or else no analysis of the program can be acceptable according to our definition. This can always be avoided by alpha-varying or relabeling appropriately.

A cache C is a mapping from labels to sets of shapes. Shapes are given by the grammar:

$$\text{Shapes: } s ::= c^i \mid (i{:}t \to j)^k \mid (\texttt{box}_t\, i)^j$$

The idea behind shapes is that each shape provides a proxy for a set of terms that might flow to a given location, describing both the shape of the values that might

flow there and the labels of the sub-components of those values. For example, for an analysis (C, ϱ), $c^i \in \mathrm{C}(k)$ indicates that (according to the analysis) the constant c, labeled with i, might flow to a location labeled with k. Similarly, if $(i{:}t \to j)^k \in \mathrm{C}(l)$, then the analysis specifies that among the values flowing to locations labeled with l might be lambdas labeled with k, whose parameter variable is labeled with i and annotated with t and whose bodies are labeled with j. If $(\mathtt{box}_t\, k)^i \in \mathrm{C}(l)$ then among the values that might flow to l (according to the analysis) are boxed values labeled with i, with meta-data t and whose bodies are labeled by some j such that $\mathrm{C}(j) \subseteq \mathrm{C}(k)$.

It is important to note that the shapes in the cache may not correspond exactly to the terms in the program, since reduction may change program terms (e.g. by instantiating variables with values). However, reduction does not change the outer shape and labeling of values—it is this reduction invariant information that is captured by shapes.

Clearly, not every choice of analysis pairs is meaningful for program optimization. While in general it is reasonable (indeed, unavoidable) for an analysis to overestimate the set of terms associated with a label, it is unacceptable for an analysis to underestimate the set of terms that flow to a label—most optimizations will produce incorrect results, since they are designed around the idea that the analysis is telling them everything that could possibly flow to them. In order to capture the notion of when an analysis pair gives a suitable approximation of the flow of values in a program we follow the general spirit of Nielson et al. [6], and define a notion of an *acceptable analysis*. That is, we give a declarative specification that gives sufficient conditions for specifying when a given analysis does not underestimate the set of terms flowing to a label, without committing to a particular analysis. We arrange the subsequent meta-theory such that our results apply to any analysis that is *acceptable*. In this way, we completely decouple our optimization from the particulars of how the analysis is computed.

Our acceptable-analysis relation is given in Figure 4 – the judgement $\mathrm{C}; \varrho \vdash (\rho, e)$ determines that an analysis pair (C, ϱ) is *acceptable* for a machine state (ρ, e), and similarly for the environment and expression forms of the judgement. We use the notation $\mathrm{lbl}(e)$ to denote the outermost label of e: that is, i where e is of the form m^i or v^i. The acceptability judgement generally indicates for each syntactic form what the flow of values is. For example, in the application rule, the judgment insists that for every lambda value that flows to the applicand position, the set of shapes associated with the parameter of that lambda is a super-set of the set of shapes associated with the argument of the application; and that the set of shapes associated with the result of the lambda is a sub-set of the set of shapes associated with the application itself.

Given this definition, we can show that the acceptability relation is preserved under reduction.

Lemma 2 (Many-step reduction preserves acceptability). *If* $\mathrm{C}; \varrho \vdash M$ *and* $M \longmapsto^* M'$ *then* $\mathrm{C}; \varrho \vdash M'$.

$\boxed{\mathrm{C}; \varrho \vdash e}$

$$\frac{\mathrm{C}(\varrho(x)) \subseteq \mathrm{C}(i)}{\mathrm{C}; \varrho \vdash x^i} \qquad \frac{\varrho(x) = j \quad \mathrm{C}; \varrho \vdash e \quad (j{:}t \to \mathrm{lbl}(e))^i \in \mathrm{C}(i)}{\mathrm{C}; \varrho \vdash (\lambda x^j{:}t.e)^i}$$

$$\frac{\begin{array}{c}\mathrm{C}; \varrho \vdash e_1 \quad \mathrm{C}; \varrho \vdash e_2 \\ \forall (k{:}t \to l)^j \in \mathrm{C}(\mathrm{lbl}(e_1)) : \\ \mathrm{C}(\mathrm{lbl}(e_2)) \subseteq \mathrm{C}(k) \wedge \mathrm{C}(l) \subseteq \mathrm{C}(i)\end{array}}{\mathrm{C}; \varrho \vdash (e_1\ e_2)^i} \qquad \frac{\mathrm{C}; \varrho \vdash e \quad (\mathtt{box}_t\, j)^i \in \mathrm{C}(i) \quad \mathrm{C}(\mathrm{lbl}(e)) \subseteq \mathrm{C}(j)}{\mathrm{C}; \varrho \vdash (\mathtt{box}_t\, e)^i}$$

$$\frac{\begin{array}{c}\mathrm{C}; \varrho \vdash e \\ \forall (\mathtt{box}_t\, k)^j \in \mathrm{C}(\mathrm{lbl}(e)) : \mathrm{C}(k) \subseteq \mathrm{C}(i)\end{array}}{\mathrm{C}; \varrho \vdash (\mathtt{unbox}\, e)^i} \qquad \frac{\mathrm{C}; \varrho \vdash \rho \quad \mathrm{C}; \varrho \vdash e \quad \mathrm{C}(\mathrm{lbl}(e)) \subseteq \mathrm{C}(i)}{\mathrm{C}; \varrho \vdash \rho(e)^i}$$

$$\frac{c^i \in \mathrm{C}(i)}{\mathrm{C}; \varrho \vdash c^i} \qquad \frac{\varrho(x) = j \quad \mathrm{C}; \varrho \vdash \rho \quad \mathrm{C}; \varrho \vdash e \quad (j{:}t \to \mathrm{lbl}(e))^i \in \mathrm{C}(i)}{\mathrm{C}; \varrho \vdash \langle \rho, \lambda x^j{:}t.e \rangle^i}$$

$$\frac{\mathrm{C}; \varrho \vdash v^j \quad (\mathtt{box}_t\, k)^i \in \mathrm{C}(i) \quad \mathrm{C}(j) \subseteq \mathrm{C}(k)}{\mathrm{C}; \varrho \vdash \langle v^j{:}t \rangle^i}$$

$\boxed{\mathrm{C}; \varrho \vdash \rho}$

$$\frac{\forall 1 \leq k \leq n : \varrho(x_k) = i_k \wedge \mathrm{C}(j_k) \subseteq \mathrm{C}(i_k) \wedge \mathrm{C}; \varrho \vdash {v_k}^{j_k}}{\mathrm{C}; \varrho \vdash x_1^{i_1}{:}t_1 = {v_i}^{j_1}, \ldots, x_n^{i_n}{:}t_n = {v_n}^{j_n}}$$

$\boxed{\mathrm{C}; \varrho \vdash M}$

$$\frac{\mathrm{C}; \varrho \vdash \rho \quad \mathrm{C}; \varrho \vdash e}{\mathrm{C}; \varrho \vdash (\rho, e)}$$

Fig. 4. Acceptable Analysis

4 Unboxing

The goal of the unboxing optimization is to use the information provided by a flow analysis to replace a boxed object with the contents of the box. Doing so may change the traceability, since the object in the box may not be a GC-managed reference. Moreover, the object in the box may itself be a candidate for unboxing; consequently, determining the traceability of boxed objects depends on exactly which objects are unboxed. Function parameters may be instantiated with objects from multiple different definition sites, some of which may be unboxed and some of which may not.

Consider again the first example from Section 1, written out with explicit GC information and labels:

$$\texttt{let } f^0\texttt{:r} = (\lambda x^1\texttt{:r}.(\texttt{box}_\texttt{r}\, x^2)^3)^4 \texttt{ in}\,(\texttt{unbox}\,(\texttt{unbox}\,(f^5\ (\texttt{box}_\texttt{b}\, 3^6)^7)^8)^9)^{10}$$

It is fairly easy to see that this program is unboxable. The binding site for x is only reached by the term labeled with 7 (the outer box introduction), and hence there should be no problems with changing its traceability annotation. Each box elimination is reached only by a single box introduction, and hence the box/unbox pairs in this program should be eliminable, yielding an optimized program:

$$\texttt{let } f^0\texttt{:r} = (\lambda x^1\texttt{:b}.x^2)^4 \texttt{ in}\,(f^5\ 3^6)^8$$

Notice that in order to rewrite the program, we have had to change the traceability annotation at the binding site for x, since we have eliminated the box introduction on its argument. This constraint is imposed on us by the need to keep the GC information consistent. If we choose (perhaps because of limitations on the precision of the analysis, or perhaps because of other constraints) to only eliminate the innermost box/unbox pair, then we must similarly adjust the traceability annotation on the remaining box introduction (labeled with 3).

$$\texttt{let } f^0\texttt{:r} = (\lambda x^1\texttt{:b}.(\texttt{box}_\texttt{b}\, x^2)^3)^4 \texttt{ in}\,(\texttt{unbox}\,(f^5\ 3^6)^8)^9$$

Not all programs can be consistently rewritten in this manner. If we consider again the second example from Section 1, we see an example of a program in which we must forgo optimization if we wish to preserve GC safety.

$$\texttt{let } f^0\texttt{:r} = (\lambda x^1\texttt{:r}.x^2)^3 \texttt{ in}\,(\texttt{unbox}\,((f^4\ f^5)^6\ (\texttt{box}_\texttt{b}\, 3^7)^8)^9)^{10}$$

It is easy to see that any acceptable analysis must include the function labeled with 3 and the boxed term labeled with 8 in the set of terms reaching the binding site for x, labeled with 1. We might naively attempt to eliminate the box/unbox pair as follows:

$$\texttt{let } f^0\texttt{:r} = (\lambda x^1\texttt{:?}.x^2)^3 \texttt{ in}\,((f^4\ f^5)^6\ 3^7)^9$$

Unfortunately, there is no consistent choice of traceability annotation for the binding site for x. If we choose b as the traceability annotation then after reduction we arrive at a state that has no defined reduction:

$$((\langle\epsilon, \lambda x^1\texttt{:b}.x^2\rangle^3\ \langle\epsilon, \lambda x^1\texttt{:b}.x^2\rangle^3)^6\ 3^7)^9$$

The first application leads to undefined behavior, since the traceability of the argument value does not match the traceability annotation on the parameter variable. If we had instead chosen r as the traceability annotation, then one further reduction would still lead us to undefined behavior.

$$(\langle\epsilon, \lambda x^1\texttt{:r}.x^2\rangle^3\ 3^7)^9$$

The requirement to preserve GC information imposes two burdens on us then: we must provide some mechanism for assigning new GC meta-data when we optimize the program, and we must also ensure that we do not optimize the program in a way that does not admit a consistent assignment of such meta-data. In the rest of this section, we first develop a framework for specifying an unboxing assignment regardless of any correctness concerns, and then separately define a judgement specifying when such an assignment is a reasonable one.

4.1 The Unboxing Optimization

We can divide the problem of specifying an unboxing into two sub-parts: choosing the particular box/unbox pairs that are valid to eliminate and assigning new traceability annotations to terms that are affected. An unboxing then is a pair (T, Υ), where Υ is a set of labels, and T is a partial function from labels to traceabilities. The unboxed set Υ is the set of labels to be unboxed, and the traceability map T specifies new traceabilities for labels affected by the unboxing. The fact that T is a partial function is essential for several reasons. On a technical level, we do not require that labels be unique in a program. Consequently, it is possible that there is no consistent choice for a specific label. More importantly, requiring that T be a total function would put unsatisfiable requirements on the flow analysis. For example, a program that allocates a mis-tagged object after going into an infinite loop is GC safe according to our specification since the bad allocation is never reached. Requiring the analysis to find a consistent traceability map for such a program is equivalent to requiring it to solve the halting problem, since it must statically prove that the set of values dynamically reaching the mis-tagged allocation site is empty. By allowing T to be a partial function, we allow for necessary imprecision in the analysis. Also of importance is the need to allow for relative imprecision in the analysis. In order to achieve faster compile times, we may choose to use less precise analyses that potentially over-approximate the set of terms reaching a use point. Consequently, even if a consistent traceability assignment exists, we may not have sufficiently precise information to construct it.

An unboxing pair defines a total function mapping labeled terms to labeled terms, as shown in Figure 5. For notational convenience, we take $\mathrm{T}(i) = t$ as asserting that i is in the domain of T, and that its image is t. We also say that: $\mathrm{T}(i) \geq t$ if and only if $\mathrm{T}(i) = t$ or $\mathrm{T}(i)$ is undefined; and $\underline{\mathrm{T}}(i, t) = \mathrm{T}(i)$ if T defined at i, otherwise t.

An important observation about the unboxing optimization as we have defined it is unlike many previous interprocedural approaches (Section 6), it only improves programs and never introduces instructions or allocation. This is easy to see, since the unboxing function only removes boxes (which allocate and have an instruction cost), and unboxes (which have an instruction cost) and never introduces any new operations at all.

$\boxed{\downharpoonleft e \downharpoonright^{\mathrm{T}}_{\Upsilon}}$

$$
\begin{array}{lll}
\downharpoonleft x^i \downharpoonright^{\mathrm{T}}_{\Upsilon} & = x^i & \\
\downharpoonleft (\lambda x^j{:}t.e)^i \downharpoonright^{\mathrm{T}}_{\Upsilon} & = (\lambda x^j{:}\underline{\mathrm{T}}(j,t).\downharpoonleft e \downharpoonright^{\mathrm{T}}_{\Upsilon})^i & \\
\downharpoonleft (e_1\ e_2)^i \downharpoonright^{\mathrm{T}}_{\Upsilon} & = (\downharpoonleft e_1 \downharpoonright^{\mathrm{T}}_{\Upsilon}\ \downharpoonleft e_2 \downharpoonright^{\mathrm{T}}_{\Upsilon})^i & \\
\downharpoonleft (\mathtt{box}_t\, e)^i \downharpoonright^{\mathrm{T}}_{\Upsilon} & = \downharpoonleft e \downharpoonright^{\mathrm{T}}_{\Upsilon} & i \in \Upsilon \\
 & = (\mathtt{box}_{\underline{\mathrm{T}}(\mathrm{lbl}(e),t)}\ \downharpoonleft e \downharpoonright^{\mathrm{T}}_{\Upsilon})^i & i \notin \Upsilon \\
\downharpoonleft (\mathtt{unbox}\, e)^i \downharpoonright^{\mathrm{T}}_{\Upsilon} & = \downharpoonleft e \downharpoonright^{\mathrm{T}}_{\Upsilon} & \mathrm{lbl}(e) \in \Upsilon \\
 & = (\mathtt{unbox}\, \downharpoonleft e \downharpoonright^{\mathrm{T}}_{\Upsilon})^i & \mathrm{lbl}(e) \notin \Upsilon
\end{array}
$$

$$
\begin{array}{lll}
\downharpoonleft \rho(e)^i \downharpoonright^{\mathrm{T}}_{\Upsilon} & = \downharpoonleft \rho \downharpoonright^{\mathrm{T}}_{\Upsilon}(\downharpoonleft e \downharpoonright^{\mathrm{T}}_{\Upsilon})^i & \\
\downharpoonleft c^i \downharpoonright^{\mathrm{T}}_{\Upsilon} & = c^i & \\
\downharpoonleft \langle \rho, \lambda x^j{:}t.e \rangle^i \downharpoonright^{\mathrm{T}}_{\Upsilon} & = & \\
 & \langle \downharpoonleft \rho \downharpoonright^{\mathrm{T}}_{\Upsilon}, \lambda x^j{:}\underline{\mathrm{T}}(j,t).\downharpoonleft e \downharpoonright^{\mathrm{T}}_{\Upsilon} \rangle^i & \\
\downharpoonleft \langle v^j{:}t \rangle^i \downharpoonright^{\mathrm{T}}_{\Upsilon} & = \downharpoonleft v^j \downharpoonright^{\mathrm{T}}_{\Upsilon} & i \in \Upsilon \\
 & = \langle \downharpoonleft v^j \downharpoonright^{\mathrm{T}}_{\Upsilon}{:}\underline{\mathrm{T}}(j,t) \rangle^i & i \notin \Upsilon
\end{array}
$$

$\boxed{\downharpoonleft \rho \downharpoonright^{\mathrm{T}}_{\Upsilon}}$

$$
\downharpoonleft x_1^{i_1}{:}t_1 = v_1{}^{j_1}, \ldots, x_n^{i_n}{:}t_n = v_n{}^{j_n} \downharpoonright^{\mathrm{T}}_{\Upsilon} = x_1^{i_1}{:}\underline{\mathrm{T}}(i_1,t_1) = \downharpoonleft v_1{}^{j_1} \downharpoonright^{\mathrm{T}}_{\Upsilon}, \ldots, x_n^{i_n}{:}\underline{\mathrm{T}}(i_n,t_n) = \downharpoonleft v_n{}^{j_n} \downharpoonright^{\mathrm{T}}_{\Upsilon}
$$

$\boxed{\downharpoonleft M \downharpoonright^{\mathrm{T}}_{\Upsilon}}$

$$
\downharpoonleft (\rho, e) \downharpoonright^{\mathrm{T}}_{\Upsilon} = (\downharpoonleft \rho \downharpoonright^{\mathrm{T}}_{\Upsilon}, \downharpoonleft e \downharpoonright^{\mathrm{T}}_{\Upsilon})
$$

Fig. 5. Unboxing

4.2 Acceptable Unboxings

While any choice of (T, Υ) defines an unboxing, not every unboxing pair is reasonable in the sense that it defines a semantics preserving optimization. Just as we defined a notion of acceptable analysis in Section 3, we will define a judgement that captures sufficient conditions for ensuring correctness of an unboxing, without specifying a particular method of choosing such an unboxing. By using analyses of different precisions or choosing different optimization strategies we may end up with quite different choices of unboxings; however, so long as they satisfy our notion of acceptability we can be sure that they will preserve correctness.

Informally, we can eliminate a box introduction if certain criteria are met. Firstly, we must be able to eliminate all of the unbox operations that it reaches. Secondly, we must be able to find a consistent traceability assignment covering each intermediate variable or field that it reaches, given all of the rest of our unboxing choices. We can eliminate an unbox operation if we can eliminate all of the box operations that reach it. Finally, we must also impose coherence requirements on traceability assignments. For every variable whose binding-site label occurs in the domain of T, we require that its new traceability assignment agree with the traceability assignment of all of its reaching definitions. Similarly, for every box introduction (or value form) that is not itself unboxed, we require that the traceability assignment for its contents agree with the traceability assignment for every reaching definition in the flow analysis.

This informal description is made precise in Figure 6. We use the notation $i \overset{\mathrm{T},\Upsilon}{\simeq} j$ to indicate when an unboxing *agrees* at two labels i and j.

$$
\begin{array}{ll}
i \overset{\mathrm{T}}{\simeq} j & \text{iff either } \mathrm{T}(i) = \mathrm{T}(j) \text{ (both defined) or } \mathrm{T}(i) \text{ and } \mathrm{T}(j) \text{ undefined} \\
i \overset{\Upsilon}{\simeq} j & \text{iff either } i, j \in \Upsilon \text{ or } i, j \notin \Upsilon \\
i \overset{\mathrm{T},\Upsilon}{\simeq} j & \text{iff } i \overset{\mathrm{T}}{\simeq} j \text{ and } i \overset{\Upsilon}{\simeq} j
\end{array}
$$

An unboxing pair (T, Υ) is acceptable relative to an analysis (C, ϱ) for a program M (judgement $\mathrm{C} \vdash M \downharpoonleft\!\downharpoonright (\mathrm{T}, \Upsilon)$) if the unboxing is *cache consistent* (judgement $\mathrm{C} \vdash (\mathrm{T}, \Upsilon)$), and *consistent* (judgement $\mathrm{T}, \Upsilon \vdash M$).

$$\boxed{\mathrm{T}, \Upsilon \vdash e}$$

$$\frac{}{\mathrm{T}, \Upsilon \vdash x^i} \quad \frac{\mathrm{T}, \Upsilon \vdash e \quad \mathrm{T}(j) \geq \mathtt{r}}{\mathrm{T}, \Upsilon \vdash (\lambda x^i{:}t.e)^j} \quad \frac{\mathrm{T}, \Upsilon \vdash e_1 \quad \mathrm{T}, \Upsilon \vdash e_2}{\mathrm{T}, \Upsilon \vdash (e_1\ e_2)^i} \quad \frac{i \in \Upsilon \quad \mathrm{T}(i) = \mathrm{T}(\mathrm{lbl}(e)) \quad \mathrm{T}, \Upsilon \vdash e}{\mathrm{T}, \Upsilon \vdash (\mathtt{box}_t\, e)^i} \quad \frac{i \notin \Upsilon \quad \mathrm{T}(i) \geq \mathtt{r} \quad \mathrm{T}, \Upsilon \vdash e}{\mathrm{T}, \Upsilon \vdash (\mathtt{box}_t\, e)^i}$$

$$\frac{\mathrm{T}, \Upsilon \vdash e}{\mathrm{T}, \Upsilon \vdash (\mathtt{unbox}\, e)^i} \quad \frac{\mathrm{T}, \Upsilon \vdash \rho \quad \mathrm{T}, \Upsilon \vdash e}{\mathrm{T}, \Upsilon \vdash \rho(e)^i} \quad \frac{\mathrm{T}(i) \geq \mathtt{b}}{\mathrm{T}, \Upsilon \vdash c^i} \quad \frac{\mathrm{T}, \Upsilon \vdash \rho \quad \mathrm{T}, \Upsilon \vdash e \quad \mathrm{T}(j) \geq \mathtt{r}}{\mathrm{T}, \Upsilon \vdash \langle \rho, \lambda x^i{:}t.e \rangle^j}$$

$$\frac{j \in \Upsilon \quad \mathrm{T}(j) = \mathrm{T}(i) \quad \mathrm{T}, \Upsilon \vdash v^i}{\mathrm{T}, \Upsilon \vdash \langle v^i{:}t \rangle^j} \quad \frac{j \notin \Upsilon \quad \mathrm{T}(j) \geq r \quad \mathrm{T}, \Upsilon \vdash v^i}{\mathrm{T}, \Upsilon \vdash \langle v^i{:}t \rangle^j}$$

$$\boxed{\mathrm{T}, \Upsilon \vdash \rho} \quad \frac{\forall 1 \leq k \leq n : \underline{\mathrm{T}}(i_k, t_k) = \underline{\mathrm{T}}(j_k, t_k) \wedge \mathrm{T}, \Upsilon \vdash {v_k}^{j_k}}{\mathrm{T}, \Upsilon \vdash x_1^{i_1}{:}t_1 = {v_1}^{j_1}, \ldots, x_n^{i_n}{:}t_n = {v_n}^{j_n}}$$

$$\boxed{\mathrm{T}, \Upsilon \vdash M} \quad \frac{\mathrm{T}, \Upsilon \vdash \rho \quad \mathrm{T}, \Upsilon \vdash e}{\mathrm{T}, \Upsilon \vdash (\rho, e)} \qquad \boxed{\mathrm{C} \vdash (\mathrm{T}, \Upsilon)} \quad \frac{\forall i : s \in \mathrm{C}(i) \implies i \overset{\mathrm{T}, \Upsilon}{\simeq} \mathrm{lbl}(s)}{\mathrm{C} \vdash (\mathrm{T}, \Upsilon)}$$

$$\boxed{\mathrm{C} \vdash M \downharpoonleft\!\downharpoonright (\mathrm{T}, \Upsilon)} \quad \frac{\mathrm{C} \vdash (\mathrm{T}, \Upsilon) \quad \mathrm{T}, \Upsilon \vdash M}{\mathrm{C} \vdash M \downharpoonleft\!\downharpoonright (\mathrm{T}, \Upsilon)}$$

Fig. 6. Consistent and acceptable unboxing

Cache consistency $\mathrm{C} \vdash (\mathrm{T}, \Upsilon)$ encapsulates the requirement that an unbox can only be eliminated if all of the reaching definitions of its target are unboxed. It requires agreement between the label of the target of the unbox and the labels of everything in the cache of the target. The results from Section 3 ensure that under any evaluation, any term reaching the unbox is in the cache of the original target label, and hence that the unboxing approximation takes into account a sufficient set of terms[2].

Cache consistency does not put any constraints on the actual choice of traceabilities in T. The consistency judgement $(\mathrm{T}, \Upsilon \vdash M)$ ensures that the traceability map T encodes choices that are compatible with the actual labeled terms in M, given a particular choice of terms to unbox Υ.

[2] See the cache refinement lemma in the extended technical report [7] for more detail.

