UNIVERSITY OF CALIFORNIA, SAN DIEGO

Predicting Performance Across Compilations

A dissertation submitted in partial satisfaction of the requirements for the degree

Doctor of Philosophy

in

Computer Science

by

Jeremy Lau

Committee in charge:

Brad Calder, Chair
Matthew Arnold
Pamela Cosman
Ranjit Jhala
Chandra Krintz
Geoff Voelker

2007

©
Jeremy Lau, 2007
All rights reserved.

The dissertation of Jeremy Lau is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

2007

DEDICATION

For Mom

EPIGRAPH

The protean nature of the computer is such that it can act like a machine or like a language to be shaped and exploited. It is a medium that can dynamically simulate the details of any other medium, including media that cannot exist physically. It is not a tool, although it can act like many tools. It is the first metamedium, and as such it has degrees of freedom for representation and expression never before encountered and as yet barely investigated. Even more important, it is fun, and therefore intrinsically worth doing.

Alan Kay

TABLE OF CONTENTS

Signature Page . . . iii

Dedication Page . . . iv

Epigraph . . . v

Table of Contents . . . vi

List of Figures . . . viii

List of Tables . . . xi

Acknowledgments . . . xii

Vita and Publications . . . xvi

Abstract . . . xviii

I Introduction . . . 1
A. Predicting Performance Across Compilations . . . 2
B. Problem Areas . . . 3
1. Cross Binary Architectural Simulation . . . 4
2. Dynamic Optimization with Performance Auditing . . . 7

II Cross Binary Architectural Simulation . . . 10
A. Background . . . 13
1. Time Varying Behavior and Phases . . . 13
2. Phase Analysis . . . 17
3. Accelerating Architectural Simulations with SimPoint . . . 21
4. Evaluating Phase Classifications . . . 23
B. Capturing Program Behavior with Fixed and Variable Length Intervals . . . 24
1. Issues with Fixed Length Intervals . . . 25
2. Hierarchical Program Behavior . . . 28
C. Software Phase Markers . . . 34
1. Capturing Hierarchical Behavior with Call-Loop Graphs . . . 34
2. Selecting Software Phase Markers . . . 38
3. Support For Variable Length Intervals in SimPoint . . . 45
4. Methodology . . . 47
5. Phase Marker Evaluation . . . 47

6. Applications: Data Cache Reconfiguration and SimPoint 56
D. Cross Binary Simulation Points 65
1. Selecting Cross Binary Simulation Points 65
2. Methodology 73
3. Cross Binary SimPoint Evaluation 74
E. Related Work 83
F. Summary 85

III Performance Audited Dynamic Optimization 87
A. Background 91
1. Adaptive Optimization in Virtual Machines 91
2. Predicting Performance in Optimization Systems 92
3. Empirical Search 94
4. Building a Mapping Between Binaries 99
B. Performance Auditing 102
1. Motivating Empirical Search 103
2. Performance Auditor Design 106
3. Methodology 110
4. Offline Convergence Study 112
5. Online Performance Auditing 119
C. Lightweight Code Markers 130
1. Inserting Lightweight Code Markers 132
2. Application to Performance Auditing 136
3. Methodology 143
4. Evaluation: Perf. Auditing with Lightweight Code Markers . . 145
5. Other Potential Uses for Lightweight Code Markers 153
D. Discussion 155
E. Summary 158

IV Conclusion and Future Challenges 160
A. Predicting Performance Across Compilations 160
1. Cross Binary Simulation Points 161
2. Performance Audited Dynamic Optimization 161
B. Building a Mapping Across Binaries 162
1. Cross Binary Simulation Points 163
2. Performance Audited Dynamic Optimization 165
C. Future Challenges 166
1. Cross Binary Architectural Simulation 167
2. Performance Audited Dynamic Optimization 168

Bibliography 171

LIST OF FIGURES

Figure II.1	Time varying graph for `bzip2-graphic` (first 64 billion instructions).	14
Figure II.2	Time varying graph for `gcc-integrate`.	15
Figure II.3	An example of what happens to a signal (top figure) when it is sampled with different interval lengths.	26
Figure II.4	Three dimensional non-accumulated representation of `gzip-graphic` and `bzip2-source`.	29
Figure II.5	Two dimensional accumulated representation of `bzip2-source`.	30
Figure II.6	Code example and Call-Loop graph for code example	37
Figure II.7	Time varying graphs with phase markers for gzip-graphic for an OSF Alpha executable	48
Figure II.8	Cross-binary time varying graphs with phase markers for gzip-graphic for Linux x86	48
Figure II.9	Bzip2 fixed length execution intervals representation	50
Figure II.10	Bzip2 variable length execution intervals representation with phase markers	51
Figure II.11	Average instructions per interval	54
Figure II.12	Number of phases detected	54
Figure II.13	Coefficient of variation of CPI. The "Whole Program" results show each program's variability if every interval is classified into a unique phase	55
Figure II.14	Average cache size with no allowed increase in cache miss rate.	58
Figure II.15	Number of instructions simulated with SimPoint and phase markers.	62
Figure II.16	Error in CPI estimates with fixed length intervals and variable length intervals from phase markers.	62
Figure II.17	Number of SimPoints for per-binary SimPoint (FLI) and cross binary SimPoint (VLI). Each bar shows the average across all four binaries.	75
Figure II.18	Interval Size for cross binary SimPoint (VLI). Each bar shows the average across all four binaries. The size of each interval in per-binary SimPoint, which uses fixed length intervals, is constant at 100 million instructions.	75
Figure II.19	CPI Error for per-binary SimPoint (FLI) and cross binary SimPoint (VLI). Each bar shows the average across all four binaries.	77

Figure II.20 Speedup error for per-binary SimPoint (fli) and cross binary SimPoint (vli). 78
Figure II.21 Speedup error for per-binary SimPoint (fli) and cross binary SimPoint (vli). Speedup is computed across pairs of binaries on the *different* platforms (32-bit vs. 64-bit). 79

Figure III.1 Per-method performance impact of moving from optimization level 4 to level 5. 104
Figure III.2 Per-method performance impact of expanded inlining heuristic. 105
Figure III.3 Performance Auditor Overview 107
Figure III.4 Misprediction rate for a simple sampling approach in which a fixed number of method invocation (entry+exit) timings are collected, for various amounts of introduced speedup. 114
Figure III.5 Convergence rate for the proposed statistical approach. This plot shows the percentage of hot methods for which confident predictions are generated, with a sampling limit of 2 minutes of CPU time for each method. 116
Figure III.6 Incorrect predictions. This plot shows the percentage of hot methods in which the analysis incorrectly predicts that A is faster than B. Results are averaged over converged methods. 117
Figure III.7 Time to converge 117
Figure III.8 Architecture for the *dispatcher*, to select method invocations for timing. 120
Figure III.9 Dispatch logic 121
Figure III.10 Per-method overhead of the dispatcher fast-path. No timing samples are being taken. 124
Figure III.11 Per-method overhead of the Performance Auditor when sampling 1 of 20 executions. Overhead includes recording and processing the timing samples. 125
Figure III.12 Performance of the online system using the proposed statistical technique to guide inlining heuristic selection. 126
Figure III.13 Accuracy of the online system using the proposed statistical technique to guide inlining heuristic selection. . 127
Figure III.14 Speedup/slowdown observed when a marker is inserted and removed in every loop or every block in every method. 135
Figure III.15 Loop Selection Algorithm 137

Figure III.16 Three CFGs selected for loop instrumentation, where the loop selection algorithm was unable to find a satisfactory timing point. 142
Figure III.17 Converge scores for hot methods, higher is better. . . . 147
Figure III.18 Perturb scores for hot methods, lower is better. 148
Figure III.19 Accuracy of our approach, where markers are inserted before optimization, and markers are replaced with full timing instrumentation after optimization. 150
Figure III.20 Accuracy of the naïve approach, where full timing instrumentation is inserted before optimization. 151
Figure III.21 Convergence time, comparing method+loop timings to method timings. Top figure: convergence time for hot methods selected for loop instrumentation. Bottom figure: convergence time for all hot methods. 152

LIST OF TABLES

Table II.1	Baseline Simulation Model.	46
Table II.2	Memory System Configuration	72
Table II.3	Phase comparison across 32-bit unoptimized and 64-bit unoptimized `gcc` binary versions.	81
Table II.4	Phase comparison across 32-bit optimized and 64-bit optimized `apsi` binary versions.	81
Table III.1	Benchmark suite	111
Table III.2	Benchmark suite	144

ACKNOWLEDGMENTS

This dissertation would not have been possible without my advisor, Professor Brad Calder. Over the last six years, he has not only taught me how to do research, but also taught me much about life. Thank you.

I must thank all my labmates in the Architecture Lab over the years. Thanks to Tim for being a second advisor, John for reminding me what it's all about, Jamison for keeping it brutal, Rakesh for the always insightful conversations, Jeff for being my partner in system administration, Satish for making everything look easy, Erez for the adventures, Stef for the moo-can, and Ganesh for all the rides to Black Mountain Road.

I must also thank my friends for putting up with too many of my work related disappearances over the last six years. Thanks to Louis for always finding time to hang out, Ethan for being down for whatever, James for dreaming, Lance, Ron, and Alex for a whole lot of bowling, and Lynne for listening.

My brother must be thanked for reminding me that there is much more to life than research.

Finally, and most importantly, my mother must be thanked for always believing in me, for always watching out for my best interests, for periodically verifying that I'm still alive, for reminding me that I am Chinese, for not letting me leave home without entirely too much food, for knowing how to grow, cook, fix, and/or clean anything imaginable... This list goes on for quite a while. For all these reasons and more, this dissertation is for you.

Cross binary simulation points, presented in section II.D, were developed in collaboration with Erez Perelman and Brad Calder at the University of California San Diego, Greg Hamerly at Baylor University, Tim Sherwood at the University of California Santa Barbara, and Harish Patil and Aamer Jaleel at Intel. I thank my co-authors for allowing me to present the results of our

collaboration in my dissertation.

The performance auditor presented in section III.B was the result of collaboration with Matthew Arnold and Michael Hind at IBM T.J. Watson, and Brad Calder at the University of California San Diego. I thank my co-authors for allowing me to present the results of our collaboration in my dissertation.

Section II.B contains material that appears in "Motivation for Variable Length Intervals and Hierarchical Phase Behavior", in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Jeremy Lau, Erez Perelman, Greg Hamerly, Timothy Sherwood, Brad Calder. The dissertation author was the primary investigator and author of this paper. Portions of Section II.B are ©2005 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Section II.C contains material that appears in "Selecting Software Phase Markers with Code Structure Analysis", in *International Symposium on Code Generation and Optimization (CGO)*, Jeremy Lau, Erez Perelman, Brad Calder. The dissertation author was the primary investigator and author of this paper. Portions of Section II.C are Copyright ©2006 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from

Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or `permissions@acm.org`.

Section II.D contains material that appears in "Cross Binary Simulation Points", in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Erez Perelman, Jeremy Lau, Harish Patil, Aamer Jaleel, Greg Hamerly, Brad Calder. The dissertation author was the secondary investigator and author of this paper. Portions of Section II.D are ©2007 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Section III.B contains material that appears in "Online Performance Auditing: Using Hot Optimizations Without Getting Burned", in *Conference on Programming Language Design and Implementation (PLDI)*, Jeremy Lau, Matthew Arnold, Michael Hind, Brad Calder. The dissertation author was the primary investigator and author of this paper. Portions of Section III.B are Copyright ©2006 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or `permissions@acm.org`.

Section III.C contains material in "A Loop Correlation Technique to Improve Performance Auditing", submitted to *Conference on Parallel Architectures and Compilation Techniques (PACT)*, Jeremy Lau, Matthew Arnold, Michael

Hind, Brad Calder. The dissertation author was the primary investigator and author of this paper.

VITA

2001	Bachelor of Arts in Computer Science University of California, Berkeley
2003	Internship Microsoft Research, Redmond
2005	Internship IBM T.J. Watson Research Center, Hawthorne
2007	Doctor of Philosophy in Computer Science University of California, San Diego

PUBLICATIONS

"Cross Binary Simulation Points" Erez Perelman, Jeremy Lau, Harish Patil, Aamer Jaleel, Greg Hamerly, Brad Calder. *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2007, San Jose, CA, USA

"Online Performance Auditing: Using Hot Optimizations Without Getting Burned" Jeremy Lau, Matthew Arnold, Michael Hind, Brad Calder. *Conference on Programming Language Design and Implementation (PLDI)*, June 2006, Ottawa, Canada

"Using Machine Learning to Guide Architecture Simulation" Greg Hamerly, Erez Perelman, Jeremy Lau, Timothy Sherwood, Brad Calder. *Journal of Machine Learning Research (JMLR)*, Volume 7, Pages 343-378, 2006

"Selecting Software Phase Markers with Code Structure Analysis" Jeremy Lau, Erez Perelman, Brad Calder. *International Symposium on Code Generation and Optimization (CGO)*, March 2006, New York, NY, USA

"Dynamic Phase Analysis for Cycle-Close Trace Generation" Cristiano Pereira, Jeremy Lau, Brad Calder, Rajesh Gupta. *International Conference on Hardware/Software Codesign and System Synthesis (CODES)*, September 2005, New York, NY, USA

"SimPoint 3.0: Faster and More Flexible Program Analysis" Greg Hamerly, Erez Perelman, Jeremy Lau, Brad Calder. *Workshop on Modeling, Benchmarking and Simulation (MOBS)*, June 2005, Madison, WI, USA

"The Strong Correlation between Code Signatures and Performance" Jeremy Lau, Jack Sampson, Erez Perelman, Greg Hamerly, Brad Calder. *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2005, Austin, TX, USA

"Motivation for Variable Length Intervals and Hierarchical Phase Behavior" Jeremy Lau, Erez Perelman, Greg Hamerly, Timothy Sherwood, Brad Calder. *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2005, Austin, TX, USA

"Transition Phase Classification and Prediction" Jeremy Lau, Stefan Schoenmackers, Brad Calder. *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2005, San Francisco, CA, USA

"Structures for Phase Classification" Jeremy Lau, Stefan Schoenmackers, Brad Calder. *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2004, Austin, TX, USA

"Reducing Code Size With Echo Instructions" Jeremy Lau, Stefan Schoenmackers, Timothy Sherwood, Brad Calder. *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, October 2003, San Jose, CA, USA

"Ninja: A Framework for Network Services" Eric Brewer, Nikita Borisov, Mike Chen, Rob von Behren, Matt Welsh, David Culler, Josh MacDonald, Jeremy Lau, Steven D. Gribble. *Usenix Technical Conference (USENIX)*, June 2002, Monterey, CA, USA

ABSTRACT OF THE DISSERTATION

Predicting Performance Across Compilations

by

Jeremy Lau
Doctor of Philosophy in Computer Science
University of California, San Diego, 2007
Professor Brad Calder, Chair

Performance comparisons are ubiquitous in computer science. The proceedings of most conferences are filled with bar charts comparing the performance of some computer system to another. For example, computer architects compare the performance of processors, and compiler writers compare the performance of generated code. It is difficult to prove that one computer system is always faster than another for all possible workloads, so these performance comparisons are used as predictors: performance is compared on several representative workloads, and the results are used to argue that one computer system is generally faster than another. Unfortunately, there are many scenarios where it is difficult to make a fair performance comparison. This dissertation focuses on two such scenarios.

The first scenario involves simulations in computer architecture. Computer architects typically evaluate new processor designs through slow cycle-level simulation. Because of the poor performance of cycle-level simulators, accelerated simulation methodologies are very popular, where small samples of a program's behavior are simulated, and the results are extrapolated to predict the results of a whole-program simulation. But with these accelerated simulation techniques,

it is difficult to meaningfully compare performance estimates when multiple compilations of a program are involved. This dissertation will show that simulation samples must be selected consistently across compilations to produce comparable results, and a technique will be presented to apply accelerated simulation across compilations to produce comparable results.

The second scenario involves dynamic optimization systems. Dynamic optimizers must predict if their optimizations will actually improve performance before applying them — if an optimization is unlikely to improve performance, or if an optimization will degrade performance, the optimization should not be applied. This dissertation presents a new approach to guide dynamic optimization decisions by performing empirical performance evaluations as programs execute. The performance of differently-compiled versions of the same code are measured, and the results of the measurements directly guide optimization decisions. The challenge is that these performance measurements are collected as programs execute, so individual measurements are not directly comparable, because the program may run the code under analysis with different inputs over time. If a single pair of performance measurements indicates that one version of the code is faster than another, it may actually be faster, or it may be that the program chose to run one version on a smaller input than the other. To overcome this challenge, this dissertation presents a statistical technique to analyze pools of timing data to determine which version is the fastest.

I

Introduction

Historically, the complexity of the average desktop computer has increased drastically, and the trend continues toward more complex systems. Hardware complexity continues to grow as more transistors become available for each chip, and software complexity continues to grow as more layers of software are introduced with the increasing popularity of application frameworks, dynamic compilation, and virtualization. These trends make it more and more difficult for computer scientists to predict the performance of their systems.

Performance prediction is a simple idea that is used extensively throughout computer science: performance measurements are collected on a small representative workload to predict the overall performance of a computer system in the general case. As a simple example, if it is known that a particular loop runs for 10,000 iterations, the overall running time of the loop can be predicted by measuring the loop's running time for 1,000 iterations, and multiplying the result by 10. As another example, computer architects measure the performance of processors as they run the SPEC2000 benchmark suite to predict how processors will perform in everyday use.

This dissertation focuses on one specific type of performance prediction problem: predicting performance across compilations.

I.A Predicting Performance Across Compilations

It is more difficult to make performance predictions when multiple binary representations of the same program are being compared. Suppose a program is run through two different compilers, producing two binaries B_1 and B_2. It is then reasonable to ask: which binary, B_1 or B_2, will run faster, on average? It is also reasonable to answer this question by evaluating small samples of B_1's and B_2's performance. This is a scenario where performance must be predicted across compilations.

When predicting performance across compilations, the primary challenge is to collect measurements of B_1's and B_2's performance that can be *meaningfully compared*. To make the example more concrete, suppose B_1 and B_2 are compiled versions of a scientific program that first reads a large dataset into memory, then analyzes the dataset. The first half of execution is spent reading the dataset from disk, and the second half of execution is spent running the analysis. The goal is to evaluate the performance of B_1 and B_2 by collecting small samples of B_1's and B_2's performance. But the samples must be selected consistently — a sample taken from B_1's analysis phase should be compared to a sample taken from B_2's analysis phase. It does not make sense to compare a sample taken from B_1's loading phase to a sample taken from B_2's analysis phase.

The problem of selecting samples consistently across binaries is surprisingly difficult. It seemed simple in the preceding example, but that is because the example provided a high level description of the program's behavior. Binaries do not contain such descriptions. Without this type of high level information, a technique is needed to select samples in each binary so that high level program behaviors are comparably represented. This dissertation will present several techniques to collect samples consistently across binaries in order to draw meaningful conclusions about the overall performance of the binaries.

I.B Problem Areas

This dissertation focuses on two instances of the problem of predicting performance across compilations: one in the area of accelerated architectural simulation and another in the area of dynamic optimization.

Computer architects evaluate new processor designs through cycle-level simulation, where a simulator program models all the low-level effects of every instruction on a new processor design. But cycle-level simulation is extremely slow, due to the complexity of modern processors — simulator programs can easily execute thousands of instructions on a host processor just to simulate a single instruction on the simulated processor. Because of the poor performance of cycle-level simulators, accelerated simulation methodologies are very popular, where small samples of a program's behavior are simulated, and the results are extrapolated to predict the results of a whole-program simulation. These accelerated simulation techniques make cycle-level simulation feasible for real-world programs, but it is difficult to compare the results of these accelerated simulation techniques when multiple binaries of a program are involved. For example, if extensions to the instruction set are considered as part of the architectural design space exploration, multiple binary representations of the same benchmark program must be considered — one binary that uses the proposed instructions, and another binary that does not use the proposed instructions. This dissertation will show that the samples of program behavior selected for detailed simulation must be selected consistently across binaries to produce comparable results, and this dissertation will present a technique to apply an accelerated simulation technique across compilations so results are comparable.

Dynamic optimization systems improve the performance of programs as they execute by recompiling portions of programs with additional optimizations. A dynamic optimizer must predict if its optimizations will actually improve

performance before applying them — if an optimization is unlikely to improve performance, or if an optimization is likely to degrade performance, the optimization should not be applied. These optimization decisions are typically guided by heuristics. For example, an optimizer may decide to inline a procedure if the procedure contains less than N instructions. But as the complexity of computer systems increases, these types of heuristics become less reliable. This dissertation presents an alternative approach to guide dynamic optimization decisions by performing empirical performance evaluations as programs execute. The performance of differently-compiled versions of the same code is measured, and the results of the performance measurements are used to directly guide optimization decisions. The challenge is that the performance measurements are collected as programs execute, so individual measurements are not directly comparable, because the program may run the code under analysis with different inputs over time. If a single pair of measurements indicates that version X of the code is faster than version Y, it may be because version X is indeed faster than version Y, or it may be because the program happened to invoke version X on a smaller input than version Y. To overcome this challenge, this dissertation presents a statistical technique to analyze pools of timing data to determine which version is the fastest.

I.B.1 Cross Binary Architectural Simulation

To design the best processor for an application, processor designers must find the best design point in a huge design space. Intuition and common sense are used to pare down the design space, but at some point it becomes too difficult to think about all the performance implications of minor design changes. At this point, cycle-level simulators are typically used to measure the performance of benchmark programs on a set of potential processor designs.

Unfortunately, cycle-level simulators are extremely slow due to the incredible complexity of modern processors. The primary goal of a cycle-level simulator is to determine how many cycles will be needed to execute a program on a candidate processor design. This means that a cycle-level simulator must model the low-level effects of every instruction on all aspects of the architecture, including branch predictors, caches, bus traffic, speculative execution, and out-of-order execution.

Modeling at this level of detail carries a significant performance cost — slowdowns on the order of 1,000 to 10,000 times are common. Simulation speed is a major concern in architecture research, because more design space explorations can be performed with faster simulators. To improve the speed of architectural simulations, a number of accelerated architectural simulation methodologies have been proposed. Several popular approaches rely on sampled simulation, where the performance of the processor on several small samples of the program is measured through cycle-level simulation, and the results are extrapolated to estimate the program's overall performance.

Accelerated simulation techniques such as SimPoint [78] work well when the goal is to estimate a single program's overall performance from small samples of the program's behavior, because the existing techniques are specifically designed for this purpose. But such techniques do not always work well when multiple binary representations of the same program are involved.

There are three main scenarios where multiple binaries must be used in architecture simulation. The first scenario involves instruction set extensions, where a new binary is created that uses new instructions, such as the 64-bit x86 extensions (x86-64). In this case, the performance of the original binary, which does not use the extensions, must be compared to the performance of the new binary, which does use the extensions. The second scenario deals with

examining completely different architectures, such as Itanium and x86-64. In this case, completely different compilers will be used. Finally, the third scenario involves developing compilers for new architectures. In this scenario, the compiler writers must evaluate the performance effects of their compiler optimizations through simulation, because working prototypes of a new processor are typically not available until very late in the development cycle. In this scenario, the new compiler may use the same instruction set as an existing compiler, but different binaries will be produced as optimizations are enabled, disabled, reordered, and adjusted.

When multiple binary versions of a program are used with an accelerated architectural simulation methodology, this dissertation will show that existing techniques do not work well, and this dissertation will show that samples of program behavior selected for cycle-level simulation must be selected consistently across the different binaries to produce comparable results. This dissertation will also present a technique to apply an accelerated simulation technique across compilations so that samples of program behavior are selected consistently across binaries, to produce comparable accelerated simulation results.

In the area of accelerated architectural simulation, the primary contributions of this dissertation are:

- A technique to identify software phase markers, which are code locations that indicate the beginning of a major change in program behavior when executed. Phase markers are associated with source code, and are thus portable across compilations.
- A technique to identify cross-binary simulation points, which allow the results of accelerated architectural simulation to be meaningfully compared across binaries. Cross-binary simulation points use phase markers to indicate where cycle-level simulation is required, so cycle-level simulation can

be run on semantically equivalent regions in all binaries, which allows for meaningful comparisons of sampled simulation results across binaries.

I.B.2 Dynamic Optimization with Performance Auditing

Dynamic optimization systems attempt to improve the performance of programs as they execute through recompilation. A dynamic optimizer must predict if its optimizations will actually improve performance before applying them — if an optimization is unlikely to improve performance, or if an optimization will degrade performance, the optimization should not be applied. But it becomes increasingly difficult to accurately predict if an optimization will actually improve performance as the complexity of software systems increases. As a simple example, when should an optimizer inline a procedure? Inlining should be done when it will improve performance, but it becomes increasingly difficult to predict if a particular inlining decision will actually improve performance as additional software layers such as application frameworks, Java virtual machines and hypervisors are introduced between a program and its processor. Optimizers typically rely on heuristics to predict if an optimization will improve performance. For example, an optimizer may inline a procedure if the procedure contains less than N instructions. But as software complexity increases, these types of heuristics become less reliable. This dissertation suggests an alternative approach called performance auditing where dynamic optimizations are guided by empirical performance measurements. With performance auditing, the performance of differently-compiled versions of the same code is measured, and the results of the performance measurements are used to directly guide optimization decisions.

In the proposed performance auditing system, when a piece of code is selected for performance auditing, several differently-compiled versions of the

code are generated. The program is allowed to run normally, but whenever the audited code region is entered, one of the compiled versions will be randomly selected and invoked. The system records the amount of time the program spent executing code in that compiled version of the code.

The main challenge in the proposed system is to meaningfully compare performance measurements that have been collected in this manner to make an accurate performance prediction. The goal is to answer the question: Which compiled version of the audited code runs the fastest in general? Answering this question is difficult because it is typically impossible to obtain single pieces of timing data for each of the differently-compiled versions of the code that are directly comparable, because timing data is collected continuously as the program executes. To overcome this challenge, this dissertation presents a statistical technique to analyze pools of timing data to determine which version is the fastest, or if there is not enough statistical evidence to make a confident decision. If statistical confidence is low, more timing data is collected, and the statistical analysis will run again later.

This challenge is very similar to the main challenge in building an accelerated architectural simulation system that produces comparable results across multiple compilations, which was discussed in the previous section. In both cases, the key challenge is to collect comparable performance samples from a set of binaries, such that the performance samples can be meaningfully compared, in order to accurately predict which binary runs the fastest.

In the area of performance auditing, the primary contributions of this dissertation are:

- A technique to empirically compare the effectiveness of optimizations online, despite changing program state.
- A description and evaluation of a prototype performance auditing system

built in IBM's J9 Java virtual machine.

- Lightweight code markers, which map between a program's source and its optimized binary without using heuristic matching, and without modifying the optimizer.
- A technique to improve statistical confidence in the proposed performance auditing system by using lightweight code markers to consistently split large timing samples into many small timing samples across compilations.

II

Cross Binary Architectural Simulation

Modern computer architecture research requires understanding the cycle level behavior of a processor as it executes a program. To gain this understanding, researchers typically employ detailed simulators that model the processor's cycle-level behavior. Unfortunately, this level of detail comes at the cost of speed. Even with the fastest simulators, modeling the full execution of a single benchmark at the level required for computer architecture research can take weeks or months, and nearly all industry standard benchmarks require simulating the execution of a *suite* of programs. Therefore, instead of simulating entire programs, a few small samples of each program's execution are typically sampled instead. In this chapter, these samples are on the scale of 10 to 500 million instructions.

With sampled simulation, the primary challenge is to determine which simulation samples most accurately represent the program's full execution. To address this problem we created a tool called SimPoint [70, 78] that uses clustering algorithms from machine learning to automatically find repetitive patterns in a program's execution. By simulating one representative of each repetitive behavior

pattern, simulation time can be reduced to minutes instead of weeks for standard benchmark programs, with very little loss in accuracy. Several researchers have shown the SimPoint approach works well when exploring architecture designs with a *single* compiled binary version of a program [55, 70, 78, 95], but this dissertation focuses on the problem of using SimPoint with *multiple* compiled binary versions of a program.

There are three main scenarios where multiple binaries must be used in architecture simulation. The first scenario involves instruction set extensions, where a new binary is created that uses new instructions, such as the 64-bit x86 extensions (x86-64). In this case, the performance of the original binary, which does not use the extensions, must be compared to the performance of the new binary, which does use the extensions. The second scenario deals with examining completely different architectures, such as Itanium and x86-64. In this case, completely different compilers will be used. Finally, the third scenario involves developing compilers for new architectures. In this scenario, the compiler writers must evaluate the performance effects of their compiler optimizations through simulation, because working prototypes of a new processor are typically not available until very late in the development cycle. In this scenario, the new compiler may use the same instruction set as an existing compiler, but different binaries will be produced as optimizations are enabled, disabled, reordered, and adjusted.

In all three of these scenarios, semantically equivalent simulation samples must be identified in all binaries, or the simulation results can not be meaningfully compared.

This chapter presents two solutions to the problem of accelerated simulation with multiple binaries. The first approach applies the baseline SimPoint approach separately for each binary. SimPoint examines an execution trace and

groups similar portions of execution into phases (clusters). The most representative interval from each phase is chosen as the simulation point to represent that cluster. SimPoint produces very accurate results when a single binary is used across different architectures, because the same code regions are simulated on each architecture, so the simulated code regions are semantically equivalent by definition.

But when SimPoint is applied to multiple binary representations of the same program, SimPoint can produce different clusterings for each binary. This means that the simulation points selected by SimPoint in each binary may actually represent different behaviors. And even if the simulation points do represent the same behaviors, the results still may not be comparable, because semantically equivalent portions of execution in one binary may be assigned to different phases in different binaries. So even if semantically equivalent simulation points are selected across binaries, those simulation points may represent different behaviors in different binaries. Results in Section II.D.3 show that these concerns are real.

To address these concerns, this chapter presents a technique called *cross binary SimPoint*. This approach identifies simulation points that are semantically equivalent across multiple binaries. The approach works by first profiling each binary and identifying a set of software phase markers in each binary that can be identified in all other binaries. These mappable simulation points are instructions in each binary corresponding to procedure calls or loop branches that can be consistently identified in all binaries. These mappable markers are potential boundaries for simulation regions. Program execution is split into intervals on these mappable markers for one of the binaries, and basic block vectors are collected. The basic block vectors are run through SimPoint, which selects a set of simulation points which are now mappable across all binaries. Then, detailed simulation can be performed on these mapped simulation points to compare per-

formance across binaries.

The outline of this chapter is as follows. First section II.A presents background and related work in the area of accelerated architectural simulation techniques. Section II.B demonstrates some issues with standard techniques for identifying repetitive program behaviors for the purposes of accelerated architectural simulation. These issues motivate the need for software phase markers, presented in section II.C. *Software phase markers* identify repetitive program behaviors based on high-level code structure, such as procedure calls and loop backedges, that mark the start of each repetitive behavior. A program's software phase markers are tightly coupled with the program's code structure, which makes them effective for applications such as dynamic hardware reconfiguration, but also makes them less effective when directly applied to accelerated architectural simulation. *Cross binary simulation points*, presented in section II.D, loosen this coupling between a program's phase markers and the program's simulation points by selecting specific dynamic instances of phase markers to identify the start and end of each simulation point. With cross binary simulation points, sampled simulation results can be meaningfully compared across multiple binaries. Section II.E presents work that is closely related to cross binary simulation points.

II.A Background

This section presents background and related work in the area of accelerated architectural simulation techniques.

II.A.1 Time Varying Behavior and Phases

Before discussing accelerated architectural simulation techniques, a brief aside is necessary to discuss the time varying behavior of programs and phase

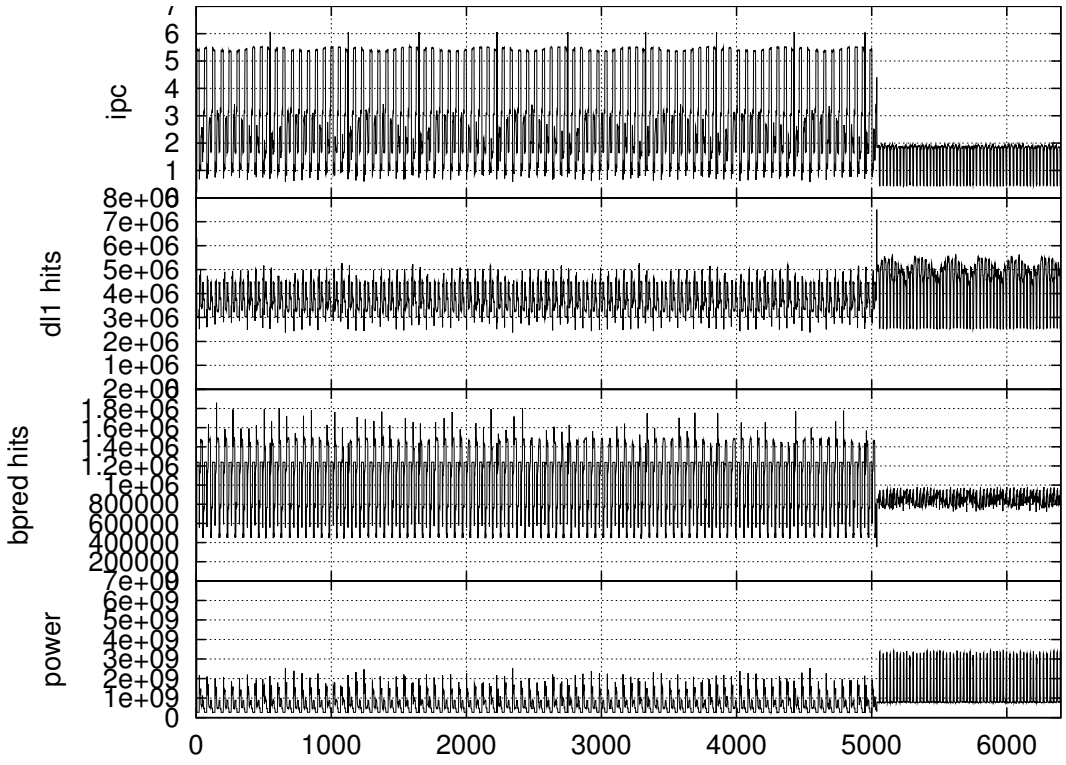


Figure II.1: Time varying graph for `bzip2-graphic` (first 64 billion instructions). Time is plotted on each X-axis, in tens of millions of instructions. Each data point on the Y-axes shows the average IPC, level 1 data cache hits, correct branch predictions, and power for each ten-million instruction interval.

behavior, which make accelerated simulation possible.

A program's behavior at any instant in time can differ drastically from the program's average overall behavior. Furthermore, programs exhibit phase behavior — their behavior will be stable for long periods of time, before moving on to another behavior pattern.

These statements about the time-varying behavior of programs may seen counterintuitive at first, but consider a block-based compression program as an example. The compression program will perform the same steps to compress each block of data. For example, the deflate compression algorithm used in gzip

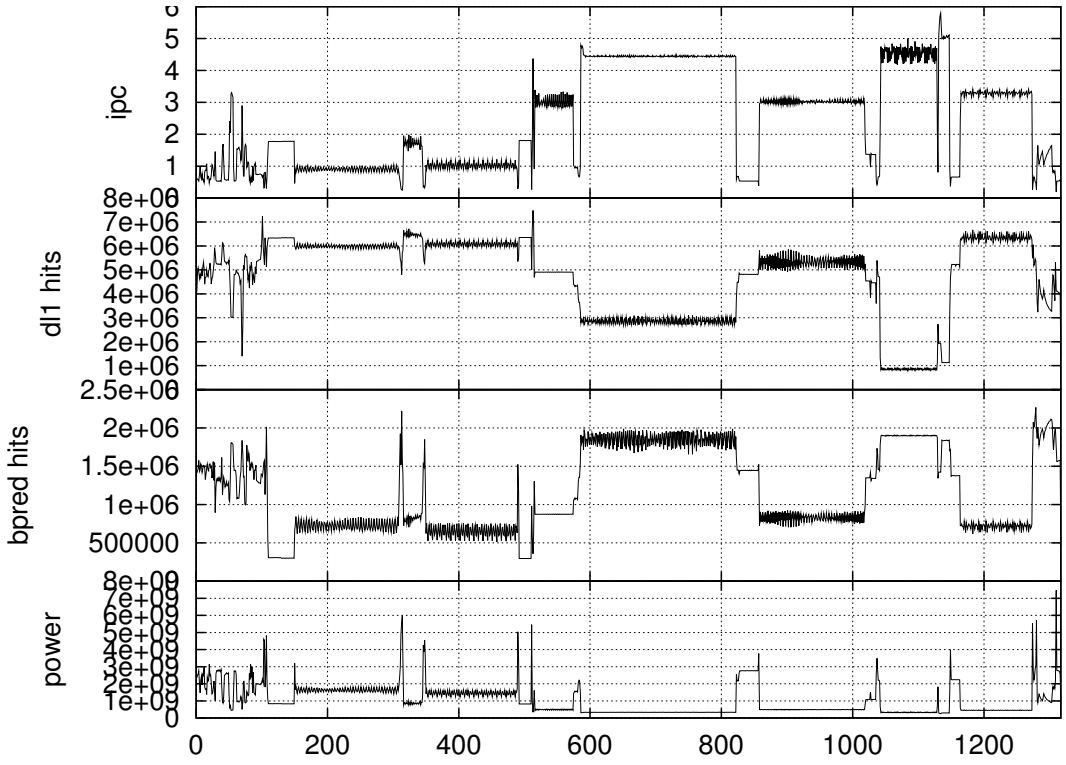


Figure II.2: Time varying graph for `gcc-integrate`.

first slides a window over the data to replace redundancies with backpointers, then applies Huffman coding to further compress the data. Different code will be executed in each of these steps, and the processor should behave differently in each of these steps. These two steps are performed for each block of data, and most inputs are split into many blocks of data, so these two steps should be performed over and over again to compress an input. These expectations lead to cyclic behavior patterns, which are common in compression programs.

Another example of phase behavior in programs are the stages of compilation. The different stages will exercise the machine in very different ways, because the algorithms change from stage to stage. For example, the algorithms used in lexing are usually very different from the algorithms used in register allocation. The compiler will do different work in each of its major stages, and it will spend a significant amount of time in each major stage, and similar work should be done within each stage. These expectations lead to phase behavior.

In general, the execution of most integer programs can be broken down into a series of major processing stages, like the stages of a compiler as discussed above. Most floating point benchmarks consist of one major "computation stage," like the compression example discussed above, but floating point benchmarks also exhibit strong phase behavior because they often spend most of their time in nested loops, which leads to cyclic behavior patterns.

Figure II.1 shows a graph of the time-varying behavior of `bzip2`. To produce this graph, `bzip2`'s execution was partitioned into intervals of 10 million instructions, and the value of each metric listed on the Y-axes (IPC, data cache hits, correct branch predictions, power) was collected for each 10 million instruction interval. Time is plotted on the X-axes, in tens of millions of instructions. This means that each data point on the graph shows the *instantaneous* behavior of `bzip2`, where an "instant" is 10 million instructions long.

Only the first half of `bzip2`'s execution is shown to make the graph more legible. The second half of `bzip2`'s execution is very similar to the first half. Inspection of this figure reveals that this benchmark exhibits phase behavior on two levels: there is one high-level behavior pattern from 0-5000 on the X-axis, and another from 5000-6400. Within each of these high level phase patterns, there are lower-level phase patterns, which are most easily seen on the power graph. Each high-level phase exhibits a highly repetitive low-level phase pattern, yet the low-level phase patterns are completely different across the high-level phases.

Figure II.2 shows a time-varying graph of `gcc`'s behavior. This figure poses more of a challenge to phase analysis. In this figure, two levels of phase behavior can also be seen. For example, `gcc`'s behavior is mostly homogeneous between 150 and 300 on the X-axis. But upon closer examination, there are small periodic variations in behavior within each high-level phase. This time-varying graph shows that `gcc` is a difficult benchmark for phase analysis, because the small behavior variations within each high-level phase are not as regular as the low-level patterns exhibited by `bzip2`, and each high level phase is seen only once.

This section showed that even complex integer benchmarks from SPEC2000 such as `bzip2` and `gcc` exhibit phase behavior: as they execute, they spend large amounts of time (millions to billions of instructions) exhibiting homogeneous behavior before quickly switching to another type of behavior.

II.A.2 Phase Analysis

Before phase behavior can be exploited for accelerated architectural simulations, phases must first be detected. The easiest way to detect phase behavior is to employ knowledge about the program to be run. For example, in an MPEG decoder, program behavior should be fairly homogeneous while decoding each MPEG frame type. Program behavior while decoding any I-frame should

be fairly similar, but behavior while decoding a P-frame should be very different. This subsection explores general techniques that do not rely on this type of program-specific knowledge.

First, some terminology:

- *Interval* — An interval is a continuous slice of program execution. A program's execution is partitioned into intervals, so intervals may not overlap. The length of intervals is commonly referred to as the "granularity" of the intervals.
- *Similarity* — A similarity metric is used to measure the similarity of program behaviors in two intervals of program execution. Similarity metrics depend on how program behavior for each behavior is represented, which will be discussed in the following subsection.
- *Phase* — A phase is a collection of intervals with similar behavior. Typically, programs are partitioned into a large number of intervals, and those intervals are grouped into a small number of phases.
- *Phase Analysis* — To detect phases in program behavior, a program's intervals must be classified into phases, typically using machine learning clustering techniques. This is called phase analysis.

Phases are identified by first partitioning a program's execution into intervals, then clustering the intervals into phases. A program's execution is *partitioned* into intervals, so intervals never overlap. Intervals can be fixed-length or variable-length. When a program's execution is partitioned into intervals, the main issue is setting an interval length. Fixed-length intervals are usually defined in terms of the number of instructions executed by the program in each interval — for example, a program may be partitioned into fixed-length intervals of 100 million instructions. This means that each fixed-length interval represents the

execution of 100 million instructions. The length of intervals is also referred to as the "granularity" — large intervals are "coarse" and small intervals are "fine." Hind et al. [37] provide a framework for defining and reasoning about program phase classifications, focusing on how to best define granularity and similarity for phase analysis.

After a program's execution has been partitioned into intervals, the intervals are clustered into phases. This process will be described in detail in the following subsections.

Phase behavior has been exploited to accelerate architectural simulations [77, 78], to save energy by dynamically reconfiguring caches and processor width [9, 79, 26, 25], to guide compiler optimizations [63, 10], to guide remote profiling [66], and to choose which core to run a process on in a multi-core architecture [54].

Basic Block Vectors

Phase analysis requires a signature of the program's behavior for each interval. This dissertation presents several techniques that rely on code-based signatures [55, 56], specifically, basic block vectors.

Basic Block Vectors (BBVs) [77] capture information about changes in a program's behavior over time. A basic block is a single-entry, single-exit section of code with no internal control flow. A *Basic Block Vector* (BBV) is a one dimensional array where each element in the array corresponds to each basic block in the program. Basic block vectors are essentially another way of looking at block profiles.

To collect a basic block vector, the profiler starts with an empty BBV (zero vector) at the beginning of each interval of execution. Throughout each interval, the profiler counts the number of times each basic block in the program

has been executed, and records the count in the BBV. For example, if the 50th basic block is executed 15 times in the current interval, then the profiler will set bbv[50] = 15. The profiler multiplies each count by the number of instructions in the basic block, so basic blocks containing more instructions will have more weight in the BBV.

Basic block vectors are used to evaluate the similarity of the intervals they were collected from [77, 78]. The intuition is that the behavior of a program in an interval is directly related to the code executed in that interval. Basic block vectors are fingerprints for each interval of execution, because each vector indicates what portions of code are executed, and how frequently those portions of code are executed: if the distance between the BBVs is small, then the two intervals spend about the same amount of time in roughly the same code, and therefore the overall behavior of the program in those two intervals should be similar.

There are many ways to collect a signature of program behavior from an interval of execution other than basic block vectors. Denning and Schwartz [23] were one of the first to analyze time-varying program behavior, by showing that patterns in program behavior can be detected by monitoring a program's data working set over time. Similarly, Dhodapkar and Smith [25, 26, 24] found relationships between patterns in program behavior and code working sets. Balasubramonian et al. [9] used hardware counters to collect miss rates, CPI and branch frequency information for every 100,000 instructions executed, and used the statistics to identify stable portions of program execution. Isci and Martonosi [45, 46] found that patterns in program behavior can be detected by monitoring the power consumption of architectural components. In [56], we examined several alternatives to basic block vectors, including several forms of code signature vectors where each dimension corresponds to procedure calls, procedure returns, or loop

branches instead of basic blocks. We found that many types of code signatures can be used effectively for phase analysis.

II.A.3 Accelerating Architectural Simulations with SimPoint

Architectural simulation is incredibly slow. Modern simulators can simulate around 400,000 instructions per hour, but typical benchmarks contain billions or even trillions of instructions [78]. The poor performance of architectural simulators makes it infeasible to perform detailed simulation for a full benchmark run. Additionally, processor designers are most interested in a processor's performance on a *suite* of benchmarks, rather than single benchmarks. This makes accelerated simulation methodologies even more important.

SimPoint [78] is a popular tool that uses phase analysis to accelerate architectural simulations. SimPoint is used by several of the techniques presented in this dissertation, so a summary of the SimPoint approach is provided in this section. [78] contains a more detailed description.

The underlying principle of the SimPoint approach by Sherwood et al. is that program behavior is a function of the code executed by the program. SimPoint detects phase behavior by examining basic block vectors (described in the previous subsection) collected from each interval of execution.

Basic block vectors can be quite large (`gcc` contains around 100 thousand static basic blocks), so the vectors are randomly projected to a lower dimensionality before analysis. Random projection first generates a M by N random projection matrix, where M is the number of static basic blocks in the program, and N is the desired lower-dimensional projection. The random projection matrix is filled with random numbers limited to a fixed range. Each vector is multiplied by this random projection matrix, resulting in smaller projected vectors. Random projection is best explained by example: to randomly project a 3-dimensional ob-

ject to 2 dimensions, you randomly position a camera at a fixed distance from the object (facing the object of course), and take a picture.

After randomly projecting the basic block vectors for each interval of execution, phases are detected by grouping the projected basic block vectors based on their similarity. Each vector is a point in N-dimensional space, where N is the number of dimensions projected to (typically 15). The Euclidean distance of a pair of vectors in this N-dimensional space is used to evaluate vector similarity.

SimPoint uses the k-means clustering algorithm [61] to partition the set of vectors into clusters of similar vectors. Each cluster produced by the clustering algorithm directly corresponds to a phase of program behavior, because intervals with similar vectors execute roughly the same code in roughly the same proportions, so they should exhibit similar behavior.

After clustering a program's intervals, SimPoint selects a single representative interval from each cluster by identifying the interval that is closest to the centroid (center) of each cluster. Only these representatives will be simulated in detail with a cycle-level simulator, and the simulation results will be weighted by the number of intervals in the phase that the representative represents. In this manner, the detailed simulation results for all the representative intervals from a program can be extrapolated to estimate the overall performance of the simulated processor when executing the program.

A summary of the SimPoint approach is provided below.

1. Profile the program by partitioning the program's execution into contiguous intervals, and collect a basic block vector for each interval. Normalize each basic block vector so that the sum of all the elements equals 1.

2. Reduce the dimensionality of the basic block vectors to a smaller number of dimensions using random linear projection.

3. Run the k-means clustering algorithm on the reduced-dimension BBVs for

a set of k values.

4. Choose from among these different clusterings a well-formed clustering that also has a small number of clusters. To compare and evaluate the different clusters formed for different values of k, SimPoint uses the *Bayesian Information Criterion* (BIC) [71] as a measure of the "goodness of fit" of a clustering to a dataset. SimPoint chooses the clustering with the smallest k, such that its BIC score is close to the best score that has been seen. The chosen clustering represents the final grouping of intervals into phases.

5. Select simulation points for the chosen clustering. For each cluster, SimPoint chooses one representative interval that will be simulated in detail to represent the behavior of the whole cluster. By simulating *only* one representative interval per phase SimPoint can extrapolate and capture the behavior of the entire program. To choose a representative, SimPoint picks the interval in each cluster that is closest to the *centroid* (center) of each cluster. Each simulation point also has an associated weight, which reflects the fraction of executed instructions that are in the cluster.

6. With the weights and the detailed simulation results of each simulation point, SimPoint computes a weighted average for the architecture metric of interest (CPI, miss rate, etc.). This weighted average of the simulation points gives an accurate representation of the complete execution of the program/input pair.

II.A.4 Evaluating Phase Classifications

Several metrics are commonly used to evaluate the effectiveness of phase analysis techniques such as SimPoint.

Phases are intervals with similar program behavior, so the effectiveness

of a phase classification can be measured by examining the similarity of program metrics within each phase. After classifying a program's intervals into phases, the average of some metric (CPI, for example) over all intervals in the phase is calculated. Next, the standard deviation of the metric for each phase is calculated, and the standard deviation is divided by the average to produce the *Coefficient of Variation* (CoV). CoV measures standard deviation as a fraction of the average.

The average and standard deviation for each interval are weighted by the number of instructions in the interval, so intervals that represent a larger percentage of the program's execution receive more weight in the CoV calculations. Per-phase CoVs can be averaged across all phases to produce an overall CoV that measures the homogeneity of a phase classification. Better phase classifications will exhibit lower overall CoV. For example, if all of the intervals in the same phase have exactly the same CPI, then the overall CPI CoV will be zero.

Unfortunately, CoV is not a perfect metric — if a program with N intervals is classified into N phases, the CoV will be zero. For this reason, the number of intervals and phases for each classification must also be considered.

Techniques to accelerate architectural simulations are evaluated by measuring the accuracy of the technique. Accelerated architectural simulation produces an estimate of the processor's performance for a program by simulating small representative portions of the program, instead of simulating the whole program. The difference between the estimated performance from accelerated simulation and the actual performance from full simulation should be small.

II.B Capturing Program Behavior with Fixed and Variable Length Intervals

Prior phase analysis work relies on fixed length intervals. Fixed length intervals are usually defined in terms of some fixed number of dynamic instruc-

tions per interval - for example, a program's execution may be partitioned into fixed-length intervals of one million instructions per interval. This means that the first interval of execution contains the first dynamic instruction to the millionth dynamic instruction, the second interval contains the 1,000,001th dynamic instruction to the two-millionth instruction, and so on.

Fixed length intervals are prone to synchronization problems, because the frequency of interval boundaries will very likely be out of sync with the frequency of behavior changes in the program under analysis. These synchronization problems make it more difficult to automatically find large scale phase behavior.

Additionally, programs exhibit phase behavior at many different granularities, and focusing on a single fixed interval length limits phase discovery to a single granularity. Some programs exhibit a hierarchy of phase behaviors, where different behavior patterns can be seen at different granularities.

This section provides motivation for variable length intervals. Contributions include new methods to graphically view program behavior, and algorithms to capture a program's hierarchy of variable length intervals.

II.B.1 Issues with Fixed Length Intervals

Prior work in phase classification has concentrated on using fixed length intervals. This subsection shows how fixed length intervals can result in sub-optimal phase classifications due to synchronization issues. Fixed length intervals profile program behavior at a fixed frequency determined by the interval length. But this profiling frequency is highly likely to be out of sync with the actual frequency of behavior changes in the underlying program. This subsection presents examples of how and why these synchronization problems occur.