For environments, the consistency judgement insists that the traceability map assign consistent traceabilities to values and the variables to which they are bound. In this way we can ensure that the result of unboxing an environment still provides good traceability information for the garbage collector.

The term consistency judgement for the most part only requires that the traceability map be consistent with the labeled values. Variable uses incur no constraints, and neither do applications nor unbox operations (beyond requiring the consistency of their sub-terms). For constants c^i, we require that the traceability assignment T, if defined at i, maps i to `b`. That is, we require that the traceability assignment for i is consistent with the actual term inhabiting the label. Functions have a similar requirement: the traceability assignment for their label, if present, must be `r` since functions are represented by heap allocated closures. In the value form the closed over environment must be consistent as well.

The only particularly interesting rules are those covering the boxed introduction form and isomorphically the boxed value form. There are two cases: one for when the boxed value is selected for unboxing (that is, its label is in Υ), and one for when it has not been selected for unboxing.

If the term is not to be unboxed ($j \notin \Upsilon$), then the consistency rule requires that its traceability assignment (if any) be `r`. In the case that the term is to be unboxed ($j \in \Upsilon$) this is not required since the unboxed value may not in fact end up having traceability `r`. Instead, we require that the traceability map have an assignment both for the label of the box (j), and for the label of the contents of the box (i), and that it assign the same traceability to both. This requirement may be somewhat unexpected at first. The intuition behind it is that the end result of unboxing will replace the outer box by the inner boxed value; therefore we wish to treat the boxed value as having the same traceability as its contents.

Our goal is to show that the unboxing function induced by any acceptable unboxing is in some sense correct as an optimization. The first part of this is to show that unboxing preserves traceability.

Theorem 1 (Consistent unboxings preserve traceability)

- *If* $\mathrm{T}, \Upsilon \vdash v^i$ *and* $\vdash_{\mathrm{v}} v^i{:}t$ *then* $\vdash_{\mathrm{v}} \lfloor v^i \rfloor_{\Upsilon}^{\mathrm{T}}{:}\underline{\mathrm{T}}(i,t)$.
- *If* $\mathrm{T}, \Upsilon \vdash e$ *and* $\vdash e\ \mathbf{tr}$ *then* $\vdash \lfloor e \rfloor_{\Upsilon}^{\mathrm{T}}\ \mathbf{tr}$.
- *If* $\mathrm{T}, \Upsilon \vdash \rho$ *and* $\vdash \rho\ \mathbf{tr}$ *then* $\vdash \lfloor \rho \rfloor_{\Upsilon}^{\mathrm{T}}\ \mathbf{tr}$.
- *If* $\mathrm{T}, \Upsilon \vdash M$ *and* $\vdash M\ \mathbf{tr}$ *then* $\vdash \lfloor M \rfloor_{\Upsilon}^{\mathrm{T}}\ \mathbf{tr}$.

Theorem 1 tells us that if we have a traceable program, then the result of unboxing it is still traceable. The second step to showing correctness is to show that unboxing does not introduce new undefined behavior.

Theorem 2 (Coherence)

- *If* $\mathrm{C}; \varrho \vdash M$, $\mathrm{C} \vdash M \downharpoonright\downharpoonleft (\mathrm{T}, \Upsilon)$, *and* $M \longmapsto^* (\rho, v^i)$ *then* $\lfloor M \rfloor_{\Upsilon}^{\mathrm{T}} \longmapsto^* (\lfloor \rho \rfloor_{\Upsilon}^{\mathrm{T}}, \lfloor v^i \rfloor_{\Upsilon}^{\mathrm{T}})$.
- *If* $\mathrm{C}; \varrho \vdash M$, $\mathrm{C} \vdash M \downharpoonright\downharpoonleft (\mathrm{T}, \Upsilon)$, *and* $M \longmapsto \cdots$ *then* $\lfloor M \rfloor_{\Upsilon}^{\mathrm{T}} \longmapsto \cdots$.

Theorem 2 shows that if two terms are related by reduction, then their images under the unboxing function are also related by the many step reduction relation given that the unboxing pair is acceptable; and that if a term diverges under reduction, then its image under the unboxing function also diverges. In other words, for an acceptable analysis and an acceptable unboxing, the induced unboxing function preserves the semantics of the original program up to elimination of boxes. Since the semantics of the core language only defines reduction steps that preserve GC safety, this theorem implies that the image of a GC safe program under unboxing is also GC safe.

5 Construction of an Acceptable Unboxing

The previous section gives a declarative specification for when an unboxing pair (T, Υ) is correct but does not specify how such a pair might be produced. In this section we give a simple algorithm for constructing an acceptable unboxing given an arbitrary acceptable flow analysis.

The idea behind the algorithm is that given a program and an acceptable flow analysis for it, we use the results of the flow analysis to construct the connected components of the inter-procedural flow graph of the program. Each connected component initially defines its own equivalence class. For each equivalence class, we then compute the least upper bound of the traceabilities of all of the introduction forms of all of the elements of the component except the box introductions. Box introductions are left initially unconstrained, since we intend to eliminate them. If the least upper bound is well-defined, then the equivalence class can potentially be eliminated. We then consider each box introduction in turn and attempt to eliminate it by combining the respective equivalence classes of the box and its contents. This is possible whenever doing so will not over-constrain the resulting combined equivalence class. When all possible boxes have been eliminated, the algorithm terminates. In the rest of the section, we make this informal algorithm concrete and show that the choice of unboxing that it produces is in fact acceptable.

For the purposes of this section we ignore environments and the intermediate forms $\rho(e)$, $\langle \rho, \lambda x^i{:}t.e\rangle^j$ and $\langle v^i{:}t\rangle^j$. These constructs are present in the language solely as mechanisms to discuss the dynamic semantics—in this sense they can be thought of as intermediate terms, rather than source terms. It is straightforward to incorporate these into the algorithm if desired.

Given a flow analysis (C, ϱ), we define the induced undirected flow graph $\mathcal{FG}$ as an undirected graph with a node for every label in C, and edges as follows.

- For every label i and every shape $s \in \mathrm{C}(i)$, we add an edge between i and $\mathrm{lbl}(s)$.
- For every box introduction in the program $(\mathtt{box}_t\, e)^i$ and every shape in the cache $(\mathtt{box}_t\, j)^i \in \mathrm{C}(i)$ we add an edge between $\mathrm{lbl}(e)$ and j.

The first set of edges simply connects up each program point with all of its reaching definitions. The second set of edges is added to simplify the proofs in

the pathological case that e has no reaching definitions (and hence the box itself is dynamically dead and unreachable): in the usual case where values reach e then the definition of an acceptable analysis implies that these edges are already present.

We define equivalence classes of labels as disjoint sets of labels in the usual way. The function $\mathcal{EC}$ maps labels i to the disjoint set containing i. We extend traceabilities t to a complete flat lattice $\hat{t}$ with a top element $\top$, a bottom element $\bot$ and the usual least upper bound function on $\hat{t}$.

$$\begin{array}{cccc} \top \sqcup t = \top & \bot \sqcup t = t & & \mathtt{b} \sqcup \mathtt{r} = \top \\ t \sqcup \top = \top & t \sqcup \bot = t & t \sqcup t = t & \mathtt{r} \sqcup \mathtt{b} = \top \end{array}$$

We initialize the mapping $\mathcal{EC}$ by finding the connected components of the induced undirected flow-graph $\mathcal{FG}$, and initializing $\mathcal{EC}(i)$ with the connected component containing i. As the algorithm proceeds, two equivalence classes may be collapsed into a single equivalence class requiring an updated mapping $\mathcal{EC}$.

We maintain a set of equivalence classes $\mathcal{CN}$ consisting of current candidates for unboxing. When equivalence classes are collapsed, the elements of $\mathcal{CN}$ are adjusted appropriately, as will be shown.

We maintain a set of labels Υ, which is an unboxing set in the sense of Section 4. The set Υ at all times contains all of the labels already selected for unboxing, and is initially empty.

We maintain an extended traceability map $\mathcal{T}$ that maps equivalence classes to extended traceabilities $\hat{t}$. For notational convenience we define $\mathcal{T}_{\mathcal{EC}}$ to be the derived function mapping labels to the extended traceabilities of their equivalence classes: $\mathcal{T}_{\mathcal{EC}}(k) = \mathcal{T}(\mathcal{EC}(k))$. The derivation of a standard traceability map T from an extended traceability map $\mathcal{T}$ is then given as follows.

$$\begin{array}{ll} \mathrm{T}(k) = \mathcal{T}_{\mathcal{EC}}(k) & \bot < \mathcal{T}_{\mathcal{EC}}(k) < \top \\ \mathrm{T}(k) = r & \mathcal{T}_{\mathcal{EC}}(k) = \bot \\ \mathrm{T}(k) = \text{undefined} & \mathcal{T}_{\mathcal{EC}}(k) = \top \end{array}$$

The general idea is that an equivalence class is mapped by the $\mathcal{T}$ function to the least upper bound of the traceabilities of all of the reaching definitions of all of the labels in the equivalence class. An equivalence class containing no reaching definitions will be unconstrained – for technical reasons we choose an arbitrary traceability ($\mathtt{r}$) for such classes. An equivalence class containing definitions with inconsistent traceabilities will have no defined traceability in the induced mapping.

During the algorithm traceability constraints imposed by box introductions in the candidate set are left out of the initial mapping and hence must be added back in before computing the induced traceability map. We write $\mathcal{T}^{\mathcal{CN}}$ for the extended traceability map obtained by adding in the delayed constraints for each equivalence class, and $\mathcal{T}^{\mathcal{CN}}_{\mathcal{EC}}$ for the extension of this to individual labels given by $\mathcal{T}^{\mathcal{CN}}_{\mathcal{EC}}(i) = \mathcal{T}^{\mathcal{CN}}(\mathcal{EC}(i))$.

$$\begin{array}{ll} \mathcal{T}^{\mathcal{CN}}(\mathcal{EC}(k)) = \mathcal{T}(\mathcal{EC}(k)) & \text{if } \mathcal{EC}(k) \notin \mathcal{CN} \\ \mathcal{T}^{\mathcal{CN}}(\mathcal{EC}(k)) = \mathcal{T}(\mathcal{EC}(k)) \sqcup \mathtt{r} & \text{if } \mathcal{EC}(k) \in \mathcal{CN} \end{array}$$

Note that by definition, if labels i and j are in the same equivalence class ($\mathcal{EC}(i) = \mathcal{EC}(j)$), then the traceability map T induced by an extended traceability map $\mathcal{T}$ agrees on i and j.

We define the immediate extended traceability of a labeled term $itr(e)$ as follows.

$$\begin{array}{llll} itr(c^i) & = b & itr((\texttt{box}_t\, e)^i) & = r \\ itr((\lambda x^i{:}t.e)^j) & = r & itr(e) & = \bot \text{ otherwise} \end{array}$$

The algorithm starts with an empty unboxing set Υ. The candidate set $\mathcal{CN}$ is initialized by including $\mathcal{EC}(i)$ for each $(\texttt{box}_t\, e)^i$ in the program. The extended traceability map is initialized by setting for each equivalence class S:

$$\mathcal{T}(S) = \bigsqcup_{i \in S,\ s \in \mathrm{C}(i),\ s \neq (\texttt{box}_t\, k)^j} itr(s)$$

That is, we take the extended traceability associated with the equivalence class S to be least upper bound of the immediate traceabilities of all of the elements of the cache of all the labels in the equivalence class, except those which are box introductions. The practical effect of this is to make the extended traceability of every label be the least upper bound of the traceability of every introduction form in its connected component (again, excepting boxes). An equivalence class that is unconstrained ($\bot$) either counts only box introductions among its definitions, or contains no definitions at all and hence is uninhabited (this can arise because of unreachable code).

The result of the initialization phase is a $(\mathcal{T}, \Upsilon, \mathcal{CN}, \mathcal{EC})$ quadruple, which induces an unboxing pair (T, Υ) where $\mathrm{T} = \mathcal{T}_{\mathcal{EC}}^{\mathcal{CN}}$. It can be shown that the unboxing pair induced in this manner is acceptable.

Lemma 3 (The initial unboxing is acceptable). *If* $\mathrm{C}; \varrho \vdash e$ *then the unboxing quadruple* $(\mathcal{T}, \Upsilon, \mathcal{CN}, \mathcal{EC})$ *computed by the algorithm in this section induces an unboxing* (T, Υ) *(where* $\mathrm{T} = \mathcal{T}_{\mathcal{EC}}^{\mathcal{CN}}$*) such that* $\mathrm{C} \vdash (\mathrm{T}, \Upsilon)$ *and* $\mathrm{T}, \Upsilon \vdash e$.

The unboxing pair created by the initial phase is acceptable, but does no unboxing. The second phase of the algorithm proceeds by incrementally moving equivalence classes from the candidate set $\mathcal{CN}$ to the unboxing set Υ, while maintaining the invariant that at every step $(\mathcal{T}, \Upsilon, \mathcal{CN}, \mathcal{EC})$ define an acceptable (and increasingly useful) unboxing. Equivalence classes of boxes that get chosen for unboxing are collapsed into the same equivalence class as the contents of the box. We use the notation $\mathcal{EC}' = \cup_{\underline{i,j}} \mathcal{EC}$ to stand for combining the equivalence classes for i and j to get a new equivalence class in the usual way.

For the unboxing steps, we consider in turn each $(\texttt{box}_t\, e)^i$ in the program. Let T be the traceability map induced by $\mathcal{T}$. The principal selection criterium for choosing which things to unbox is that $\mathcal{T}_{\mathcal{EC}}(i) \sqcup \mathcal{T}_{\mathcal{EC}}^{\mathcal{CN}}(\mathrm{lbl}(e)) < \top$. The idea is that under the assumption that no further unboxing is done, combining the equivalence classes for i and $\mathrm{lbl}(e)$ will not over-constrain the resulting equivalence class, and hence that the final induced traceability map will be well-defined at i and $\mathrm{lbl}(e)$. The extended traceability $\mathcal{T}_{\mathcal{EC}}(i)$ is the extended traceability associated with i under the assumption that $\mathcal{EC}(i)$ is unboxed, while $\mathcal{T}_{\mathcal{EC}}^{\mathcal{CN}}(\mathrm{lbl}(e))$

is the extended traceability associated with $\text{lbl}(e)$ under the assumption that no further unboxing is done. If i is either unconstrained, or constrained to something compatible with $\text{lbl}(e)$, then it is safe to unbox it.

Formally, if we have that $\mathcal{EC}(i) \in \mathcal{CN}$, $\mathcal{EC}(\text{lbl}(e)) \neq \mathcal{EC}(i)$, and $\mathcal{T}_{\mathcal{EC}}(i) \sqcup \mathcal{T}_{\mathcal{EC}}^{\mathcal{CN}}(\text{lbl}(e)) < \top$ then we select i for elimination. We then take the new unboxing to be the updated quadruple $(\mathcal{T}', \Upsilon', \mathcal{CN}', \mathcal{EC}')$ where:

$$\begin{array}{lll}
\mathcal{EC}' & = \overline{\cup_{\text{lbl}(e),i}\mathcal{EC}} & \\
\mathcal{CN}' & = (\mathcal{CN} - \{\mathcal{EC}(\text{lbl}(e)), \mathcal{EC}(i)\}) \cup \{\mathcal{EC}'(i)\} & \text{if } \mathcal{EC}(\text{lbl}(e)) \in \mathcal{CN} \\
 & = (\mathcal{CN} - \{\mathcal{EC}(\text{lbl}(e)), \mathcal{EC}(i)\}) & \text{if } \mathcal{EC}(\text{lbl}(e)) \notin \mathcal{CN} \\
\Upsilon' & = \Upsilon \cup \{\mathcal{EC}(i)\} & \\
\mathcal{T}'(s) & = \mathcal{T}_{\mathcal{EC}}(i) \sqcup \mathcal{T}_{\mathcal{EC}}(\text{lbl}(e)) & \text{if } s = \mathcal{EC}'(i) \\
 & = \mathcal{T}(s) & \text{otherwise}
\end{array}$$

For $\mathcal{EC}'$, we repartition the graph so that the equivalence classes for the box and its contents are combined into a single equivalence class. We remove the two original equivalence classes from the candidate set, and if the contents of the box was a candidate for unboxing we add back in the new equivalence class, which is the union of the two original classes. All of the elements of the original equivalence class of the box introduction are added to the unbox set. The extended traceability map is updated to map the new equivalence class (including both i and $\text{lbl}(e)$) to the extended traceability of the contents of the box.

If the conditions for unboxing i are not satisfied, then we take $\mathcal{CN}' = \mathcal{CN} - \{\mathcal{EC}(i)\}$ and take $\mathcal{T}'(\mathcal{EC}(i)) = \mathcal{T}(\mathcal{EC}(i)) \sqcup r$ and leave the rest of the data structures unchanged. Since we only consider each box introduction in the program once, the algorithm terminates.

Lemma 4 (Unboxing steps preserve acceptability). *If* $(\mathcal{T}, \Upsilon, \mathcal{CN}, \mathcal{EC})$ *define an acceptable unboxing as constructed by the initial phase of the algorithm and maintained by the unboxing phase, then the* $(\mathcal{T}', \Upsilon', \mathcal{CN}', \mathcal{EC}')$ *quadruple produced by a single step of the algorithm above also define an acceptable unboxing.*

Lemma 4 states that each step of the unboxing phase of the algorithm preserves the property that the induced unboxing pair is acceptable. Consequently, the algorithm terminates with an acceptable unboxing.

Theorem 3 (The algorithm produces an acceptable unboxing). *If* $\mathrm{C}; \varrho \vdash e$ *then the algorithm defined in this section produces a quadruple* $(\mathcal{T}, \Upsilon, \mathcal{CN}, \mathcal{EC})$ *such that* $\mathrm{C} \vdash e \downharpoonright\!\downharpoonleft (\mathrm{T}, \Upsilon)$ *where* $\mathrm{T} = \mathcal{T}_{\mathcal{EC}}^{\mathcal{CN}}$*.*

This construction demonstrates that the specification defined in Section 4 is a useful one in the sense that it is satisfiable. While the algorithm defined here is unlikely to be optimal, it has proved very effective in our compiler: on floating-point intensive benchmarks we have measured an order of magnitude reduction in allocation and substantial performance and parallel scalability gains.

6 Related Work

This paper provides a modular approach to showing correctness of a realistic compiler optimization that rewrites the structure of program data structures in significant ways. Our approach uses an arbitrary inter-procedural reaching definitions analysis to eliminate unnecessary heap allocation in an intermediate representation in which object representation has been made explicit. Our optimization can be staged freely with other optimizations. Unlike any previous work that we are aware of, we account for correctness with respect to the metadata requirements of the garbage collector. For presentational purposes, we have restricted our attention to the core concern of GC safety, but additional issues such as value size, dynamic type tests, etc. are straightforward to incorporate.

There has been substantial previous work addressing the problem of unboxing. Peyton Jones [3] introduced an explicit distinction between boxed and unboxed objects to provide a linguistic account of unboxing, and hence to allow a high-level compiler to locally eliminate unboxes of syntactically apparent box introduction operations. Leroy [4] defined a type-driven approach to adding coercions into and out of specialized representations. The type driven translation represented monomorphic objects natively (unboxed, in our terminology), and then introduced wrappers to coerce polymorphic uses into an appropriate form. To a first-order approximation, instead of boxing at definition sites this approach boxes objects at polymorphic use sites. This style of approach has the problem that it is not necessarily beneficial, since allocation is introduced in places where it would not otherwise be present. This is reflected in the slowdowns observed on some benchmarks described in the original paper. This approach also has the potential to introduce space leaks. In a later paper [5] Leroy argued that a simple untyped approach gives better and more predictable results.

Henglein and Jørgensen [2] defined a formal notion of optimality for local unboxings and gave two different choices of coercion placements that satisfy their notion of optimality. Their definition of optimality explicitly does not correspond in any way to reduced allocation or reduced instruction count and does not seem to provide uniform improvement over Leroy's approach.

The MLton compiler [10] largely avoids the issue of a uniform object representation by completely monomorphizing programs before compilation. This approach requires whole-program compilation. More limited monomorphization schemes could be considered in an incremental compilation setting. Monomorphization does not eliminate the need for boxing in the presence of dynamic type tests or reflection. Just in time compilers (e.g. for .NET) may monomorphize dynamically at runtime.

The TIL compiler [1,9] uses intensional type analysis in a whole-program compiler to allow native data representations without committing to whole-program compilation. As with the Leroy coercion approach, polymorphic uses of objects require conditionals and boxing coercions to be inserted at use sites, and consequently there is the potential to slow down, rather than speed up, the program.

Serrano and Feeley [8] described a flow analysis for performing unboxing substantially similar in spirit to our approach. Their algorithm attempts to find

a monomorphic typing for a program in which object representations have not been made explicit, which they then use selectively to choose whether to use a uniform or non-uniform representation for each particular object. Their approach differs in that they define a dedicated analysis rather than using a generic reaching definitions analysis. They assume a conservative garbage collector and hence do not need to account for the requirements of GC safety, and they do not prove a correctness result.

References

1. Harper, R., Morrisett, G.: Compiling polymorphism using intensional type analysis. In: Twenty-Second ACM Symposium on Principles of Programming Languages, San Francisco, CA, pp. 130–141 (January 1995)
2. Henglein, F., Jørgensen, J.: Formally optimal boxing. In: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1994, pp. 213–226. ACM, New York (1994)
3. Jones, S.L.P., Launchbury, J.: Unboxed Values as First Class Citizens in a Non-Strict Functional Language. In: Hughes, J. (ed.) FPCA 1991. LNCS, vol. 523, pp. 636–666. Springer, Heidelberg (1991)
4. Leroy, X.: Unboxed objects and polymorphic typing. In: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1992, pp. 177–188. ACM, New York (1992)
5. Leroy, X.: The effectiveness of type-based unboxing. Tech. rep., Boston College, Computer Science Department (1997)
6. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus (1999)
7. Petersen, L., Glew, N.: GC-safe interprocedural unboxing: Extended version (2012), http://leafpetersen.com/leaf/papers.html
8. Serrano, M., Feeley, M.: Storage use analysis and its applications. In: Proceedings of the First ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, pp. 50–61. ACM, New York (1996)
9. Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., Lee, P.: Til: a type-directed, optimizing compiler for ml. SIGPLAN Not. 39, 554–567 (2004)
10. Weeks, S.: Whole-program compilation in MLton. In: Proceedings of the 2006 Workshop on ML, ML 2006, p. 1. ACM, New York (2006)

Compiler Support for Value-Based Indirect Branch Prediction*

Muhammad Umar Farooq[1], Lei Chen[2], and Lizy Kurian John[1]

[1] Department of ECE, The University of Texas at Austin
ufarooq@utexas.edu,ljohn@ece.utexas.edu
[2] Intel Architecture Group, Intel Corporation
lei777@gmail.com

Abstract. Indirect branch targets are hard to predict as there may be multiple targets corresponding to a single indirect branch instruction. *Value Based BTB Indexing (VBBI)*, a recently proposed indirect branch prediction technique, utilizes the compiler to identify a 'hint instruction', whose output value strongly correlates with the target address of an indirect branch. At run time, multiple targets are stored at different branch target buffer (BTB) locations indexed using the branch PC and the hint instruction output value.

In this paper, we present compiler support for the VBBI prediction scheme. We also propose compiler and run time optimizations to increase the dynamic instruction count between the indirect branch and its corresponding hint instruction. The more the dynamic instructions between the hint-jump instruction pair, the more likely that the hint value will be available when making the prediction.

Our evaluation shows that the proposed compiler and run time optimizations improve the VBBI prediction accuracy from 66% to 80%. This translates into performance improvement from 17.2% (baseline VBBI) to 24.8% (optimized VBBI) over the traditional BTB design and from 11% (baseline VBBI) to 17.3% (optimized VBBI) over the best previously proposed indirect branch prediction scheme.

Keywords: branch prediction, indirect branches, compiler guided branch prediction, compiler optimizations, compiler-microarchitecture interaction.

1 Introduction

Several high level programming language constructs such as virtual function calls, switch-case statements, function pointers are implemented using indirect branches. With object oriented programming languages gaining more popularity in various computing arenas, indirect branches will become more prevalent in future applications. As a result, whether or not the indirect branches can be

* This research was partially supported by NSF grant 1117895. The opinions and views expressed in this paper are those of the authors and not those of NSF.