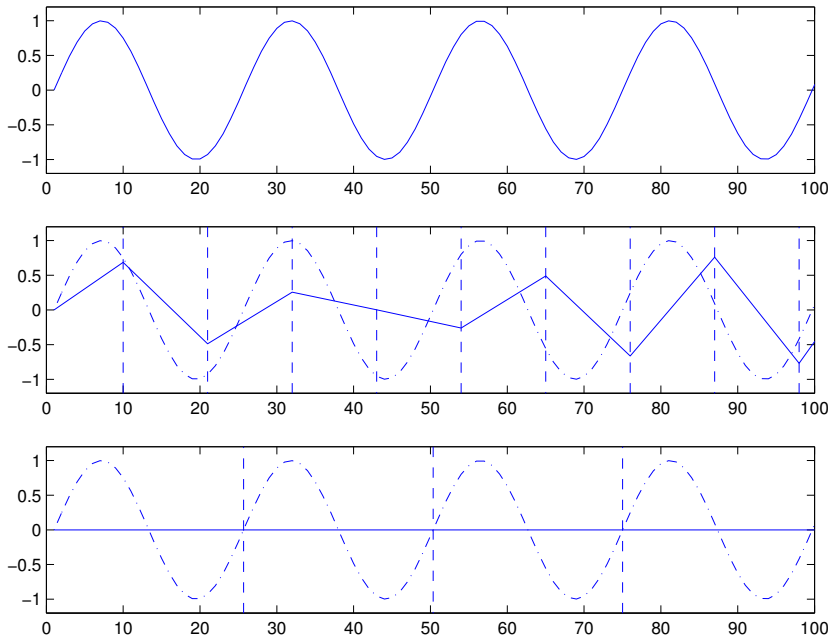


Figure II.3: An example of what happens to a signal (top figure) when it is sampled with different interval lengths. The signal in this example is a sinusoid, shown in the top figure, and the intervals it is broken into are drawn vertically in the lower two figures. The average signal for each interval is shown as the straight line within an interval. When the interval is dissonant with the period of the signal, it results in a jagged and unstable characterization as can be seen in the central figure. The optimal interval duration, shown in the bottom figure, captures exactly one cycle of the repetitive behavior, which results in a concise and stable characterization of the signal.

Interval Dissonance and Harmony

Figure II.3 shows how fixed length intervals can distort the view of time-varying program behavior. The figure shows a simple sinusoid sampled at different frequencies. The top figure shows the signal, a sine wave with a constant period of 25. The middle figure shows what happens when the signal is sampled at a constant period of 11. The original signal is shown in the background, with vertical dashed lines depicting where the intervals are split. The signal average for each interval is plotted as a point at the end of that interval, and the solid line connects these averages to show the interval-based representation of the signal. In this figure the solid line is very jagged, because the period of the intervals is out of sync with the period of the sinusoid.

Fixed length intervals may require a high profiling frequency to capture a signal. The number of intervals required to accurately represent a signal is the ratio of Least Common Multiple between the interval length and the period of the signal, and the length of the interval:

$$\frac{LCM(|interval|, |SignalPeriod|)}{|interval|}$$

In this example, 25 intervals are needed to accurately represent the signal using intervals of fixed length 11.

The bottom figure shows an interval length of 25, equal to the period of the signal. In the bottom figure, each interval captures an entire period of the signal. In this scenario, only one interval is needed to accurately represent the signal.

In this simple example, fixed intervals with length 25 can accurately represent the signal. But most programs do not exhibit simple fixed-frequency phase behavior. For example, the `gzip` benchmark from the SPEC2000 benchmark suite exhibits low-frequency phase behavior in its low-IPC phases, and high-frequency

phase behavior in its high-IPC phases. It is unlikely that a single fixed interval length can accurately capture phase behavior at both these frequencies. Additionally, there are some benchmarks where the period changes over time - `vpr` for example. As `vpr`'s simulated annealing algorithm converges on a solution, it spends less and less time evaluating each solution.

II.B.2 Hierarchical Program Behavior

The previous subsection presented a simple example where fixed length intervals can distort a signal. This subsection examines the effects of fixed length intervals on actual program execution, demonstrating issues with fixed length intervals, and also demonstrating hierarchical phase behavior.

This subsection presents two representations of a program's execution by examining the code executed with basic block vectors collected from fixed length intervals: a three-dimensional non-accumulative representation, and a two-dimensional accumulative representation.

3D Non-Accumulated Representation

In the non-accumulated representation, a basic block vector is captured from each interval of execution. The basic block vector for an interval indicates the number of times each basic block was executed in that interval, as described in subsection II.A.2.

One basic block vector captures program behavior for a single interval, so a set of basic block vectors captures a program's overall behavior. To create the 3D non-accumulated representation, each vector in the set is projected to three dimensions with random linear projection [78]. The projected vectors are then plotted as points in 3-dimensional space, and lines are drawn to connect temporally adjacent points, to show the order in which the intervals are executed.

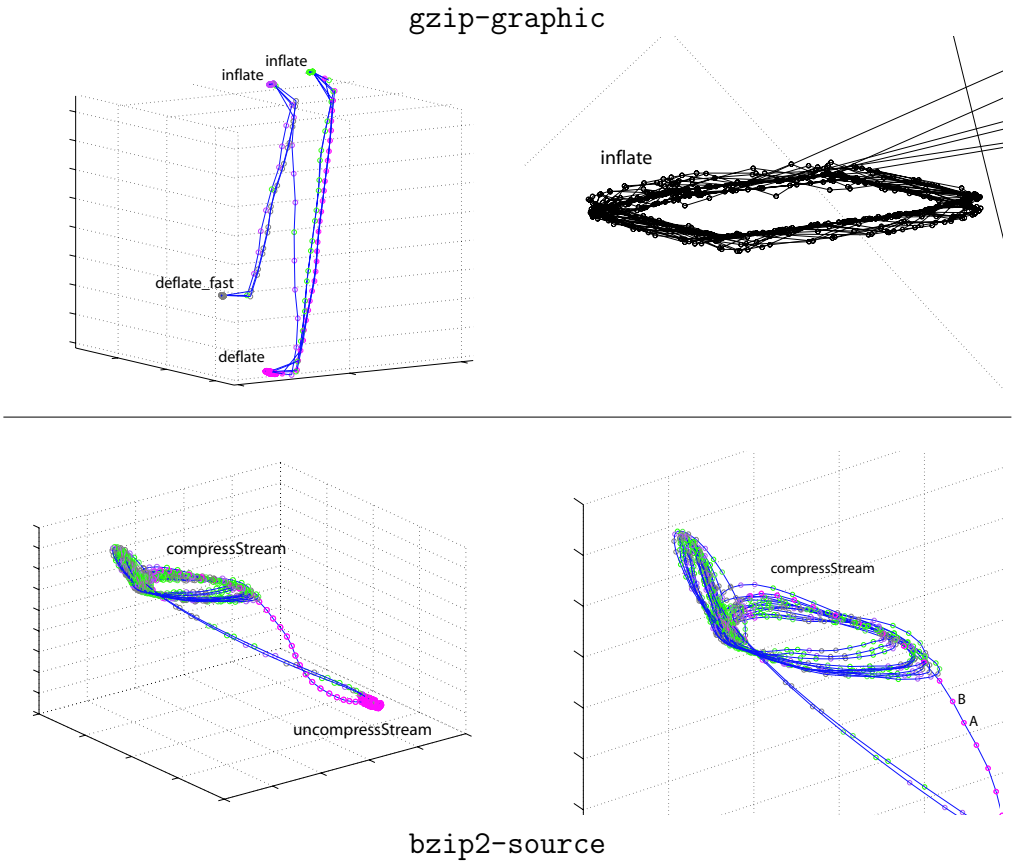


Figure II.4: Three dimensional non-accumulated representation of `gzip-graphic` and `bzip2-source`. Each point represents an interval during execution, and the line connecting the points shows the order in which the intervals are executed in time. The right figure of `bzip2-source` has two points labeled A and B, which indicate two temporally adjacent intervals of program execution (A is executed first). The figures on the left plot the entire execution, while the figures on the right zoom in on a looping region in the execution.

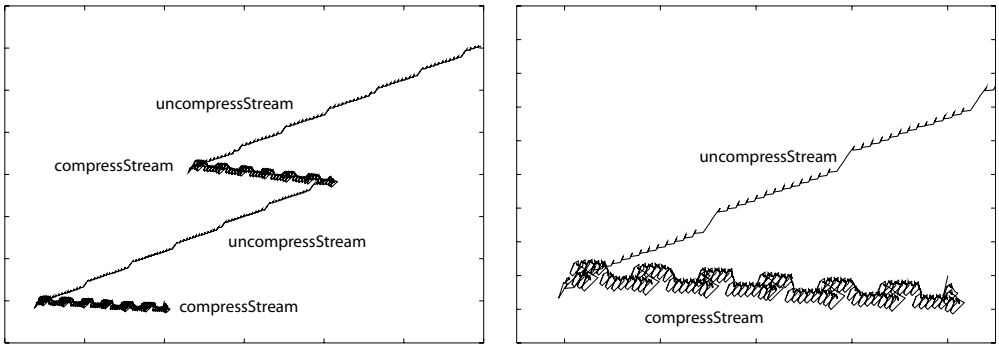


Figure II.5: Two dimensional accumulated representation of `bzip2-source`. The figure on the left was produced by calculating the running sum of the vector data in Figure II.4. Each point in this figure is a projected basic block vector representing the program's execution from start of execution to the interval represented by the point. The figure on the right shows detail of the bottom left corner of the figure on the left.

The resulting figure provides a graphical representation of how the program's usage of its code changes over time. If a program spends many consecutive intervals executing the same code in the same proportions, those intervals will appear very close together in the figure. If interval boundaries were always synchronized with phase transitions, then the figure would show a small number of interconnected tight clusters.

If interval boundaries are poorly synchronized with the program's phase transitions, then oscillating patterns will appear, as discussed in the prior subsection. The oscillations should appear as circular patterns when their vectors are plotted. Figure II.4 shows a 3-dimensional representation of the execution of two benchmarks: `gzip-graphic` and `bzip2-source`. A fixed length interval size of 100 million instructions was used.

The `gzip` benchmark from the SPEC2000 benchmark suite repeatedly compresses and decompresses its data 5 times, at compression levels 1, 3, 5, 7, and 9. At compression levels 1 and 3, a faster version of the `deflate` algorithm is

used. This time-varying program behavior is clearly visible from the `gzip` graphs shown in Figure II.4. For example we see that execution transitions back and forth between `deflate_fast` and `inflate` 3 times (`deflate_fast` → `inflate` → `deflate_fast` → `inflate`), corresponding to compression and decompression at levels 1 and 3. There are 5 transitions between `deflate` and `inflate`, corresponding to compression and decompression at levels 5, 7, and 9.

The vectors collected from each deflation and inflation phase form a torus. Each cycle around the torus corresponds to the compression or decompression of a block of data. If the interval length was properly synchronized with program behavior, each operation on a block of data could be represented with a single interval. But Figure II.4 shows that these fixed length intervals are out of sync with the behavior of the program, resulting in a large number of irregular intervals capturing the behavior pattern.

Similarly, SPEC2000's `bzip` compresses and decompresses the data twice, at compression levels 7 and 9. Thus, execution transitions between compression and decompression three times, as seen in Figure II.4. Each iteration around the circular structures corresponds to compression of a block of data, as seen in the `gzip` plots. There are two looping structures within the compression phase - these correspond to compressing blocks with different entropy properties. `bzip` performs run-length encoding on its front end, and more time is spent in the run-length encoder on blocks with more contiguous sequences of repetitive bytes.

As seen from these two examples, the majority of a program's execution is spent in loops. The average number of instructions per loop iteration can change over time. For example, fewer instructions are typically needed to decompress a block of data than to compress a block of data. This means that the period changes over time. An ideal fixed interval length for one section of execution (compression) may not be ideal for another portion of the program's

execution (decompression). It is unlikely to find a single fixed interval that can accurately represent a whole program, which motivates the need for variable length intervals that adjust to the period of the program's current behavior pattern.

Figure II.4 shows that it is possible to see periodic behavior in programs by looking at a non-accumulative representation of the program's code space usage over time: the periodic behavior of programs results in cyclic patterns in these graphs. Both these programs also exhibit hierarchical behavior: there is a high-level behavior pattern between compression and decompression, and within each compression and decompression phase, there is a low-level behavior pattern corresponding to each block of data compressed or decompressed.

2D Accumulated Representation

Another way to examine a program's execution is with an accumulative representation. With this approach, each interval of execution is represented with a basic block vector that tracks the total number of times each basic block is executed from the beginning of execution to the current interval. A fixed length interval size of 1 million instructions was used. Figure II.5 shows a two-dimensional projection of accumulated basic block vector data for `bzip2-source`. Each point represents the accumulated basic block vector from the start of execution up to a fixed length interval boundary, and the lines connecting the points indicate the order in which the intervals were executed.

This graph shows that it is easy to find stable program behavior by looking at an accumulated representation, because stable behavior results in a straight line. If the program is executing the same distribution of basic blocks, the accumulated representation will show a straight line, because the same dimensions of the accumulated vector will be increased by the same quantities. Whenever the line bends, the program is executing a different distribution of basic blocks,

and is therefore exhibiting a different behavior pattern.

SPEC2000's `bzip2-source` benchmark fills a 58MB buffer with back-to-back copies of a tarfile containing source for some SPEC benchmarks. The 58MB buffer is large enough to hold 6.4 copies of the source tarfile. The tarfile contains a large number of null bytes at the end. This buffer is first compressed with a block size of 700KB, then decompressed, then compressed again with a block size of 900KB, and finally decompressed.

All these properties are visible in Figure II.5. The 6.4 copies of the input file can be seen most easily in the decompression phases. The line shifts upwards every time the end of the original input file is reached, because of the large number of null bytes present at the end of the input file. This changes the entropy of the block, which causes `bzip2` to execute code in different proportions - more time is spent doing RLE decompression, and less time spent inverting the Burrows-Wheeler transform. Each circle during compression corresponds to compressing a block of data, and each spike during the decompression phase occurs when writing a decompressed block of data. Thus, the block size is not directly visible, but the number of blocks is.

When a block size of 700KB is used, there are 14 blocks per copy of the original input file, and if you look closely, there are 14 little spikes within each "plateau" during the first decompression phase. When a block size of 900KB is used during the second decompression phase, there are 11 blocks per copy of the original input file, and there are 11 little spikes per plateau visible in the second decompression phase. The same patterns can be seen by counting loops in the compression phases.

II.C Software Phase Markers

The last section motivated the need for variable length intervals. This section proposes software phase markers, which are used to align interval boundaries with a program's phase transitions, instead of relying on fixed length intervals. The key idea presented in this section is to use the code at each procedure or loop boundary as a *software phase marker* that, when executed, signals a phase change, without any hardware support. These software phase markers are selected by directly analyzing each program's procedure call and loop iteration patterns.

The analysis is driven by a hierarchical call-loop graph. A call-loop graph is much like a traditional call graph, except that it has additional nodes for loops, and additional annotations noting the variance on all paths to each call or loop. Call-loop graphs can be quickly collected from a running program through the use of a profiling tool. A simple and fast algorithm is used to select code structures to serve as software phase markers from the Call-Loop graph. Software phase markers accurately identify phase changes at the binary level, with no hardware support, across different inputs to a program.

II.C.1 Capturing Hierarchical Behavior with Call-Loop Graphs

This subsection describes the hierarchical call-loop graph which guides the selection of software phase markers. The call-loop graph is a call graph extended with nodes for loops, where each node and edge is annotated with hierarchical instruction counts and standard deviation of the hierarchical instruction count.

Procedures, Loops, and Phase Behavior

Huang et al. [41] found that program behavior tends to be fairly homogeneous across different invocations of the same procedure. This section shows that this result extends to loops as well: program behavior across loop iterations and across different executions of the same loop nest are fairly homogeneous. This will be shown in the next section by measuring the variance in program metrics across a program's procedures and loops. By placing phase markers on loops in addition to procedures, smaller interval sizes can be achieved compared to procedures alone.

While procedures are sufficient for detecting phase behavior in many programs, such as Java applications with small object oriented routines [33], in general it is important to also examine a program's loop structure to identify repetitive program behaviors, because using procedures alone places a dependence on the application programmer to divide their code into meaningful subroutines. Prior work has shown that tracking loops and procedures together can effectively detect phase behavior, while tracking procedures alone was not as effective [56].

The call-loop graph has nodes for both procedures and loops. Each node and edge in the call-loop graph carries the call count, total local and hierarchical dynamic instruction count, as well as the average and standard deviation of the hierarchical dynamic instruction count.

Creating a Call-Loop Graph for Finding Phase Behavior

The call-loop graph is constructed by analyzing binaries with ATOM [81], a binary instrumentation tool. Procedures are detected by ATOM, and loop back edges are detected by identifying non-interprocedural backwards branches. A loop is the static code region from the backwards branch to its target. There are nodes for both procedures and loops in the call-loop graph.

The call-loop graph tracks the hierarchical instruction counts on each edge. For a call, for example, this is the total number of instructions executed between call and return. For each edge the graph tracks the following four pieces of data:

1. for procedures, the number of times the procedure is called, and for loops, the number of times the loop iterates
2. the average number of instructions executed on each edge
3. the maximum number of instructions executed on a single traversal of the edge
4. the standard deviation of the number of instructions executed on each edge

In the call-loop graph, each procedure and loop is represented with two nodes to handle recursion and iteration. Each procedure and loop is represented with a head node and a body node. Every head node always has exactly one child, which is its corresponding body node.

Loop head nodes track how many instructions execute between loop entry and exit, while loop body nodes track how many instructions execute between loop iterations. If a loop head node is selected as a phase marker, then loop entry points are marked, and if a loop body node is selected as a phase marker, then loop backedges are marked.

Similarly, procedure head nodes track the number of overall instructions executed for recursive procedures between entry to the recursive procedure and exit from the recursive procedure, similar to loop head nodes. Procedure body nodes track instructions executed for each recursive iteration, similar to the loop body nodes. For non-recursive procedures, procedure head and body nodes carry identical information.

```
Proc foo()
   ...
   loop
      ...
      if (cond) call X;
      else call Y;
      ...
   end loop
   ...
   call X;
   ...
end proc

Proc X()
   ...
   call Z;
   ...
end proc
```

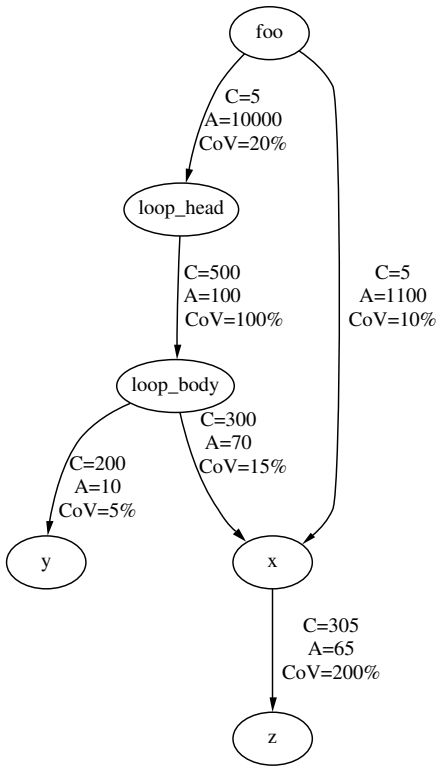


Figure II.6: Code example and Call-Loop graph for code example. C is the number of times each edge is traversed, A is the average number of hierarchical instructions executed each time the edge is traversed, and CoV is the hierarchical instruction count coefficient of variation. For procedures, only the procedure-head nodes are shown, since the example does not have recursion. Maximum instruction counts are also not shown to save space.

By representing each procedure and loop with head and body nodes, the call-loop graph allows for the detection of more stable program behaviors. For example, a loop body node will indicate if a loop does similar work on each iteration, and the loop head node will indicate if a loop does similar amounts of work between each loop entry and loop exit.

Figure II.6 shows a piece of code and its corresponding call-loop graph. The call-loop graph has been simplified to improve its legibility. Because there is no recursion in this example, only procedure head nodes are shown in this

call-loop graph. To improve legibility, the maximum instruction counts are not shown. Procedure `foo` contains a loop and the `foo` → `loop_head` edge notes the hierarchical instruction count from loop entry to loop exit. In comparison, the `loop_head` → `loop_body` edge notes the hierarchical instruction count for each loop iteration. Two nodes are used to represent the loop, and the weight of the `foo` → `loop_head` edge indicates the number of times the loop is entered, and the `loop_head` → `loop_body` edge indicates the number of times the loop iterates.

Each edge in Figure II.6 is annotated with three values: the number of times the edge was traversed (C), the average number of instructions executed each time the edge is traversed (A), and the standard deviation of the number of instructions across invocations, which is represented by the Coefficient of Variation (CoV). CoV is just standard deviation divided by average. CoV is used to identify edges with low variance, which are candidates for phase marker selection, as described in the next subsection.

II.C.2 Selecting Software Phase Markers

Software phase markers are points in the binary that can be instrumented (branches, procedure calls, returns, loop entries, the start of a procedure, etc) to reliably indicate the beginning of an interval of repeating program behavior when executed. These software phase markers can be used to easily and accurately predict program phase changes at run-time with no hardware support. In addition, software phase markers can be used to predict phase changes across different inputs to a program, and across different compilations of the same source code.

In this subsection, the software phase marker selection algorithm is presented. Given a call-loop graph as described in the prior section, the algorithm selects phase markers that can be monitored in a program with a static or dy-

namic compiler or binary instrumentation.

Selecting Markers from the Call-Loop Graph

Many programs exhibit different repetitive behaviors at different time scales. When selecting software phase markers, a time scale must be specified by the user - the selection algorithm needs to know whether the user is interested in large or small scale behaviors. For example, optimizations with high overheads will be more interested in large-scale phase behavior, since they require more time between optimizations to recoup the cost of applying each optimization. The call-loop graph can be used to find both large and small scale phase behaviors. The algorithm presented in this subsection identifies repetitive phase behaviors at the desired granularity by first considering small granularities in the call-loop graph and moving upwards toward larger granularities. The algorithm seeks to satisfy two conflicting constraints:

1. There must not be too much or too little distance between marker points
2. Each marker must indicate the start of a repetitive program behavior with high probability

The call-loop graph in Figure II.6 provides an example that illustrates how the hierarchical instruction count CoV guides the selection of software phase markers. Each edge in the Figure shows `C` - the number of times the edge was traversed, `A` - the average hierarchical number of instructions executed each time the edge was traversed, and `CoV` - the hierarchical instruction count coefficient of variation (standard deviation divided by average). In this example, the `X` $\rightarrow$ `Z` edge has a CoV of 200%, and each time `Z` is executed on the path from `loop_body`, `Z` executes 50 instructions on average. On the other hand, each time `Z` is executed on the `foo` $\rightarrow$ `X` $\rightarrow$ `Z` path, `Z` executes 1000 instructions on average.

Therefore, the `X` → `Z` edge exhibits high CoV in hierarchical instruction count per edge traversal. But `X`'s incident edges allow for additional path differentiation that allow `X`'s behaviors to be cleanly separated into different classes based on the callsite. Because the `loop_body` → `X` and `foo` → `X` edges both have a low CoV, the number of hierarchical instructions executed on each edge is similar every time the program traverses one of these edges. These two edges are good candidates for software phase marker selection.

The example in Figure II.6 will also be used to demonstrate the phase marker selection algorithm. The algorithm takes as input a threshold for the minimum average interval size (hierarchical instruction count). For this example, the instruction count threshold is set to 90, so the algorithm searches for edges where `A` ≥ 90. The edges `foo` → `loop_head`, `foo` → `X` and `loop_head` → `loop_body` are the only edges that qualify as potential software phase markers. In addition, the algorithm also uses a CoV threshold, which is set based on the call-loop graph as described below. Returning to the example shown in Figure II.6, if the CoV threshold is set to 50%, then the `loop_head` → `loop_body` edge would not be considered, because its CoV is 100%. This indicates that there is too much variation in hierarchical instruction count for each iteration of the loop. In comparison, each time the loop was entered and exited (on the `foo` → `loop_head` edge), the CoV was low, indicating the hierarchical instruction counts across all loop `invocations` were similar. Therefore, it is better to place the phase marker on the `foo` → `loop_head` edge. The end result would be the placement of the phase markers on edges `foo` → `loop_head` and `foo` → `X`.

The phase marker selection algorithm runs in two passes. The first pass prunes the call-loop graph based on the desired average number of instructions per interval. When a marker is placed on an edge in the call-loop graph, the average number of instructions per interval for that marker will equal the average

hierarchical instruction count on that edge (shown as `A` in Figure II.6). The second pass of the algorithm searches the remaining nodes in the call-loop graph in reverse depth order, looking for edges with a low average hierarchical instruction count CoV. This algorithm therefore takes two inputs: a call-loop graph and *ilower*. *ilower* specifies the minimum allowed interval size, which is the minimum number of instructions allowed per interval. A CoV threshold is used to limit the variability in instruction count, but it is automatically calculated by the first pass of the algorithm.

Pass 1 - Pruning based on the average hierarchical number of instructions: The algorithm first estimates the maximum depth of each node in the call-loop graph, by performing a modified depth-first search. In this depth-first search, a node can be traversed more than once when a longer path to that node is found. The algorithm never revisits a node on the current path, to ensure the algorithm terminates for graphs that contain cycles. All nodes in the call-loop graph are then placed in a queue which is sorted by decreasing estimated maximum depth. The queue ensures that the algorithm will consider children before parents, to the best of its ability.

Ties are broken by sorting by increasing out-degree, so the algorithm processes leaf nodes before internal nodes. As each node is taken off the queue, its incoming edges are examined. If the average number of instructions executed on an edge satisfies the *ilower* requirement, the edge is noted as a potential software phase marker. After all the incoming edges on a node are examined, the algorithm continues to the next node in the queue.