M. O'Boyle (Ed.): CC 2012, LNCS 7210, pp. 185–199, 2012.
© Springer-Verlag Berlin Heidelberg 2012

accurately predicted will be a limiting factor of the overall system performance. This trend is recognized by commercial microprocessor manufacturers including Intel, whose recent processor includes a dedicated indirect branch predictor [8]. Figure 1 shows the mispredictions per 1K instructions (MPKI) for different applications using different indirect branch prediction schemes. On average, indirect branch mispredictions account for 38%, 31% and 22% of the overall mispredictions, using the branch target buffer (BTB) [13], the tagged target cache (TTC) [2] and the value-based BTB indexing (VBBI) [7] designs respectively.

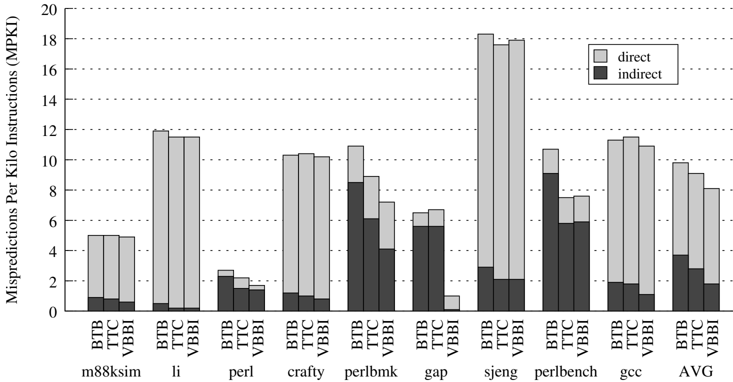


Fig. 1. MPKI for BTB, TTC and baseline VBBI prediction schemes

Prior research on indirect branch prediction has mainly focused on history-based target prediction schemes [2,4,5,6,11,12]. In these schemes, branch history information is used to distinguish different dynamic instances of the same indirect branch. These purely dynamic schemes have the advantage of not requiring compiler support and invisible to the software. However, hardware has limited view of program execution and may not be able to capture certain program behavior with reasonable cost.

The recently proposed VBBI scheme [7] shows that by tracing back the indirect branch data dependence chain, an instruction can be found whose output is directly related to the target taken by the indirect branch. This correlated instruction is referred to as the *hint instruction*, and its output as *hint value*. The key idea of VBBI is to store multiple targets of an indirect branch at different BTB indices computed by hashing the branch PC with the hint value.

Previous work on VBBI presented performance improvements without details on compiler implementation [7]. In this paper, we propose compiler support for the VBBI prediction scheme. For every static indirect branch instruction, the compiler analyzes the source code to find the 'most recent definition' of the variable on which the indirect branch is dependent. During code generation this information is encoded in the indirect branch to be used at run time. In order to maintain strong correlation between the target and the hint value, the current hint value should be used for making the prediction, i.e. the hint instruction should have finished its execution before the indirect branch is fetched. To this

end, we propose the compiler and run time optimizations for improving the VBBI prediction accuracy by increasing the dynamic instruction count between the hint instruction and the corresponding indirect jump instruction.

We show the performance improvement from these optimizations and compare with the traditional BTB design and the tagged target cache (TTC) design [2]. Our evaluation shows that the proposed compiler and run time optimizations improve the VBBI prediction accuracy from 66% (baseline VBBI) to 80% (optimized VBBI). In terms of performance, the optimized VBBI improves the performance from 17.2% (baseline VBBI) to 24.8% (optimized VBBI) over the traditional BTB design and from 11% (baseline VBBI) to 17.3% (optimized VBBI) over the TTC design.

This paper makes the following contributions:

1. We added compiler support for a recently proposed indirect branch prediction technique, the VBBI scheme. The implementation is based on GCC v4.2.1.
2. We propose compiler and run time optimizations to improve the VBBI prediction accuracy. These optimizations are applicable to similar schemes, as well as other design ideas exploiting data dependences and improving memory operations.

Rest of the paper is organized as follows. Section 2 gives the VBBI background. Compiler analysis for the VBBI scheme is introduced in section 3. Section 4 presents compiler and run time optimizations. Our simulation methodology is outlined in section 5. We discuss our results in section 6. Section 7 presents the related work and we conclude the paper in section 8.

2 Value Based BTB Indexing (VBBI) Background

The VBBI prediction scheme relies on the compiler to identify a *'hint instruction'* whose output value strongly correlates with the target taken by the indirect jump instruction. Dynamically, multiple targets are stored at different BTB locations indexed using the jump PC and the hint instruction output value. When a hint instruction is executed, its output value is stored in a buffer. Subsequently, when the corresponding jump instruction is fetched, it reads the hint value and uses it to compute the BTB index. When the branch commits, the BTB is updated with the correct target (if different from the predicted target) using the same index.

Figure 2 shows the overall operation of the VBBI prediction scheme. Each entry in the *Hint Instruction Buffer (HIB)* has 3 fields, branch PC (jmp_pc), corresponding hint instruction PC (hint_pc), and the hint value. HIB is accessed in fetch and write-back (WB) stages. In the fetch stage, indirect jump instructions read the hint value from the HIB to compute the BTB index, while other instructions access HIB to see if they are the hint instruction for an indirect jump instruction. In the write-back stage, hint instructions write their output value into the HIB.

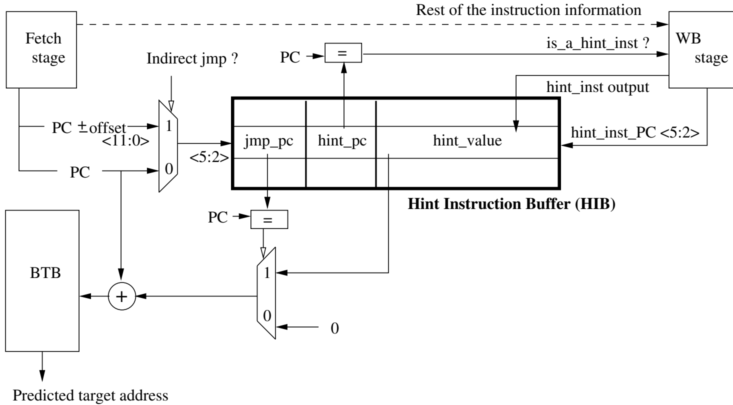


Fig. 2. VBBI Hardware Design (from [7])

Target Prediction Overriding. VBBI scheme will be more accurate, if the prediction is made using the *current* output of the hint instruction. In cases where the jump instruction is fetched before the hint instruction has produced its output, the jump instruction will use *stale* hint value for making the prediction. When the latest hint value becomes available, another prediction is made using the updated hint value, if it is different from the old value. This prediction will override the initial prediction and redirect the fetch to the correct path, cycles before the jump resolution.

3 Compiler Analysis for VBBI

VBBI prediction scheme relies on compiler to identify the 'most recent definition' of the variable on which an indirect jump instruction depends on. This variable can be a switch-case control variable, a pointer to a function or an object, etc. During code generation, *offset* of the jump instruction from the instruction holding the 'most recent definition' (i.e. the hint instruction) is encoded in the indirect jump instruction. Since an indirect jump instruction specifies its target using an architectural register instead of an absolute address, some of the bits in the instruction encoding are unused and are available for providing hints [3].

Implementation Details. We modified GCC v4.2.1 to support hint instruction identification for the VBBI scheme. Figure 3 explains the modified passes using a hand written example code. In Figure 3(a), the switch-case statement will be compiled into an indirect jump instruction. The target taken by the jump depends on the definition of the underlined variable **p**.

1. **pass_uncprop:** This pass is executed before coming out of the SSA form (in pass_del_ssa). While the code is still in the SSA form, we modified this pass to identify the 'last definition' of the variable (i.e., the hint instruction), on which the indirect jump is dependent on. Figure 3(b) shows that in the SSA

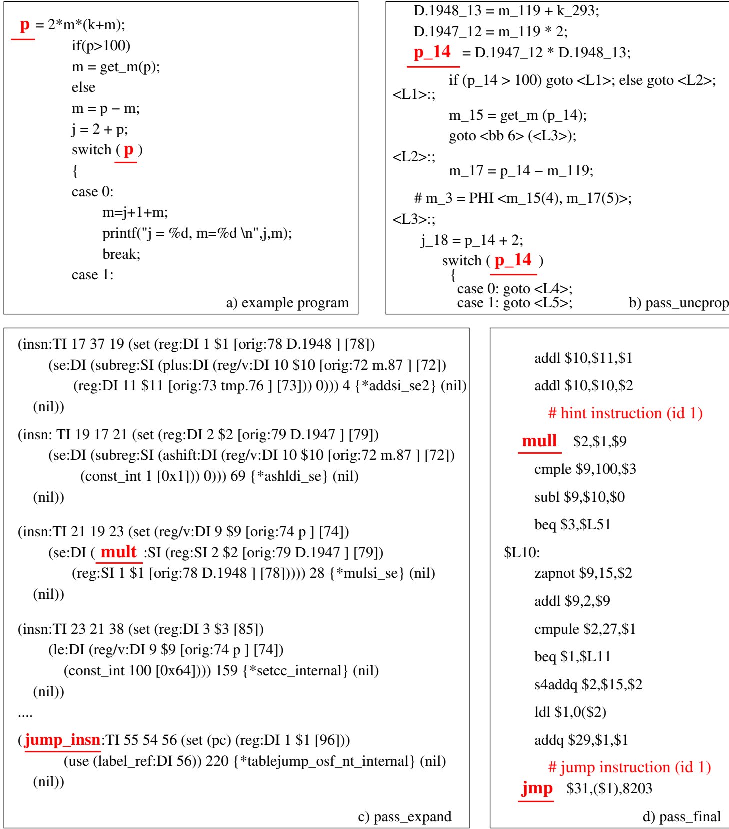


Fig. 3. Modified compiler passes for VBBI hint instruction analysis

form, the program variable **p** is renamed to **p_14**. All the uses of **p** including the switch-case statement reached by **p's** assignment are also renamed to **p_14**. Since in the SSA form, each USE has a unique DEF, we identify the hint instruction as the one containing the DEF of the SSA variable **p_14**.

2. **pass_expand:** This pass converts program statements from TREE format into the RTL code. For every hint instruction identified in the earlier pass, its corresponding RTL code is also marked as hint instruction. Figure 3(c) shows RTL code for some of the statements in the example program. Note the underlined **mult** code generated for statement 'p_14 = D.1947_12 * D.1948_13'. Similarly, the underlined **jump inst** code is generated corresponding to the 'switch (p_14)' statement.
3. **pass_final:** This pass looks at the list of instructions in the RTL format and outputs their corresponding assembly code. While going through the list of instructions, a counter keeps track of the number (and size) of instructions between the hint instruction and its corresponding jump instruction. This gives the PC offset of the hint instruction from the jump instruction, which

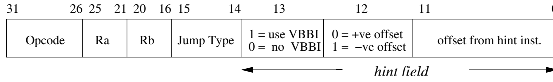


Fig. 4. Alpha indirect branch instruction augmented with VBBI hint bits

is then encoded in the jump instruction assembly code. Figure 3(d) shows the assembly code generated for the example program. The underlined **mull** instruction is the identified hint instruction for the underlined **jmp** instruction. The third argument of the **jmp** instruction is the VBBI related hint information encoded in the jump instruction format shown in Figure 4, i.e. bits 0 through 13 in the instruction. In this example, the hint information shows that the offset between the hint-jump instruction pair is 11 instructions (8203 = 2ˆ13 + 11).

4 Optimizations for Improving VBBI Prediction Accuracy

As indicated by the almost perfect prediction accuracy with VBBI target prediction overriding [7], the VBBI scheme is highly accurate when the current hint value is used, i.e. the hint instruction output is available when making the prediction. We apply compiler and run time optimizations to increase the dynamic instructions between the hint-jump instruction pair. This results in higher probability of the hint value being available when making the prediction. The techniques presented in this section are applicable to similar prediction schemes, as well as other design ideas exploiting data dependences to improve instruction level parallelism (ILP) and memory operation performance.

4.1 Compiler Optimizations

Instruction Hoisting. In this optimization, the hint instruction is moved away from its dependent jump instruction, thus creating more dynamic instructions between the hint-jump pair. Figure 5(a) shows a code example from SPEC95 099.go benchmark that is suitable for such an optimization. In this example, the switch-case statement will be compiled into an indirect jump instruction. The target taken by the jump instruction depends on the underlined computation of **shapes[sh].where** which can be hoisted up to the beginning of the **for** loop, thus increasing the dynamic distance between the hint instruction and the jump instruction.

Function Inlining. If the control variable of an indirect jump instruction is passed as a function argument, that function can be inlined to increase the dynamic instruction count between the hint and the jump instruction. Figure 5(b) shows a code example from SPEC2000 186.crafty benchmark that can benefit

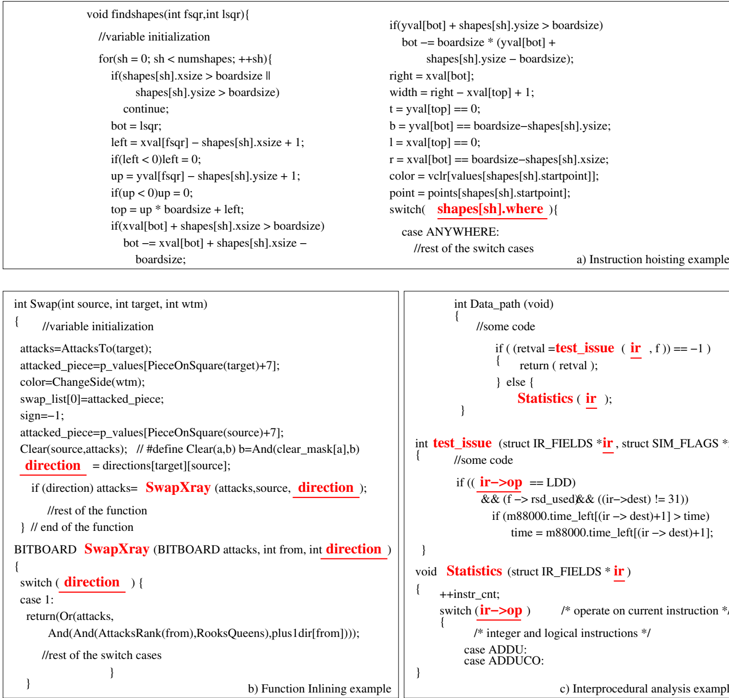


Fig. 5. Code examples showing compiler optimizations for VBBI

from function inlining optimization. In this example, the indirect jump instruction in function **SwapXray** depends on the underlined variable **direction** which is passed as an argument to the function. Without inlining, the instruction that POPS the variable **direction** from the stack will be marked by the compiler as the hint instruction. However, after inlining, this indirect jump instruction is now dependent on the *def* of the variable **direction** in function **Swap**. Note that this *def* of variable **direction** can be further moved to the beginning of the function using the previous instruction hoisting optimization.

Inter-procedural Dataflow Analysis. This optimization aims at identifying the hint instruction in a function other than the one containing the indirect jump instruction. Figure 5(c) shows a code example suitable for such an optimization. In this example, taken from SPEC95 124.m88ksim benchmark, the function **Data_path** calls two other functions, **test_issue** and **Statistics**, passing them the same pointer **ir**. The indirect jump instruction in function **Statistics** depends on the underlined computation **ir->op**. The same computation is also performed in an earlier function **test_issue**. Marking the hint instruction in function **test_issue** instead of **Statistics** greatly increases the possibility that the latest hint instruction outcome is available for making the prediction.

```
int Search(int alpha, int beta, int wtm, int depth, int ply, int do_null)
{
// some code
     while ((current_phase[ply]=(in_check[ply]) ? NextEvasion(ply,wtm) :
                                   NextMove (ply,wtm))) {
        extended_reason[ply]&=check_extension;
           // some other code
       MakeMove (ply, current_move[ply] ,wtm);
          //rest of the code
     } //end of while
 } //end of Search
```

```
void MakeMove (int ply,  int move , int wtm)
{
  register int piece, from, to, captured, promote;
    // some code
 piece = Piece( move ); // define Piece(a) (((a)>>12)&7)
    // some code
  switch ( piece ) {
      case pawn:
```

```
int NextMove (int ply, int wtm)
{
 register int *bestp, *movep, *sortv, temp;
 register int history_value, bestval, done, index;

 switch (next_status[ply].phase) {

 case HASH_MOVE:
  next_status[ply].phase=GENERATE_CAPTURE_MOVES;
  if (hash_move[ply]) {
   current_move[ply] = hash_move[ply];
   if (ValidMove(ply,wtm,current_move[ply])) return(HASH_MOVE);
   else printf("bad move from hash table, ply=%d\n",ply);
  }
 case CAPTURE_MOVES:
  if (next_status[ply].remaining) {
   current_move[ply] = *(next_status[ply].last);
   *next_status[ply].last++=0;
   next_status[ply].remaining--;
   if (!next_status[ply].remaining) next_status[ply].phase=KILLER_MOVE_
   return(CAPTURE_MOVES);
  }
  next_status[ply].phase=KILLER_MOVE_1;
```

Prediction Accuracy (%)			
Jump PC	**BTB only**	**VBBI baseline**	**VBBI with load–store address matching**
0x120022930	55%	61%	88%

Fig. 6. Load to store address matching example (taken from SPEC2000.186.crafty)

4.2 Hardware Optimization

Load to Store Address Matching. Dynamic tracking of load and store dependences has been used to support data speculation in code scheduling [9] [14]. We apply this technique to improve target prediction for indirect branches. When the hint instruction for an indirect jump is a *load* instruction, the address of the load instruction can be recorded in a *Hint Store Buffer (HSB)*. Subsequently, each store address is compared against load addresses in the HSB. If a match is found, the store value will be placed as the hint value in the corresponding HIB entry. Figure 6 shows an example taken from SPEC2000 186.crafty benchmark. In this example, the *while* loop in function *Search()* computes the next move by calling function *NextMove()*, which stores the computed move in **current_move**[**ply**]. Subsequently, **current_move**[**ply**] is passed as an argument to the function MakeMove() where **current_move**[**ply**] is used as the switch-case control variable. Dynamically tracking the load-store dependence and using the *store* value as opposed to the *load* value as the hint value increases the possibility of making the prediction using the latest hint value. As shown in Figure 6, the load-to-store address matching optimization improved the prediction accuracy of this jump to 88% compared to 61% when the hint instruction originally identified by the compiler was used for making predictions (VBBI baseline).

5 Simulation Methodology

We extended SimpleScalar [1] to simulate a 4-issue, 24-stage pipeline for evaluating VBBI prediction scheme. Table 2 shows the baseline parameters for our

Table 1. Characteristics of evaluated benchmarks

	m88ksim	li	perl	crafty	perlbmk	gap	sjeng	perlbench	gcc
Static Indir. br.	59	39	4	8	59	35	8	62	543
Dynamic Indir. br. (K)	161	62	289	215	1252	1238	532	1170	474
Baseline IPC	0.98	0.69	0.78	0.88	0.74	0.73	0.62	0.62	0.59

processor. Our workload includes nine benchmarks, three each from SPEC95, SPEC2000 and SPEC2006 suites [17]. Currently our compiler work for identifying hint instruction only support benchmarks written in C language. We plan to extend the support for C++ and Java benchmarks.

We use SimPoint [15] to find a representative program execution slice for each benchmark using the reference input data set. All binaries are compiled using modified GCC v4.2.1 with -O3 optimization running on Compaq Tru64 UNIX V5.1B. Each benchmark is run for 100M Alpha instructions. Table 1 shows the characteristics of simulated SimPoint for each benchmark.

Table 2. Processor parameters

Pipeline depth	Evaluated multiple configurations ranging from 8 to 24 stages;
Instr. Fetch	4 instructions/cycle; fetch ends at first pred. taken br;
Execution Engine	4-wide decode/issue/execute/commit; 512-entry RUU; 128-entry LSQ;
Branch Predictor	12KB hybrid pred. (8K-entry bimodal and selector, 32K-entry gshare); 4K-entry, 4-way BTB with LRU repl.; 32-entry return addr. stack; 15 cycle min. br mispred. penalty; 16-entry HIB; 32-entry HSB;
Caches	16KB, 4-way, 1-cycle L1 D-cache; 16KB, 2-way, 1-cycle L1 I-cache; 1MB, 8-way, 10-cycle unified L2 cache; All caches have 64B block size with LRU replacement policy;
Memory	300-cycle memory latency (first chunk), 15-cycle (rest);

6 Results

In this section we compare the performance of the baseline VBBI [7] with the optimized VBBI. Sections 6.1 and 6.2 use the traditional BTB and the TTC designs respectively as the reference point for comparison.

6.1 VBBI versus Traditional BTB

We compare the optimized VBBI prediction accuracy with the baseline VBBI and with the traditional BTB scheme in Figure 7. On average, the VBBI prediction accuracy is improved from 66% (baseline VBBI) to 80% (optimized VBBI).

Table 3. Average dynamic instruction count between hint-jump instruction pair

	m88ksim	li	perl	crafty	perlbmk	gap	sjeng	perlbench	gcc	AVG
VBBI baseline	45	58	10	63	12	31	47	15	51	37
VBBI optimized	168	110	16	158	43	39	174	24	64	88

Table 4. MPKI with traditional BTB, VBBI baseline and optimized VBBI

	m88ksim	li	perl	crafty	perlbmk	gap	sjeng	perlbench	gcc	AVG
Indir. br. MPKI (BTB only)	0.9	0.5	2.3	1.2	8.5	5.6	2.9	9.1	1.93	3.7
Indir. br. MPKI (VBBI baseline)	0.6	0.2	1.4	0.8	4.1	0.1	2.1	5.9	1.1	1.8
Indir. br. MPKI (VBBI optimized)	0.1	0.1	0.5	0.5	3.2	0.1	0.8	4.9	1.1	1.3

The BTB mispredictions are reduced by 2.1x using the optimized VBBI scheme. The increase in prediction accuracy is due to the fact that more predictions are made using the current and highly correlated hint value. Table 3 shows that the proposed optimizations increase the average dynamic distance between hint-jump instruction pair from 37 instructions to 88 instructions. Table 4 shows the number of mispredictions for indirect branches per 1K instructions (MPKI) for different prediction techniques. The optimized VBBI slashes the indirect branch MPKI by a third compared with traditional BTB design, and by a half compared with the baseline VBBI design.

Performance comparison of the optimized and the baseline VBBI scheme over the traditional BTB design is shown in Figure 8. For a 4-issue, 24-stage pipeline, the proposed VBBI optimizations enhance the baseline VBBI performance by 5.5%, achieving a 20.7% performance improvement over the traditional BTB design. When target prediction overriding is also enabled, the VBBI scheme achieves an overall performance improvement of 24.8% over the traditional BTB technique.

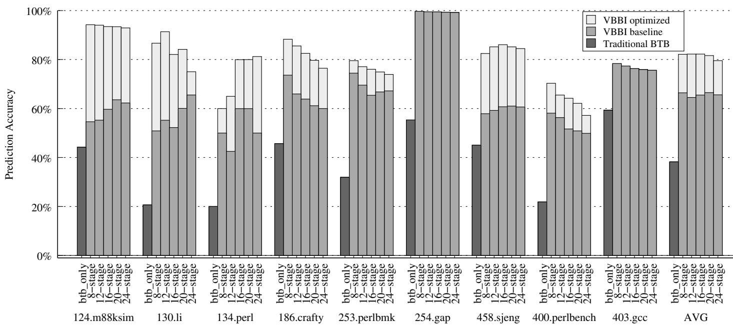


Fig. 7. VBBI Indirect branch prediction accuracy

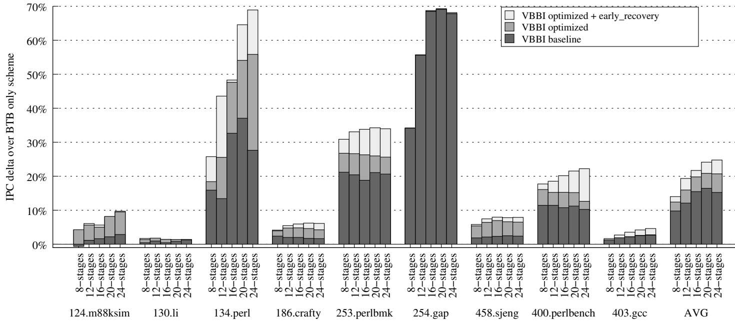


Fig. 8. IPC improvement of VBBI over traditional BTB scheme

6.2 VBBI versus Tagged Target Cache

We also compare the VBBI design with the TTC predictor [2], which is shown to be the best previously proposed jump predictor in a recent study by [10] (in Figure 17). In the TTC scheme, target addresses from recently executed indirect jump instructions are recorded in a target history register. When an indirect jump is fetched, the target cache is indexed using the XOR of the indirect branch PC and target history register, and the address stored at that index is predicted as the next target address. When the indirect jump retires, the computed target address is written into the target cache using the same index. When updating the history information, few bits from the target address are shifted into the global target history register. Farooq et al. [7] show that the TTC gives the best prediction accuracy with 14-bit global target history register. Upon update, 5 bits of target address (starting from the 3rd bit) are shifted into the target history register.