When there are no nodes remaining in the queue, the algorithm has generated a list of potential software marker edges, where all of the edges are above the average number of instructions allowed per interval. The second pass will use these edges to calculate a CoV threshold, and select a subset of these

edges to use as phase markers.

Pass 2 - Setting and applying the hierarchical instruction count CoV threshold: The first pass of the algorithm pruned the lower parts of the call-loop graph. The pruned nodes and edges represent low-level behavior patterns that are too small to be of interest, according to the *ilower* threshold. The results of the first pass are used to set a CoV threshold for phase marker selection.

The candidate software phase markers identified in the first pass of the algorithm are used to calculate a CoV threshold. The CoV threshold is calculated independently for each program, because different programs have different levels of variability. For example, floating point programs tend to have more stable instruction counts within each loop and procedure, while integer programs tend to be more variable. By tuning our CoV thresholds to the variability detected in each program, the algorithm can find relatively stable behavior patterns even in highly variable programs.

The base CoV threshold is set to the average CoV ($avg(CoV)$) across all edges in the list of potential phase markers. The standard deviation of the CoVs in the list of potential phase markers is also calculated, and the CoV threshold applied to an edge is in the range $[avg(CoV) \ldots avg(CoV)+stddev(CoV)]$, scaled linearly with the current edge's average hierarchical instruction count, to encourage the algorithm to select phase markers with instruction counts close to *ilower*, by allowing more variability as the average instruction counts become larger.

After a *cov_threshold* is determined for an edge, edges are examined as in the first pass, except that now an edge must satisfy both the *ilower* minimum instruction count threshold, and the *cov_threshold* variability limit to qualify as a phase marker.

Complexity of the Algorithm: The algorithm's running time is $O(E + N * log(N))$, where N and E are the numbers of nodes and edges in the graph. The $N * log(N)$ is due to the sort of all of the nodes to create a total call-loop depth ordering of the nodes during the first pass of the algorithm. The algorithm runs in seconds on every call-loop graph we have collected. The approach should be significantly faster and less complex than the approach of Shen et al. [76], where wavelet analysis [17] is applied to reuse distance traces, and Sequitur [68] is applied to the results of the wavelet analysis.

Limiting Maximum Interval Size for SimPoint

The algorithm described in the preceding subsection is effective for finding homogeneous behavior to guide reconfiguration optimizations. But the algorithm does not limit the maximum size of intervals. For this reason, when evaluating this algorithm later, it will be referred to as *no-limit*. Because there is no limit on the maximum interval size, the algorithm may create large intervals.

Phase markers are attractive for accelerated architectural simulation because they clearly identify transition points between regions of stable program behavior. Interval boundaries can be set at phase markers, resulting in variable length intervals. Unfortunately, the phase marker selection algorithm described in the preceding subsection does not limit maximum interval length, which is not acceptable for accelerating simulation - the goal of accelerated simulation is to simulate a small representative portion of program behavior. If a program contains large intervals, it will be difficult to simulate a small portion of the program if any of the large intervals are selected for detailed simulation.

This subsection describes modifications to the phase marker selection algorithm that limit the maximum interval size. Maximum interval size is limited by adding two additional steps to the second pass of the algorithm. This modified

algorithm will be referred to as *max-limit* when evaluating the algorithm later.

Maximum Interval Limit: While collecting the call-loop graph, the profiler also records the maximum hierarchical instruction count on each edge. While searching for phase markers in the call-loop graph, if the maximum hierarchical instruction count on a node's incoming edge exceeds max_limit, all of the edge's successor's outgoing edges are marked, because those edges must be below the maximum interval size limit, and there is no other way to remain under the maximum interval size limit - if the algorithm were to continue searching upwards in the call-loop graph, the interval sizes would only become larger.

Merging Loop Iterations: The second pass of the algorithm also attempts to group consecutive loop iterations if each iteration has a similar average hierarchical instruction count. If the edge from a loop-head to a loop-body is below the CoV threshold, consecutive loop iterations are grouped together until they are greater than *ilower* and less than max_limit. Typically there are many ways to group loop iterations within these limits. For example, the algorithm might be able to group pairs of consecutive loop iterations together, or possibly arrange iterations in groups of three or four. The call-loop graph indicates the number of times each loop iterates on average, so the modified algorithm attempts to group N iterations, where $avg_iterations \bmod N \approx 0$. In other words, the modified algorithm searches for a value of N that evenly divides the number of loop iterations per entry to the loop nest that satisfies the above interval size constraints.

The modified phase marker selection algorithm presented in this subsection is designed specifically for use with SimPoint to reduce simulation time. Markers found with these additional constraints can be fairly input specific, so this approach is only suggested for use in scenarios where the inputs to the programs will remain the same. This approach is not designed to capture behavior

across inputs.

For the `limit` phase marker results presented in this section, a minimum interval length ($ilower$) is set to 1 million instructions and the maximum interval length (max_limit) is set to 200 million instructions. After phase markers have been selected with the modified marker selection algorithm, one variable length interval basic block vectors is collected for each phase marker, and these basic block vectors are run through SimPoint to select simulation points.

II.C.3 Support For Variable Length Intervals in SimPoint

With variable length intervals, different intervals may represent different amounts of program execution, as opposed to fixed length intervals, where every interval represent the same amount of execution. Each variable length interval has an associated weight w_i, which is the fraction of total program execution represented by that interval. SimPoint 3.0 contains modifications to support these weights [36].

The k-means clustering algorithm has two steps that it repeats: determining which cluster each interval belongs to (called the expectation step), and repositioning each cluster center to the mean of the intervals that it owns (called the maximization step). The expectation step is not changed by variable length intervals. The maximization step handles weights w_i by using the weights during the recomputation of cluster centers - the center of a cluster is computed by using the weights w_i to compute a weighted mean over the vectors corresponding to the variable length intervals in the cluster.

When SimPoint 3.0 is run with variable length intervals, larger intervals have more influence than smaller intervals when determining cluster centers.

Table II.1: Baseline Simulation Model.

I Cache	16k 4-way set-associative, 32 byte blocks, 1 cycle latency
D Cache	16k 4-way set-associative, 32 byte blocks, 1 cycle latency
L2 Cache	128K 8-way set-associative, 64 byte blocks, 12 cycle latency
Main Memory	120 cycle latency
Branch Predictor	Hybrid - 8-bit gshare with 2K 2-bit predictors and an 8K bimodal predictor
Out of Order Issue	Up to 8 operations per cycle, 64 entry re-order buffer
Memory Disambiguation	Load/store queue, loads may execute when all prior store addresses are known
Registers	32 integer, 32 floating point
Functional Units	2-integer ALU, 2-load/store units, 1-FP adder, 1-integer MULT/DIV, 1-FP MULT/DIV
Virtual Memory	8KB pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

II.C.4 Methodology

The results presented in sections II.C.5 and II.C.6 evaluate software phase markers for several benchmarks from SPECINT2000 benchmark suite.

The baseline architecture modeled is described in Table II.1. Each of the SPEC programs were simulated to completion to collect baseline results. Section II.C.6 also presents results for a data cache reconfiguration experiment, which uses a lightly modified version of the ATOM [81] cache simulator used in [76]. For the data cache reconfiguration experiment, phase markers are compared to the reuse distance-based software phase marking approach of Shen et al. described in [76]. Shen was very gracious to provide the authors with his binaries, his reuse distance phase markers, and his ATOM-based Cheetah simulator, which allow for a fair comparison with his approach. For this comparison, the same binaries are presented as in [76], which are `tomcatv`, `swim`, `compress95`, `mesh`, and `applu`.

II.C.5 Phase Marker Evaluation

This section presents an evaluation of the proposed software phase marker selection algorithm.

Using Software Phase Markers

Software phase markers are selected so that they reliably predict the beginning of a repetitive interval of program execution. The most straightforward application of software phase markers is to use them as triggers for dynamic reconfiguration or optimization. This can be done by inserting code into the binary at each phase marker to trigger reconfiguration or optimization. This can be done with a binary modification tool such as OM [82] or ALTO [64].

Figure II.7 shows how phase markers predict the beginning of repetitive program behaviors. In this figure, time is on the X-axis, measured by instructions

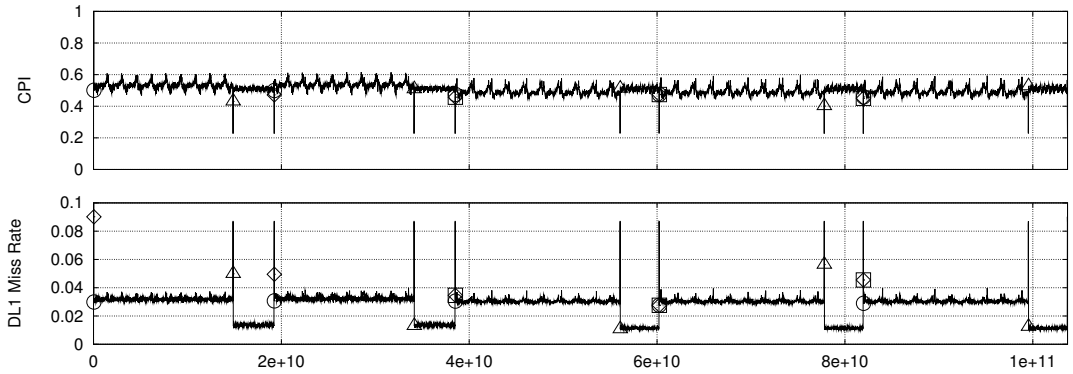


Figure II.7: Time varying graphs with phase markers for gzip-graphic for an OSF Alpha executable. Time is on the X-axis, measured in instructions executed. Phase marker locations are indicated with symbols. Each marker is assigned a unique symbol. If a marker occurs many times in a row, we only plot the first instance from each repeating run to make the graph more readable.

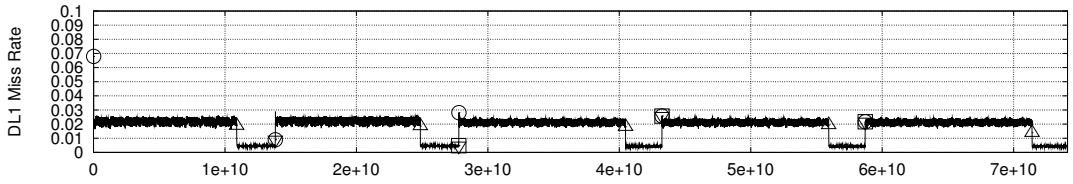


Figure II.8: Cross-binary time varying graphs with phase markers for gzip-graphic for Linux x86. The phase markers were selected from the call-loop graph profile from the Alpha binary, were mapped back to source code level, and then used to mark the x86 binary. No call-loop graph was created for the x86 binary. The markers detect the same high-level patterns in the x86 binary.

executed, and CPI and level 1 data cache miss rates are on the Y-axes. Phase marker locations are indicated with symbols (circles, squares, etc) plotted on top of the CPI or data miss rate. Each phase marker is assigned a unique symbol. In Figure II.7, the 2 large sized phases found are between the circles and triangles. In Figure II.7, the beginning of each long high miss rate phase is marked with a circle, and the beginning each short low miss rate phase is marked with a triangle.

There are many more phase markers than shown in Figures II.7 and II.8, but to improve the legibility of the graphs, only the first marker is shown at the start of each consecutive run of marker invocations. There are actually phase markers at each ridge shown during the long stable regions of high miss rate (circle) and low miss rate (triangle).

Figure II.8 shows how phase markers selected for `gzip-graphic` from an OSF Alpha binary can be used with a Linux x86 binary. For this result, the markers are selected on an Alpha and mapped back to source code level, using debug line number information. Then the symbol information from the x86 binary is used to map the source-level markers to the corresponding assembly instructions in the x86 binary. This matching technique was used with other benchmarks and similar results were found. This figure shows how phase markers can potentially be used across different compilations of the same source program.

Figures II.9 and II.10 are visual representations of the complete execution of `bzip2-graphic`, as previously described in Section II.B.2. To recap, these figures are a three-dimensional projection of the basic block vectors collected from each execution interval, where each interval is represented with a single point in the figures. Each interval is projected to three dimensions with random linear projection and plotted as a point in the graph. Figure II.9 shows fixed length 100 million instruction intervals and Figure II.10 shows variable length intervals generated by software phase markers. In both of these figures a similar number

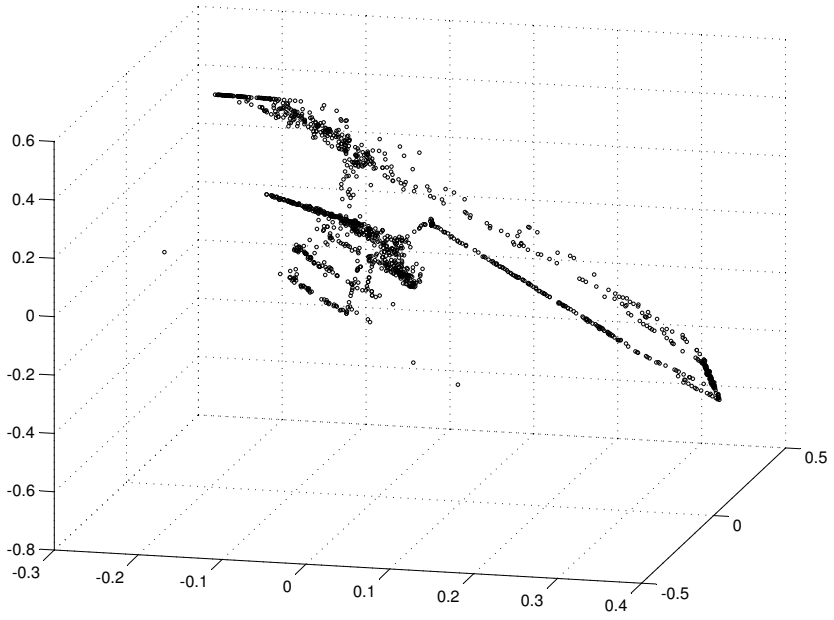


Figure II.9: Bzip2 fixed length execution intervals representation. The scattered representation with points spread across the space is a direct consequence of using fixed length intervals across the execution.

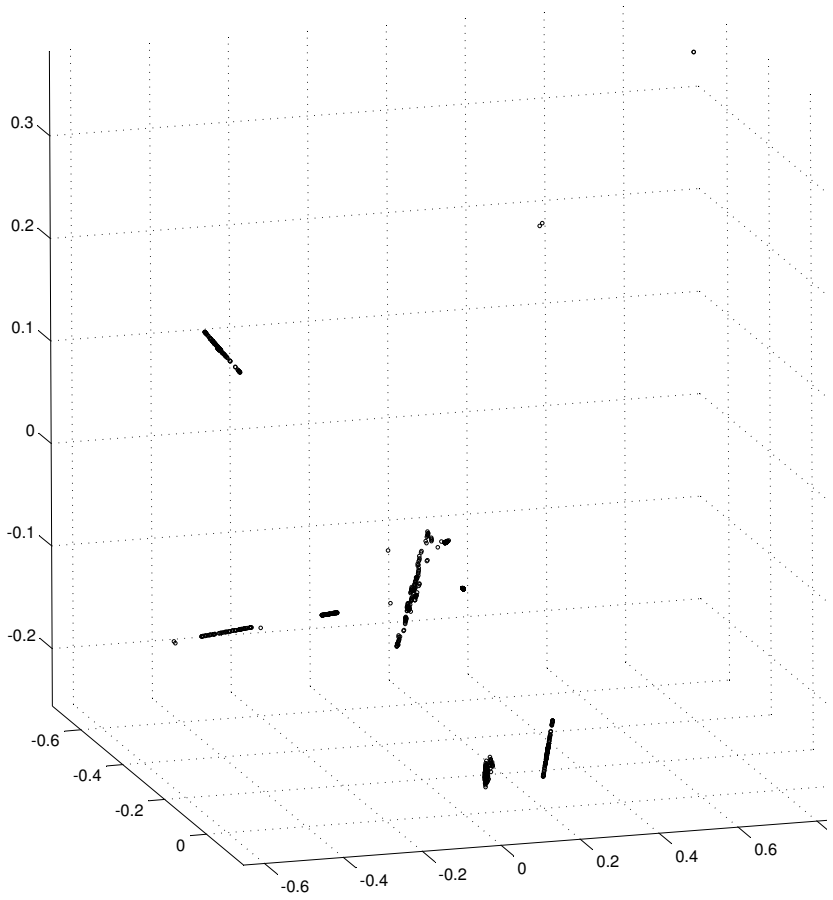


Figure II.10: Bzip2 variable length execution intervals representation with phase markers. The tight clustering of intervals is from marking regions of the hierarchical call-loop graph that have fairly homogeneous behavior each time that edge is traversed during execution.

of intervals are required to represent the entire execution of the program. The same random projection matrix was used for Figures II.9 and II.10, but they are shown at different angles. The angle was chosen for each graph to best show how each set of intervals captures the program's use of its code space.

Bzip2 spends the majority of execution in several code regions, and transitions between these regions just a few times. The dominant code regions can be seen in both figures as dense clouds of points. These code regions are substantially more clear in Figure II.10 compared to Figure II.9. The transitions between these code regions can be seen in Figure II.9 as a string of points connecting the denser regions. These transitions are not visible in Figure II.10 since the entire phase representation is synchronized with the program behavior where transitions between dominant regions are encapsulated by unique intervals.

These figures provide visual evidence that software phase markers are partitioning the execution into naturally occurring intervals that are synchronized with the program's behavior changes. On the other hand, the fixed length intervals are not synchronized with program behavior. This is why program execution appears more chaotic with fixed length intervals in this figure.

Behavior Characteristics of Software Phase Markers

This subsection presents results from the marker selection algorithm. *ilower* is set to one million instructions. Two scenarios are presented:

1. Phase markers are selected from training inputs and applied to reference inputs (cross-train),
2. Phase markers are selected and applied to the reference input (self-train).

All results presented were produced by running the reference input. In this section, the results of the proposed phase marking algorithm are compared

to SimPoint [78], an offline phase analysis tool based on the k-means clustering algorithm from machine learning described in section II.A.3. For experiments with SimPoint, basic block vectors were collected from fixed length intervals of ten million instructions, and SimPoint was run with a 15 dimension random projection and $k_{max} = 100$. SimPoint classifies the intervals of execution into phases. This comparison favors SimPoint because SimPoint analysis cannot be used across inputs. This section also presents results where our phase marking algorithm is only allowed to mark procedures. The result is similar to the approaches of Huang et al. [41] and Georges et al. [33], except that our approach uses a call-loop graph, rather than the dynamic call stack approach used in the work of Huang et al. and Georges et al.

In this section, the `no-limit` results correspond to phase markers selected with the algorithm described in Section II.C.2, where no upper bound is placed on the maximum interval length. In comparison, the `limit 1-200m` results correspond to phase markers selected with the modified algorithm described in Section II.C.2 which limits the maximum interval length. For the `limit 1-200m` results, the minimum interval length is 1 million instructions and the maximum interval length is 200 million instructions.

Figure II.11 shows the average length of intervals produced by each approach. The leftmost bar, BBV, uses fixed-length intervals of 10M instruction. The next two bars show the results for the proposed phase marking approach when the selection algorithm is only allowed to select edges entering procedure-head and procedure-body nodes in the call-loop graph. This limits the marker selection algorithm to selecting procedure callsites. The last three bars show results for the suggested phase marking algorithm when any edge may be selected in the call-loop graph.

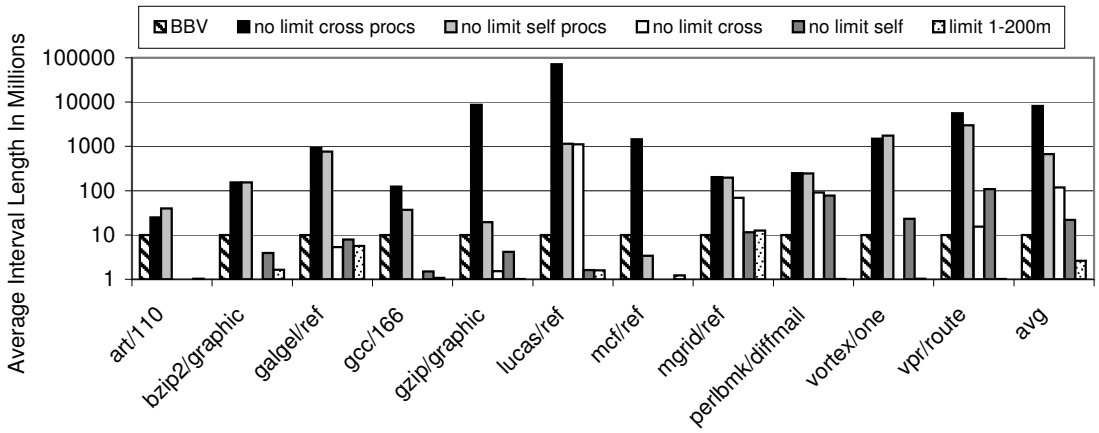


Figure II.11: Average instructions per interval. BBV uses fixed 10M instruction intervals. The remaining results use software phase markers with no limit on the maximum interval length, except for the last bar which limits maximum interval length to 200 million instructions. The second and third bar show results when only procedures can be marked, while the last three bars show results when both procedures and loops can be marked

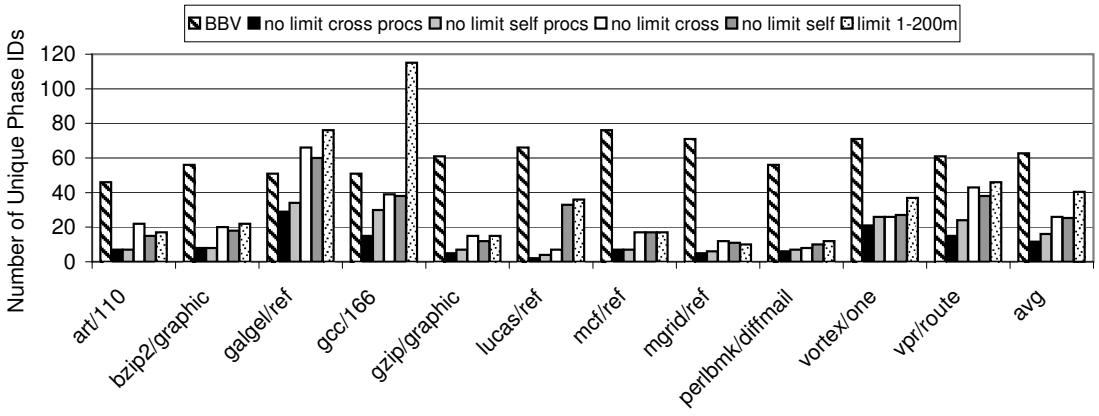


Figure II.12: Number of phases detected

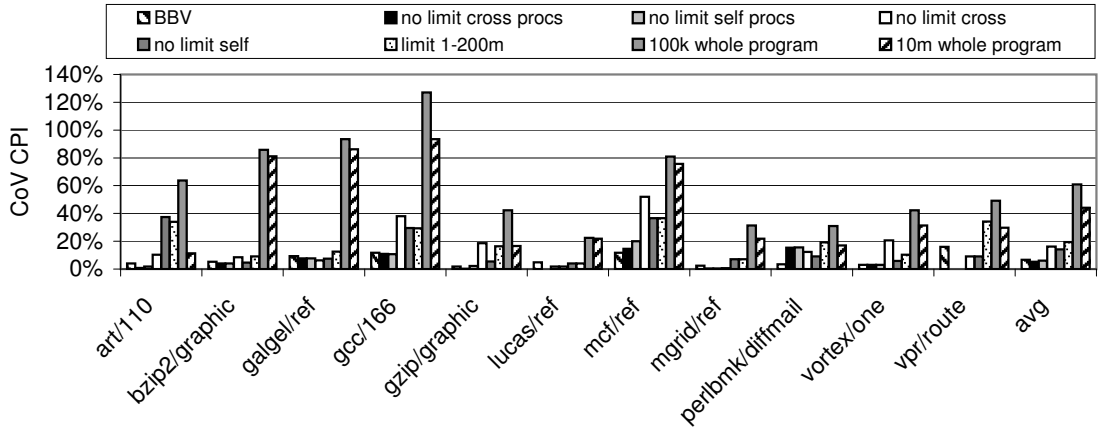


Figure II.13: Coefficient of variation of CPI. The "Whole Program" results show each program's variability if every interval is classified into a unique phase

The last bar in figure II.11 limits the maximum interval length to 200 million instructions. Bars that may appear to be missing actually have an average interval size of 1 million instructions. For the self-train results, the selection algorithm examines a call-loop graph collected by profiling the reference input, and for the cross-train results, the selection algorithm examines a call-loop graph collected by profiling the training input, and the selected markers are used with the reference input.

When the selection algorithm is only allowed to select procedure call-sites, the average interval sizes increase dramatically - they range from 1 billion for self-train to 10 billion for cross-train. Allowing the marker selection algorithm to select loops reduces the average interval size to 10-100 million. The last bar shows that limiting the maximum interval length in the phase marker selection algorithm further reduces the average interval length to 3 million instructions.

Figure II.12 shows the number of phases detected by each approach. The BBV approach detects the most phases, and it also has the lowest variation

in phases as Figure II.13 will show. In most cases, the suggested phase marking approach detects half as many phases as the BBV approach. Limiting the maximum interval length results in more phase markers as expected, because the modified algorithm limits maximum interval length by selecting many markers when necessary.