Figure 9 compares the VBBI prediction accuracy with the TTC design. On average, the optimized VBBI achieves a prediction accuracy of 80%, compared to 56% achieved by the best performing TTC configuration. Figure 10 compares the performance of the VBBI predictor with the TTC predictor. On average, the baseline VBBI outperforms the TTC design by 9.1% with just 130B of additional storage [7] compared to 384KB storage of the TTC design. The optimized VBBI further enhance the baseline VBBI performance by 4.7%, achieving 13.8% improvement over the TTC design. With target prediction overriding, VBBI achieves an overall performance improvement of 17.3% over the TTC predictor.

7 Previous Work

Lee and Smith [13] proposed branch target buffer (BTB) to predict indirect branches. This technique predicts the same target for the current execution of the branch that was taken in the last execution of that branch. Though simple in design, this scheme does not work well for indirect branches that may switch between multiple targets at run time.

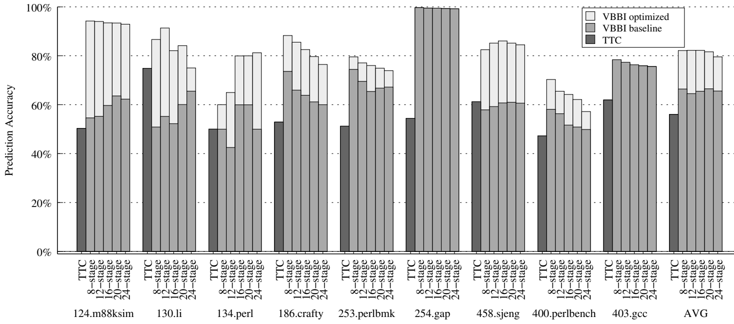


Fig. 9. Indirect branch prediction accuracy: VBBI prediction vs. Tagged Target Cache (TTC)

History based two-level indirect branch predictor was first proposed by Chang et al. [2]. This mechanism, known as 'target cache', uses the branch history information to distinguish different dynamic instances of the same indirect branch, a concept similar to 2-level conditional branch predictor [18]. When an indirect jump is fetched, the jump address and the global target history register are used to form an index into the target cache. The target cache is accessed and the resident address is predicted as the target address. Upon retiring the indirect jump, the target cache entry and the target history register is updated with the actual target address.

Driesen et al. [5] [6] focused on improving the indirect branch prediction accuracy by combining multiple predictors using a cascaded predictor. Cascaded predictor is a hybrid predictor consisting of a simple predictor for easy-to-predict indirect branches, and a more complex predictor for hard-to-predict indirect branches.

Kalamatianos et al. [11] proposed predicting indirect branches via data compression. Their predictor uses prediction by partial matching (PPM) algorithm of order three, which is a set of four Markov predictors of decreasing size, indexed by an indexing function formed by a decreasing number of bits from previous targets in the target history register.

Kim et al. [12] utilized the existing conditional branch predictor for predicting indirect branches as well. The mechanism, known as the 'VPC prediction', treats an indirect branch instruction with t targets as t direct branches, each with its own unique target address. On fetching an indirect jump, the VPC prediction algorithm makes *MAX_ITER* attempts for predicting an indirect branch target, each time as a different *'virtual direct branch'* of the same indirect branch. This iterative process stops either when a *'virtual direct branch'* is predicted to be taken, or *MAX_ITER* number is reached, in which case the processor is stalled until the indirect branch is resolved. *MAX_ITER* determines the maximum number of attempts made to predict an indirect branch. Each attempt takes one cycle during which no new instruction is fetched. A more recent study ([10] in Figure 13 and 14) shows that performance of VPC prediction degrades significantly for workloads with higher number of dynamic targets.

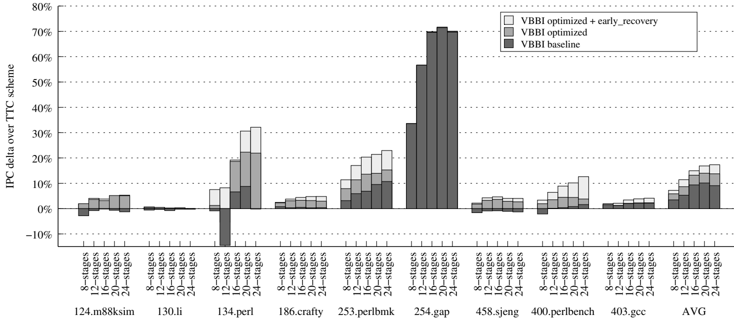


Fig. 10. Performance improvement: VBBI prediction vs. Tagged Target Cache (TTC)

Roth et al. [16] took a different approach for predicting indirect branch targets, precomputating them in anticipation of having to make a prediction. Proposed specifically for virtual function calls, the scheme dynamically captures the sequence of instructions involved in the target generation process. Whenever the first instruction in the sequence completes, it uses a separate, fast execution engine and computes the target before the actual call instruction is encountered. Although this technique avoids using specialized jump predictor, it requires significant hardware for capturing the target generation instructions along with a fast execution engine to pre-compute the target. Furthermore, this technique is very specific to target prediction of virtual function calls , as their target generation process consists of a fixed pattern of three dependent loads followed by an indirect call.

Joao et al. [10] proposed a new way of handling indirect jumps, dynamically predicating them. Instead of fetching from a single control path, when a hard-to-predict indirect jump instruction is fetched, the processor starts fetching from N different targets of the jump instruction. By fetching from more than one target, the processor increases the probability of fetching from the correct target path at the expense of executing more instructions. They showed that N=2 is a good trade-off between performance and complexity.

Recently, Farooq et. al [7] proposed a compiler-guided, correlation-based target address prediction scheme that combines data dependences with indirect branch target prediction. The proposed technique, known as Value based BTB indexing (VBBI), relies on the compiler's ability to statically capture data dependences, and uses the hardware to exploit the correlation between the data and branch target at run time. The key idea of VBBI is to identify a 'hint instruction' whose output is highly correlated with the target taken by the jump. At run time multiple targets of an indirect branch are stored at different BTB indices computed by hashing the branch PC with the output of the hint instruction. They show that by off-loading dependence analysis to the compiler, the hardware predictor size can be kept much smaller.

8 Conclusion

The recently proposed VBBI prediction scheme uses a novel BTB indexing technique that allows multiple targets of an indirect branch to be stored at different BTB indices. This technique relies on the compiler to identify a *'hint instruction'* whose output strongly correlates with the target taken by the indirect branch. At run time multiple targets are stored at different BTB indices computed by hashing the branch PC and the hint instruction output value.

In this paper we propose the compiler support for identifying the hint instruction for the VBBI prediction scheme. We also propose the compiler and run time optimizations that improve the VBBI prediction accuracy by increasing the dynamic instructions between the hint-jump instruction pair. The more the dynamic instructions between this instruction pair, the more likely that the hint instruction outcome will be available when making the prediction.

Our evaluation shows that the proposed optimizations improve the VBBI prediction accuracy from 66% to 80%. Compared to traditional BTB design, this translates into average performance improvement from 17.2% (baseline VBBI) to 24.8% (optimized VBBI). We also compare the VBBI with the best previously proposed indirect jump predictor, the tagged target cache (TTC). Compared to the TTC design, the proposed optimizations improve the performance by 6.3%, from 11% (baseline VBBI) to 17.3% (optimized VBBI).

References

1. Burger, D., Austin, T.M.: The SimpleScalar Tool Set, Version 2.0. SIGARCH Comput. Archit. News 25(3), 13–25 (1997)
2. Chang, P., Hao, E., Patt, Y.N.: Target Prediction for Indirect Jumps. In: ISCA-24, pp. 274–283 (1997)
3. COMPAQ. Alpha Architecture Handbook, V4 (October 1998)
4. Driesen, K., Hölzle, U.: Accurate Indirect Branch Prediction. In: ISCA-25, pp. 167–178 (1998)
5. Driesen, K., Hölzle, U.: The Cascaded Predictor: Economical and Adaptive Branch Target Prediction. In: MICRO-31, pp. 249–258 (1998)
6. Driesen, K., Hölzle, U.: Multi-stage Cascaded Prediction. In: Amestoy, P.R., Berger, P., Daydé, M., Duff, I.S., Frayssé, V., Giraud, L., Ruiz, D. (eds.) Euro-Par 1999. LNCS, vol. 1685, pp. 1312–1321. Springer, Heidelberg (1999)
7. Farooq, M.U., Chen, L., John, L.K.: Value Based BTB Indexing for Indirect Jump Prediction. In: HPCA-16, pp. 1–11 (January 2010)
8. Gochman, S., Ronen, R., Anati, I., Berkovits, A., Kurts, T., Naveh, A., Saeed, A., Sperber, Z., Valentine, R.C.: The Intel Pentium M Processor: Microarchitecture and Performance. Intel Technology Journal 7(2) (May 2003)
9. Intel. Intel software college: http://developer.intel.com/software/products/college/itanium/
10. Joao, J.A., Mutlu, O., Kim, H., Agarwal, R., Patt, Y.N.: Improving the Performance of Object-Oriented Languages with Dynamic Predication of Indirect Jumps. In: ASPLOS-13, pp. 80–90 (2008)
11. Kalamatianos, J., Kaeli, D.R.: Predicting Indirect Branches via Data Compression. In: MICRO-31, pp. 272–281 (1998)

12. Kim, H., Joao, J.A., Mutlu, O., Lee, C.J., Patt, Y.N., Cohn, R.: VPC Prediction: Reducing the Cost of Indirect Branches via Hardware-Based Dynamic Devirtualization. In: ISCA-34, pp. 424–435 (2007)
13. Lee, J.K.F., Smith, A.J.: Branch Prediction Strategies and Branch Target Buffer Design. Computer 17(1), 6–22 (1984)
14. Lin, J., Chen, T., Hsu, W.C., Yew, P.C.: Speculative Register Promotion Using Advanced Load Address Table (ALAT). In: Proceedings of First Annual IEEE/ACM International Symposium on Code Generation and Optimization, pp. 125–134. IEEE Computer Society (2003)
15. Perelman, E., Hamerly, G., Biesbrouck, M.V., Sherwood, T., Calder, B.: Using SimPoint for Accurate and Efficient Simulation. In: SIGMETRICS 2003, pp. 318–319 (2003)
16. Roth, A., Moshovos, A., Sohi, G.S.: Improving Virtual Function Call Target Prediction via Dependence-Based Pre-Computation. In: ICS-13, pp. 356–364 (1999)
17. SPEC. Standard Performance Evaluation Corporation, http://www.spec.org.
18. Yeh, T.-Y., Patt, Y.N.: Two-Level Adaptive Training Branch Prediction. In: MICRO-24, pp. 51–61 (1991)

Compiler Support for Fine-Grain Software-Only Checkpointing

Chuck (Chengyan) Zhao[1], J. Gregory Steffan[1],
Cristiana Amza[1], and Allan Kielstra[2]

[1] Department of Electrical and Computer Engineering, University of Toronto
{czhao,steffan,amza}@eecg.toronto.edu
[2] IBM Canada Toronto Laboratory
kielstra@ca.ibm.com

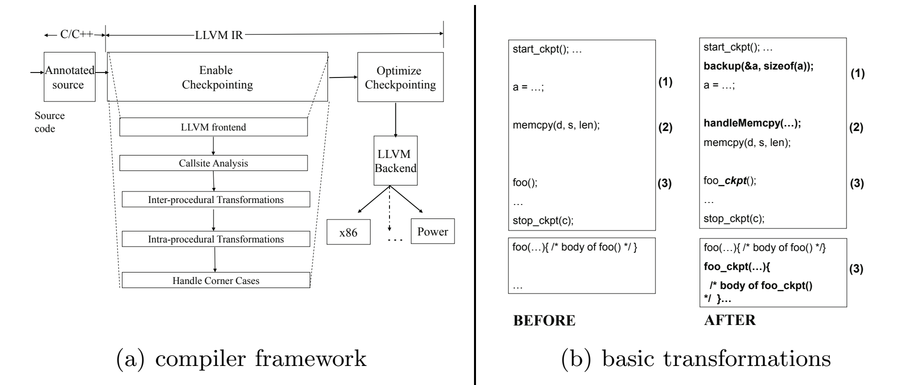
Abstract. Checkpointing support allows program execution to roll-back to an earlier program point, discarding any modifications made since that point. Existing software-based checkpointing methods are mainly libraries that snapshot all of working-memory, and hence have prohibitive overhead for many potential applications. In this paper we present a lightweight, fine-grain checkpointing framework implemented entirely in software through compiler transformations and optimizations. A programmer can specify arbitrary checkpoint regions via a simple API, and the compiler automatically transforms the code to implement the checkpoint at the granularity of individual stores, optimizing to remove redundancy. We explore two application areas for this support. First, we investigate its application to debugging, in particular by providing the ability to rewind to an arbitrarily-placed point in a buggy program's execution. A study using BugBench applications shows that our compiler-based approach is more than 100x less overhead than full-process checkpointing. Second, we demonstrate that compiler-based checkpointing support can be leveraged to free the programmer from manually implementing and maintaining software rollback mechanisms when coding a back-tracking algorithm, with runtime overhead of only 15% compared to the manual implementation.

1 Introduction

Checkpointing [7,16,19,24,27,30,31] is a technique to back-up program state such that execution can later revert to the backup, to recover from program failure or mis-speculation. While proposed hardware-based checkpointing solutions [4,13] show promising performance, they are not yet available in commodity systems. Software-based checkpointing solutions [16,19,25,31] can be used on commodity hardware, but can also have prohibitive overheads as they are typically coarse-grained, meaning that they back-up large ranges of memory if not the entire process image.

In this paper we propose a software-only method for checkpointing program execution that is implemented in a compiler. In particular, our transformations implement checkpointing at the level of individual variables, as opposed to previous work that checkpoints entire ranges of memory or entire objects [4,7,16,25].

M. O'Boyle (Ed.): CC 2012, LNCS 7210, pp. 200–219, 2012.
© Springer-Verlag Berlin Heidelberg 2012



(a) compiler framework (b) basic transformations

Fig. 1. Framework and basic transformations

The intuition is that such fine-grain checkpointing can (i) provide many opportunities for optimizations that reduce redundancy and increase efficiency, and (ii) facilitate uses of checkpointing that demand minimal overhead. We present a complete checkpointing framework and optimization infrastructure that can (i) enable software-only checkpointing over arbitrarily large and complex program regions and (ii) leverage compiler optimizations to reduce overhead. We show that our fine-grain scheme is more efficient than coarse-grain approaches, and that up to 98% of checkpoint buffer space and up to 95% of `backup` memory calls can be eliminated.

We demonstrate the utility of our compiler-based checkpointing infrastructure via two different applications of this support. The first is support for debugging, in particular by giving the programmer the ability to roll-back execution to repeatedly examine the state of a program prior to the manifestation of a bug. We study several flawed applications from the BugBench [20] suite and demonstrate the low overheads of checkpointing support for rollback. The second is support for back-tracking algorithms, where the programmer can avoid manually implementing support for rewinding data-structures, instead leveraging compiler-based checkpointing to provide it automatically. We study VPR [5,6], in particular the simulated-annealing-based place-and-route algorithm for FPGAs, which optimistically swaps blocks and either keeps or discards the swap depending on whether a cost function is improved. We compare the original manual implementation of back-tracking support to our automatic compiler-based approach.

2 Basic Checkpointing

Figure 1(a) presents an overview of our checkpointing system, implemented as passes in the `LLVM` [17,18] compiler infrastructure. It takes as input a source program with programmer annotations, and outputs transformed `LLVM IR` code that can target the multiple native platforms that LLVM supports. LLVM provides a `C` back-end that allows the conversion of optimized IR back to `C` source code.

This source-to-source approach allows us to capitalize on all of the optimizations of the back-end compilers.

Programmer Interface. We assume a very simple programmer interface to checkpointing: the user delimits the desired checkpointing region via the interface calls `start_ckpt()` and `stop_ckpt(c)`, where `c` is a boolean variable that indicates whether the checkpoint should be rewound/re-executed or committed. The compiler then instruments all relevant write operations with `backup` calls, each taking as arguments a pointer to the destination's address and its size in bytes. These `backup` calls are later optimized and inlined, but for now we show them in the code for illustration.

Callsite Analysis. Our compiler needs to know all user-defined functions that may be called directly or indirectly from the checkpoint region. We call the process of discovering such functions *callsite analysis*. The callsite analysis visits each node in the application's sub call graph originated from the annotated checkpoint region. It recursively identifies all user-defined functions in this partial call graph and marks them as requiring the creation of a checkpoint-enabled version.

Intra-procedural Transformation. The compiler then converts code in the user-annotated region into its checkpoint-enabled version in three steps. Step 1 is to precede each write with code to backup the write. Figure 1(b)(1) shows that variable `a` is modified and thus preceded with a `backup` operation. Step 2 is to handle certain system functions that have implicit memory writes. Figure 1(b)(2) illustrates the handling of one such routine, `memcpy`, by placing a handling function, `handleMemcpy`, immediately before it. Step 3 is to rename any user-defined function callsite within the region. Figure 1(b)(3) shows that a user callsite `foo` is renamed to its checkpoint-enabled version, `foo_ckpt`.

Inter-procedural Transformation. The final step is to enable checkpointing on all user-defined routines that are identified through the callsite-analysis phase. For each identified function, we clone its function body and rename it by appending `_ckpt` to its name, as shown in Figure 1(b)(3). Within the body of the cloned function, we recursively and repetitively apply the same three actions introduced in `Intra-procedual Transformation` above. In the end we produce a checkpoint-enabled version for every user-defined function that can potentially be called from the checkpoint region.

Handling Function Pointers. Since our checkpointing scheme clones user-defined functions, the compiler needs to identify the precise callee function at compile time. However, calls via function pointers might only be resolved at runtime. As shown in Figure 2(a), we handle this function pointer ambiguity by changing from a function pointer call to a normal function wrapper call with function pointer arguments. Within the wrapper function, each possible callee is explicitly examined through a list of parameter-matched candidates.

Handling Early Exits. Another special case deals with early exits from the checkpointing region, as shown in Figure 2(b). A `return` within the checkpoint

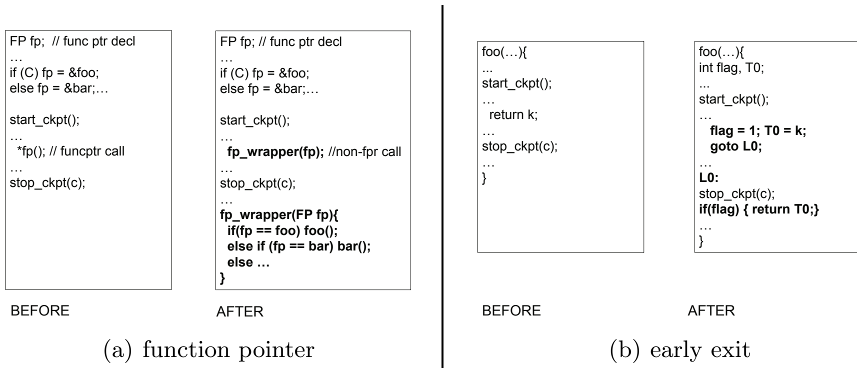


Fig. 2. Examples of handling function pointers and early exits

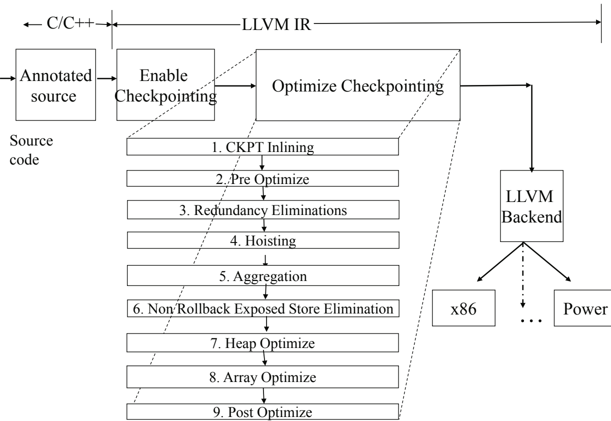


Fig. 3. Overview of optimization passes

region may prematurely terminate the checkpoint process without visiting the `stop_ckpt` marker. This violates the rule that the checkpoint region markers must be visited in pairs. Figure 2(b) suggests a possible solution: code is transformed to have a `goto` that branches to the `stop_ckpt` marker and reserves the appropriate return value.

3 Optimizations

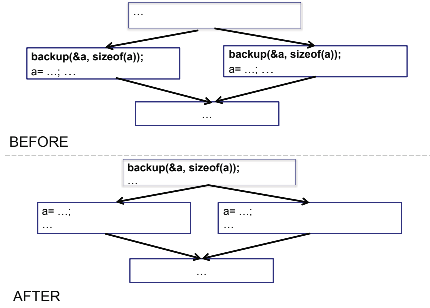
Base transformations enable checkpointing on any user-annotated region by backing up the memory contents before each explicit or implicit write. This creates a large number of `backup` calls that are potentially redundant and leaves ample opportunities for optimization. Figure 3 provides an overview of our checkpointing optimization framework, that takes as input the checkpoint-enabled code produced as described in the previous section. The framework includes

```
start_ckpt();
...
 backup(&a, sizeof(a));
 a = ...;
 ...
 backup(&a, sizeof(a));
 a = ...;
 ...
 foo_ckpt();
 ...
 if (C){
   backup(&a, sizeof(a));
   a = ...; ...
 }
...
stop_ckpt(c);
...
foo_ckpt(){
 int x;
 ...
 backup(&a, sizeof(x));
 x = ...;
...}
```
(i) code with CKPT enabled

```
start_ckpt();
...
 backup(&a, sizeof(a));
 a = ...;
 ...
 backup(&a, sizeof(a));   [struck out]
 a = ...;
 ...
 foo_ckpt();
 ...
 if (C){
   backup(&a, sizeof(a));   [struck out]
   a = ...; ...
 }
...
stop_ckpt(c);
...
foo_ckpt(){
 int x;
 ...
 backup(&a, sizeof(x));
 x = ...;
...}
```
(ii) redundancy-elim case1

```
start_ckpt();
...
 backup(&a, sizeof(a));
 a = ...;
 ...
 backup(&a, sizeof(a));   [struck out]
 a = ...;
 ...
 foo_ckpt();
 ...
 if (C){
   backup(&a, sizeof(a));   [struck out]
   a = ...; ...
 }
...
stop_ckpt(c);
...
foo_ckpt(){
 int x;
 ...
 backup(&a, sizeof(x));   [struck out]
 x = ...;
...}
```
(iii) redundancy-elim case1+2

(a) cases 1 and 2



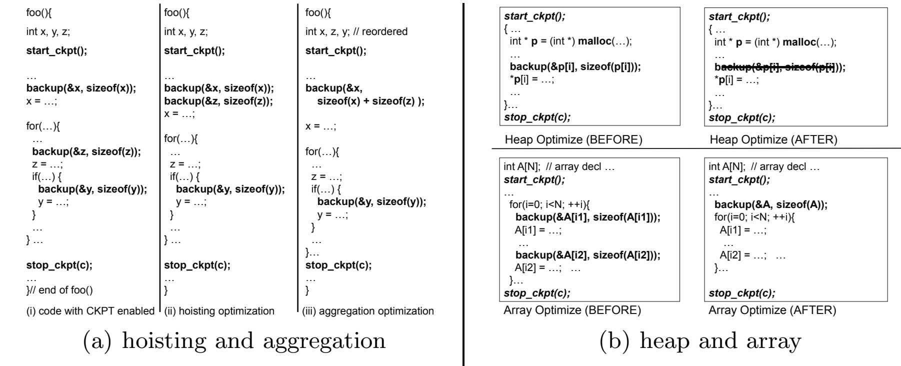
(b) case 3

Fig. 4. Redundancy elimination cases 1-3 via code examples

more than 10 different optimizations and we introduce them in order of importance.

Redundancy Elimination. The most important optimizations are three cases of redundancy eliminations ($RE1$, $RE2$, and $RE3$), as illustrated in Figure 4. $RE1$ uses *dominating* relationships among `backup` calls. It identifies all `backup` calls with the same address and length that dominate the `stop_ckpt` region marker (e.g., the first three `backup` calls in Figure 4(a)), establishes the first in the sequence as the *leading* `backup` call, and then removes any remaining ones that are dominated by the leader. $RE2$ identifies all `backup` operations on a function's non-pointer-type local variables (i.e., the fourth `backup` call in Figure 4(a)). Since local variables are allocated on the stack and have no memory footprint in its enclosing function's calling context, it is safe to remove backups on local variables within any checkpoint-enabled function without impacting the correctness of checkpointing. $RE3$ performs similarly to common sub-expression elimination (*CSE*) by finding duplicate `backup` operations on both sides of a branch (as shown in Figure 4(b)). Once it finds a suitable pair, it hoists one of the backup calls into the immediate dominator block, and removes the other backup call.

Hoisting. Hoisting optimization aims to reduce redundant backup calls within loops (as illustrated in Figure 5(a)), by hoisting the backup of any variable written unconditionally within a loop to the loop header (e.g., variable `z` in the example). Such hoisting would not be performed by a normal compiler hoisting pass since the write to the variable is not necessarily loop invariant. The decision to hoist conditionally-modified backup calls is a trade-off, the conditional code must be executed frequently enough to be worth the cost of the non-conditional backup call in the hoisted version. Through experiment we found that it is generally not worth hoisting such conditionally-modified variables, at least not without profile feedback as a guide. To illustrate, in the example we choose not to hoist variable `y`.



(a) hoisting and aggregation (b) heap and array

Fig. 5. Examples of hoisting, aggregation, heap, and array optimizations

Aggregation. Aggregation examines `backup` calls for variables that are adjacent in memory, potentially rearranging the layout of the variables to ensure that they are adjacent. Figure 5(a)(iii) shows that two individual `backup` operations on variable `x` and `z` can be merged into a single one, covering the entire memory range for both variables.[1] Aggregation reduces the overhead of managing adjacent variables individually.

Dynamic Memory Optimization. Opportunities exist for any `backup` call that operates on dynamically allocated (heap) memory. If the heap allocation site is within the checkpoint region and it dominates the write, the `backup` operation on this write into heap-allocated memory can be eliminated. Figure 5(b) demonstrates the process of removing a `backup` on heap-allocated variable `p[i]`. Since the heap allocation happens within the checkpoint region, the heap-allocated contents have no memory footprint before checkpoint starts. Hence such backup calls can be eliminated since they are unneccesary.

Array Optimization. More interesting cases occur among backup operations on writes to array-based data inside a loop, as shown in Figure 5(b). Both writes into `A[i1]` and `A[i2]` are correlated with loop index variable `i`. It could be beneficial to merge multiple backups on individual array elements into a single backup operation, potentially covering a continuous array sub-range or even the entire array. We develop an algorithm that considers not only the array size, loop trip count and store intensity, but also a tolerance factor that a user can control through command-line options. Non-continuous array writes may happen when the program executes inside the loop, thus the tolerance factor specifies the trade-off in buffer-space used versus performance.

Non-rollback-Exposed Store Elimination. Given any variable that is written inside the checkpoint region, if there is *no* read of that variable on any path

[1] Note that for a source-to-source transformation this isn't necessarily a safe optimization as the back-end compiler may further rearrange the variable layout—an implementation in a single unified compiler would not have this problem.

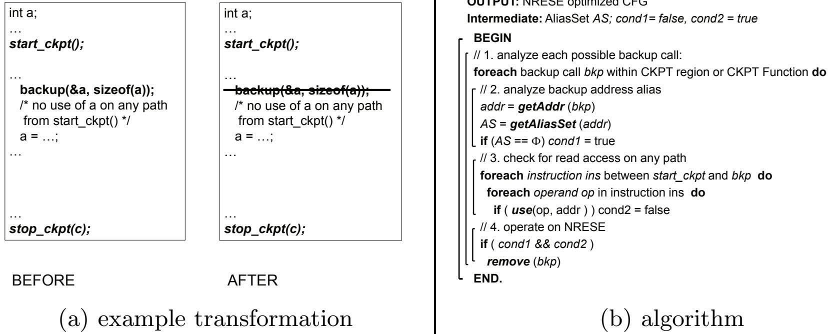


Fig. 6. Non-rollback-exposed store elimination optimization (NRESE)

from the beginning of the region, and its address has no alias, then an optimization can remove the respective `backup` operation for this variable without impacting checkpointing correctness. We call this optimization *non-rollback-exposed store elimination* (*NRESE*). Figure 6(a) shows an example of NRESE. Notice that the `backup` operation on variable `a` can be safely removed, since there are no direct or aliased reads of `a` along any path from the beginning of the checkpoint region. The value of `a` is recomputed each time and this re-computation is essentially independent of the current value of `a`. The algorithm presented in Figure 6(b) relies on performing an alias analysis to that `a` has no alias—we use the basic alias analysis (basic_aa) provided with `LLVM`.

Miscellaneous Optimizations. Inlining is applied to all remaining `backup` operations, allowing later standard optmiizations to schedule and optimize the contained instructions. `Pre-Optimize` and `Post-Optimize` passes perform miscellaneous clean-up operations, such as removing zero-length `backup` calls).

4 Buffering Implementation

The most important design decision in a checkpointing scheme is the approach to buffering: whether it will be based on *write-buffering* [11,21] or *undo-logging* [14,23]. A write-buffer approach buffers all writes from main memory, and therefore requires that the write-buffer be searched on every read. Should the checkpoint commit, the write-buffer must be committed to main memory; should the checkpoint fail, the write-buffer can simply be discarded. Hence for a write-buffer approach the checkpointed code proceeds more slowly, but with the benefit that parallel threads of execution can be effectively checkpointed and isolated (e.g., for some forms of optimistic transactional memory [11,22]). An undo-log approach maintains a buffer of previous values of modified memory locations, and allows the checkpointed code to otherwise read or write main memory directly. Should the checkpoint commit, the undo-log is simply discarded; should the checkpoint fail, the undo-log must be used to rewind main memory. Therefore an undo-log approach

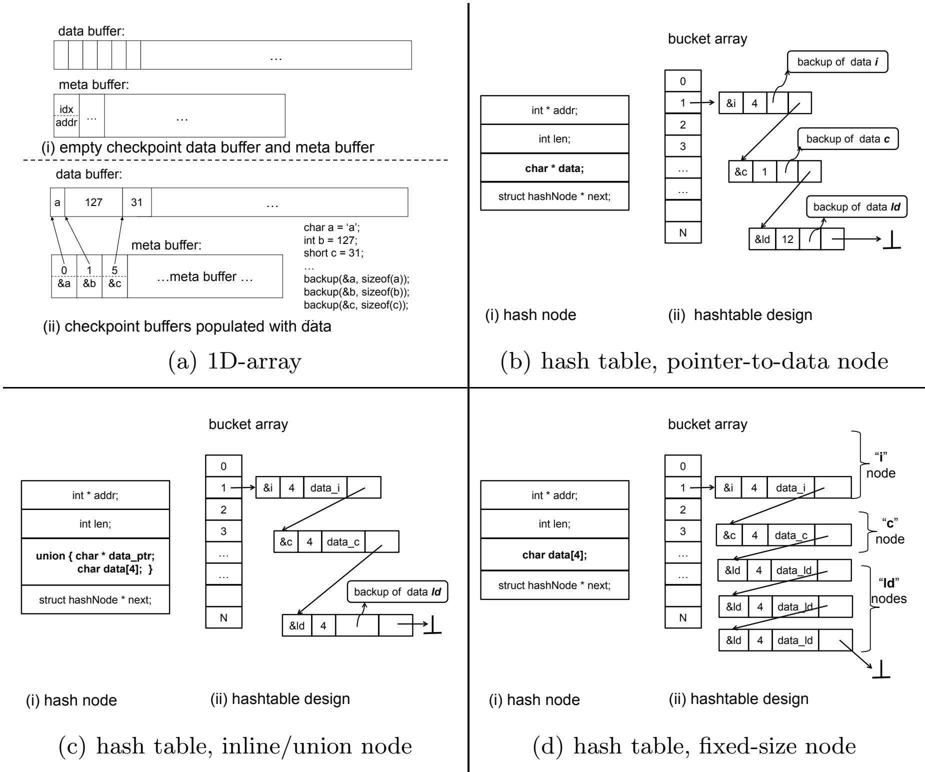


Fig. 7. Design options for an undo-log implementation

is best for the case of single-threaded code where checkpoint-rewind is uncommon, hence we focus solely on an undo-log approach for the remainder of this paper.

Figure 7(a)(i) illustrates a straightforward design of an undo-log based on the use of 1D arrays, where we have divided the undo-log buffer into two structures: (i) an array that is a concatenation of all backed-up data values of arbitrary sizes; and (ii) a meta-data array that stores the length and starting address of each element. As an example, Figure 7(a)(ii) shows the contents of an undo-log after three `backup` calls. When a checkpoint commits, we simply move the data and meta-data pointers back to the start of each array; when a checkpoint must be rewound, we use the meta buffer to walk backwards through the data buffer, writing each data element back to main memory.

While simple, a 1D-array-based undo-log suffers from redundancy, as each new backup call simply appends a value to the log without searching for an existing entry for that location; to search the array linearly would be prohibitively expensive. An alternative is to use a hash-table to allow fast search of prior entries for matches, to eliminate all redundancy in the undo-log. There will be a trade-off in the performance savings of reduced storage (due to reduced redundancy) versus the performance cost of hash-lookups.

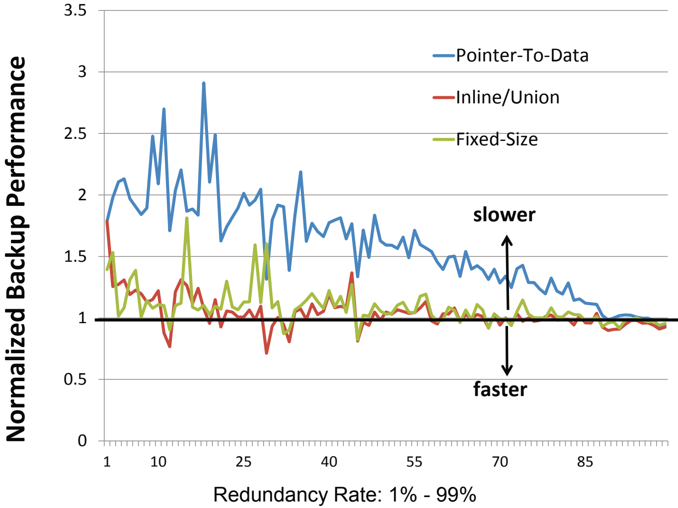


Fig. 8. Performance impact of four different buffer schemes over a wide range of redundancy rates. The x-axis represents redundancy rate from 1% to 99%; the y-axis is the relative checkpoint performance normalized to using a 1D-array. The figure represents checkpoint buffer with 1024 unique backup addresses, with only 4-byte backup length.

Hence we consider three hash-table designs, as illustrated in Figure 7, based on the options for the design of a hash table node: *pointer-to-data* (*PTD*), that stores a pointer to dynamically-allocated data storage; *inline/union* (*union*), that stores a union field that can be used either to directly store a 32-bit value inline, or instead as a pointer to dynamically-allocated data storage larger than 32 bits; and *fixed-size* (*fixed*), that always stores 32 bits of data per node and requires a list of nodes to store larger data values.

To compare the potential undo-log implementations we measure their *redundancy rate*, defined as follows. Let $Access(R)$ denote the total number of backups of a particular variable R that is written at least once within the checkpoint region, then the redundancy rate (RR) for this region can be defined as

$$RR = \frac{\sum_1^n (Access(R_i) - 1)}{\sum_1^n Access(R_i)} \quad (1)$$

where n is the total number of unique addresses that are checkpointed within the region. RR quantifies the amount of checkpointing redundancy as a floating point value between 0 and 1. In an ideal region where each unique variable address is checkpointed exactly once, its RR rate will be 0. The higher the RR rate, the more redundancy remains.

In Figure 8 we evaluate the trade-offs between the four buffering implementations above on microbenchmarks. We vary the microbenchmark access patterns to produce redundancy rates that vary from 1% to 99%, and report

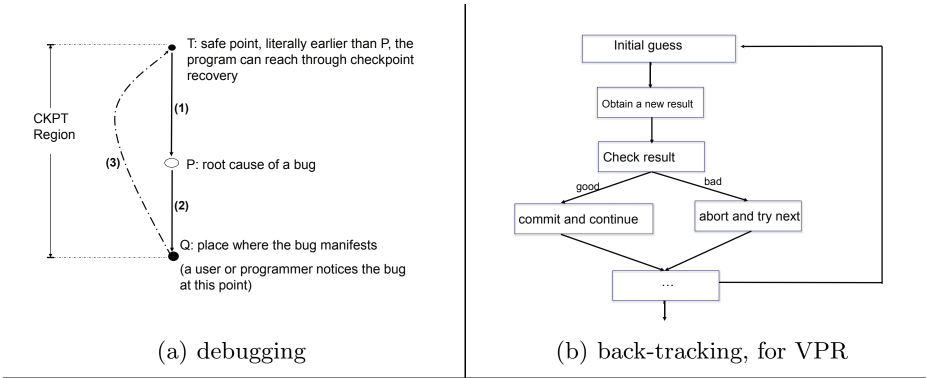


Fig. 9. Overview of applications enabled by fine-grained checkpointing support

checkpoint performance normalized to that of using a 1D array. Overall, the solution based on a 1D array almost always outperforms all hashtable-based solutions. All three curves converge at a very high *RR* rate (close to 99%). With increasing redundancy rates, the performance difference among different backup schemes diminishes. The three different hashtable-based implementations have perfect storage behaviors; however this comes at a performance cost, mainly due to the poor cache locality of their link-list accesses. *Union* and *fixed* are both heavily optimized for dynamic memory management, thus their performance is considerably and consistently better than *PTD*. In summary, because of its superior performance, we focus on the 1D-array implementation of the undo-log for the remainder of this paper.

5 Checkpoint-Enabled Applications

Our compiler-based fine-grained checkpointing scheme can be leveraged in a wide range of applications. In this section, we introduce two important application domains that can benefit by either gaining additional functionality or through a simplified programming interface: checkpoint support for debugging, and checkpoint-enabled automatic back-tracking.

5.1 Checkpoint Support for Debugging

Program debugging is used to identify and resolve software bugs. A normal debugging session begins with user placing breakpoints at multiple pre-determined program locations, and stops execution at each location to examine the program's logic and states. However, once execution passes a certain breakpoint, it is normally difficult to rewind execution to a previous location though a user may often find that the root cause of a bug is likely located close to a previous breakpoint. Frequently restarting execution can be impractical because it may take a long time to reach the suspicious bug location.

Debuggers enhanced with our checkpointing support can help alleviate this situation. We expose the checkpoint APIs on the source-code level so that a programmer can selectively mark a checkpoint region that likely contains the bug, as shown in Figure 9(a). The programmer first inserts a end-region marker slightly after the bug-trigger location. Properly identifying a start-region position requires some understanding of the code. The region needs to be big enough to contain the root cause of the bug, but can't be too big so that the programmer is lost in unrelated details. In practice, we often place breakpoints overlapping with the checkpoint region boundaries. Once execution reaches the end of the region, the programmer decides whether he wants to finish debugging this region (by issuing a `commit_ckpt` command), or rewind and re-examine the current region (by issuing an `abort_ckpt` command).

Debuggers with our checkpointing support can rewind execution to a previously identified program location and re-examine the program region with unlimited number of retries. There is no restriction on the size of the region because we checkpoint into main memory and can dynamically grow the checkpoint buffer when needed. Eliminating program restart not only avoids all problems related with non-deterministic execution and availability of input, but also helps to reduce debugging cycle time. In practice we find it easy to use such rewind-capable debugger. The restart-free debugger with checkpointing support leads to shorter debugging cycle – allowing a programmer to rapidly identify root causes of a bug, thus converting the checkpointing capability into improved productivity.

5.2 Checkpoint Support for Automated Back-Tracking

Back-tracking refers to a set of algorithms that search for solutions in a given space of possible choices. A partial result may be either committed or discarded, depending on the evaluation result from it.

We study Versatile Placement and Route (*VPR*) [5,6], a CAD tool for generating high-quality circuit layouts on array-based FPGAs. VPR places and routes on a wide variety of FPGAs and facilitate comparisons among different architectures. VPR implements a software back-tracking algorithm in its placement phase, as shown in Figure 9(b). The algorithm starts as its input a randomly generated guess. It evaluates the result based on this attempted input. If the result is positive, it will be incorporated into the current system. Otherwise, the negative result is discarded. This process continues until a desired terminating condition is satisfied. Current implementation of VPR saves all necessary program states before attempting a new input. Shell a discard happen, it manually restores all saved program states from various complex data structures. VPR designers need to understand not only the placement algorithms, but also pay close attention to details of manually save and restore necessary program states. This is a tedious and error-prone process that often has a negative impact on productivity, especially when improving the algorithm that results data structure changes.

By exposing the checkpoint APIs at the source-code level, our fine-grain checkpoint framework frees VPR from details of conducting back-tracking operations.

Table 1. Benchmarks and Checkpoint Region Properties

Apps	Region	avg insts	avg source lines	entries
bc-1.05	S	2.2 K	3	3
	M	208 K	430*	1
	L	305 K	1200*	1
gzip-1.24	S	0.9 K	1	1
	M	2.7 K	89*	1
	L	194 M	119*	1
man-1.5h1	S	1.4 K	14	1
	M	1.6 K	30*	1
	L	645 K	89*	1
ncompress-4.2	S	0.8 K	2	1
	M	149 K	149*	1
	L	231 K	163*	1
polymorph-0.4.0	S	1.5 K	2	1
	M	3.1 K	49*	1
	L	148 K	76*	1
VPR-5.02		67.1 K	268*	371 K

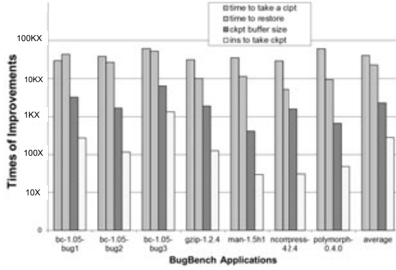
VPR designers can ignore all details of manual checkpointing and instead call `abort_ckpt()` or `commit_ckpt()`, which performs checkpoint abort and commit actions respectively. The simple APIs enable automatic software back-tracking on VPR, as well as all applications that have a need to perform back-tracking. VPR designers can instead focus on improving the algorithm – an step that simplifies application programming interface and improves end-user productivity.

6 Evaluation

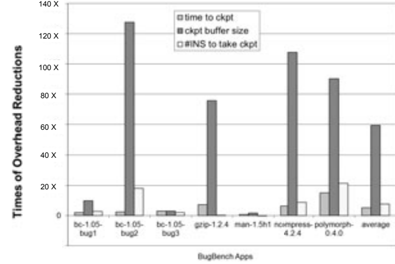
In this section we evaluate our fine-grain software-only checkpointing framework. Our compiler infrastructure builds on the LLVM [17,18] open-source compiler infrastructure release 2.9—all analyses, transformations, and optimizations are organized as LLVM passes. For debugging support we consider Bugbench [20] applications, a suite containing various known software bugs plus program inputs that trigger them; we select five BugBench applications that contain buffer-overflow bugs. To evaluate back-tracking support we study a recent version of VPR-5.02 [5], as described in Section 5.2. We measure on an Intel Core i7 920 CPU, with 4GB of DDR3 RAM, running Debian6-i386 with `g++` version 4.4.5.

6.1 Checkpoint Region Selection

Table 1 summarizes the checkpoint regions for each benchmark application. For the selected applications from the BugBench suite, we enclose the root cause and manifestation of each bug in a minimal checkpoint region called the *small (S)* region. We then grow the small region by both forward-and-backward extending



(a) our approach relative to coarse-grain libCKPT



(b) our approach relative to runtime-based ICCSTM

Fig. 10. Overhead reduction relative to conventional checkpointing methods for BugBench applications

the region boundaries, covering increasing granularity and complexity of the source code. The result is a *medium (M)* region that contains a significant portion of the program, and a *large (L)* region that can potentially cover the entire application. VPR has only one checkpoint region as appropriate for properly implementing back-tracking within the `try_swap` function, although we have two implementations, *medium (M)* and *large (L)*, depending on whether the region is marked from the function callee's perspective or the caller's perspective, respectively. Checkpoint regions are vastly different in size: for example, a small region usually contains around 1000 instructions and spans 2–3 lines of source code, while a large region can contain up to 195 million instructions (e.g., gzip-1.24) and covers 1000+ lines of source code (e.g., bc-1.05)[2].

6.2 Comparison with Existing Checkpointing Solutions

In this section we compare our compiler-based checkpointing solution with two alternative software approaches to checkpointing: a checkpointing library, and a software transactional memory library supported by a commercial compiler.

Library-based schemes back-up all of the memory used by the running process—thus the checkpointing overhead closely correlates to the size of memory at checkpointing time. We use libCKPT [26] as the representative of a library-based software checkpointing solution. Figure 10(a) shows that our fine-grained checkpointing approach provides over 1000X overhead reduction compared to coarse-grain checkpointing, for both the time-to-take a checkpoint and the time-to-restore a checkpoint. The corresponding improvement in terms of the checkpointing metrics of checkpoint buffer size and the number of instructions needed to service a checkpoint are within the range of 100X to 1000X.

[2] Note that M and L regions always contain user-defined functions, thus the number of source lines presented in Table 1 marked with * only indicates the lower bound of possible source-code span.

We further compare software overheads for supporting single-threaded speculative optimization in Intel's Software Transactional Memory (STM) [1,28] (*ICCSTM*) versus our compiler-based checkpointing solution. (*ICCSTM*) is a software solution for supporting optimistic parallelism, based on Intel's production-quality `C/C++` compiler. Just like other STM systems, *ICCSTM* supports speculative parallel execution through write-buferring and dependence tracking of the reads and writes of multiple threads at run-time. The differences in performance between the two software packages are expected to come from the different focus and specialization on their respective main use cases.

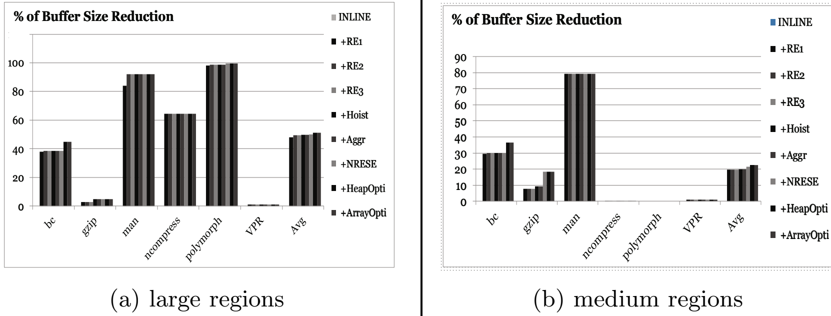
ICCSTM is mainly optimized to support program parallelization based on relatively short transactional regions. On the other hand, our checkpointing software is optimized to support single-thread speculation, or debugging for larger program regions. Based on the limited description available [1,28], *ICCSTM* uses only basic compiler optimizations such as inlining and a very simple form a partial redundancy elimination. Furthermore, to the best of our knowledge, *ICCSTM* does not optimize for the single-threaded speculative execution case. In this special case of speculation support, tracking of the single thread's read-set could be safely omitted. In contrast, our checkpointing scheme benefits from being specialized for the single-thread case. Specifically, we track only the write set for the speculative thread via an efficient implementation based on undo-logging. In the common case where speculation is successful, undo-logging avoids expensive lookups on reads for matching prior writes, and also the copies of writes to shared memory on commit. Overall, it is expected that our fine-grain checkpoint support will have lower overheads, and/or better cache behavior than a write-buffering STM.

Figure 10(b) compares *ICCSTM* to our baseline compiler-based checkpointing solution (with no optimizations). We find that our solution outperforms *ICCSTM* in almost all cases. On average, our solution outperforms the time-to-take a checkpoint for *ICCSTM* by 5X, and the number of instructions needed to take a checkpoint by 8X. The largest difference is in terms of the checkpoint buffer size, which is almost 60X lower for our solution.