Figure II.13 shows the average coefficient of variation of CPI per phase. The last two bars in these graphs show the overall CoV for each program with fixed length intervals of 100,000 instructions and 10 million instructions. These graphs show that both the BBV and the software phase marking approach successfully partition execution into phases of homogeneous behavior. The procedures-only results have a lower CoV CPI for some programs compared to the results that use both loops and procedures. This occurs because only marking procedures results in fewer phases but significantly larger intervals compared to marking procedures and loops. Recall from the discussion in section II.A.4 that CoV does not tell the whole story - both CoV and interval length must be considered when evaluating a phase classification. In general, more behavior variability must be tolerated with smaller interval sizes. For example, it is easy to achieve zero CoV by treating a whole program as one large interval. This is essentially what is happening for programs like `vpr` in the procedure-only results - the programs are being partitioned into a small number of large intervals. In general, program behavior variability decreases as interval size increases. In all cases, the average behavior variation within each phase is much lower than the program's overall behavior variability.

II.C.6 Applications: Data Cache Reconfiguration and SimPoint

This section evaluates two applications of software phase markers: a data cache reconfiguration experiment as described in [76], and accelerating ar-

chitectural simulations with SimPoint.

Data Cache Reconfiguration and Comparison to Data Reuse Markers

Data cache reconfiguration dynamically adjusts a processor's cache size to reduce energy consumption without increasing the miss rate. This section presents the same data cache reconfiguration experiment performed by Shen et al. [76] except that reconfigurations are triggered by software phase markers instead of data reuse markers. Shen provided the authors with the benchmarks, data reuse markers, and simulation infrastructure used in [76] as discussed in Section II.C.4. This subsection presents results that simulate the same adaptive cache hardware: 64-byte blocks, 512 sets, 32KB to 256KB cache size. The cache reconfigures by changing associativity from 1 to 8 ways.

The dynamic cache reconfiguration approach of Shen et al. [76] is briefly described here. When a phase marker is encountered for the first time, the first two intervals of its execution are spent experimenting with different cache configurations. At the end of the first two intervals of each phase, the best cache configuration is determined, and subsequent intervals in that phase will automatically switch to the best cache configuration. This dynamic cache reconfiguration approach is also used in this section with software phase markers.

This section also presents results for dynamic cache reconfiguration driven by an ideal SimPoint [78] based approach. SimPoint classifies the intervals of execution into phases, assigning each 10M instruction interval a *phaseid*. For the ideal SimPoint-based cache reconfiguration experiments, the *phaseid*s produced by SimPoint are used to trigger cache reconfiguration at each 10M interval boundary. This ideal SimPoint-based approach has oracular knowledge of the *phaseid* for each upcoming interval, as determined by the offline k-means algorithm. This approach is similar to an idealized version of the hardware BBV phase

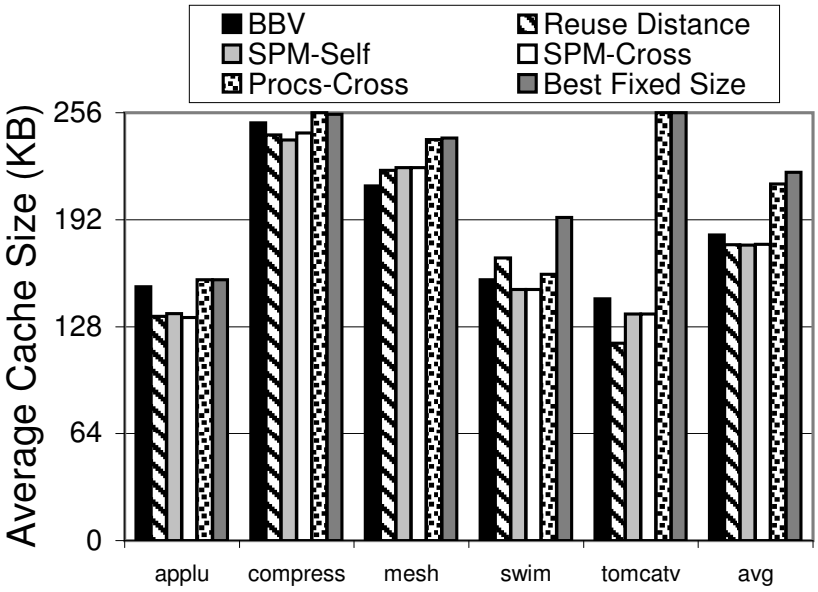


Figure II.14: Average cache size with no allowed increase in cache miss rate. BBV is the idealized SimPoint-based approach, Reuse Distance is the approach of Shen et al. (trained with train input), and Software Phase Marker (SPM) is our approach (trained with ref input (self) and train input (cross)). "Procs only" only allows procedures to be selected as phase markers. Best Fixed Size is the smallest fixed cache size with the maximum hit rate out of all cache configurations considered.

classification approach presented in [79, 57] with perfect next-phase prediction.

Figure II.14 shows the average cache size used by each cache reconfiguration approach on the benchmarks used by Shen et al. [76]. This figure shows the average cache size used over the execution of each program. The Self results use the reference input for phase marker selection and for the cache reconfiguration experiment, while the Cross results use the training input for phase marker selection and apply those phase markers to the reference input for cache reconfiguration.

The benchmarks presented in Figure II.14 have very stable repetitive

behavior patterns: the average coefficient of variation of hierarchical instruction count in marked procedures and loops is less than 1% for all of these programs. This means that it is very easy to predict the number of instructions executed within the program's main procedures and loops with very high accuracy. On these benchmarks, our phase marking approach outperforms the idealized BBV approach, indicating that the fixed-length intervals used by this approach are out of sync with the phase behavior exhibited by these programs, resulting in inferior phase classifications. For example, our phase marking approach selects intervals with an average of 4M instructions for `applu`, yet the idealized BBV approach uses fixed length intervals of 10M instructions. 10M instruction intervals will not work well in this case, and the results show that the average cache size with the idealized BBV approach is near the best fixed cache size for `applu`.

The results also show that the proposed simple software phase marking approach is as effective as the more complicated reuse distance-based approach of Shen et al. [76] for cache reconfiguration on these programs. The results also show that selecting phase markers from the training input is as effective as selecting markers from the reference input for these benchmarks, which is to be expected due to the highly regular behavior patterns exhibited by these programs. Examining only procedures does not work well for some of these programs because these programs tend to spend most of their time in loops, resulting in a small number of marked procedure call edges, as discussed in the previous subsection. Shen et al. [76] did not provide the authors with their reuse-distance analysis tools, so the authors were unable to experiment with their reuse-distance approach for benchmarks with more irregular behavior such as `gcc` and `vortex`. For `gcc`, the best fixed cache was 256KB and with dynamic cache reconfiguration based on software phase markers, the average cache size was 240KB. For `vortex`, the best fixed cache was 245KB and the average cache

size with dynamic cache reconfiguration with software phase markers was 200KB.

Using Phase Markers with SimPoint

The baseline SimPoint approach is highly accurate even with fixed-length intervals. SimPoint works very well even with imprecise phase classifications because its goal is to select a single representative interval from each phase. It is not a major concern if the intervals in a phase exhibit somewhat irregular behaviors, as long as a representative can be selected that captures the average behavior of the intervals in the phase. Simulation benchmarks typically run for billions or even trillions of instructions, which results in plenty of intervals for SimPoint analysis, and SimPoint only needs to select a tiny fraction of these intervals as representatives. As more intervals are available, it becomes increasingly likely that a good representative for each phase will be present, regardless of how well the interval boundaries are synchronized with program behavior.

Therefore, phase markers should not be expected to significantly improve SimPoint's accuracy. Rather, the major advantage of using phase markers with SimPoint is that simulation points based on phase markers become portable across inputs, and across recompilations. Because phase markers correspond to source-level code structures, simulation points based on phase markers can even be used across different architectures.

This section examines how phase markers can be used to partition a program's execution into intervals by setting interval boundaries at phase markers. The modified phase marker selection algorithm with a limit on maximum interval length described in Section II.C.2 is used in this subsection, with a maximum interval size threshold of 200 million instructions. The average interval length, number of phase IDs and CoV of CPI results for these `limit` phase markers are shown in Figures II.11, II.12, and II.13.

To produce the variable length interval SimPoint results, phase markers are used to partition each program's execution into variable length intervals. Whenever a marker is encountered during execution, a new interval is created. As the program's execution is partitioned into intervals in this manner, a basic block vector for each variable-length interval is also collected.

These variable-length basic block vectors are run through SimPoint 3.0 [35] to select simulation points. Older versions of SimPoint will not work for this purpose, because they assume that each interval represents an equal fraction of program execution, which is not the case with the variable-length intervals produced by phase markers.

Figure II.15 shows the number of instructions simulated, and Figure II.16 shows error in estimated CPI. In these graphs, the first three bars show the number of instructions simulated with the baseline SimPoint [78] approach with fixed length intervals of 1, 10 and 100 million instructions. For the fixed length interval results, k_{max} was set to 300 for the 1 million interval, 30 for 10 million interval, and 10 for 100 million interval as described in [72]. To produce the estimated CPIs used in Figure II.16, each representative interval was simulated with perfect warm up, and the CPI results for each representative interval were weighted by the size of the clusters they represent to produce an overall CPI estimate.

The last three bars in these graphs shows results for variable length intervals (VLIs) created by phase markers. The last bar (VLI 100%) shows the results when a simulation point is selected from each cluster. These graphs also present results for a common optimization where clusters are sorted by their weight, then simulation points are selected from the heaviest N clusters which account for 95% or 99% of execution. This is a common technique to trade simulation time for accuracy [35].

For the VLI results in Figures II.15 and II.16, the numbers of clusters

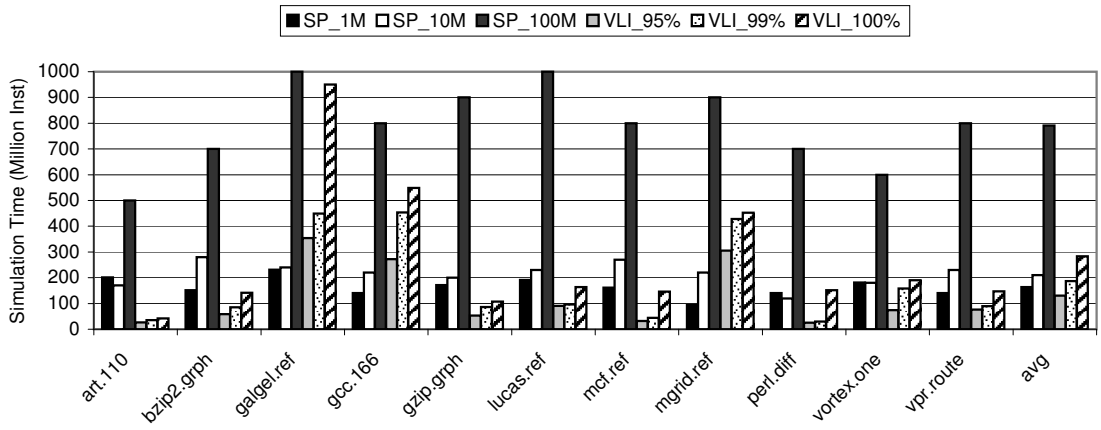


Figure II.15: The number of simulated instructions when fixed length intervals of 1, 10 and 100 million instructions are used with baseline SimPoint, and when phase markers are used to partition program execution into variable length intervals.

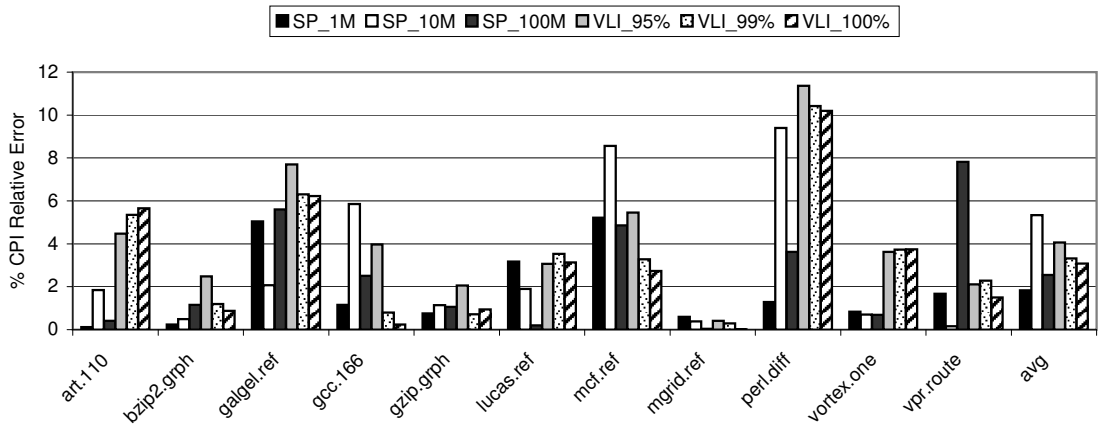


Figure II.16: Error in CPI estimates with fixed length intervals and variable length intervals from phase markers.

and instructions simulated are highly correlated with the number of phase markers shown in Figure II.12. This is because the phase markers mark boundaries between different code behaviors, so the basic block vectors created for different phase markers are expected to be quite different from each other. In Figure II.12, `galgel` and `gcc` have a large number of phase markers for the `limit` approach, because the modified phase marker selection algorithm is forced to select a large number of edges from the call-loop graph to limit the maximum interval length. The call-loop graphs for these programs have several points where the selection algorithm is forced to choose between creating a very large interval, or creating many small intervals. In these cases, to enforce the maximum interval size, the selection algorithm chooses to mark many small intervals.

Overall, these results show that the variable length intervals created by phase markers result in about the same simulation time as 10m fixed length SimPoint, with similar level of error in estimated CPI. The results show that the proposed variable length interval approach does not improve on baseline SimPoint with 1m or 100m fixed length intervals, but the results are comparable, as expected, because variable length intervals improve the homogeneity of intervals in each phase, but SimPoint only requires a single representative from each phase that captures the phase's average behavior.

Cross Binary Simulation Points with Phase Markers

When phase markers are used to define interval boundaries, simulation points can be mapped to source code, and simulation points can be identified across different compilations of the same source. This is useful for using SimPoint to guide architectural studies with new instructions, and to study compiler optimizations.

To evaluate the potential of our approach, a simple experiment was per-

formed where the same software phase markers are used across two compilations of each program. Two Alpha binaries were produced for `gzip`: one without optimization and another with full optimization. Phase markers were selected from the unoptimized binary and used with the optimized binary.

To verify that these phase markers are portable across compilations, traces were collected of each executed phase marker in each binary, similar to the traces shown in Figure II.7. The phase marker traces for the optimized and unoptimized binaries were then compared to verify that they were identical - that the same phase markers were encountered in the same order in both binaries. For `gzip`, the traces were exactly the same.

These results suggest that phase markers can indeed be used to identify portable simulation points, because each simulation point is associated with source-level code structures. Phase markers identify the start of semantically equivalent code regions across compilations of the same source code, so simulation points based on phase markers can be identified across compilations.

Phase markers work well for applications that tolerate long intervals, such as dynamic cache reconfiguration. Unfortunately, accelerated simulation is very sensitive to interval length. A modified algorithm was proposed which limits the maximum length of intervals, but the modifications reduce interval length by introducing a large number of small intervals, which are also not useful for accelerated simulation.

It is difficult to control interval length in a pure software phase marker approach, because call-loop graphs often present situations where the marker selection algorithm must choose between creating a single large interval, or a large number of small intervals. An approach is needed that decouples the portability of software phase markers and the issues with interval length. The next section presents such an approach.

II.D Cross Binary Simulation Points

This section describes cross binary SimPoint, which adapts the phase markers described in the previous section for use with accelerated simulation, so the results of accelerated simulation can be meaningfully compared across multiple binaries compiled from the same source code.

II.D.1 Selecting Cross Binary Simulation Points

This subsection describes the cross binary SimPoint approach to identify semantically equivalent simulation points across a set of binaries for a program/input pair.

Why Variable Length Intervals Are Necessary

To select a single set of simulation points that represent major program behaviors across multiple binaries, the baseline SimPoint approach with fixed length intervals is inadequate. With fixed length intervals, the start and end of each simulation point are identified by instruction counts.

For example, if SimPoint is used with fixed-length intervals of one million instructions, and the first interval is selected as a simulation point, then the simulation point spans dynamic instructions $0 \ldots 999,999$. Because simulation points are tied to dynamic instruction counts, simulation points are not portable across binaries - a different binary will very likely execute a different number of dynamic instructions to perform the same task, which makes it difficult to determine where a semantically equivalent simulation point starts and ends in the different binary.

The main problem is that a simulation point in binary A may start at dynamic instruction count X, but the semantically equivalent part of execution in binary B might start at dynamic instruction count Y (where $X \neq Y$). A

secondary problem is that the semantically-equivalent sample for binary B may execute a different number of instructions than the sample in binary A. Therefore, dynamic instruction counts can not be used as-is to identify the beginning and end of samples with multiple binaries — the beginning and end of each sample must be consistently located across binaries.

Because the binaries are compiled from the same source, this section proposes identifying sample regions where the start and end of each sample region corresponds to a high-level source code construct, such as a procedure call or loop branch, so semantically equivalent sample regions can be identified across all binaries.

Steps for Mappable Simulation Points

The algorithm to identify cross binary simulation points operates as follows:

1. **Collect a Call-Loop Graph from Each Binary:** Collect a call-loop graph as described in Section II.C.1 from each binary. The annotations in the call-loop graph, along with symbol information, are used to identify mappable markers.

2. **Find Mappable Markers that Exist in All Binaries:** Using symbol information and call-loop graph annotations, identify high-level program structures that are present in all binaries. Each node and edge in the call-loop graph corresponds to some high-level program structure, such as a procedure call site, loop backedge, loop entry point, etc.

3. **Select a Binary for Phase Analysis:** With the cross binary SimPoint approach, phase analysis is performed on one binary, and the results are applied to all binaries. The binary selected for phase analysis is called the

"primary binary."

4. **Create Variable Length Intervals Using Mappable Markers:** The primary binary is profiled again to partition its execution into appropriately-sized variable length intervals. Every variable length interval starts and ends on a mappable marker. Basic block vectors are collected for each variable length interval.

5. **Pick Simulation Points for the Primary Binary:** SimPoint is run on the basic block vectors collected from the primary binary's variable length intervals to produce a set of portable simulation points.

6. **Map the Simulation Points to All Binaries:** The simulation points from the primary binary are mapped to the other binaries. This mapping is possible because every interval in the primary binary is mappable (because they all start and end on mappable markers). Because the simulation points selected for the primary binary are just a subset of the primary binary's intervals, the selected simulation points are also mappable.

7. **Recalculate Weights for Mapped Simulation Points:** Different binaries may spend different amounts of their execution in each phase, so the simulation results need to be weighted by the actual amount of execution represented by each simulation point in each binary - not by the amount of execution the simulation point represents in the primary binary.

Each of these steps is discussed in more detail below.

Collect a Call-Loop Graph from Each Binary

Call-loop graphs are collected from each binary for the input being examined using Pin [60], a dynamic instrumentation tool. The call-loop graphs

collected are the same as described in the previous section, with two minor differences: the profiler was ported to profile x86 binaries with Pin (instead of Alpha binaries with ATOM [81]), and loops are detected by identifying natural loops instead of simply looking for backwards branches. Natural loops make it much easier to identify the same loop backedge across multiple binaries compared to the simple loop detection approach described in the previous section.

Find Mappable Markers that Exist in All Binaries

The call-loop graphs collected from each binary are intersected to identify code structures that can be found in all binaries. Code structures that exist in all call-loop graphs and that are executed the same number of times in all binaries are mappable markers.

To intersect the call-loop graphs, the first step is to identify identical procedure head and body nodes across binaries, by identifying procedure nodes with matching procedure names and invocation counts. Procedure names are extracted from the debug symbol information in each binary. If the procedure names and invocation counts for a pair of procedure nodes match across all binaries, then the procedure nodes represent the same part of execution in all binaries.

The next step is to identify identical loop head and body nodes across binaries. Two pieces of information are used to find the same loop across all binaries: invocation counts from the call-loop graph annotations, and debug line number information associated with each branch. If the invocation counts and line numbers for a loop node match across all binaries, then that loop node represents the same part of execution across all binaries.

For both procedure entry points and loop branches, the execution count across all binary versions must match. This guarantees that the mappable markers will execute the same number of times across all binaries. Because every

mappable marker executes the same number of times, semantically equivalent regions of execution can be specified in a portable way by defining start and end of the region in terms of mappable markers - for example, a region can start at mappable marker A after it has executed X times and end when mappable marker B has executed Y times. By defining regions of execution in this manner, semantically equivalent portions of program execution can be identified across binaries.

Select a Binary for Phase Analysis

With the cross binary SimPoint approach, phase analysis is performed on one binary, and the results of phase analysis are applied to all binaries. The binary selected for phase analysis is called the "primary binary."

The primary binary can be selected arbitrarily from the set of binaries available. However, interval sizes can expand or contract across binaries depending on which binary is chosen as the primary. If the other binaries execute more or fewer instructions between interval boundaries compared to the primary binary, then intervals in the other binaries will become bigger or smaller when mapped from the primary.

Create Variable Length Intervals Using Mappable Markers

In this step, the goal is to partition the execution of the primary binary into intervals that are close to the desired size specified by the user (e.g. 100 million instructions).

Each variable length interval must start and end on one of the mappable markers identified in the second step. So to partition the execution of the primary binary into intervals, the primary binary is run, and as it executes, the profiler identifies a set of mappable markers that will result in variable length intervals

that are approximately the size requested by the user.

The profiler must track the number of times every mappable marker is encountered, because any mappable marker may be used to start a new interval. For example, if the desired interval size is 100 million instructions, and the program has just executed 100 million instructions, the profiler must create an interval boundary on the next mappable marker encountered. When the next mappable marker is reached, the current interval is ended by recording the marker ID and the number of times the marker has executed since the start of execution.

This is done throughout the execution of the primary binary: whenever the current interval grows beyond the desired interval size, the current interval is ended when the next mappable marker is reached. Recording the execution count is critical, because markers are typically executed many times as programs execute. Each (marker ID, execution count) pair uniquely identifies a specific point in execution that can be mapped to other binaries.

As the profiler partitions the execution of the primary binary into variable length intervals on mappable markers, the profiler also collects a basic block vector for each interval, which will be run through SimPoint in the next step. Only the primary binary is profiled in this step.

Pick Simulation Points for the Primary Binary

Next, SimPoint is run on the basic block vectors collected in the previous step to pick simulation points. SimPoint 3.0 or later is needed, because earlier versions do not support variable length intervals. SimPoint 3.0 supports variable length intervals by considering the number of instructions in each interval throughout the clustering process and the search for simulation points.

SimPoint generates a set of simulation points, weights for each simulation point, and phase labels for every interval. Each simulation point represents

a unique phase, and the weight associated with the simulation point reflects the fraction of executed instructions in that phase in the primary binary.

Map the Simulation Points to All Binaries

Next, the simulation points selected for the primary binary are mapped to the other binaries, resulting in simulation points that represent semantically equivalent portions of execution in all the binaries.

The start and end of each simulation point are defined by a (marker ID, execution count) pair. These pairs represent the simulation point across all binaries, and can be used during simulation to represent the start and end of that simulation point when executing any of the binaries.

Recalculate Weights for Mapped Simulation Points

Finally, the simulation points must be appropriately weighted relative to the size of the clusters they represent in each binary. The weights need to be readjusted because the amount of execution in each phase can change across binaries.

A simulation point's weight is the fraction of the total dynamic instructions that the program executes in the phase it represents. For example, if a program executes 60% of its dynamic instructions in phase P, the simulation point for phase P will have a weight of 60%. So, to calculate the correct weight for each simulation point in each binary, the number of dynamic instructions executed in each phase is counted with Pin. Weights do not need to be recalculated for the primary binary.

Table II.2: Memory System Configuration

Cache Level	Capacity	Associativity	Line Size	Hit Latency	Type
FLC(L1D)	32KB	2-way	64 bytes	3 cycles	WriteBack
MLC(L2D)	512KB	8-way	64 bytes	14 cycles	WriteBack
LLC(L3D)	1024KB	16-way	64 bytes	35 cycles	WriteBack
DRAM				250 cycles	

Dealing With Optimized Code Regions

Call-loop graphs are collected through binary instrumentation, so compiler optimizations can change call-loop graphs. For example, if a procedure is inlined in the optimized version of a binary, the name of the inlined procedure and the entry point to the inlined procedure may not exist in the optimized binary. The suggested approach is tolerant of these types of optimizations - if a procedure disappears from one binary due to optimizations, then the nodes associated with that procedure are not mappable, and will never be used as interval boundaries.

Additionally, the call-loop graph matching algorithm described in step 2 has been extended to handle some optimization cases. Inlined procedures are detected by their parent nodes and loop structure. Consider a procedure that has a loop that executes N times, which is called from another procedure that has a loop that executes M times. If this procedure has been inlined and its loop structure is maintained, then the caller should now have two loops, executing N and M times respectively. The loop in the inlined procedure can still be mapped because it can be uniquely identified based on iteration counts. Of course, if $N = M$, the matching algorithm can not determine which loop belongs to which procedure based on iteration counts alone.

II.D.2 Methodology

The proposed cross binary simulation point approach is evaluated with CMP$im [48], a Pin [60] based multi-core simulator. CMP$im models an in-order processor and can simulate the performance of applications run to completion. CMP$im is configured to model a single-core processor with a three-level non-inclusive cache hierarchy with parameters as shown in Table II.2. All caches use a 64B line-size and LRU replacement policy.

To evaluate the cross binary simulation points, SPEC2000 was compiled with debug information (`-g` compiler flag) on 32-bit (x86) and 64-bit (x86_64) Linux. The programs were compiled using version 9.0 of Intel's C/C++ and Fortran compilers. For each program, unoptimized and optimized versions were also compiled, for a total of four binaries per SPEC program: 32-bit Optimized, 32-bit Unoptimized, 64-bit Optimized, and 64-bit Unoptimized. Cross binary simulation points are used to compare the performance of these binaries and examine how well the proposed technique estimates the speedup across the different binaries. All programs are evaluated while running reference inputs.