6.3 Optimization Effectiveness

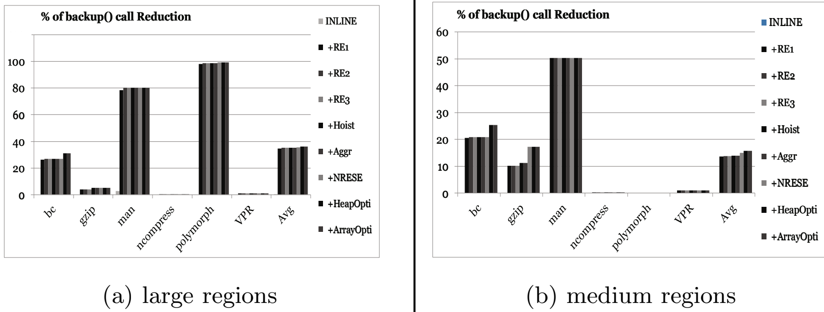
To evaluate our checkpointing optimization framework, we run optimizations over each application's *M* and *L* regions. We gradually increase the number of optimizations on each test region until all available optimizations are applied. We focus the evaluation on the effectiveness of checkpointing overhead reduction as measured by the following metrics: checkpoint buffer size reduction, the reduction in the number of `backup` calls, and the impact on the redundancy rate.

Checkpoint Buffer Size Reduction. Figure 11 shows the compiler optimization impact on checkpoint buffer size when all optimizations are incrementally applied. The effectiveness of our optimizations depends on the region size, as well as the frequency of stores within the region. Normally, a larger region has more opportunities for optimization. We observe that *RE1* is the most effective of all optimizations: as shown in Figures 11(a), and 11(b), respectively, *RE1*



(a) large regions (b) medium regions

Fig. 11. Incremental/cumulative impact of optimizations on buffer size



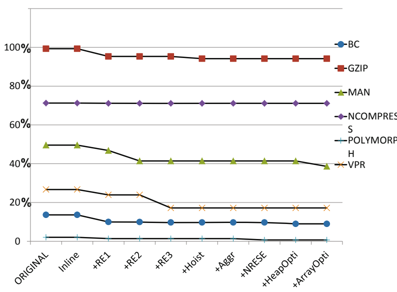
(a) large regions (b) medium regions

Fig. 12. Incremental/cumulative impact of optimizations on number of backup calls

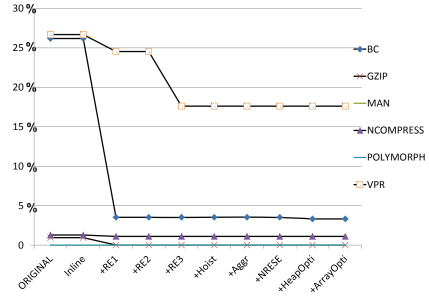
reduces the checkpoint buffer size by 92% in `polymorph`, and by almost 80% in `man`. When optimizations are incrementally applied, we observe a stable trend of buffer size reduction for both M and L regions. All performance numbers show that our compiler optimizations either exploit opportunities for optimization and hence improve checkpoint efficiency, or at least do not introduce negative effects (regressions). On average, the optimizations reduce checkpoint buffer size by an average of 52% for the L regions and 22% for the M regions.

Backup Call Reduction. in addition to buffer size reduction, our compiler optimizations also reduce the total number of `backup` calls—another metric for estimating the checkpointing overhead. Figure 12 shows that our optimizations reduce the total number of `backup` calls by an average of 36% for the L region and by 15% for the M region.

Redundancy Rate Impact. After all optimizations have been applied, it is interesting to understand how much redundancy remains in the checkpoint buffer, as a measure of what further optimization opportunities remain. We

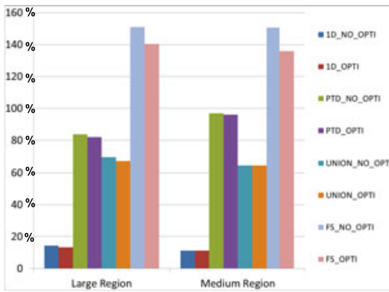


(a) large regions

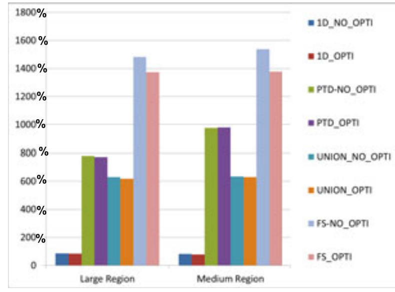


(b) medium regions

Fig. 13. Incremental/cumulative impact of optimizations on redundancy rate



(a) entire program



(b) within `try_swap` only

Fig. 14. Performance overhead of automated back-tracking support via our compiler-based checkpointing, relative to manually-implemented back-tracking support

quantify this by studying the region's *redundancy rate* (RR), as defined earlier in Section 4. Figure 13 illustrates the impact of our compiler optimizations on RR for both M and L regions when incrementally applying available optimizations. Figure 13(b) indicates that our optimizations are more effective in eliminating redundancy in M regions, since the highest RR is around 18% after optimizations. This is because the opportunities in M regions are more likely to be captured by our optimizations. Although optimizations do reduce redundancy for L regions as well, in most cases the impact is small. Three applications (`gzip`, `ncompress`, and `man`) still have very high RR even after applying all available optimizations. After manually examining the source code for each case we conclude that the high RR is due to extensive use of pointers, the presence of which hinders our optimization framework.

6.4 Overhead of Back-Tracking Support

In this section we evaluate the use of our checkpointing framework for implementing automatic back-tracking support in the VPR application, as introduced earlier in Section 5.2. Automatic back-tracking support frees the developer from having to manually implement support for checkpoint and restore, and also allows for source code that is easier to read and maintain.

We focus our evaluation on the `try_swap` function that implements the back-tracking portion of VPR. This routine spans almost 300 lines of source code including function calls and data access through link-list structures. Figure 13 shows that the final *RR* for VPR is 18%, and that *RE1* exploits most of the optimization opportunities. In addition to the 1D array buffer scheme, alternatively we can try a hash table solution to achieve a perfect redundancy behavior.[3] Section 4 introduces the details of our hash table designs, and their performance impact on checkpoint-enabled VPR is given in Figure 14.

Figure 14(a) presents the overall performance impact on VPR when our compiler-based automatic back-tracking is enabled. We observe that the 1D-array scheme achieves the best run-time performance after optimization. The entire VPR program has a mild 15% slowdown with 18% of buffer redundancy after applying all optimizations. When switching to buffers utilizing hash table schemes, we measure increased performance penalty to 62%, 95% and 140% when utilizing *union*, *PTD*, and *fixed* hash-table buffer schemes respectively.

We further zoom-in to the function level and measure the checkpointing performance impact on the `try_swap` routine alone (in Figure 14(b)) by comparing its execution time with and without our compiler-based checkpointing. Figure 14(b) is very similar to Figure 14(a). They can be treated as scaling on different levels of granularity. When measuring against `try_swap` function only, enabling automatic checkpointing will have performance penalties of 90%, 600%, 950% and 1500% for *1D array*, *union*, *PTD*, and *fixed* respectively.

7 Related Work

Our techniques leverage prior work in related areas, including support for software checkpointing, thread-level speculation (TLS), transactional memory (TM) and program back-tracking.

Checkpointing. Checkpointing is a process of taking program snapshots to facilitate later recovery. Feldman *et al.* [8] present the IGOR system capable of conducting full-process checkpointing, optimized for checkpointing only dirty pages. King's time-traveling VM [15] discusses an OS-level debugging facility by checkpointing entire OS states into disk files. Fine-grain refinement includes both undo-log and redo-log, to reach any specific program location between two consecutive full checkpoints. Xu *et al.* [32] demonstrate a re-tracing tool that

[3] Note that a hashtable always performs a search before any insert, thus the redundancy rate for all hashtable-based checkpoint buffer implementations is always 0.

uses VMWare's deterministic replay technique to collect only non-deterministic events during program execution and later expanding the collection into full program traces using replay. In contrast to existing approaches that checkpoint entire VM or application, we checkpoint on a per-store granularity to memory within a single application – a fine-grain checkpointing scheme that hasn't received much attention.

Speculation. Thread-level speculation [10,29] (TLS) and Transactional Memory [11,12,28] (TM) are optimistic program execution whose result might not be needed. TLS and TM approaches provide for each optimistic thread the ability to checkpoint and rollback, although this support is also intertwined with support for tracking and detecting inter-thread conflicts. Hardware buffering support for hardware TLS and TM implementations has the challenge that it can overflow. Software implementations can be less limited in buffer capacity, but suffer from high instrumentation overheads. In contrast to most TM or TLS solutions that using hardware buffering for multi-threaded workload, we instead focus on using software buffering for single-thread application. We further leverage compiler optimizations to aggressively reduce checkpointing overhead.

Program Back-Tracking. Debugging often requires revisiting passed program state while trying to locate the root cause of a bug. A checkpoint-enabled debugger can greatly simplify the debugging process by eliminating the need for program restart to look backwards. Agrawal *et. al.* [2,3] presented a prototype debugging tool that is based on dynamic program slicing and execution back-tracking—it provides a structured view of dynamic events through runtime traces, but is constrained by storage limitations. Recent versions of `gdb` [9] allow inverse execution by conducting program replay, but are limited to one million instructions. In contrast, our checkpointing scheme allocates its buffer in main memory so that it can grow dynamically. This allows a checkpoint region of relatively unbounded size and complexity. We expose the checkpointing functionality to the user, so that programmers can have explicit control of rewind by issuing debugger commands, to help reduce develop-run-debug cycle time and improve productivity.

8 Conclusion

We have designed, implemented and evaluated a comprehensive checkpointing framework that automatically enables software-only checkpointing over any user-specified source program region. In this paper, we presented compiler analyses and transformations that enable and optimize user-level checkpointing over programs of arbitrary size and complexity, and demonstrated that compiler optimizations are effective at eliminating checkpointing overhead. In particular, they reduce checkpoint buffer size by up to 98% and remove up to 95% of redundant `backup` calls. We showed that by leveraging our checkpointing framework, a debugger can conduct unlimited retries of execution rewind over arbitrarily large regions. We also showed that we can enable automatic back-tracking,

with a moderate performance overhead of only 15% for VPR's place-and-route algorithm.

Future Work. We plan to enhance our checkpointing API by allowing users to specify non-checkpointable code within a checkpoint region; this will have an immediate use for VPR because users will gain manual control within an otherwise automatically-checkpointed region. Redundancy rates remain high for a few applications after all optimizations due to extensive use of pointers, hence we plan to develop deep pointer analyses to better understand such pointer behaviors and help to further reduce checkpointing overhead. We also plan to extend our framework with multi-threading support, including evaluating a write-buffer approach.

References

1. Adl-Tabatabai, A., Lewis, B.T., Menon, V.S., Murphy, B.R., Saha, B., Shpeisman, T.: Compiler and runtime optimizations for efficient software transactional memory. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI (2006)
2. Agrawal, H., Demillo, R., Spafford, E.: An execution-backtracking approach to debugging. IEEE Transactions on Software (May-June 1991)
3. Agrawal, H., Demillo, R., Spafford, E.: Debugging with dynamic slicing and backtracking. Software: Practice and Experience (October 2006)
4. Akkary, H., Rajwar, R., Srinivasan, S.: Checkpoint processing and recovery: An efficient, scalable alternative to reorder buffers. IEEE Computer Society (2003)
5. Betz, V., Rose, J.: Vpr: A new packing, placement and routing tool for fpga research. In: VPR: A New Packing, Placement and Routing Tool for FPGA Research (1997)
6. Betz, V., Rose, J., Marquardt, A.: Architecture and cad for deep-submicron fpgas. Kluwer Academic Publishers (February 1999)
7. Elnozahy, W., Johnson, D., Zwaenepoel, W.: The performance of consistent checkpointing. In: 11th Symposium on Reliable Distributed Systems, pp. 39-47 (October 1992)
8. Feldman, S.I., Brown, C.I.: Igor: A system for program debugging via reversible execution. In: ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging (1989)
9. Free Softwar Foundation. Gdb: the gnu debugger manual 7.0 (September 2009)
10. Hammond, L., Willey, M., Olukotun, K.: Data speculation support for a chip multiprocessor. In: ACM SIGOPS Operating Systems (December 1998)
11. Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional memory coherence and consistency. In: CM SIGARCH Computer Architecture News (March 2004)
12. Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N.: Software transactional memory for dynamic-sized data structures. In: The Twenty-Second Annual Symposium on Principles of Distributed Computing (2003)
13. Hwu, W., Patt, Y.: Checkpoint repair for out-of-order execution machines. In: Computer Science Division. ACM, University of California at Berkeley (1987)
14. Jagadish, H.V., Silberschatz, A., Sudarshan, S.: Recovering from main-memory lapses. In: Procs. of the International Conf. on Very Large Databases, VLDB (1993)

15. King, S.T., Dunlap, G.W., Chen, P.M.: Debugging operating systems with time-traveling virtual machines. In: Annual USENIX Technical Conference (2005)
16. Kingsley, G., Beck, M., Plank, J.: Compiler-assisted checkpoint optimization using suif. In: First SUIF Compiler Workshop (1995)
17. Lattner, C., Adve, V.: Llvm a compilation framework for lifelong program analysis and transformation. In: Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO) (March 2004)
18. Lattner, C., Adve, V.: The LLVM Compiler Framework and Infrastructure Tutorial. In: Eigenmann, R., Li, Z., Midkiff, S.P. (eds.) LCPC 2004. LNCS, vol. 3602, pp. 15–16. Springer, Heidelberg (2005)
19. Li, C., Stewart, E., Fuchs, W.: Compiler-assisted full checkpointing. Software-practice and Experience 24(10), 871–886 (1994)
20. Lu, S., Li, Z., Qin, F., Tan, L., Zhou, P., Zhou, Y.: Bugbench: Benchmarks for evaluating bug detection tools. In: Workshop on the Evaluation of Software Defect Detection Tools (2005)
21. Mcdonald, A., Chung, J., Carlstrom, B.D., Minh, C.C., Chafi, H., Kozyrakis, C., Olukotun, K.: Architectural semantics for practical transactional memory. Computer Architecture News (2006)
22. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A.: Logtm: Log-based transactional memory. In: High-Performance Computer Architecture (2006)
23. Eliot, J., Moss, B.: Log-based recovery for nested transactions. In: Proceedings of the 13th International Conference on Very Large Data Bases (1987)
24. Ng, W., Chen, P.: The symmetric improvement of fault tolerance in the rio file cache. In: Proceedings of 1999 Fault Tolerance Computing, FTC (1999)
25. Plank, J., Beck, M., Kingsley, G.: Compiler-assisted memory exclusion for fast checkpointing. In: IEEE Technical Committee on Operating System and Application Environments, Special Issue on Fault-Tolerance (1995)
26. Plank, J.S., Beck, M., Kingsley, G., Li, K.: Libckpt: Transparent checkpointing under unix. In: Usenix Winter Technical Conference (1995)
27. Chandra, S.: An evaluation of recovery related properties of software faults. Ph.D. thesis (2004)
28. Saha, B., Adl-Tabatabai, A.-R., Hudson, R.L., Minh, C.C.: Mcrt-stm: A high performance software transactional memory system for a multi-core runtime. In: Principles and Practice of Parallel Programming, PPOPP (2006)
29. Gregory Steffan, J., Colohan, C.B., Zhai, A., Mowry, T.C.: A scalable approach to thread-level speculation. In: International Symposium on Computer Architecture (ISCA) (June 2000)
30. Wang, Y., Huang, Y., Vo, K., Chung, P., Kintala, C.: Checkpointing and its applications. In: 25th Int. Symp. On Fault-Tol. Comp., pp. 22–31 (June 1995)
31. Whaley, J.: System checkpointing using reflection and program analysis
32. Xu, M., Malyugin, V., Sheldon, J., Venkitachalam, G., Weissman, B.: Retrace: Collecting execution trace with virtual machine deterministic replay. In: 3rd Workshop on Modeling, Benchmarking and Simulation (2007)

VMAD: An Advanced Dynamic Program Analysis and Instrumentation Framework

Alexandra Jimborean, Luis Mastrangelo,
Vincent Loechner, and Philippe Clauss

INRIA Nancy-Grand Est (CAMUS), LSIIT,
University of Strasbourg, CNRS, France
{firstname.lastname}@inria.fr

Abstract. VMAD (Virtual Machine for Advanced Dynamic analysis) is a platform for advanced profiling and analysis of programs, consisting in a static component and a runtime system.

The runtime system is organized as a set of decoupled modules, dedicated to specific instrumenting or optimizing operations, dynamically loaded when required. The program binary files handled by VMAD are previously processed at compile time to include all necessary data, instrumentation instructions and callbacks to the runtime system. For this purpose, the LLVM compiler has been extended to automatically generate multiple versions of the code, each of them tailored for the targeted instrumentation or optimization strategies. The compiler chooses the most suitable intermediate representation for each version, depending on the information to be acquired and on the optimizations to be applied. The control flow graph is adapted to include the new versions and to transfer the control to and from the runtime system, which is in charge of the execution flow orchestration.

The strength of our system resides in its extensibility, as one can add support for various new profiling or optimization strategies, independently of the existing modules. VMAD's potential is illustrated by presenting several analysis and optimization applications dedicated to loop nests: instrumentation by sampling, dynamic dependence analysis, adaptive version selection.

1 Introduction

Runtime code analysis and optimization becomes the main strategy for facing the ever extending and changing variety of processor architectures and execution environments that an application can meet. Unlike static compilers, that have to take conservative decisions from restricted information extracted from the source code, runtime profilers and optimizers rely on information captured at execution time. While today's processors provide more and more computing resources at the price of increasing usage complexity, particularly with the advent of multicore processors, efficient program optimizations such as adaptive and speculative parallelism, require accurate and advanced runtime analyses. However, such analyses inevitably incur time overhead that has to be minimized.

M. O'Boyle (Ed.): CC 2012, LNCS 7210, pp. 220–239, 2012.
© Springer-Verlag Berlin Heidelberg 2012

In this paper, we present a framework called VMAD, that includes a virtual machine handling x86_64 binary files, tailored at compile time thanks to dedicated passes developed in the LLVM compiler [22]. One of VMAD's specific feature is that low level profiling is initiated from the source code by the programmer through the insertion of a dedicated *pragma*. The inserted pragmas delimit the regions of the source code of interest and specifies the type of analysis to be performed at runtime. This approach provides the programmer a direct view of the actual execution behavior of his source code. We have extended the LLVM compiler and the Clang front-end to handle such pragmas.

VMAD has the ability of dynamically loading separated modules that implement various analysis strategies. The modules are loaded and unloaded on demand, and several instances of the same module can be loaded simultaneously to handle different code regions. Profiling by sampling, as well as code transformations, are achieved using the same mechanism of multi-versioning and chunking. At compile-time, several versions of the targeted code are automatically generated depending on the semantics of the pragma. For instrumenting by sampling, original and instrumented versions are prepared, while for optimizations, a pattern that can support various code transformations is built. Optimizations are usually preceded by a profiling phase. In this respect, the multiple versions are prepared such that they can be launched successively in chunks of code of various sizes. For example one chunk might represent a subset of iterations of a loop, or a number of function calls. The runtime system then takes the decision concerning the version to be run and adjusts the chunk size correspondingly.

To show VMAD's potential, we present some advanced analysis processes. The first one consists in collecting all memory addresses that are accessed during a selected number of successive iterations of each loop of a loop nest. This instrumentation is rather specific since it occurs on non-contiguous phases of the loop nest execution. The analysis process tries to interpolate addresses successively accessed through each memory reference as a linear function. The second application extends the previous one by performing a dynamic dependence analysis: when all accessed memory addresses can be represented as linear functions, these are transmitted to a dependence analysis module which determines if the loop nest may be parallelized. Finally, the third application is a runtime version selector handling distinct versions of loop nests generated by applying different optimizations. Samples of each version are launched successively and the performance of each version is evaluated by accessing the CPU time stamp counter. The best version is then run for the remaining iterations.

The remainder of the paper is organized as follows. In Sect. 2, we present an overview of our static-dynamic framework. The runtime component is detailed in Sect. 3, and the static preparation of the code is presented in Sect. 4. The loop analysis processes implemented in VMAD are further described in Sect. 5. Finally, we summarize related work in Sect. 6 and conclude in Sect. 7.

2 Framework Overview

VMAD has been built by taking great care of its performance and its runtime overhead. Hence, we avoided the use of software dynamic translation that would delay the execution of the input program. Further, instrumentation instructions are not inserted on-the-fly by replacing some NOP instructions that have been previously inserted at compile time, as done with PEBIL [21]. Rather, we use multi-versioning: several copies are built from the targeted code extracts at compile time. The price to pay with this approach is the larger size of the program binary file. However, great care can also be taken to minimize the size of the copies by inserting branches to the original code whenever possible. Besides performance, another noticeable benefit is the opportunity of implementing advanced analyses which can use versions far different from the original code.

The static-dynamic collaborative framework is depicted in Fig. 1. At compile time, the C/C++ source code, annotated with dedicated pragmas, is translated into the LLVM intermediate representation (IR) with additional specific metadata. An LLVM pass creates copies of the targeted code extracts, sometimes customized with instrumentation instructions. Depending on the type of information intended to be captured during the analysis phase, instrumentation instructions may be inserted either in the LLVM IR, for high-level information, or in the final assembly code, for low-level information. As an example, tracing memory accesses in LLVM IR is not possible, as register allocation is not yet done at this level.

Besides instrumentation instructions, we insert *decision blocks*, providing the means to toggle between versions. We also insert callbacks, in order to invoke VMAD and its related modules when necessary. To be generic, callbacks are inserted as indirect calls and the address of the function to be called is patched at start-up by the virtual machine.

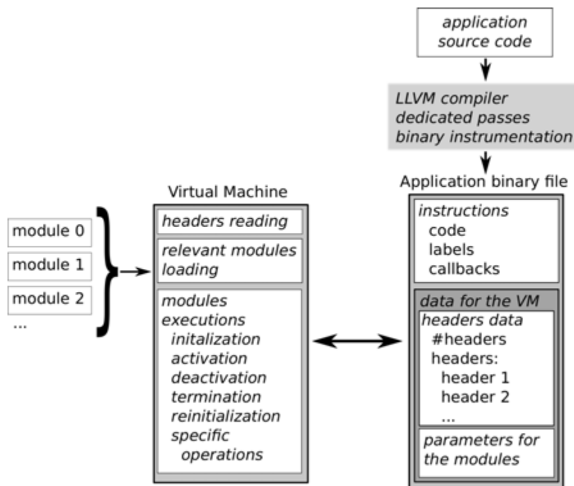


Fig. 1. Framework overview

Moreover, the compiler inserts data in the final binary file to inform the VM regarding the analysis process to be performed and to provide the necessary static information.

At runtime, we use `LD_PRELOAD` to load VMAD's dynamic shared library at startup. `LD_PRELOAD` provides its own version of the C-library entry point *__libc_start_main*, allowing VMAD to read the information statically prepared, to load the required modules and to patch the binary file. Next, control is given to the input program. When necessary, VMAD is reactivated through the callbacks.

3 The Virtual Machine VMAD

The virtual machine makes use of three kinds of information inserted at compile time in the program binary file: instrumentation code that has been inserted before or after original instructions in order to monitor their run; decision blocks to control the selection of the versions; static data corresponding to module parameters, and pointers to the addresses of the inserted code and the callbacks.

Each analysis collaborates with a dynamic module. They share information using a fixed size header which resides inside the binary file. At startup, as soon as the headers and their associated parameters have been read, the VM loads the relevant modules and instantiates them. Parameters fetched at this point are *static data*, and are accessed either by the VM, in order to know which modules are required, or by the modules loaded by VMAD, for instance for obtaining the addresses of the code snippets that have to be patched, or by instrumenting instructions, for allocating memory to backup registers. On the other hand, *dynamic data* is accessed through pointers set up by the corresponding VM modules which allocate the memory they require.

Each module is structured with at least five main entry points: *init* to instantiate an analysis process, *quit* to kill such an instance, *on*, *off* and *reset* to activate, deactivate and reset an analysis process. Additional operations can also be provided by a module. They are invoked thanks to callbacks patched initially by *init* in order to point to the relevant instance of the module and operation. These callbacks are inserted at some control points in the program binary file, as detailed in Sect. 4 and Sect. 5, and have the common form shown in Fig. 2.