Simulation regions are represented with PinPoints [70] files. PinPoints is a Pin tool chain that generates basic block vectors for each interval and then runs them through SimPoint 3.0 to generate simulation points and weights. Each binary was run through CMP$im configured as described in Table II.2 with a PinPoints file describing the simulation regions for the binary for the given input. Whole-program statistics are estimated using statistics reported by CMP$im and weights reported by SimPoint for each simulation region, and these estimates are compared to the actual whole-program statistics reported by CMP$im.

II.D.3 Cross Binary SimPoint Evaluation

This subsection evaluates SimPoint performance estimates across different binaries compiled from the same program source. This subsection will show that the proposed cross binary SimPoint technique improves on the per-binary SimPoint approach because semantically equivalent simulation points will be used across all binaries.

SimPoint Performance Estimation

SimPoint has many options that can affect phase classification and simulation point selection. To fairly compare per-binary SimPoint and cross binary SimPoint, the same SimPoint options are used for both approaches. The settings for these options are described below.

The maximum number of clusters, max_k, is limited to 10. max_k is an upper bound on the number of clusters SimPoint may use to characterize the phase behavior of a program. One simulation point is generated for each cluster (phase) identified, so the number of simulation points selected will always be less than or equal to max_k. SimPoint rarely uses the maximum allowed number of clusters because it usually finds a good phase characterization with fewer clusters, so SimPoint typically selects less than max_k simulation points.

Figures II.17 and II.18 show the number of simulation points selected and the average interval size for each benchmark. Figure II.17 shows results for per-binary SimPoint (FLI) and cross binary SimPoint (VLI), while Figure II.18 only shows results for cross binary SimPoint, because per-binary SimPoint uses fixed length intervals of 100 million instructions, so the interval size for per-binary SimPoint is constant at 100 million instructions. Four different binaries were compiled for each benchmark as described in Section II.D.2, and the results in these figures are averaged over the four binaries.

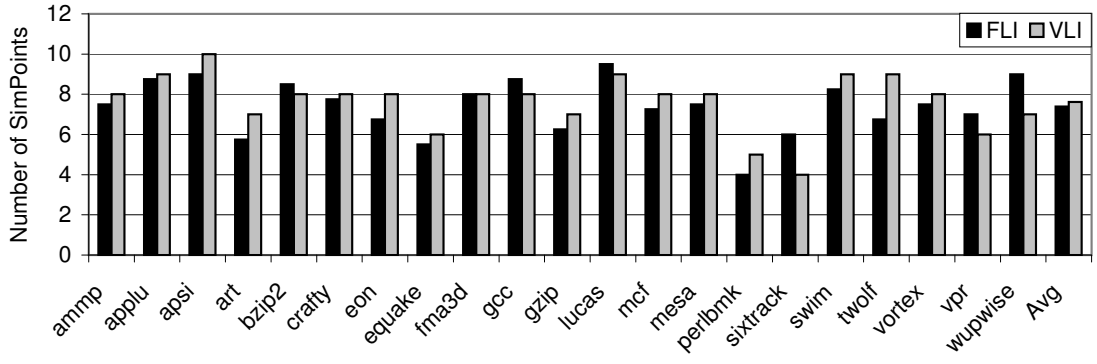


Figure II.17: Number of SimPoints for per-binary SimPoint (FLI) and cross binary SimPoint (VLI). Each bar shows the average across all four binaries.

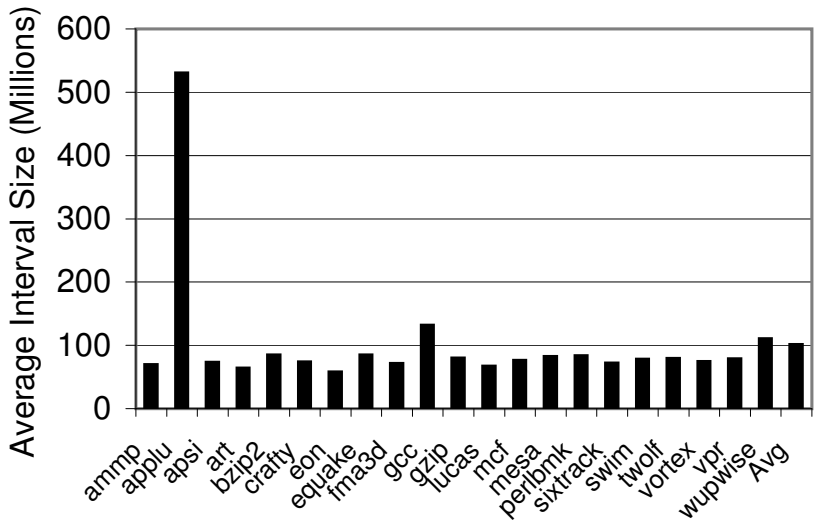


Figure II.18: Interval Size for cross binary SimPoint (VLI). Each bar shows the average across all four binaries. The size of each interval in per-binary SimPoint, which uses fixed length intervals, is constant at 100 million instructions.

Figure II.17 shows that both techniques select a similar number of simulation points on average. This is expected, because the four binaries for each program are compiled from the same source, so similar numbers of high-level behaviors are identified in all four binaries.

The differences in interval size between per-binary SimPoint and cross binary SimPoint shown in figure II.18 occur because per-binary SimPoint and cross binary SimPoint use different algorithms to partition execution into intervals. Per-binary SimPoint partitions program execution into fixed length intervals of 100 million instructions, while cross binary SimPoint produces intervals of *at least* 100 million instructions. Cross binary SimPoint creates intervals for the primary binary only at mappable markers, so intervals in the primary binary may be larger than 100 million instructions.

Cross binary SimPoint maps the primary binary's intervals to the remaining binaries. This means that a single interval may have many different lengths across binaries. Suppose, for example, that an unoptimized binary executes 10 times more instructions than an optimized binary. If the unoptimized binary is used as the primary binary, mappable intervals of 100 million instructions will be constructed, but when the intervals are mapped to the optimized binary, the intervals will shrink to 10 million instructions on average.

This effect is visible in Figure II.18, where most benchmarks have intervals that are smaller than 100 million instructions on average, because interval lengths tend to shrink when mapped from unoptimized to optimized binaries.

`applu` has a much larger interval size because our technique was unable to find mappable markers across all four binaries for some large portions of execution. In these parts of execution, a loop calls five procedures that each solve a partial differential equation. Each of the five procedures has a similar looping structure because they perform similar operations. In the optimized version of

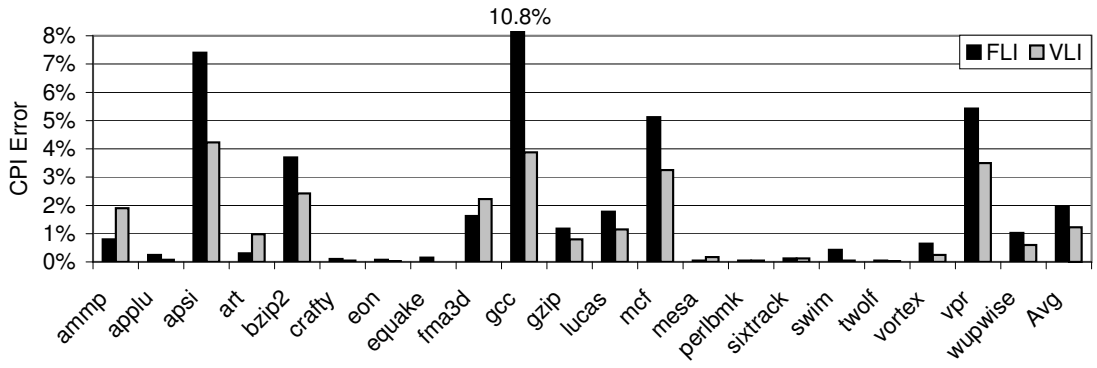


Figure II.19: CPI Error for per-binary SimPoint (FLI) and cross binary SimPoint (VLI). Each bar shows the average across all four binaries.

this binary, all five procedures are inlined into the loop. Furthermore, the loops were split by the optimizer, and code was moved within this loop. While our technique can deal with simple cases of inlining, in this case there was no loop structure remaining after optimization to build a mapping between the unoptimized and optimized code. It may be possible to extend the cross binary SimPoint approach to handle more of these difficult optimization cases with more powerful mapping techniques, such as those presented in Section III.C.

As in the previous figure, each bar averages across all four binary versions of each program. For each binary, a full simulation is conducted to determine the actual overall CPI, and the simulation points are used to generate an estimated overall CPI. The error in each CPI estimate is then calculated relative to the actual CPI, and the CPI errors are averaged across the four binary versions to produce Figure II.19.

Figure II.19 shows that both per-binary SimPoint and cross binary SimPoint accurately estimate the CPI of each program. This is not surprising, because no cross binary comparisons are done in this experiment - per-binary SimPoint is being used to estimate the CPI of each binary, which is the usual way in

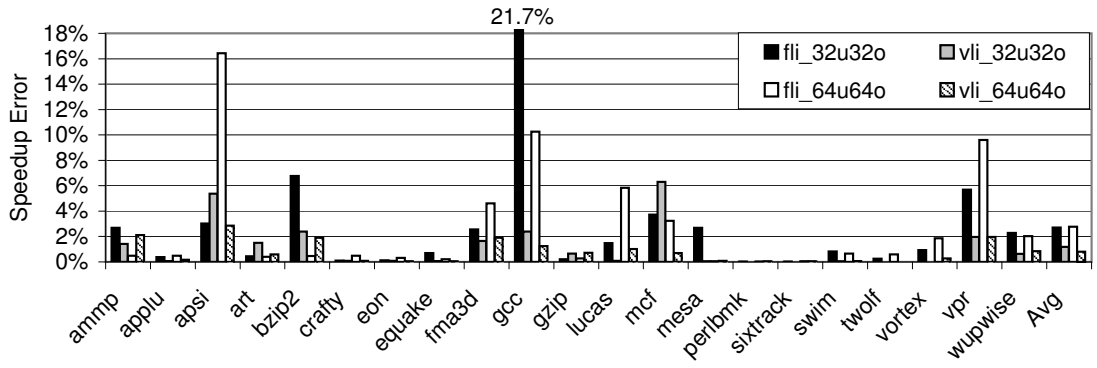


Figure II.20: Speedup error for per-binary SimPoint (fli) and cross binary SimPoint (vli). Speedup is computed across pairs of binaries on the *same* platform (optimized vs. unoptimized). Speedup error is the error in estimated speedup relative to the actual speedup. 32U is 32-bit Unoptimized, 32O is 32-bit Optimized, etc.

which SimPoint is used. Figure II.19 shows that the errors in CPI estimates are low, but it does not show that these errors are consistent across binaries. This is the focus of the following subsection.

Speedup Comparison

When accelerated architectural simulation techniques are used with multiple binaries, the samples of program behavior selected for simulation must be chosen consistently across binaries in order to make a meaningful performance comparison. This subsection presents an experiment where the actual speedup is calculated between binaries, and the actual speedup is compared to the estimated speedup between binaries with the per-binary and cross binary SimPoint techniques.

Figures II.20 and II.21 show the error in speedup estimates across several pairs of binaries. Figure II.20 considers pairs of binaries on the same platform with different optimization levels, while Figure II.21 considers pairs of binaries

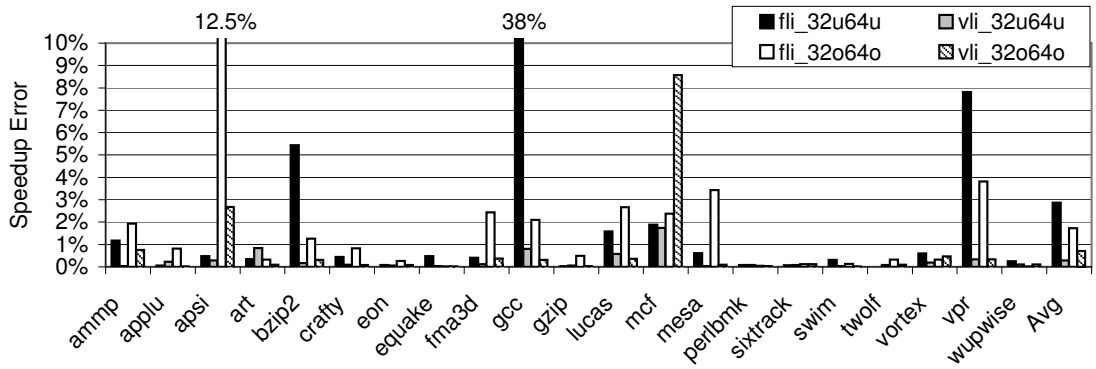


Figure II.21: Speedup error for per-binary SimPoint (fli) and cross binary SimPoint (vli). Speedup is computed across pairs of binaries on the *different* platforms (32-bit vs. 64-bit).

on different platforms at the same optimization level. Each figure shows how well the per-binary and cross binary SimPoint approaches estimate speedup across pairs of binaries.

Speedup error is computed as

$$SpeedupError = \frac{|(TrueSpeedup - EstimatedSpeedup)|}{TrueSpeedup}$$

$TrueSpeedup$ is the ratio of the total number of cycles needed to execute the two binaries. For example, the $TrueSpeedup$ for the `32u32o` configuration is the number of cycles needed to execute the 32-bit unoptimized version divided by the number of cycles needed to execute the 32-bit optimized version. Similarly, $EstimatedSpeedup$ is the ratio of the estimated total number of cycles needed to execute the two binaries. $EstimatedSpeedup$ is computed in the same manner as $TrueSpeedup$, except that estimated cycle counts from accelerated simulation are used instead of actual cycle counts. $SpeedupError$ measures the difference between the estimated speedup and the actual speedup, relative to the actual speedup, for a pair of binaries.

Figure II.20 shows speedup errors for (32-bit unoptimized, 32-bit optimized) and (64-bit unoptimized, 64-bit optimized), and Figure II.21 shows speedup errors for (32-bit unoptimized, 64-bit unoptimized) and (32-bit optimized, 64-bit optimized). For each pair of binaries, speedup error is calculated for both the per-binary and cross binary SimPoint methods.

These figures show that cross binary SimPoint results in less speedup error on average than per-binary SimPoint. This can be explained by the lack of consistency in samples that are chosen as simulation points across the different binaries with the per-binary approach. When small samples of program behavior are used to approximate a program's overall behavior, errors in the approximations are to be expected, because the samples can not accurately represent *all* program behaviors - the only way to guarantee zero error is to examine the entire program, instead of small samples of the program.

This approximation error due to sampling is called *bias*, and it occurs simply because it is impossible for a set of samples to fully represent all behaviors. SimPoint selects samples with a cost-benefit analysis: samples are selected for frequently occurring program behaviors. This implies that infrequent behaviors are not likely to be represented in the samples selected by SimPoint.

Cross binary SimPoint selects semantically equivalent execution regions across different binaries, so errors in performance estimates due to lack of representation will occur consistently across all the binary executions. Thus, errors due to bias will be consistent across all binaries. This allows cross binary SimPoint to obtain performance estimates that are more accurate when compared across binaries.

Table II.3: Phase comparison across 32-bit unoptimized and 64-bit unoptimized `gcc` binary versions.

	32-bit Unoptimized					64-bit Unoptimized				
	Phase	Weight	True CPI	SP CPI	CPI Error	Phase	Weight	True CPI	SP CPI	CPI Error
VLI	1	35%	3.16	3.15	0.2%	1	28%	2.97	2.97	-0.1%
	2	26%	3.99	2.93	27%	2	21%	4.11	2.93	29%
	3	14%	4.47	5.17	-16%	3	17%	5.49	6.34	-16%
FLI	1	36%	3.16	3.16	0%	1	22%	2.98	2.97	0.5%
	2	31%	6.54	2.90	56%	2	18%	6.04	7.04	-17%
	3	9%	5.00	4.04	19%	3	16%	6.66	7.19	-8.0%

Table II.4: Phase comparison across 32-bit optimized and 64-bit optimized `apsi` binary versions.

	32-bit Optimized					64-bit Optimized				
	Phase	Weight	True CPI	SP CPI	CPI Error	Phase	Weight	True CPI	SP CPI	CPI Error
VLI	1	52%	3.04	2.91	4.5%	1	52%	2.59	2.44	5.9%
	2	19%	3.57	3.10	13%	2	18%	3.16	2.66	16%
	3	5%	4.66	4.70	-0.9%	3	5%	3.64	3.63	0.3%
FLI	1	71%	3.50	3.00	14%	1	65%	2.77	2.50	0.9%
	2	5%	4.58	4.61	-0.7%	2	8%	5.34	3.39	37%
	3	5%	4.60	4.63	-0.7%	3	6%	7.61	7.55	0.8%

Bias Analysis

Per-binary SimPoint selects a different set of simulation points for each binary version, and these sets of simulation points are not guaranteed to represent semantically equivalent portions of program behavior in each binary. Each set of simulation points can be used to accurately represent the overall execution of its corresponding binary, but SimPoint may represent specific program behaviors more accurately in one binary than another. Bias, the unavoidable error due to sampling, will not be consistent across all the binary versions with the per-binary SimPoint approach. Per-binary SimPoint may select semantically *similar* simulation points that represent common program behaviors across binaries, but there are no guarantees.

To demonstrate the benefits of the consistent bias achieved by cross binary SimPoint, this subsection considers two benchmarks in detail: `gcc` and `apsi`. Both of these benchmarks have higher speedup error with per-binary SimPoint than cross binary SimPoint. Tables II.3 and II.4 show phase statistics across two binary versions of `gcc` and `apsi`. Table II.3 compares the top three phases found with per-binary SimPoint and cross binary SimPoint across 32-bit unoptimized and 64-bit unoptimized `gcc` binary versions. Similarly, Table II.4 compares the top three phases found with per-binary SimPoint and cross binary SimPoint across 32-bit optimized and 64-bit optimized `apsi` binary versions. Both tables show for each phase the phase ID, the weight of the phase (the percentage of executed instructions in that phase), the true CPI of the phase (the average CPI across all intervals in that phase), the estimated CPI using per-binary and cross binary SimPoint, and the relative error between the true CPI and the estimated CPI.

Table II.3 shows the problem with per-binary SimPoint by showing how independent SimPoint runs on different binaries can produce different phase classifications. The table shows that the weights for the top three phases differ by 11% on average with per-binary SimPoint (FLI), while the difference in weight is 5% on average with cross binary SimPoint (VLI). This shows that larger portions of execution are being grouped into different phases in the two binaries. The effects of different biases are even more apparent in the changes in CPI error for the top three phases: with per-binary SimPoint the average difference in CPI error is 20%, but with cross binary SimPoint the average difference in CPI error is only 0.8%.

With the cross binary SimPoint approach, the weight of each phase is adjusted for each binary as described in section II.D.1, which is why the weights vary across binaries even with the cross binary approach. Small changes in weight-

ing are necessary due to differences in compilation - different binaries typically require different amounts of time to accomplish the same subtasks.

Table II.4 shows similar results for `apsi` as Table II.3 shows for `gcc`: in this table, the weights for the top three phases differ by 3% on average for the per-binary SimPoint approach (FLI), and 0.3% on average for cross binary SimPoint. Examining the CPI errors, the average difference in CPI error is 17.4% for per-binary SimPoint and 1.9% for cross binary SimPoint.

When a single binary is used to explore a design space, the bias does not change, but when different binaries are used to explore a design space, the biases can change, and the simulation methodology must ensure that the biases are consistent across binaries. Table II.3 shows that per-phase biases can change significantly across binaries with the per-binary SimPoint approach. For example, the per-binary SimPoint approach estimates the CPI of the second phase in `gcc` with an error of 56% for the 32-bit binary and -17% for the 64-bit binary, and similarly for the third phase, the 32-bit binary has 19% error and the 64-bit binary has -8% error. These changes in bias are responsible for the 38% speedup error for `gcc` shown in Figure II.21.

On the other hand, the cross binary SimPoint approach proposed in this chapter has a much more consistent bias across phases, because the simulation points are selected consistently across binaries, ensuring that each simulation point represents semantically equivalent portions of execution in each binary.

II.E Related Work

This subsection presents work closely related to cross binary simulation points.

Huang et al. [41] proposed a hardware approach to track procedure calls with a call stack, which is used with a set of thresholds to divide a program's

execution into phases at the procedure level. Georges et al. [33] implemented the approach of Huang et al. to perform offline phase analysis of Java programs. Huang et al.'s algorithms are used off-line to provide a workload case study on phase behavior in Java programs.

Liu and Huang [59] used procedures and loops to divide a program's execution into phases. The partitioning determined where and when statistical samples should be taken during architecture simulation. Their analysis divided a program's execution at static call sites, and if a procedure executes for too long, then the procedure's execution was further divided into its major loops. The sample rate for sampled architectural simulation was determined by examining the variability of several architecture metrics for each program region.

Shen et al. [76] used wavelets [17] and Sequitur [68] to build a hierarchy of phases to represent the program's behavior patterns at many levels. The goals of the work of Shen et al. are similar to the goals of cross binary simulation points, but the approaches are very different. Shen et al. perform their phase analysis on data reuse distance traces, while the approaches presented in this chapter are based on a program's code use. Another key difference is that the phase marker approach presented in this chapter analyzes a program's code structure to directly build a hierarchy of phases, instead of rediscovering the hierarchy by analyzing traces.

This chapter focused on simulation scenarios involving multiple binary representations of the same program. This scenario was not addressed in any of the above research: all of the work described in this section only considers simulation scenarios involving single binaries, while the work in this chapter focuses on handling multiple binaries.

II.F Summary

This chapter presented cross binary simulation points, which represent simulation points in a binary-independent manner. With cross binary simulation points, semantically equivalent code regions are selected across binaries in sampled architectural simulations, which allows for meaningful comparisons of the sampled simulation results across binaries.

This chapter discussed synchronization issues with fixed length intervals, where fixed length intervals may be out of sync with patterns in program behavior. These issues motivated the need for variable length intervals.

This led to software phase markers, which are code locations that mark the beginning of a major change in program behavior when executed. Phase markers are associated with source code, and are thus portable across compilations. Direct use of phase markers works well for applications that are tolerant of long intervals such as dynamic cache reconfiguration, but not so well for accelerated architectural simulation. With a pure software phase marker approach, it is difficult to control maximum interval length, which directly affects the number of instructions that must be simulated.

To address this issue, cross binary simulation points were presented, which use specific dynamic instances of phase markers to bound simulation points to control maximum interval length. Cross binary simulation points identify semantically equivalent behaviors across binaries, which results in consistent sampling bias, which allows for meaningful comparisons of sampled simulation results.

Acknowledgements

Cross binary simulation points were developed in collaboration with Erez Perelman and Brad Calder at UC San Diego, Greg Hamerly at Baylor University, Tim Sherwood at UC Santa

Barbara, and Harish Patil and Aamer Jaleel at Intel. I thank my co-authors for allowing me to present the results of our collaboration in my dissertation.

Section II.B contains material that appears in "Motivation for Variable Length Intervals and Hierarchical Phase Behavior", in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Jeremy Lau, Erez Perelman, Greg Hamerly, Timothy Sherwood, Brad Calder. The dissertation author was the primary investigator and author of this paper. Portions of Section II.B are ©2005 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Section II.C contains material that appears in "Selecting Software Phase Markers with Code Structure Analysis", in *International Symposium on Code Generation and Optimization (CGO)*, Jeremy Lau, Erez Perelman, Brad Calder. The dissertation author was the primary investigator and author of this paper. Portions of Section II.C are Copyright ©2006 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or `permissions@acm.org`.

Section II.D contains material that appears in "Cross Binary Simulation Points", in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Erez Perelman, Jeremy Lau, Harish Patil, Aamer Jaleel, Greg Hamerly, Brad Calder. The dissertation author was the secondary investigator and author of this paper. Portions of Section II.D are ©2007 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

III

Performance Audited Dynamic Optimization

The goal of a compiler optimization is to improve program performance. Ideally, an optimization would improve performance for *all* programs, but some optimizations can also degrade performance for some programs. Thus, it is sometimes acceptable for an optimization to improve performance *on average* over a set of programs, even if a small performance degradation is seen for some of these programs. This often leaves aggressive optimizations, which can produce substantial performance gains and losses, turned off by default in production compilers because it is difficult to know when to choose these optimizations, and the penalty for a wrong decision is high. Therefore, developing a compiler involves tuning a number of heuristics to find values that achieve good performance on average, without significant performance degradations.

Today's virtual machines (VMs) perform sophisticated online feedback-directed optimizations, where profile information is gathered during the execution of the program and immediately used during the same run to bias optimization decisions toward frequently executing sections of the program. For

example, many VMs capture basic block counts during the initial executions of a method and use this information later to guide optimizations such as code layout, register allocation, inlining, method splitting, and alignment of branch targets [6]. Although these techniques are widely used in today's high-performance VMs [69, 86, 1, 47, 34], their speculative nature further increases the possibility that an optimization may degrade performance if the profile data was incorrect or if the profile data does not accurately represent future behavior.

The empirical search community [11, 30, 92, 73, 43, 58, 88, 20, 53, 89] takes a different approach toward optimization. Rather than tuning the compiler to find the best "compromise setting" to be used for all programs, they acknowledge that it is unlikely any such setting exists. Instead, the performance of various optimization settings, such as loop unroll factor, are measured on a particular program, input, and environment, with the goal of finding the best optimization strategy for that program and environment. This approach has been very successful, especially for numerical applications. Computer architectures, memory hierarchies, and runtime environments vary greatly from computer to computer, resulting in tremendous potential for improved performance if a program's compilation is tuned for the machine on which it will actually run.