4 Preparing the Code at Compile Time Using LLVM

Code analysis and optimization starts, in our approach, at the level of the original source code, where the programmer guards interesting regions of code with a specific pragma. The code is then statically shaped to enable the analysis phase, by performing the following steps, detailed below: code tracking, multi-versioning, customizing the versions for instrumentation or optimization, inserting callbacks to the VM, and appending static information required by the VM. An extra challenge is taken when a high optimization level is applied, such as -O3, due to the aggressive code transformations being performed. We first optimize the code

```
sub     $0x80,%rsp     // backup the stack red zone
// backup the scratch registers here...
mov     %rsp,%rbp      // stack adjustement (x86_64 convention)
mov     $0xfffffffffffffff0,%rsi
and     %rsi,%rsp
mov     $0x0,%rax      // x86_64 convention
mov     $0x0,%rdi      // address of the module ($0x0 will be patched)
mov     $0x0,%rsi      // address of the operation ($0x0 will be patched)
callq   *%rsi          // function call
mov     %rbp,%rsp      // stack readjustement
// restore the scratch registers here...
add     $0x80,%rsp     // restore the stack red zone
```

Fig. 2. callback in x86_64 assembly code

(O3), then we generate multiple versions and run a few optimization passes to optimize them, without altering the code which requires patching.

Code tracking. Our work relies on the LLVM compiler and on the Clang front-end, which must be extended in order to handle the newly defined pragma. The semantics of the pragma and the delimited region must be preserved from the source code, through the internal phases of the compiler, until the code generation. The original source code is converted into the LLVM IR, annotated with metadata information to mark the code enclosed in the pragma scope. The difficulties of tracking code throughout optimization phases is that metadata information is not entirely preserved, and that code suffers significant transformations. For instance, if one marks the instructions building up a loop and performs loop optimizations, additional code is included (*e.g.* due to loop fusion) or excluded (*e.g.* loop invariants, loop split) from its original body. Therefore, identifying all original instructions is not always possible. Focusing on loops, the conservative solution we propose is to consider that the original loop is transformed into the code region containing

- all loops that include ...
    - at least one basic block containing ...
        - at least one instruction that carries metadata

The consequence is that more code than the one originally marked for multi-versioning might be considered. However, in this manner, we ensure that all instructions of the targeted code region are safely enclosed.

Multi-versioning. Once the region is identified, several clones are generated. We designed an LLVM pass which creates clones of these regions and builds a selection mechanism. The steps to follow for building multiple versions are described in Fig. 3. Using the LLVM copying utilities, we build clones of the instructions found in the region. By default, the clones represent identical copies of the original values. To restore the control flow graph between the copies, a map of the original values and the corresponding clones needs to be built. Based on it, we replace all uses of the original values inside the cloned region, with the corresponding cloned value, thus obtaining a clone of the entire region, as

depicted in Fig. 3B. Finally, each version is extracted into a new function and the selection mechanism is inserted.

Each code version is converted into a suitable intermediate representation, depending on its objectives. Versions created for high level code analysis and optimizations are preserved in the LLVM IR, while versions targeting low-level information are transformed into x86_64 assembly code. For instance, tracing memory behaviour is a difficult task in LLVM IR, because register allocation is not available at this compilation step, and because LLVM IR is in SSA form, containing Φ-nodes. The number of "load" and "store" instructions from LLVM IR is greatly reduced by the optimizers during the code generation, hence they do not represent actual memory accesses. A lower level representation, such as x86_64 assembly, is required for this type of instrumentation. On the other hand, for performing dependence analysis, it suffices to track the "load" and "store" LLVM instructions since it is not relevant for this purpose whether a register or a memory location from the internal memory is accessed.

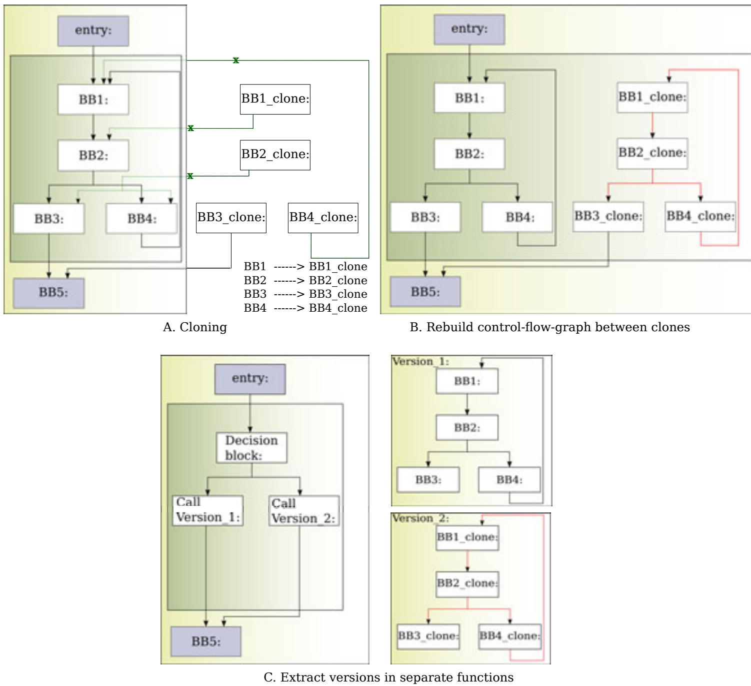


Fig. 3. Multi-versioning

Converting regions of code into different intermediate representations justifies the necessity of extracting the versions in separate functions, further compiled and processed independently. Also, in this manner, we have a clean separation between the regions marked for multi-versioning and the embedding code, from the source level to the LLVM IR and till the x86_64 assembly form. The clones

represented in the LLVM IR are inlined in the original code once they have been customized, to reduce the runtime overhead, but this is not possible for the versions in x86_64 assembly representation, due to register allocation. Nevertheless, the overhead incurred by extracting the versions in separate functions is negligible in most situations.

Chunking. The applications we currently propose for evaluating our framework regard analysis and instrumentation of loop nests, as well as dynamic selection between different optimized versions of loop nests. For these purposes, we build chunks of successive iterations by inserting a virtual iterator vi and new conditions $lowerBound \leq vi < upperBound$ in each loop, in order to manage the number and the position of the executed iterations, relatively to the preceding executed chunk. Under these circumstances, sampling is achieved by executing a chunk of the instrumented version, and then switching to the original version. More generally, our strategy provides the means to switch between different versions of a loop nest while completing its execution. After the execution of each chunk, based on the information registered from instrumentation, the VM is invoked to perform additional computations and to guide the decision concerning the next chunked version to be executed. By adjusting the values of the $lowerBound$ and $upperBound$ dynamically, one can control the sampling rate, restart instrumentation when necessary, and, more generally, can vary the chunk size and the occurring frequency of any version.

Invoking the VM. The selection mechanism consists in preceding each set of versions by a *decision block*, which invokes the VM to decide upon the version to be executed. Additional callbacks to the VM (Fig. 2) are performed at runtime to transmit the data collected via instrumentation, as presented in section 3. The callbacks are placed statically at the beginning and end of each version, and for loops, in addition, they mark the beginning and the end of each instrumented iteration.

Following a generic approach, all callbacks have a standard form, using indirect calls. This approach enables the compiler to generate multiple versions in a generic form, and relies on the VM to patch the address of the corresponding function, at runtime.

Since we patch the code dynamically, our strategy is to inline these code snippets in x86_64 assembly code, ensuring that the size of the code to be patched is fixed. The inline code contains new labels and jumps and the callbacks to the VM. First, this ensures that the VM can track the position of the callbacks in the binary file by using the addresses of the labels. And second, it prevents the modification of the code snippets in the last phases of the code generation, as all jumps and callbacks are inserted in fixed size hexadecimal representation. What we obtain is a partial control flow graph managed as x86_64 assembly code, inlined in the LLVM IR. On the other hand, the inline code is not accessible to the LLVM compiler in this compilation phase. As a consequence, to preserve the validity of the LLVM IR code, we have to maintain both the original CFG, represented as LLVM branches, as well as the jumps inserted in the inline assembly code. The CFG expressed in inline code is the one actually executed, however,

the LLVM branches are preserved to avoid compile time errors, or false cases of dead-code elimination.

Finally, handling inline assembly code throughout the optimizations requires considerable efforts to protect the inlined code against duplications, relocations or dead code eliminations, and to minimally perturb the optimizing passes.

The steps presented above, for statically processing the code, are independent of the type of profiling or optimization to be performed. Next, the clones are modified following the individual specifications and a list of parameters is appended to transmit relevant static information to the VM.

Customizing the Versions. Each function containing a code version is customized according to the type of instrumentation or optimization, either by inserting snippets of instrumenting code, or by performing various code transformations. This is the step when the most suitable intermediate representation is selected. For the examples we address in this paper, various representations are preferred. For tracking the memory locations and building interpolating linear functions, we insert the instrumenting instructions in the final assembly code, after the register allocation. For performing dependence analysis, we instrument the memory accessing instructions at the LLVM IR level and verify pair-wise dependences of load and store instructions. Finally, to perform runtime version selection, for the purpose of this example, the versions are embedded in the source code. The application benefits on the possibility of executing samples of each version and evaluate them using processor counters. The code snippets for evaluating the performance of each version are automatically inserted in the LLVM IR.

Inserting Static Information. In addition to preparing the code, a set of headers and parameters is annexed to the generated binary code (Fig. 4). The list of headers is specific to the type of instrumentation, as they determine the modules to be loaded in the VM (*vmad_0_entry*, *vmad_loop_entry*). Headers are linked to the corresponding parameters (*vmad_0_param*), containing higher level information statically available, but which would be time-expensive to identify in the binary representation (for example, the loop depth). Furthermore, the compiler transmits as parameters instrumentation specific information, for instance the addresses of the code snippets inserted in the original code (*vmad_0_loop_reinstru*, *vmad_0_instru_call*).

5 Illustrating Applications

5.1 Analyzing Memory Accesses in Loop Nests

The first application we propose for evaluating VMAD consists in instrumenting and profiling the memory accesses of critical pieces of code, with a minimal time overhead. In our examples, the target is non-statically analyzable code. We focus on loop nests, as they represent a significant part of the total execution of compute-intensive applications. The goal of the instrumentation framework

```
# number of headers.
.global vmad_headers_nb
 vmad_headers_nb:
        .long   1
```

```
#list of headers
. global vmad_headers
 vmad_headers:
. global vmad_0_entry_lb
 vmad_0_entry_lb:
        .quad   vmad_0_entry
        .quad   vmad_loop_entry
        .quad   vmad_0_param
```

```
#list of parameters
. global vmad_0_param
 vmad_0_param:
     .long  3          # loop depth
     .quad  vmad_0_end_original
     .quad  vmad_0_loop_reinstru
     .quad  vmad_0_instru_call
```

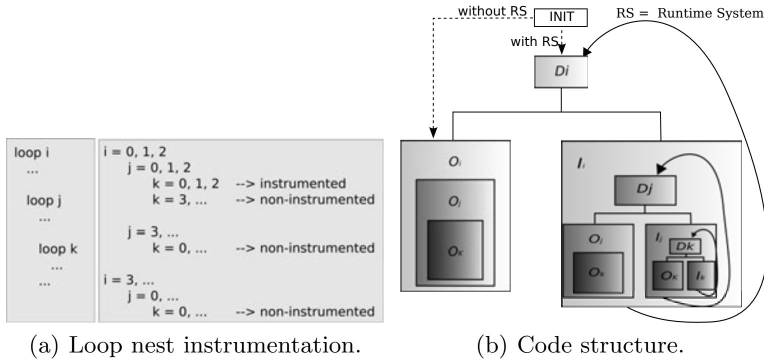
Fig. 4. Headers and parameters

is to collect the memory addresses accessed during samples of iterations and, if possible, to compute linear functions interpolating their values. Additionally, the loop trip counts of each loop, except the outermost ones, are collected for a few runs to be linearly interpolated. Such a profiling is particularly useful for nested loops, either *while*-, *for*- or *goto*-loops, accessing memory through indirect references or pointers. If the memory accesses and the loop bounds can be expressed by means of linear functions, the enclosing loop nests can be optimized and parallelized using *for*-loop dedicated approaches, such as the polyhedral model [4,5]. To handle all loop types in the same manner, we introduce "virtual" iterators, which are maintained to mirror the number of executed iterations.

Loop Nest Instrumentation. Efficient loop nest instrumentation by sampling consists in profiling only a subset of the executed iterations of each loop. The complexity of the method is outlined in the case of nested loops, as instrumentation depends not only on the iteration of the current loop, but also on the parent loops. For a thorough understanding, consider the loop nest in Fig. 5(a). In this example, the first three iterations of each loop are instrumented. One may easily notice that instrumented and non-instrumented iterations alternate, hence the execution has to switch from one code version to another at runtime. Once the outermost loop profile has been completed, the execution can continue with a non-profiled version of the loop nest, thus inducing no overhead for the remaining iterations.

For linear interpolation of memory accesses, each memory instruction should be profiled during the execution of at least three iterations, in order to get sufficient address values. However, since some memory instructions can be guarded by conditional branches, it is required to profile such instructions for more iterations, to increase the chances of collecting enough, *i.e.*, at least three, address values. This contributes to the accuracy of the computed interpolating functions. In our experiments, we fixed the number of instrumented iterations to 10, which was a good trade-off between overhead and accuracy. The sampling rate can be set by a parameter. The first two collected values are dedicated to computing the affine function coefficients, while the remaining values are used to verify the interpolation correctness.

Statically, our LLVM pass creates copies of the loop nests, extracts them in new functions and converts them to x86_64 assembly code. A second pass analyzes the functions and precedes each instruction accessing memory with instrumentation code that computes the actual memory location being accessed,



(a) Loop nest instrumentation. (b) Code structure.

Fig. 5. Instrumenting loop nests

and makes a call to the VM to transmit the collected data. Fig. 5(b) illustrates the structure of the code from Fig. 5(a) and the links between different versions. Blocks O_i, O_j and O_k represent the original versions, while I_i, I_j and I_k represent the instrumented bodies of each loop. The instrumented and original versions are connected together at their entry point, where a choice is made at runtime deciding which version to run, based on the values of the virtual iterators. One decision block is associated to each loop, represented by D_i, D_j and D_k, correspondingly, containing a callback to the VM. The VM is also invoked when entering or exiting a version of the loop, to retrieve dynamic information. At compile time, we mark the beginning and end of the original and instrumented versions with labels, and append them to the list of parameters given to the VM.

Lastly, a list of headers and parameters is prepared, notifying the VM which are the modules required for this instrumentation: module *vmad_handle_loop*, *vmad_gather_memory_addresses* and *vmad_interpolation*. One instance of each of these modules is created per loop, at runtime. The first module encloses the mechanisms necessary for handling loops, the second one collects the memory accesses performed inside the loops, while the last module performs the interpolation. Each module uses the information from the previous one to complete its task, however, they are decoupled, hence each module may be employed in performing new types of analysis. The list of parameters contains specific information, such as addresses of the code to be patched at startup, or the structure of the loop nest. At startup, the VM parses the list of headers, loads the solicited modules and patches the code to enable instrumentation.

Analyzing Memory Accesses. Since the number of memory locations accessed inside loops can be very high, considering a memory intensive loop nest, it is recommended that the acquired data is processed immediately by the interpolation process, rather than stored for a later utilization.

For each instrumented loop, a buffer is created at compile time, to (re)store the state of the machine before the interpolation process. At runtime, the VM allocates space to be populated dynamically with the accessed memory locations and to store the coefficients of the linear functions.

As the instrumented iterations of a loop are executed, the VM reads the values of the memory locations from the designated buffer and the corresponding function coefficients are computed and stored in the associated positions. Subsequent instrumented iterations are used to verify the linearity of these functions.

Communication with the VM is achieved by means of a dirty flag, which indicates that a new memory location is available in the buffer.

Experiments. For our experiments, we targeted the C codes from the SPEC CPU 2006 benchmark suite [29] and four codes from the Pointer-Intensive benchmarks [27]. We inserted a dedicated pragma in the source codes, marking the loop nests (`#pragma instrument_mem_add { // loop nest}`) in the most time consuming functions [31]. We ran the benchmarks using the `ref` input files to compute VMAD's runtime overhead, and using the `test` input files to get output files with the interpolation results, since runs using the `ref` files produce an amount of data too large to be stored on the disk, but suitable for online consuming. We have carried out the experiments using the O0 and the O3 optimization levels. The execution platform is a 3.4 Ghz AMD Phenom II X4 965 micro-processor with 4GB of RAM running Linux 2.6.32. We ran each program in its original form and in its instrumented form to compute the runtime overhead introduced by using VMAD. For each instrumented loop nest, the dynamic profiling is activated each time its enclosing function is invoked, for the experiments using O0 optimization level. In the experiments with a higher optimization level (O3) we instrument the first eight calls of each function.

Our measurements are shown in Tab. 1. The columns show for each program: the program name (first part of the table: SPEC CPU 2006, second part: Pointer-Intensive); VMAD's runtime overhead, both with O0 and with O3; the code size increase; the number of instructions performing linear memory accesses; the number of instrumented memory instructions; the percentage of memory accesses that were identified to be linear.

For most programs, VMAD induces a very low runtime overhead, which is even negligible for `bzip2`, `milc`, `hmmer`, `h264ref` and `lbm`. For the programs `sjeng` and `sphinx3`, the significant overheads are mainly due to the fact that the instrumented loops execute only a few iterations, but they are enclosed by functions that are called many times (with O0). Thus, all iterations are run while being fully instrumented. However, the profiling strategy is improved in order to manage such cases by deactivating the instrumentation after a few calls (with O3). Program `milc` shows an opposite behavior since a few memory instructions are executed many times. In such a case the runtime overhead is very low. For the Pointer-Intensive benchmarks, the execution times are too small – of the order of milliseconds – to get relevant overhead measurements: either a large runtime overhead is obtained since VMAD inevitably induces a fixed minimum overhead (`bc`), or even a speedup is obtained (`ft`), which may be explained by cache locality, new alignments or new optimization opportunities. The overhead is higher when instrumenting optimized code (-O3), since modifying optimized code impacts its performance, still, the execution time is better than with O0.

Table 1. Measurements made on some of the C programs of the SPEC CPU 2006 (first part) and Pointer-Intensive (second part) benchmark suites

Program	Runtime overhead (-O0)	Runtime overhead (-O3)	code size increase	# linear m.a.	# instrumented m.a.	Percentage of linear m.a.
bzip2	0.24%	12.31%	218%	608	1,053	57.74%
mcf	20.76%	17.23%	213%	2,848,598	4,054,863	70.25%
milc	0.081%	3.61%	44%	1,988,256,000	1,988,256,195	99.99%
hmmer	0.062%	0.76%	63%	845	0	0%
sjeng	182%	11.13%	80%	1,032,148,267	1,155,459,440	89.32%
libquantum	3.88%	2.76%	21%	203,078	203,581	99.75%
h264ref	0.49%	4.59%	0.44%	30,707,102	32,452,013	94.62%
lbm	0%	0.93%	170%	358	0	0%
sphinx3	172%	27.62%	20%	51,566,707	78,473,958	65.71%
anagram	-5.37%	34.88%	73%	134	159	84.27%
bc	183%	36.79%	11%	243,785	302,034	80.71%
ft	-8.46%	176%	86%	22	36	61.11%
ks	29.7%	2.98%	268%	29,524	42,298	69.79%

We also noticed that this particular instrumentation process increases the size of a program's binary file by 400 bytes per instrumented memory instruction, on average. However, the code size variation strongly depends on the depth of the loop nests and on the percentage of code selected for instrumentation.

5.2 Dynamic Dependence Analysis

The second application is an extension of the previous one. It adds a module to determine, for a loop nest, which are the loop levels that might be parallelized, according to the memory behavior observed during profiling. Such information can be a useful indication for a developer in order to identify and further analyze such loops, to decide whether they can be effectively parallelized. Our framework identifies the candidate loops by speculatively analyzing dependences between iterations, based on the linear functions interpolating the memory addresses accessed during profiling. The module considers each couple of memory instructions and their associated linear functions, where at least one of them is a write.

We use a simple value range analysis method to determine if the two referenced address spaces can overlap, using the linear functions to compute the minimal and the maximal values of the memory addresses accessed by each instruction. Each write instruction is also considered solely since it can carry an output self-dependence. A loop level not carrying any dependence is then identified as a candidate for parallelization. We used the OmpSCR benchmark suite [26] for our experiments, a set of scientific kernels that are already manually parallelized by the programmer using OpenMP pragmas. Even if these have been deactivated for our runs, they indicate loops being effectively parallel. Loops inside these

Table 2. Dynamic dependence analysis and parallel loop detection in the OmpSCR benchmark suite

Benchmark	#OMP pragmas	#Linear loop nests	#Detected as parallel	Parallel loop levels
FFT	2	2	0	
FFT6	3	10	4	1 / 3 / 1,2 / 1,2
Jacobi	2	4	1	1,2
LUreduction	1	2	2	1,2 / 2,3
Mandelbrot	1	2	1	1
Md	2	2	1	1,2
Pi	1	1	0	
QuickSort	1	2	1	1

kernels contain memory references through pointers, through parameterized array accesses and references to dynamically allocated arrays. Such memory references cannot be handled statically by a compiler. Results are shown in table 2. For two benchmarks, FFT6 and LUreduction, more loop nests than the ones with OpenMP pragmas where detected as parallel. When less parallel loop nests are detected, it is due to dependences induced by reductions.

5.3 Dynamic Version Selection

A loop nest can be optimized using different kinds of transformations such as loop fusion/fission, interchange, skewing, tiling, unrolling, etc. A subset of those transformations can be applied, in different order, or with different parameters (unrolling factor, tile size, ...) to generate distinct versions. Hence many versions can be obtained in this way, and each of them may be the best performing one in some execution contexts, while being slower in some others. Such a phenomenon can occur, for example, when the amount of accessed data generates a lot of cache misses if the computation size exceeds a given threshold. Another case is when the locality of the data accesses depends on some input parameters, or when the control flow traverses costly branches in some circumstances depending on intermediate computations. More exactly, it is a combination of such phenomena that impacts the global performance. Hence, it is in general impossible to predict in advance which version would yield the lowest execution time.

The implemented runtime mechanism consists in first measuring the time per iteration when executing a small chunk of each version, and then running the fastest one for the remaining iterations. Different versions are provided in the source code, delimited by dedicated pragmas. Each version includes an additional condition in the outermost loop, constraining the iterator between a lower and an upper bound, which is required for the chunking mechanism.

At compile-time, the multiple versions are identified and a callback to the dedicated runtime selector module is added, as well as the mechanism to switch between the versions.

The runtime module performs the following operations: for each version, one by one, it sets the chunk bounds such that each new chunk will continue the execution of the previous one, it gets the processor's time stamp counter using the RDTSC instruction, launches the version, gets the new CPU time information, computes the execution time per iteration and stores a reference to the fastest version so far. Finally, when all versions have been evaluated, the fastest version is launched to complete the execution. This naive approach already selects the best version in most cases, but the algorithm can be further refined. Similarly to the sampling rate in the first example, the size of the instrumented chunk can be set by a parameter.

The benchmark programs contain 12 loop nests. The code `2mm` consists of two matrix multiply ($D = A \times B \times C$), `adi` is the ADI kernel provided as an example with the automatic optimizer Pluto [7], `covariance` is a covariance matrix computation, `gemm` is taken from BLAS [6], `jacobi-1d` and `jacobi-2d` are the 1D and 2D versions of the Jacobi kernel, `lu` is a LU decomposition kernel, `matmul` is a simple matrix multiply, `matmul-init` is a matrix multiply combined with the initialization of the result matrix, `mgrid` is a kernel extracted from the `mgrid` code in SPECOMP [3] and `seidel` is a Gauss-Seidel kernel also provided with Pluto.

Such loops are good candidates for loop optimizations such as skewing, loop interchange or tiling. We generated 6 or 7 different versions for each benchmark, either using Pluto or manually. Some versions are tiled, some others are tiled two times in two levels, some others are just skewed or their loops have been interchanged, and finally some are the result of a combination of these transformations. All versions, as well as VMAD's code selector, have been run on a Intel Xeon W3520 at 2.67Ghz under Linux 2.6.38. Results are shown in table 3. For each benchmark, it shows the execution time of the best and of the worst version, the average execution time of all versions, the time when executing with VMAD, and finally a comparison between VMAD and the average execution time.

In most cases, VMAD selects the best version and its execution time is close to the best execution times, and very far from the worst ones. Although it does not select the best version in all cases, it still selects one of the best ones. The overhead is higher when some versions are very slow compared to others.