The majority of empirical searches have been performed *offline*. But with the rich runtime environment provided by a VM, it becomes possible to perform fully automatic empirical search *online* as a program executes. Such a system would compile multiple versions of each method, compare their performance, and keep the fastest version for future use. Examples of such an online system are the Dynamic Feedback [27] and ADAPT [90] systems, and the work by Fursin et al. [32].

The most significant challenge to such an approach is that an online system does not have the ability to run the program (or method, etc.) multiple

times with the exact same program state. Traditional empirical search, and optimization evaluation in general, is performed by holding as many variables constant as possible, including: 1) the program, 2) the program's inputs, and 3) the underlying environment (operating system, architecture, etc.). In an online system, the program state is continuously changing; each invocation of a method may have different parameter values and different global program state. Without the ability to re-execute differently-optimized versions of the code with the exact same parameters and program state, meaningful performance comparisons seem impossible. Previous online systems [27, 90, 32] do not provide a general solution to address the issue of changing inputs or workloads during the comparison. Because of this challenge, today's VMs make no attempt to apply any form of empirical search at runtime.

This chapter presents *performance auditing*, which overcomes the challenge of evaluating the performance of multiple versions of an optimized method online in a production VM. The proposed technique allows for online empirical optimization, greatly improving the ability of runtime systems to *increase performance* and *prevent degradations*.

Performance auditing works by compiling several versions of a method, and, as the program executes, invocations of these different method versions are timed and a statistical analysis is performed to determine the fastest version. However, methods that are infrequently invoked will have a small number of timing samples, resulting in a larger number of inconclusive statistical analyses. These situations occur when a method spends significant time in loops, rather than being frequently invoked, because the baseline performance auditing system only collects method invocation timings. Thus, for infrequently invoked methods, a solution is needed to time loop iterations rather than method invocations.

It is easy to collect loop timings, but it is surprisingly difficult to collect

comparable loop timings from differently compiled versions of the same loop. Loop optimizations are particularly troublesome: for example, if the optimizer unrolls the loop in one compiled version of the method, the loop's timing data must be adjusted accordingly for that compiled version of the method if the timing data is to be meaningfully compared across the differently compiled versions.

This problem is an instance of the general problem of mapping between program source and its optimized binary instructions. Prior work in this more general area falls into two categories: heuristic matching algorithms to correlate the source and binary [49, 91, 97], and modifications to the optimizer to maintain a mapping between the program's source and the optimizer's output [4, 28]. Both of these approaches have significant shortcomings that make them inappropriate for the problem addressed in this chapter. The cross binary simulation point methodology proposed in chapter II under the category of heuristic matching algorithms, and a comparison of the cross binary simulation approach and the approach proposed in this chapter will be discussed further in section III.C.

This chapter presents a new solution to the problem of mapping between a program's source and its optimized binary: *lightweight code markers* are injected into a method before optimization. Lightweight code markers have the semantics of an arithmetic operation on a specially allocated global variable. The optimizer is allowed to freely manipulate these markers, and the markers are removed after optimization, so no code is generated for lightweight code markers. The optimizer's semantics-preserving property automatically maintains the mapping between source and binary. Because lightweight code markers are operations on global variables, the optimizer may not remove the markers, although it may move, copy, or combine markers.

This chapter shows how lightweight code markers are used in a performance auditing system to collect comparable loop timings for infrequently

invoked methods, which results in fewer inconclusive statistical analyses.

The outline of this chapter is as follows. First section III.A presents background and related work in the area of performance audited dynamic optimization. Section III.B motivates the performance auditor approach, presents the baseline performance auditor system, and evaluates the system. Section III.C discusses issues with infrequently invoked methods in performance auditing systems, describes the need for a technique to map between a program's source and its optimized binary, presents a technique that maps between source and binary without using heuristic matching, and without modifying the optimizer, and applies the technique to improve performance auditing for infrequently invoked methods.

III.A Background

This section presents background and related work in optimization systems and mapping between a program's source and binary.

III.A.1 Adaptive Optimization in Virtual Machines

All high-performance VMs use a selective optimization strategy, where methods are initially executed using an interpreter or a non-optimizing compiler. A coarse-grained profiling mechanism, such as method counters or timer-based call-stack sampling, is used to find "hot" methods. Most programs spend most of their time executing code from a small number of "hot" methods. These hot methods are dynamically compiled with a JIT compiler.

VMs use the JIT compiler's multiple optimization levels to tradeoff the cost of high-level optimizations with their benefit; when a method continues to consume cycles, higher levels of optimization are employed. Some VMs, like J9 [62], will compile the hottest methods twice: once to insert instrumentation [7]

to gather detailed profile information about the method, and then a second time to take advantage of the profile information after the method has run for some duration. Thus, in modern VMs a particular method may be compiled many times (with different optimization strategies) during a program's execution. Overhead is kept low by performing such compilations on only a small subset of the executing methods, and often by compiling concurrently using a background compilation thread.

III.A.2 Predicting Performance in Optimization Systems

In a traditional optimizing compiler, every transformation (or optimization) is fundamentally a performance prediction. For example, redundant load elimination makes the seemingly obvious prediction that removing load instructions will reduce execution time. Often, performance predictions are based on an abstract metric, with the assumption that there is a correlation between the metric and bottom-line performance. As another example, compilers often attempt to minimize the number of instructions executed along the critical path, relying on the assumption that fewer instructions results in faster execution.

Some of the compiler's performance predictions are made explicit through the use of a *performance model*. Examples of compiler optimization that have used explicit models are inlining, instruction selection, instruction scheduling, and register allocation. Inlining models the cost of an inlining decision (code growth, resulting in instruction cache pressure) weighed against its benefits (reduced call overhead and increased analysis scope); instruction selection models the cost of choosing various instructions; instruction scheduling models the instruction pipeline, the latency of instructions, and architectural hazards; and register allocation models the cost of spilling a register.

Implicit models are more prevalent than explicit models. The dozens

of transformations and heuristics used throughout a compiler make performance predictions based on an implicit model. The compiler writer may not have explicitly formulated, or even fully understood the model, but it is still inherently hard-coded into the optimization. For example, many compilers contain an inlining heuristic that ensure methods larger than a certain size are never inlined; this simple rule can be considered an implicit performance model that predicts a lack of benefit (or potential performance decline) from inlining large methods.

Most optimizations that use profiling information (often referred to as feedback-directed optimization, or FDO) still employ a model, whether explicit or implicit. Although feedback-directed optimizations observe dynamic information, they generally do not measure program performance directly (wall clock time, etc.), but instead measure a dynamic metric, such as basic block frequencies or method invocation frequencies.[1] These dynamic metrics are then used as input to the optimization's model (usually an implicit model) to predict performance. For example, it is widely assumed that placing frequently-executed basic blocks close to each other will improve instruction cache performance, which will improve bottom-line performance. Optimizations are designed to assume that branch predictors will predict forward and backward branches in a certain way.

Although performance models are often effective, they do not always correlate with bottom-line performance. This results in performance degradations, which are common in a compiler's highest optimization levels. Section III.B.1 provides empirical examples. Heuristics are often tuned extensively to avoid performance degradations, resulting in compromise settings that are not the best in any one configuration.

One way to address the shortcomings of performance models is to refine the models to more accurately predict performance, but we believe this is a

[1]Notable exceptions include the work of Adl-Tabatabai et al. [2], where information from hardware performance monitors is used to trigger dynamic optimizations.

never-ending task. *All* aspects of the execution stack must be correctly modeled, including the operating system, the hardware architecture and implementation, and even the compiler itself, as one optimization may interfere with another downstream optimization. With the current trend of adding more levels of complexity and virtualization to all levels of the execution stack, the problem will only become more difficult as true performance continues to diverge from the predictions of any model. Ten years ago it was not such a daunting task to model the performance impact of instruction scheduling; on today's hardware it is nearly impossible.

III.A.3 Empirical Search

Optimization systems based on empirical search use performance measurements to guide optimization decisions, rather than relying on predictive models. Empirical search has been applied to many optimization scenarios, including optimizing libraries, adjusting compiler phase orderings, and choosing optimization levels.

Adaptive Compilation with Empirical Search

Diniz and Rinard [27] proposed a Dynamic Feedback approach that generates code for several different optimization strategies for important sections of a program. They examine different synchronization strategies for a program's parallel code sections. During execution of these parallel sections, execution alternates between training and production periods for fixed time intervals set in the compiler. The system measures the amount of overhead relative to the performance of the alternative implementations during this training period. It then chooses the implementation with the lowest overhead for the corresponding production period. To choose between implementations, they measure the *overhead*

from one execution of the parallel section of the code. By overhead they measure all of the stalls that occur during execution (e.g., stalls due to locking). This approach is only feasible if (a) all of the overhead can be measured, and (b) the number of stalls seen relative to the overall execution time is independent of the input being run.

Similarly, Voss and Eigenmann [90] perform dynamic optimization on hot spots through empirical search. They use a domain-specific language to specify how to search the optimization space for a specific optimization. As an example, for loop unrolling, a hot spot will be optimized for each level of unrolling. Each of these compiled versions of the hot spot will be run and timed, and the fastest overall time will be kept and used for the hot spot. They time each compiled version only once to decide if it should be used. They deal with varying inputs by partitioning the timings into different bins based on the loop bounds, which relies on loop bound values characterizing varying inputs/workloads.

Both of the above techniques examine the performance of the alternatives only once to choose the better performing one. They argue that performing the timing once for an alternative is sufficient, since the granularity of a single sample can account for a significant amount of execution for the programs they examined. In comparison, for general purpose applications that are run on a JVM it is much harder to create a single large sample that represents the same code being executed. Section III.B.4 will show that, for JVM workloads, it is difficult to correctly choose which alternative has the best performance with a small number of samples for hot methods in a production Java Virtual Machine. The performance auditor approach, based on statistical analysis and randomization, determines how many samples are needed to make a confident evaluation.

Fursin et al. [32] explore online empirical search for scientific programs. Prior to the program's execution, a set of optimization strategies are created to be

explored during execution. Their system uses phase detection to identify periods of stable, repetitive behavior. During a stable phase of program execution, each optimized version is run once and timed, and the best performing version is chosen. This approach exploits the repetitive behavior of scientific applications. General purpose Java applications have a much higher variance in invocation timings, which motivates the performance auditor approach of taking a large number of samples.

Dynamic Optimization Using Heuristics and Feedback Directed Profiling

There is also a large collection of online optimizations that have built-in mechanisms to adapt their strategy based on past program behavior. Such techniques include inlining [22], garbage collection [5], virtual method dispatch [39], object models [8], object layout [50, 74, 42], and prefetching [14, 75, 16]. The techniques used are specific to each optimization and adapt their strategies based on program behavior rather than execution time. More general purpose systems for continuous optimization [51, 15] have also been proposed.

None of these techniques time and compare multiple optimizations at once to find the best. They instead use profiling information to heuristically guide what to try next, for the specific optimization being examined. In comparison, the goal of the performance auditor is to provide an accurate and general method to dynamically measure the actual performance results for different optimizations applied to a hot method.

Offline Empirical Search

The performance impacts of optimizations can be used to guide performance tuning in libraries and kernels. For example, ATLAS [92], PHiPAC [11],

and SPARSITY [44] provide highly-tuned libraries for matrix multiplication and linear algebra kernels, and SPIRAL [73] and FFTW [31] provide digital signal processing solutions. There are even proposals for finding the best-performing sorting routine [58]. STAPL [88] presents a general framework for representing and searching the algorithm space for these types of optimizations.

These techniques work by exploring the algorithm and optimization spaces to tailor the library to the machine it will run on, and even to the application and the inputs the library will be used with. This can result in 50% to an order of magnitude speedup. The best-performing algorithm is found by timing differently compiled versions on test inputs, and this search can take hours to tens of hours. To reduce the time overheads, recent techniques [96, 13] consider using a model-based approach for searching the optimization space.

Another form of empirical optimization search is offline search of optimization phase orderings and optimization levels. Cooper et al. [20, 19] examined reordering phases with adaptive random sampling and genetic algorithms. These techniques compile a version of the program under different phase orderings and optimization levels, and then time the execution of the resulting program with a representative input to determine which is better.

Kulkarni et al. [53] used genetic algorithms combined with memoization to reduce their search space. They were able to achieve 4% speedups on average by searching the compilation space for 3 hours for embedded applications on an ARM processor. Recently they examined probabilistic pruning of the search space to reduce this search time to 1/3 of their previous approach [52]. Triantafyllis et al. [89] built a decision tree to decide which optimizations to apply, and in what order. The tree guides the search, starting with the most important optimizations. They achieved 5% speedups on average compared to -O2 compilation for the Itanium, while doubling the baseline compilation time.

The above studies have shown that offline search of the optimization space, guided by performance data, can lead to speedups. Recent results [89] show that this search can be done quickly with intelligent decision trees to guide the search. The proposed performance auditing technique parallels this work: with performance auditing, the goal is to achieve similar results with an *online* search, by exploring different optimizations as the program executes.

Other Techniques for Improving Optimization Decisions

The previous subsection described approaches to improve code quality by performing offline experiments based on measuring execution time. Another approach is to use machine learning to create a model that correlates method properties with effective optimization decisions as measured by execution time [12, 85, 3]. The result is a predictive model that can be used to determine optimization decisions at runtime for any program. This approach has less overhead than performance auditing because performance predictions are generated by simply consulting the predictive model instead of performing multiple compilations and executing the resulting methods; however, its effectiveness is susceptible to the deficiencies of a model as described in Section III.A.2.

Another approach is to determine the effectiveness of an optimization based on an evaluation of the quality of the generated code. For example, Dean and Chambers [22] monitor the effectiveness of inlining decisions in the Self system based on how many optimization opportunities are created by subsequent optimization phases. This information is used when considering future decisions to inline a method during the same or subsequent executions. Nethercote et al. [67] also evaluate code quality and use it to possibly reconsider optimization decisions for a method by re-executing earlier optimization phases with different settings to produce better generated code. Compared to performance auditing,

this approach also has less runtime overhead; all decisions are made during compilation, no runtime bakeoff occurs. However, because this approach examines the quality of the generated code, it does use an implicit model, and thus, may also be less robust in the presence of varying machine environments.

Performance auditing does not use an explicit or implicit model, but instead measures execution time during an online bakeoff to guide optimization decisions.

These four approaches, offline empirical search, offline machine learning, compile-time evaluation, and performance auditing, are complementary techniques on a continuum. As one moves along the continuum from offline empirical search to performance auditing, the cost of the technique increases (such as number of compilations or runtime experiments) as well as the robustness (likelihood for being correct despite changing inputs and environments). Combining these techniques is both possible and attractive. For example, one could use machine learning or compile-time evaluation to create a small subset of optimization strategies, which could then be explored online by performance auditing.

III.A.4 Building a Mapping Between Binaries

Many situations call for a mapping between binaries. For example, the cross binary simulation points presented in II.D required a mapping between binaries in order to identify semantically equivalent simulation points across binaries.

Compiler optimizations introduce major obstacles to building a mapping between binaries. Different optimizations may be applied to different binaries, making it difficult to identify which portions of the binaries correspond to the same source code.

There are two general classes of solutions to the problem of building a

mapping between binaries: *matching*, which uses heuristics to match the source and binary, and *bookkeeping*, which requires the optimizer to maintain this mapping.

The primary advantage of the matching approach is that it works if there are source-level changes. Both the bookkeeping approach and lightweight code markers are intolerant of source-level changes. The primary disadvantage of the matching approach is that it is very difficult to achieve high accuracy, especially if different optimizations are used, and that it is fragile — matching is driven by heuristics, and the heuristics will likely need to be updated as compilers change.

The primary advantage of the bookkeeping approach is accuracy. The optimizer has full knowledge of all the transformations that it will perform to convert the source to binary, so it can maintain a fully accurate mapping between the two. The primary disadvantage, of course, is that this requires modifying the optimizer to track changes, which requires significant effort.

Matching with Heuristics

Heuristics can be used to correlate source and binary. Wang et al. [91] propose BMAT, a technique to match binaries. BMAT is used to recycle profile data: if quality profile data has been collected from an old binary of a program, but programmers have made relatively small source-level changes to the code and produced a new binary, BMAT can effectively use the old binary's profile data with the new binary.

BMAT's matching algorithm is complex, yet it works well for the application presented. This is most likely because BMAT is designed to handle source-level changes, and not optimization-level changes. Minor source-level changes do not change binaries nearly as much as minor changes in optimization settings. In

addition, BMAT's use case is fairly tolerant of errors — if the algorithm produces an incorrect match, there will be errors in the new profile data.

In performance auditing, on the other hand, an incorrect match is very dangerous, because it can lead the system to make wrong decisions. Performance auditing relies on accurate timing data to evaluate the effectiveness of optimizations. The system will not work if the timing data does not reflect reality.

Before developing lightweight code markers, we experimented with a matching algorithm driven by bytecode index data, which is similar to line number information, and loop structure to match loops in binaries produced by the J9 JIT system. A program was compiled twice with different optimizations, producing two binaries, and the matching algorithm was used to match the loops in the optimized binaries. It was very difficult to achieve over 70% match accuracy, and much higher accuracy is required to collect comparable loop timings for performance auditing.

Kim et al. [49] present a survey of many code matching techniques.

Bookkeeping: Modifying the Optimizer

Albert [4] collects profile data from optimized binaries. For the compiler to use the profile data, a mapping from basic blocks in the optimized binary to source-level basic blocks is required. Basic block split, merge, and delete operations are tracked so they maintain a mapping from blocks in the optimized binary to source-level blocks.

In Albert's work, only basic block operations are modified, which greatly reduces the amount of effort required to implement this solution. However, this approach also decreases its accuracy. Heuristics are needed to prevent information loss, because it is impossible to accurately track block mappings across block operations without higher-level knowledge of what the optimizer is doing.

Engblom et al. [28] present a variant on the modify-the-optimizer approach: they define their optimizer in a domain-specific language (Optimization Description Language, ODL), and generate code for their optimizer from ODL. Because they are generating code for their optimizer, they can easily generate additional code to accurately maintain mappings between source and binary. The downside of this approach is that the optimizer must be written in ODL, and ODL is not general enough to describe all possible optimizations.

We considered modifying the optimizer to maintain an accurate mapping between source and optimized binary for performance auditing, but it would require too much effort to achieve high match accuracy in the J9 optimizer.

Heavyweight Code Markers

A variant of lightweight code markers has been used to implement on-stack replacement (OSR) [40, 29]. In a system that supports OSR, certain points in a method are designated as OSR points, i.e., points where a mapping back to unoptimized code is available. Like lightweight markers, OSR points are inserted into the IR before optimization and the semantics of inserted instructions dictate what optimizations are possible. Unlike lightweight markers, OSR points will likely preclude many optimizations because they are modeled similar to a call instruction that uses all live variables needed to recover unoptimized state. Because they will impact the applicability of optimizations, they are not a viable solution to the problem addressed by lightweight markers.

III.B Performance Auditing

This section introduces the *Performance Auditor*, a framework that enables online performance evaluation. The performance auditor tracks the bottom-line performance of multiple implementations of a region of code to determine

which one is fastest. The analysis is performed online: as the application executes, performance evaluations are done and the results are immediately used to guide optimization decisions. No offline training runs or hardware support are required.

III.B.1 Motivating Empirical Search

This subsection motivates the need for empirical search for dynamic optimizations by presenting two examples: an analysis of the effectiveness of optimization levels in a JIT compiler, and an analysis of the effects of changing inlining heuristics.

For these examples, IBM's J9 JIT compiler was modified to read the hardware cycle counter on entry and exit of selected program methods to measure the total number of cycles spent in a given method. Further details on the methodology will be described in Section III.B.3.

This infrastructure allows the VM to directly evaluate the performance of generated code by measuring the amount of time needed to execute the generated code. In this subsection, the timing infrastructure will be used to evaluate the impact of optimizations on individual methods. To evaluate the quality of the code produced by the compiler, these experiments measure *steady-state* performance, which excludes benchmark start-up. By ignoring benchmark start-up behaviors, steady-state performance measurements exclude the effects of dynamic compilation on timing measurements.

This chapter focuses on *hot* methods, which are a subset of the methods where the program spends most of its time. Further details about the benchmark suite and the identification of hot methods will be presented in Section III.B.3. In our 20 benchmarks, there are 101 hot methods.

Most compilers group optimizations into *levels*, where higher levels pro-

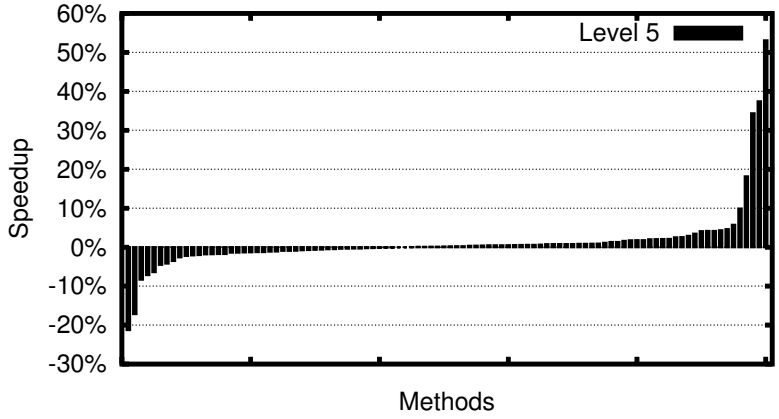


Figure III.1: Per-method performance impact of moving from optimization level 4 to level 5.

vide better performance at the cost of longer compilation times. In a JIT setting, an optimization level must be selected whenever a method is compiled.[2] The J9 JIT has five such levels, which we will refer to as O1–O5.

The timing infrastructure is used to compare the performance of the two highest levels, O4 and O5. On average, across the entire benchmark suite, level O5 improves performance by 1.5% over level O4 with a maximum improvement of 12.5%, and does not significantly degrade performance for any benchmark. However, when measuring performance of O4 versus O5 for individual *methods*, the results vary substantially.

Figure III.1 compares the performance of optimization levels O4 and O5 for each of the 101 hot methods. Each bar represents a method, and the y-axis reports the percentage speedup of O5 over O4 (higher means that O5 is faster). As is typical in compiler optimizations, the highest level (O5) offers substantial speedups for a small number of methods (more than 10% speedups for the 5 rightmost methods), with modest effects on most of the methods.

However, level O5 actually *degrades* performances relative to O4 for

[2]Because of Java's dynamic nature, traditional static interprocedural analysis is not performed because the complete call graph for the program's execution is not known until the program terminates [38].

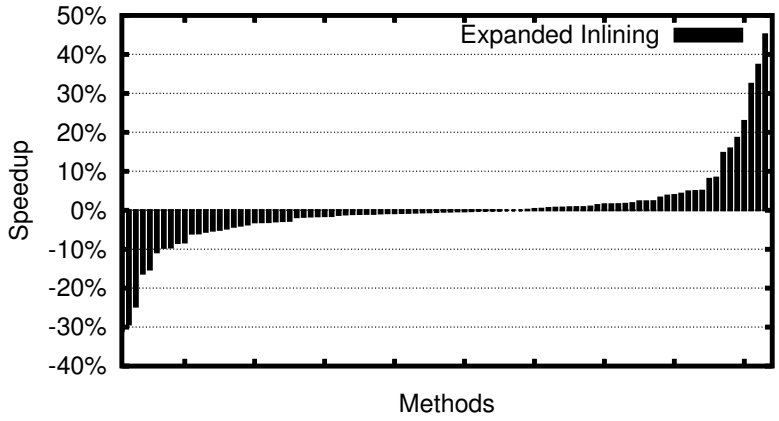


Figure III.2: Per-method performance impact of expanded inlining heuristic.

about a third of the methods. The leftmost method is degraded by 21%. This result is not a byproduct of a poor performing JIT compiler; J9 performs competitively with industry leading JVMs and substantial time and effort has gone into tuning its optimization levels to maximize performance. These results are an inevitable byproduct of tuning compiler heuristics for the *average* case, and evaluating performance at the program level. Similar results are expected from any optimizing compiler.

This subsection presents a second study that examines a specific optimization, method inlining. Conceptually, method inlining has a fairly clear cost-benefit model. The potential benefit of inlining is the performance gained by removing call overhead, and optimizing the callee in the context of the caller. The potential cost of inlining is a decrease in instruction cache locality, and increasing the pressure on downstream optimizations such as register allocation. These performance effects are nearly impossible to predict accurately, so the inlining component in optimizing compilers typically contains dozens of heuristics and ad-hoc tuning knobs. These heuristics are often tuned on large benchmarks suites to find values that work well, on average for these benchmarks.

To demonstrate the difficult task of predicting the performance effects of method inlining, the thresholds used by the inlining heuristic were increased substantially, and the timing infrastructure was used to evaluate the performance effects. Specifically, the threshold that limits the size of callees considered for inlining was quadrupled; thus the modified heuristic allows for substantially larger callees to be inlined. Figure III.2 shows the performance of the new modified inlining heuristic compared to the original inlining heuristic for the hot methods in our benchmarks. As in Figure III.1, each bar represents a method, and the y-axis reports the percentage speedup with expanded inlining thresholds. As in Figure III.1, the impact of expanded inlining varies greatly among methods. On the right side of Figure III.2, there are 7 methods that improve by over 10%, 4 of which are over 20%. On the left side of the figure, there are 5 methods that degrade by over 10%, with 2 larger than 20%. Reducing the inlining thresholds produced similar, inconsistent performance across the set of hot methods.

These large variations are a compiler writer's nightmare, because the higher threshold offers great promise, but also introduces a significant risk of performance degradation. These examples motivate the need for a technique that can automatically tune optimizations to produce performance improvements, while avoiding optimization settings that degrade performance.

III.B.2 Performance Auditor Design

As further described in Section III.B.5, this work uses method boundaries to delimit code regions (as is common in virtual machines) and focuses on comparing only two versions of each method at a time; however, the ideas presented are not limited to these design choices.