5.4 Other Possible Applications

In addition to the examples presented above, the VMAD platform can find its applications in debugging or instrumentation, distributed among multiple users. Thanks to the sampling approach and the multiple versions, the selection mechanism can be adjusted such that the version chosen for execution differs from one user to another. Moreover, each version contains only a subpart of the instrumenting or debugging instructions, which ensures a very low overhead, but together, the instrumentation inserted in all versions cover the entire targeted code. Distributed debugging or instrumentation becomes attractive when there is a high number of testers, as each version is executed at least by one user. Moreover, since the overhead is negligible, users are not hindered from executing the versions multiple times. On the other hand, when overhead is not a concern,

Table 3. Dynamic code selection with VMAD

Benchmark	#Versions	Best exec. time	Worst exec. time	Average exec. time	VMAD exec. time	Gap to the average version
2mm	6	2.68	19	8.29	4.80	-42.09%
adi	7	32.99	34.17	33.24	33.10	-0.42%
covariance	6	9.71	145.55	55.81	17.54	-68.5%
gemm	6	7.21	57.10	15.79	9.94	-37.04%
jacobi-1d	6	8.34	11.05	9.70	9.72	0.2%
jacobi-2d	6	2.74	5.24	4.12	4.22	2.42%
lu	6	3.94	51.26	12.11	6.31	-47.89%
matmul	7	4.96	31.49	16.90	6.96	-58.81%
matmul-init	6	3.29	27.04	7.38	4.72	-36.04%
mgrid	6	11.58	16.50	13.45	13.03	-3.12%
seidel	6	76.59	87.71	85.07	86.66	1.86%

the framework can be employed for fully tracing the behavior of the code. This can be achieved by setting the chunk size to a maximal value and selecting the instrumented version.

6 Related Work

VMAD's goal is to be a generic platform running advanced low-level analyses of programs that are initiated from the source code. To our knowledge, there are no previous works directly comparable. However, VMAD can still be related to frameworks that are similar in some important aspects: code instrumentation, code tracking, code cloning and multi-versioning. We also reference a few proposals related to our illustrating applications.

Code Instrumentation. Most of the noticeable code instrumentation tools apply on binary codes. One of the most popular is Pin [23], a software system that performs runtime binary instrumentation. It enables the user to build a wide variety of program analysis tools, known as pintools. A pintool consists of instrumentation, analysis, and callback routines. The insertion of instrumenting instructions is based on software dynamic translation (SDT): a just-in-time compiler recompiles small chunks of binary instructions immediately prior to executing them. Dynamic instrumentation, such as the interpolation of memory accesses in loops presented in this paper, would be impossible to be implemented efficiently with Pin. Of course, the compile-time phase of our framework plays an important role in providing a wider scope of analysis opportunities and in the runtime overhead minimization.

The PEBIL toolkit [21] is more similar to VMAD since it does not use SDT, but static binary instrumentation. PEBIL performs function relocation to acquire enough space at instrumentation points to insert branch instructions at runtime. We use two different strategies to transfer control from the application to the instrumentation code: at compile time, we insert branch instructions

branching initially to the next instruction and that are patched at runtime; we also insert callbacks in the instrumented code snippets that are patched at start-up with the address of the corresponding functions of VMAD.

The above mentioned tools are designed for instrumenting and profiling the code, nevertheless the goal of VMAD goes beyond code analysis. We aim code instrumentation followed by optimization *on the fly*. This emphasizes the need of a mechanism for creating multiple versions of code and switching between them at runtime (multi-versioning). On the contrary, PIN and PEBIL are tailored to instrument the code for the whole execution time. Their advantage is that they provide easy-to-use APIs allowing the programmer to develop new instrumentation tools, with the cost of an increased overhead at runtime. VMAD requires a new LLVM pass and a new module to support additional instrumentation types. In favour of VMAD comes the fact that it is more flexible in handling multiple instrumented or optimized versions simultaneously. It also allows sampling, by enabling/disabling instrumentation at any time. Moreover, the target code delegates instrumentation related tasks, such as processing the acquired information, to the virtual machine.

Code Tracking. Tracking code has always been a necessary technique, evolving from the simple strategies employed in the early debuggers, to complex approaches meant to correlate the original source code with dynamically optimized code. In our framework, tracking the code through the optimization phases plays a key role, both for identifying the region marked for instrumentation in the source code, and the code that must be patched.

Tracking the suite of code transformations performed in the optimization phase has early been identified as an impractical solution, since compilers reorder, replicate, delete, merge, transform the code, eliminate variables or synthesize new ones. A viable alternative is presented by Brooks *et al.* [8] as a method for acquiring extended debugging information, communicated from one optimization phase to another.

More recent and daring work tackling debugging of dynamically optimized code has been reported [17,19]. The challenge consists in discerning between the optimized code and the optimizers dynamically, and to map it back with the source code, which is no longer available at runtime.

In the gcc compiler [12], generating debug information is possible via the option -g. Also, one can control the amount of information transmitted to the debugger by specifying the level, from -g0 to -g3. This option has been implemented in LLVM [22] and in the Clang front-end [10] and the result consists in populating the code represented in LLVM IR with a significant amount of metadata information, which is then transformed into debug information.

We have adopted a similar approach in tracking code from the source level to the intermediate representation, by marking interesting code regions with metadata information.

The next step in performing multi-versioning is cloning, associated with the construction of a selection mechanism.

Cloning, Multi-versioning, Instrumentation by Sampling. Multi-versioning is a widely adopted strategy to reduce the cost of code instrumentation by sampling. A selection mechanism periodically switches execution between a number of versions embedding instrumentation code and the original version. Chilimbi and Hirzel [14,9] add finer control on the sampling rate and eliminate redundant checks to decrement the overhead. They operate directly on the x86 assembly code using Vulcan [11] for capturing sequences of data references (dynamic executions of loads or stores).

An interesting use of sampling is presented by Chilimbi and Hauswirth [13] for checking program correctness. They develop an adaptive profiling where the sampling rate is the inverse of the frequency of execution of each code region. They adapt the framework introduced by Arnold and Ryder [2] to detect memory leaks. Marino *et al.* [24] extend this solution to multi-threaded programs to find data races.

Our goal is to create a static-dynamic framework that supports multi-versioning and sampling, by means of a generic runtime system that patches the code to enable various types of profiling, instrumentations and code optimizations. We plan to extend our work to accommodate all frameworks described above.

Similarly, ICI [16] has been developed with the aim of providing access to the internal functionalities of compilers. Extensions to ICI [15] provide generic function cloning, program instrumentation, pass reordering and control of individual optimizations. Patching is used to insert an event call before and after the execution of each version, either only for transferring information for further processing, or to change the selection decision of the compiler. In these regards, we have a very similar approach, as we insert callbacks to a runtime system to guard the execution of each code version. However, ICI makes multi-versioning available at function call level only, while we target more precise control for example to enable/disable instrumentation at loop level.

Runtime Code Selection. Several studies proposed a runtime selection between various algorithms, or code extracts, or versions of a function. PetaBricks [1] provides a language and a compiler where having multiple implementations of multiple algorithms to solve a problem is the natural way of programming. Mars and Hundt's static/dynamic SBO framework [25] consists in generating at compile-time several versions of a function that are related to different dynamic scenarios. The STAPL adaptive selection framework [30] runs a profiling execution at install time to extract architectural dependent information. In [28], Pradelle et al. propose a framework to select between versions of loop nests resulting from various polyhedral transformations.

Dynamic Dependence Analysis. The analyzer *pp* [20] is one of the earliest work that proposed hierarchical dependence testing to estimate the parallelism in loop nests. Some recent works are Alchemist [32] and SD3 [18] where runtime and memory overhead is reduced through the use of parallelization and compression.

7 Conclusion

In this paper, we presented VMAD, an infrastructure for dynamic profiling, where advanced analyses can be implemented with almost negligible runtime overhead, since it does not use software dynamic translation like most of the dynamic profiling tools. We extended the LLVM compiler to handle specific pragmas allowing the developer to initiate low-level analyses from selected parts of the source code. Dedicated LLVM passes duplicate the targeted code regions into various versions: instrumentation instructions, version selection mechanism, and callbacks are inserted. At runtime, and when activated, the virtual machine of VMAD loads the necessary analysis modules and patches the callback addresses in the application code. To our knowledge, VMAD is the first proposal allowing developers to initiate low-level analyses from the source code.

Regarding the API, there are two scenarios to be emphasized. If one chooses already defined code analyses, the instrumentation process is totally invisible. The only task is to mark the regions of code of interest with a pragma. We underline that the programmer is not required to annotate the source code with callbacks to the VM, nor to write the decision blocks. These code transformations are handled automatically by our framework. On the other hand, for the compilation expert to develop new types of analysis, it is necessary to write an LLVM pass and to add a module in the virtual machine, containing analysis specific operations. Both the pass and the module are programmed in C/C++ and may include inline assembly code.

VMAD's potential has been shown by implementing the following analyses. First, we instrumented memory accesses in a targeted loop nest, by using sampling. The dedicated LLVM passes duplicate each loop into instrumented and non-instrumented versions, and the control switches from instrumented to non-instrumented code until having collected enough data. Then, we implemented an analysis strategy interpolating those memory accesses as linear functions. Using the results of the interpolation, the next application was to perform dependence analysis and offer hints to the programmer regarding the loops which are good candidates for parallel execution. The last application that we implemented is a runtime adaptive version selector, that takes as input several differently optimized code versions, and selects the best performing one. In this respect, a sample of each version is executed and evaluated based on the processor counters, and the best one is selected to execute until the end of the computations.

Our experiments for interpolating memory accesses as linear functions have been conducted on the SPEC CPU 2006 and on the Pointer Intensive benchmark suites. They reveal almost negligible overhead in most cases, of less than 4%, with -O0 optimization level, and varying between 0,5% and 27% with -O3 optimization level. The two other experiments, on different benchmark suites, also show good results.

We plan to extend the framework to support new types of code instrumentation and optimization. For instance, using the results of the data dependence analysis, we target speculative parallelism by generating code on-the-fly.

References

1. Ansel, J., Chan, C., Wong, Y.L., Olszewski, M., Zhao, Q., Edelman, A., Amarasinghe, S.: Petabricks: a language and compiler for algorithmic choice. In: PLDI 2009, pp. 38–49. ACM (2009)
2. Arnold, M., Ryder, B.G.: A framework for reducing the cost of instrumented code. SIGPLAN Notices 36(5), 168–179 (2001)
3. Aslot, V., Domeika, M.J., Eigenmann, R., Gaertner, G., Jones, W.B., Parady, B.: SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In: Eigenmann, R., Voss, M.J. (eds.) WOMPAT 2001. LNCS, vol. 2104, pp. 1–10. Springer, Heidelberg (2001)
4. Banerjee, U.: Loop Transformations for Restructuring Compilers - The Foundations. Kluwer Academic Publishers (1993) ISBN 0-7923-9318-X
5. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT 2004: Proc. of IEEE Int. Conf. on Parallel Architectures and Compilation Techniques (2004)
6. Blackford, L.S., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Kaufman, L., Lumsdaine, A., Petitet, A., Pozo, R., Remington, K., Whaley, R.C.: An updated set of basic linear algebra subprograms (blas). ACM Transactions on Mathematical Software 28, 135–151 (2001)
7. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: PLDI (2008)
8. Brooks, G., Hansen, G.J., Simmons, S.: A new approach to debugging optimized code. In: ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI (1992)
9. Chilimbi, T.M., Hirzel, M.: Dynamic hot data stream prefetching for general-purpose programs. In: PLDI 2002: Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation (2002)
10. Official website of clang: a C language family frontend for LLVM, http://clang.llvm.org
11. Edwards, A., Vo, H., Srivastava, A.: Vulcan binary transformation in a distributed environment. Tech. rep. (2001)
12. The GNU Compiler Collection, http://gcc.gnu.org
13. Hauswirth, M., Chilimbi, T.M.: Low-overhead memory leak detection using adaptive statistical profiling. In: 11th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, ASPLOS-XI. ACM (2004)
14. Hirzel, M., Chilimbi, T.: Bursty tracing: A framework for low-overhead temporal profiling. In: 4th ACM Workshop on Feedback Directed and Dynamic Optimization FDDO4 (2001)
15. Huang, Y., Peng, L., Wu, C., Kashnikov, Y., Rennecke, J., Fursin, G.: Transforming GCC into a research-friendly environment: plugins for optimization tuning and reordering, function cloning and program instrumentation. In: 2nd Int. Workshop on GCC Research Opportunities (GROW 2010), Pisa Italy (2010), Google Summer of Code 2009 (2010)
16. Interactive Compilation Interface, http://ctuning.org/ici
17. Jaramillo, C., Gupta, R., Soffa, M.L.: FULLDOC: A Full Reporting Debugger for Optimized Code. In: SAS 2000. LNCS, vol. 1824, pp. 240–260. Springer, Heidelberg (2000)

18. Kim, M., Kim, H., Luk, C.K.: Sd3: A scalable approach to dynamic data-dependence profiling. In: Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, pp. 535–546. IEEE Computer Society, Atlanta (2010)
19. Kumar, N., Childers, B., Soffa, M.L.: Transparent debugging of dynamically optimized code. In: Int. Symp. on Code Generation and Optimization, CGO 2009. IEEE Computer Society (2009)
20. Larus, J.R.: Loop-level parallelism in numeric and symbolic programs. IEEE Trans. Parallel Distrib. Syst. 4, 812–826 (1993)
21. Laurenzano, M., Tikir, M., Carrington, L., Snavely, A.: PEBIL: Efficient static binary instrumentation for linux. In: ISPASS-2010: IEEE Int. Symp. on Performance Analysis of Systems and Software (2010)
22. LLVM compiler infrastructure, http://llvm.org
23. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: PLDI 2005: Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation (2005)
24. Marino, D., Musuvathi, M., Narayanasamy, S.: Literace: effective sampling for lightweight data-race detection. In: PLDI 2009: Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation (2009)
25. Mars, J., Hundt, R.: Scenario based optimization: A framework for statically enabling online optimizations. In: CGO 2009, pp. 169–179. IEEE Computer Society
26. OmpSCR: OpenMP source code repository, http://sourceforge.net/projects/ompscr
27. Pointer-intensive benchmark suite, http://pages.cs.wisc.edu/~austin/ptr-dist.html
28. Pradelle, B., Clauss, P., Loechner, V.: Adaptive runtime selection of parallel schedules in the polytope model. In: ACM/SIGSIM High Performance Computing Symposium (HPC 2011). ACM (April 2011)
29. SPEC CPU (2006), http://www.spec.org/cpu2006
30. Thomas, N., Tanase, G., Tkachyshyn, O., Perdue, J., Amato, N.M., Rauchwerger, L.: A framework for adaptive algorithm selection in stapl. In: PPoPP 2005, pp. 277–288. ACM (2005)
31. Weicker, R.P., Henning, J.L.: Subroutine profiling results for the CPU2006 benchmarks. SIGARCH Comput. Archit. News 35(1) (2007)
32. Zhang, X., Navabi, A., Jagannathan, S.: Alchemist: A transparent dependence distance profiling infrastructure. In: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2009, pp. 47–58. IEEE Computer Society, Washington, DC (2009)

Sambamba: A Runtime System for Online Adaptive Parallelization

Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack

Saarland University, Saarbrücken, Germany
{streit,hammacher,zeller,hack}@cs.uni-saarland.de

Abstract. How can we exploit a microprocessor as efficiently as possible? The "classic" approach is *static optimization* at compile-time, optimizing a program for all possible uses. Further optimization can only be achieved by anticipating the actual *usage profile*: If we know, for instance, that two computations will be independent, we can run them in parallel. In the *Sambamba* project, we replace anticipation by *adaptation*. Our runtime system provides the infrastructure for implementing runtime adaptive and speculative transformations. We demonstrate our framework in the context of adaptive parallelization. We show the *fully automatic parallelization* of a small irregular C program in combination with our adaptive runtime system. The result is a parallel execution which *adapts to the availability of idle system resources*. In our example, this enables a 1.92 fold speedup on two cores while still preventing oversubscription of the system.

Keywords: program transformation, just-in-time compilation, adaptation, optimistic optimization, automatic parallelization.

1 Introduction

A central challenge of multi-core architectures is how to leverage their computing power for programs that were not built with parallelism in mind—that is, the vast majority of programs as we know them. Recent years have seen considerable efforts in automatic parallelization, mostly relying on *static program analysis* to identify sections amenable for parallel execution (often restricted to small code parts, such as nested loops). There also have been *speculative approaches* that execute certain code parts (identified by static analyses) in parallel and repair semantics-violating effects, if any.

While these efforts have shown impressive advances, we believe that they will face important scalability issues. The larger a program becomes, the harder it gets to precisely identify dependences between code parts statically, resulting in conservative approximations producing non-parallel and overly general code. The problem is that the actual environment and usage profile cannot be sufficiently anticipated [2]. Of course, one could resort to dynamic runtime techniques to determine dependences, but the initial overhead of dynamic analysis so far would not be offset by later performance gains. All of this changes, though, as soon as one moves the analysis and code generation from compile-time to runtime.

M. O'Boyle (Ed.): CC 2012, LNCS 7210, pp. 240–243, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Rather than analyzing and compiling a program just once for all anticipated runs, we can now reanalyze and recompile programs in specific contexts, as set by the input and the environment. Interestingly, it is the additional power of multi-core architectures that makes such a continuous adaptation possible: While one core runs the (still sequential) programs, the other cores can be used for monitoring, learning, optimization, and speculation. Moving from anticipation to adaptation enables a number of software optimizations that are only possible in such dynamic settings. First and foremost comes adaptive parallelization—that is, the execution of the program in parallel depending on the current environment.

2 The *Sambamba* Framework

The *Sambamba*[1] project aims to provide a reusable and extendable framework for online adaptive program optimization with a special focus on parallelization. With *Sambamba*, one will be able to introduce run-time adaptive parallelization to *existing large-scale applications* simply by recompiling; no annotation or other human input is required. Indeed, we aim to make parallelization an optimization as transparent and ubiquitous as, say, constant propagation or loop unrolling.

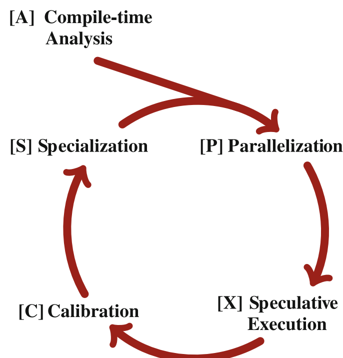


Fig. 1. *Sambamba* execution steps

Sambamba is based on the LLVM compiler framework and consists of a static (compiler) part and a runtime system. The framework is organized in a completely modular way and can easily be extended. Modules consist of two parts: *Compile-time parts* handle costly analyses such as inter-procedural points-to and shape analysis as used by our parallelization module. These results are fed into the *runtime parts*—analyses conducted at runtime which adapt the program to runtime conditions and program inputs. Obviously, it is crucial for the runtime analyses to be as lightweight as possible.

The flow of execution in the *Sambamba* framework is depicted in Figure 1:

- **[A]** We use static whole-program **analyses** to examine the program for potential optimizations and propose a first set of parallelization and specialization candidates that are deemed beneficial. For long-running programs it might be a viable alternative to also run these analyses at runtime.
- **[P]** The runtime system provides means for speculatively **parallelizing** parts of the program based on the initial static analysis and calibration information.
- **[X]** We detect conflicts caused by speculative **executions** violating the program's sequential semantics and recover using a software transactional memory system [1] which we adapted to our special needs.

[1] *Sambamba* is Swahili for *parallel*, *simultaneously* or *side by side*.

[C] We gather information about the execution profile and misspeculations to **calibrate** future automatic optimization steps.

[S] Based on the calibration results, *Sambamba* supports generating different function variants that are **specialized** for specific environmental parameters and input profiles. These can then again be individually parallelized in the next step.

3 Adaptive Parallelization

3.1 Data Dependence Analysis

The main obstacle for parallel execution of program parts is data dependences over the heap. Parallel computation cannot start before all input data has been computed. In large irregular programs, the interprocedural data flow is hard to determine statically, so all known analyses only provide overapproximations.

In order to get a sound over-approximation of the existing data dependences, we use a state of the art context-sensitive alias analysis called *Data Structure Analysis* [3]. This information allows us to statically prove the absence of certain dependences.

3.2 Parallel CFG Construction

Given a regular control flow graph in SSA form, *Sambamba* creates the so-called parallel control flow graph (*ParCFG*). Unnecessary structural dependences defining an execution order are removed and replaced by real dependences caused by possible side effects.

We formulated the graph partitioning related problem as integer linear program (ILP). The solution of this ILP is then used to introduce so-called parallel sections (*ParSecs*). Each *ParSec* defines at least one fork point π_s and exactly one join point π_e for later parallel execution. Side-effect-free instructions might be duplicated in this step in order to facilitate parallelization.

We do not put special emphasis on loop parallelization and deal with general control flow instead. Very strong approaches of loop parallelization have been proposed and implemented during the last 30 years. Enriching some of these methods, like for example polyhedral loop optimization, with speculation is one of our ongoing projects.

3.3 Scheduling and Parallel Execution

In this step, *Sambamba* generates executable code from the *ParCFG*. This task includes the creation of an execution plan for concurrently executed parts as well as the generation of LLVM bitcode, which is translated into machine code by a just-in-time compiler.

In this demonstration, we only partition a region into parallel tasks if we could prove the absence of data dependences between them. Thus, the execution order of these tasks is not relevant. This will change as soon as we allow to speculate

on the absence of dependences. Then it may be beneficial to delay the execution of a task T until all tasks that T might depend on complete.

The assignment of tasks to processors is done dynamically by using a *global thread pool* initialized during load time of the program.

4 State of the Project

The demonstrated tool is a working prototype. Not every planned feature is fully implemented yet. Especially the features of the runtime-system are implemented on demand as we work on the modules for automatic parallelization.

At the time of writing, the following module independent parts are examples of implemented features:

- Method versioning and a general method dispatch mechanism
- A software transactional memory system supporting speculative execution
- Integration of the LLVM just-in-time compiler.

Concerning automatic parallelization, the demonstrated implementation is able to statically find sound candidates for parallelization. It identifies and rates data dependences which could not be statically proven to exist (may dependences) but prevent further parallelization. Execution adapts to the available system resources by dispatching between the sequential and a sound parallel version of parallelized methods.

For further details and news on the *Sambamba* framework please refer to the project webpage: *http://www.sambamba.org/*.

Acknowledgments. The work presented in this paper was performed in the context of the Software-Cluster project *EMERGENT* (www.software-cluster.org). It was funded by the German Federal Ministry of Education and Research (BMBF) under grant no. "01IC10S01". The authors assume responsibility for the content.

References

1. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP 2008), p. 237. ACM Press, New York (2008)
2. Hammacher, C., Streit, K., Hack, S., Zeller, A.: Profiling Java programs for parallelism. In: Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering, pp. 49–55. IEEE Computer Society (2009)
3. Lattner, C., Lenharth, A., Adve, V.: Making context-sensitive points-to analysis with heap cloning practical for the real world. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2007), pp. 278–289. ACM, New York (2007)

Author Index

Amza, Cristiana 200

Balakrishnan, Gogul 144
Banerjee, Anindya 41

Chen, Lei 185
Clauss, Philippe 220

Farooq, Muhammad Umar 185
Fauzia, Naznin 101
Fujimoto, Richard 81

Glew, Neal 165
Gupta, Aarti 144

Hack, Sebastian 1, 240
Hammacher, Clemens 240
Hou, Cong 81

Ivančić, Franjo 144

Jefferson, David 81
Jimborean, Alexandra 220
John, Lizy Kurian 185

Karrenberg, Ralf 1
Kielstra, Allan 200
Kwon, Taeho 122

Lhoták, Ondřej 41
Loechner, Vincent 220

Maeda, Naoto 144
Marron, Mark 41
Mastrangelo, Luis 220

Nasre, Rupesh 61

Petersen, Leaf 165
Pouchet, Louis-Noël 101
Putta, Sandeep 61

Qasem, Apan 21
Quinlan, Daniel 81

Ramanujam, J. 101

Sadayappan, P. 101
Sankaranarayanan, Sriram 144
Sarkar, Vivek 101
Shaltz, Christopher 21
Sharma, Kamal 101
Sharma, Naveen 144
Shirako, Jun 101
Sinha, Nishant 144
Steffan, J. Gregory 200
Streit, Kevin 240
Su, Zhendong 122

Unkule, Swapneela 21

Vuduc, Richard 81
Vulov, George 81

Yang, Jing 144

Zeller, Andreas 240
Zhao, Chuck (Chengyan) 200