The performance auditor conducts *bakeoffs* to identify the faster method variant. A *client* generates method variants and uses the performance auditor to

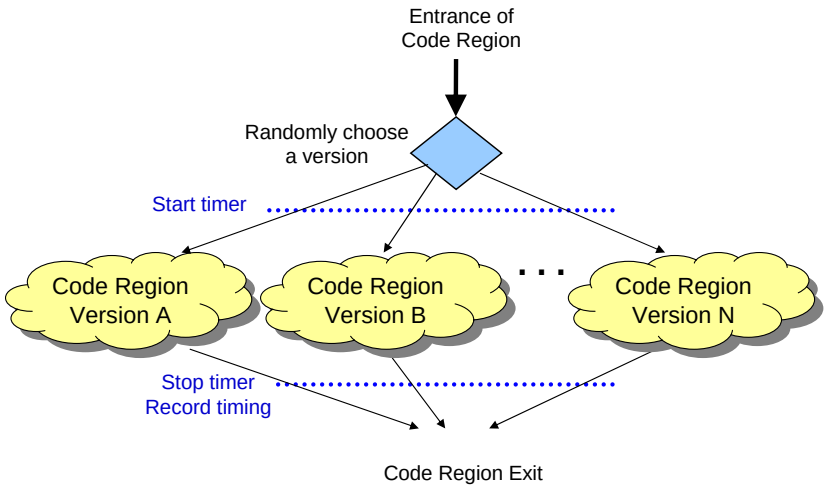


Figure III.3: Performance Auditor Overview

evaluate their performance. The client of performance auditor is responsible for 1) providing the alternative optimized implementations of the code region to be compared and 2) deciding what action to take with the outcome of the bakeoff. For example, a client could evaluate aggressive optimizations that significantly improve performance for some code regions, but degrade performance for others. With the performance auditor, the client could identify situations where performance degrades, and revert to the original implementation. As another example, a client could empirically search the tuning space for important optimization parameters, such as the inlining thresholds described in Section III.B.1, and choose the parameter that results in the best performance.

The performance auditor has two main components: 1) a technique to collect sets of execution times for an implementation of a code region, and 2) a statistical mechanism to determine which set of timings is fastest.

Figure III.3 presents a high-level view of our approach. The client provides N optimized versions of the code region to evaluate. The key idea is to randomly select one of these implementations whenever the code region is en-

tered, and record the amount of time spent. As the program executes, timings for each implementation are collected for later analysis.

The second component of our framework analyzes the collected timings to determine which is fastest. The biggest challenge is that the program is very likely to invoke the audited code region with different program state (parameters and/or memory values) over time. This implies that the different implementations of this code region may take different paths or operate on different data during their executions, which will makes it difficult to meaningfully compare timing data collected from these executions.

For example, a particular execution of version A of the code region may run for less time than version B of the code region, not because A has a better implementation, but because an argument to the method had a value that resulted in A doing less work. For example, if the code region is a sorting algorithm, version A may have been invoked with a list of 10 elements, while version B was invoked with a list of 1,000 elements.

The law of large numbers is used to overcome this challenge. Specifically, as the framework accumulates more timing samples for versions A and B, variations in timings will be evenly distributed across the two versions if the versions are selected randomly to avoid correlations with patterns in program behavior. The performance auditor performs statistical analysis on the timing data and concludes which version is faster, with some degree of statistical confidence.

The performance auditor decides which version (A or B) is faster by averaging the timing data collected from each version of the code. By comparing the averages, the system can determine which version is faster on average, but the comparison is meaningful only if there are enough timing samples. To determine if there are enough samples, a statistical confidence metric calculates the probability that the actual performance difference between versions A and B is consistent

with the observed performance difference in the timing samples from versions A and B.

To calculate the statistical confidence metric, the mean and variance are first computed for each set of timing data. Then the absolute difference between the means $MeanDiff$ is computed:

$$MeanDiff = |mean(A) - mean(B)|$$

And the variance of the difference between the means $MeanDiffVariance$ is computed:

$$MeanDiffVariance = \frac{variance(A)}{sizeof(A)} + \frac{variance(B)}{sizeof(B)}$$

This computed mean and variance define a normal probability distribution for the absolute difference between the means of A and B. A confidence value is computed by integrating this distribution from 0 to ∞. The distribution is for the absolute difference between the means, which is always positive, so this confidence value is the probability that the difference between the means for the entire dataset is consistent with the difference between the means for our samples.

The number of samples needed to make a confident conclusion is directly correlated with the variance in the amount of time spent in the code region. If a constant amount of time is spent in the code region across all invocations, a small number of samples will be needed to conclude that the performance difference is statistically significant. And of course, if very different amounts of time are spent in the code region across invocations — for example, if the code region is sorting lists of very different lengths as the program executes — then a larger number of samples will be needed to detect a statistically significant performance difference. As the variance in the code region's runtime increases, so does the number of samples needed before a confident conclusion can be drawn.

For the proposed approach to succeed, there must not be any correlations between the timing data and program behavior. Any such correlations could bias the timings for one version of the code region. If there are no correlations, this technique can identify arbitrarily small performance differences with arbitrarily high confidence. Of course, if very high confidence is desired, a very large amount of timing data may be necessary.

Sections III.B.4 and III.B.5 explore this concept in more detail. Section III.B.4 investigates the feasibility of the proposed statistical approach through an offline study, and Section III.B.5 describes an online system based on this approach.

III.B.3 Methodology

The experiments in this section were performed using a development version of IBM's J9 VM and its high-performance optimizing JIT compiler [34]. The VM was run on a Pentium 4 3.0 GHz machine with 2 processors and 1 GB RAM running Linux. A suite of 20 benchmarks composed of the complete SPECjvm98 benchmark suite [84], the SPECjbb2000 benchmark [83], and several other Java applications, including seven of the DaCapo benchmark suite [87][3] were used to evaluate the proposed system. Table III.1 reports the number of methods executed,[4] as well as the total size (in KB) of all bytecodes executed for each benchmark. These numbers report dynamic metrics, i.e., they are based on what is executed, not what could be executed.

The last column shows the number of hot methods for each benchmark. Hot methods are a small subset of a program's methods where most of execution occurs, and hot methods can be selected in many different ways. For this work,

[3]The development version of the VM that we used did not run the remaining three benchmarks from the DaCapo suite.

[4]Number of methods executed includes all Java methods (including library methods) executed by the JVM while loading and executing the application.

Table III.1: Benchmark suite

Program	Methods Executed	Bytecodes Exe (KB)	Hot Methods
antlr [87]	1702	228	2
bloat [87]	1063	105	7
compress [84]	770	66	2
daikon [21]	2108	171	6
db [84]	782	67	3
hsqldb [87]	1416	147	8
ipsixql [18]	828	61	6
jack [84]	746	56	3
javac [84]	1467	133	3
jbb2000 [83]	1197	115	8
jess [84]	1140	86	4
jython [87]	1777	186	15
mpegaudio [84]	866	78	4
mtrt [84]	853	76	4
phase [65]	450	31	2
pmd [87]	2030	128	4
ps [87]	946	75	3
soot [80]	2061	235	4
xalan [87]	2108	171	5
xerces [94]	521	36	8

hot methods are defined to be methods that consume enough cycles to be selected by J9 for aggressive feedback-directed optimizations. For these methods, J9 first performs an additional compilation to instrument the method for profiling, then optimizes the method at one of the two highest optimization levels (O4 or O5).

The Read Time-Stamp Counter (`RDTSC`) instruction was used to collect timing samples for a compilation of a method. The `RDTSC` instruction reads the processor's 64-bit cycle counter and stores its value in a register. Method entry is instrumented to read the cycle counter and store the cycle count in the method's stack frame. Method exit is instrumented to read the cycle counter again, subtract the current cycle count from the cycle count on the stack, and

store the difference in a circular buffer.

This methodology measures the total time that the method was active on the stack to ensure fair comparisons in the presence of changing inlining decisions. Measuring only time spent in the instrumented method, excluding callees, would produce incorrect results in the presence of method inlining.

III.B.4 Offline Convergence Study

This subsection presents an offline feasibility study that demonstrates the potential of a performance auditing system. Section III.B.4 describes how timing samples are collected and used to evaluate our approach. Section III.B.4 considers using a fixed number of samples, and shows that thousands of samples are needed to accurately detect speedups during a bakeoff. Section III.B.4 demonstrates that the proposed confidence analysis can correctly detect performance differences between two different compilations of the same method.

Experimental Setup

How many timing samples are needed to accurately detect a performance difference between two compilations of a method? To answer this question, timing data is collected for invocations of each hot method. These timing data are used to represent two hypothetical implementations of the hot method. For a hot method, half of the samples are randomly assigned to set A, and the other half to set B. Each sample in set B is artificially sped-up by $X\%$, where $X > 0$. A and B then represent two potential implementations of the method: A is the baseline, and B is an optimized version that runs exactly $X\%$ faster. The proposed statistical technique is applied to these two sets of timing data to determine how many samples are needed to recognize that B is faster than A.

Method timing samples are collected for the hot methods in each bench-

mark. For these experiments, if a method accumulates 2 minutes of CPU time, no additional samples will be collected for the method. After collecting a set of method invocation timings, the timing values are sorted, and the highest 10% are eliminated. This step filters out noise introduced by events such as garbage collection and context switches. Next, the remaining timing samples are randomly partitioned into two disjoint sets A and B, each containing half of the timings collected, and the timings in set B are artificially decreased by $X\%$.

Fixed Number of Samples

Prior online systems that performed a bakeoff between different compiler optimizations took either one [27, 90] or two [32] timing samples for each optimized version of the code. These systems determined which version was faster based on these one or two samples. In the more general setting explored in this chapter, thousands of samples are needed to accurately determine which version is faster.

This is demonstrated through an experiment that uses a fixed number of timing samples to decide which version is faster.

The sets A and B are created as described above, and a fixed-size subset A' is randomly selected from A, and an equally sized random subset B' is selected from B. The means of the times in subsets A' and B' are calculated, and used to predict if A is faster than B. Recall that in this experiment, B has been artificially sped up by $X\%$, and $X > 0$, so B is *always* faster than A. If the mean for B' is less than the mean for A', the samples correctly predict that B is faster than A, so a correct prediction is reported. Otherwise, a misprediction is reported. To compensate for noise introduced by the use of randomness in this experiment, each experiment is repeated 100 times with different random seeds, and the results are averaged over these 100 trials.

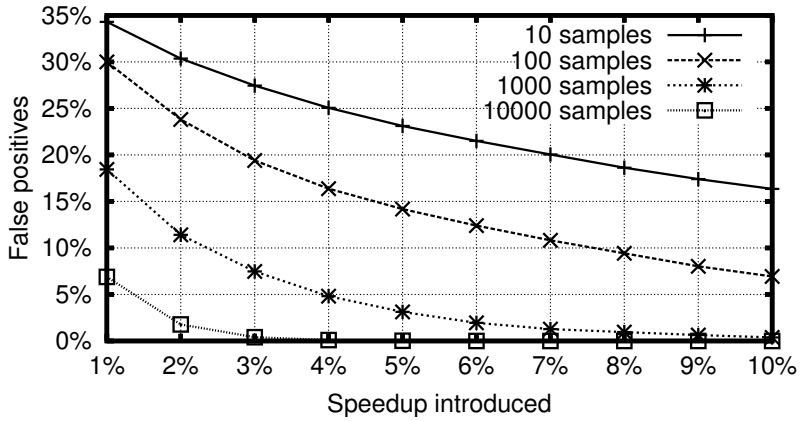


Figure III.4: Misprediction rate for a simple sampling approach in which a fixed number of method invocation (entry+exit) timings are collected, for various amounts of introduced speedup.

Figure III.4 shows the number of mispredictions that occur when 10, 100, 1000, and 10000 samples are used for A' and B'. The x-axis shows the speedup introduced in set B, and the y-axis shows the percentage of experiments in which the fixed-size samples fail to predict that B is faster than A. Method invocation timings are used for this graph as described in the previous section.

Figure III.4 shows that a large number of samples must be collected to detect small speedups, and fewer samples are needed to detect larger speedups. For example, to reliably detect a 2% speedup with less than 2% false positives, 10,000 samples are needed, but 1,000 samples are sufficient to detect a 10% speedup with less than 2% false positives. The results show that both large and small speedups can be detected with 10,000 samples, but 10,000 samples are really only necessary to detect small speedups — 1,000 samples can detect a 10% speedup in an order of magnitude less time, with only 0.4% additional mispredictions. This motivates the proposed approach, which can efficiently detect both large and small speedups by collecting only as many samples as needed.

Statistical Approach

This subsection uses the confidence-based technique described in Section III.B.2 to predict if B is faster than A. For this experiment, sets A and B are used again, as described above.

To determine the number of samples necessary to confidently predict a performance difference between A and B, a confidence threshold $Z\%$ (80% and 99.99% in this study) must first be set. In this experiment, samples are added to A' from A, and to B' from B until $Z\%$ confidence is achieved in the performance prediction, based on the formulas in Section III.B.2.

Initially, sets A' and B' each contain 100 samples. Confidence evaluation is performed. If the confidence is not above the confidence threshold $Z\%$, 100 more samples are added to both A' and B' from A and B respectively, and the confidence evaluation is performed again. More samples are added (100 at a time) to sets A' and B' until the confidence is above the confidence threshold $Z\%$. In this experiment, it is possible to exhaust the supply of timing samples before the confidence threshold is reached. When this happens, the experiment did not *converge* on a confident performance prediction. A performance prediction can not be made, and this event is noted. The convergence experiment is repeated 100 times with different random seeds, and the results over the 100 trials are averaged.

When the statistical analysis converges on a performance prediction, the time to converge is reported, which is the sum of the cycle counts in subsets A' and B'. For these confident predictions, if the analysis incorrectly predicts that A is faster than B, a misprediction is reported. If the supply of samples is exhausted before reaching the confidence threshold, a failure to converge is reported.

Figure III.5 shows the percentage of experiments that converged for

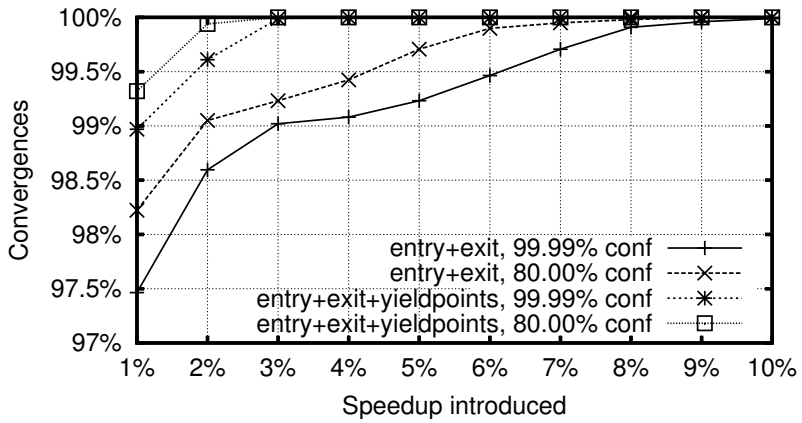


Figure III.5: Convergence rate for the proposed statistical approach. This plot shows the percentage of hot methods for which confident predictions are generated, with a sampling limit of 2 minutes of CPU time for each method.

various amounts of artificial speedup introduced in set B. The y-axis shows the percentage of hot methods for which performance predictions are generated for the two confidence levels examined. The `entry+exit` results use the per method timing, and the `yieldpoints` results will be discussed shortly. These results show that it is more difficult to detect small speedups in 2 minutes of CPU time for our hot methods, if high confidence is desired, as expected.

Figure III.6 shows the percentage of experiments in which the analysis incorrectly predicts that A is faster than B. The results in this figure and figure III.5 show the tradeoff between the number of accurate predictions and the number of confident predictions. As expected, incorrect predictions occur more frequently when the performance difference between A and B is small.

Figure III.7 shows the number of cycles needed to detect a performance difference. The y-axis shows the total amount of time needed to make a confident performance prediction. For experiments that did not converge, the total time spent before giving up is shown. In other words, this graph includes experiments that did not converge, and for those experiments, it shows a lower bound on the

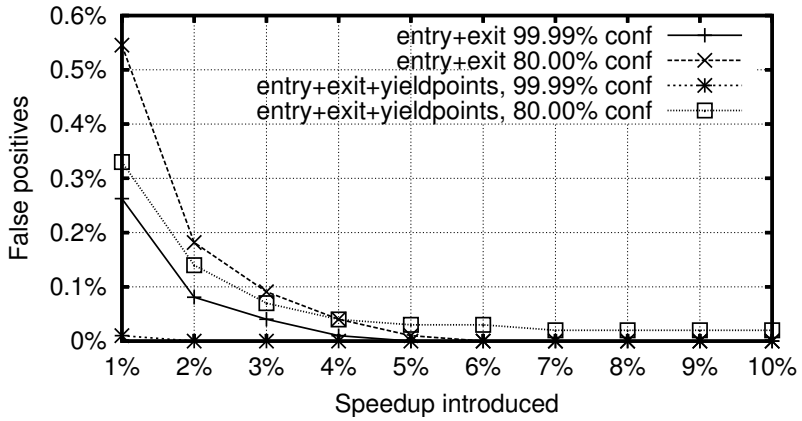


Figure III.6: Incorrect predictions. This plot shows the percentage of hot methods in which the analysis incorrectly predicts that A is faster than B. Results are averaged over converged methods.

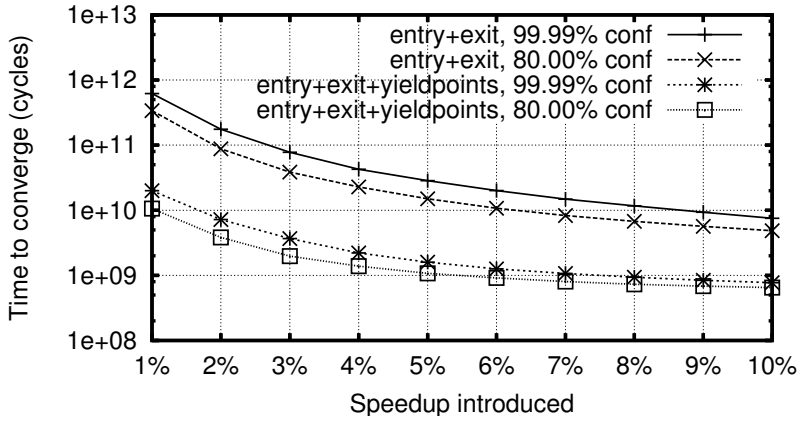


Figure III.7: Time to converge. This plot shows the number of cycles needed to make a prediction. This graph includes experiments that did not converge, and for those experiments, it shows the amount of time spent in the experiment before giving up, which is a lower bound on the convergence time.

convergence time. The results show that it takes significantly longer to detect small speedups than large speedups. The results also show that by adjusting the confidence threshold, convergence time can be further reduced by sacrificing accuracy.

Using Yield Points for Timing

The proposed system collects method invocation timings, but it is also possible to collect timing samples at every loop branch. Collecting one timing sample from each method invocation works well for methods that are invoked frequently, but infrequently invoked methods may pose a problem because they generate timing samples more slowly, which means it will take longer to detect a performance difference.

To address infrequently invoked methods, many timings can be collected from a single method invocation by collecting timing samples between temporally adjacent pairs of yield points. A yield point is a compiler-inserted statement that checks if the application needs to be temporarily suspended to perform a virtual machine service, such as garbage collection. The compiler inserts yield points on loop back-edges, so this approach approximates collecting timing samples for each loop iteration. If a hot method is infrequently invoked, that method is spending a lot of time in loops, and timing samples can be collected for those loop iterations by collecting timing samples between yield points.

Figures III.5, III.6, and III.7 show results with yield-point timings. These results suggest that collecting many timings per method invocation by instrumenting yield points reduces false positives as well as the time to converge, making it an attractive option. However, this offline study was conducted with simulated speedups applied to method timings. Collecting yieldpoint timings when comparing differently optimized versions of code is more challenging, as it

requires keeping the yieldpoint placement consistent in both optimized versions. For example, if inlining occurs in the method being timed, it may inline additional yieldpoints into the method, resulting in more frequent sampling. If any instructions can be identified uniformly in both optimized versions of the code, they would be candidates for timing instrumentation. A second problem is that instrumenting frequently executed instructions, such as loop branches, can introduce more overhead in an online system. Careful engineering can be used to reduce this effect, such as timing groups of N yieldpoints, instead of timing each temporally adjacent pair of yieldpoints. These issues will be discussed in detail in section III.C.

For the remainder of this section, method invocation timings are used.

III.B.5 Online Performance Auditing

This subsection describes the implementation of an online system that utilizes the proposed statistical technique to determine which compilation of a method performs best. The implementation consists of two parts: 1) a dispatch mechanism to select which version of code to run, and 2) a background thread to process and analyze the timing data collected. This is followed by an empirical evaluation of our online technique with a sample client.

Method Dispatch Mechanism

As described in Section III.B.2, methods are the code regions used for each performance bakeoff. Thus, method dispatch for the method being timed (M) must be intercepted so that execution jumps to one of the implementations of the method M at random. There are a number of ways this could be implemented in a VM. We implemented the dispatch as shown in Figure III.8, where a full method body is compiled to act as the dispatcher. The dispatcher method is not

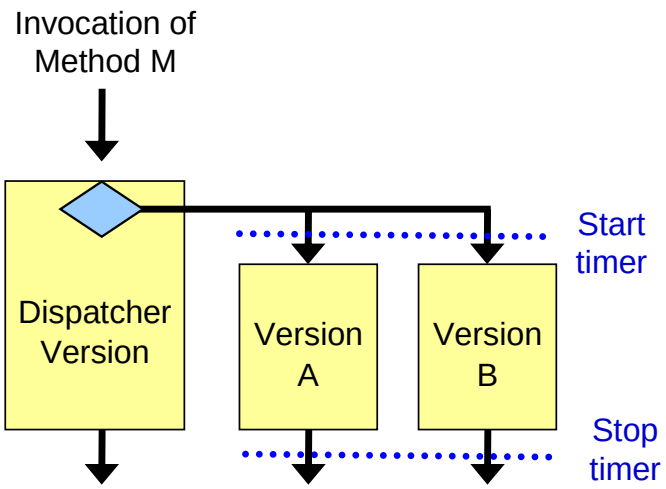


Figure III.8: Architecture for the *dispatcher*, to select method invocations for timing.

timed, but contains a conditional test to decide whether a timed method should be executed, and if so, to invoke the correct one.

Compiling a full method body for the dispatcher is not necessary, but doing so makes it easy to reduce the overhead of the timing instrumentation, because the dispatcher can choose to invoke a "clean" version of the method without any timing instrumentation to arbitrarily reduce overhead, at the cost of convergence time. In situations where compiling a third version of the method is not feasible, alternative implementations can exclude the method body from the dispatcher, so it always invokes one of the timed methods. The dispatch logic could also be placed at the beginning of one of the timed methods, or inlined at the method's call sites.

Figure III.9 shows the dispatch method logic. The prologue checks a sampling condition to determine if any timed version should be executed. We use a count-down sampling mechanism, similar to the approach of Arnold and Ryder [7]. This makes the fast-path of the dispatcher method reasonably efficient, containing only a decrement and check. Once the sample counter reaches zero,

```
METHOD ENTRY:
  sampleCountdown --;
  if (sampleCountdown < 0) goto BOTTOM;

  ... body of method M ...

BOTTOM:
  if (sampleCountdown < (BURST_LENGTH * -1)) {
    // Burst completed. Toggle and reset count
    toggle = !toggle;
    sampleCountdown = SAMPLE_COUNT_RESET;
  }
  if (toggle)
     return invokeVersionA();
  else
     return invokeVersionB();
```

Figure III.9: Dispatch logic

it jumps to the bottom of the code to determine which timed version should be executed. This code is on the slow path and can afford to execute slightly more complicated logic.

To minimize timing errors caused by caching effects due to jumping to a cold method, the dispatcher invokes the timed methods in consecutive bursts. The burst threshold (`BURST_LENGTH`) specifies a number of invocations that are to be performed in a row. When a burst is complete, tracked by allowing the sample counter to go negative, the sample counter is reset. A toggle flag is maintained to switch between optimization versions A and B. If more than two versions are being compared, this toggle can easily be replaced by a switch statement. All of the counters are thread-specific to avoid race conditions and to ensure scalable access for multi-threaded applications.

If a pure counter-based sampling mechanism is used, it could correlate

with patterns in program behavior, and even a slight correlation can skew the timing results. One solution would be to randomly set the toggle flag on each invocation, but this is difficult to implement with low overhead. Instead, the number of samples profiled (`SAMPLE_COUNT_RESET`) and the burst length (`BURST_LENGTH`) are varied by periodically adding a random epsilon. In the prototype performance auditing system implemented in J9, the VM sampling thread is used to update the number of samples profiled and the burst length every 10ms, the shortest timing interval available in a default Linux kernel. This is sufficient to avoid deterministic correlations.

Data processing

As method timings are collected, they are written into a circular buffer, one buffer per thread. The timing data is processed by a background processing thread that wakes up periodically and scans through each thread's buffer every 10ms. The processing thread starts from the position in the buffer where it left off last time, and scans the buffer until it catches up with the producer. No synchronization is used, so race conditions are possible, but dropping a sample occasionally is not a concern. The system is designed so that races cannot corrupt the data structures; slots in the buffer are zeroed after being read to avoid reading a buffer twice if the consumer advances past the producer.

Two operations must be performed on the timing data: 1) discard outliers, and 2) maintain a running average and standard deviation. Outliers cannot be identified by collecting and sorting the entire dataset because this would require too much space and time for an online system. Instead, the system discards outliers by examining windows of N samples. The outliers are identified as the slowest M times in the window. Maintaining the running average and standard deviation is accomplished by keeping a running total of the times, as well as the

sum of the squares.

The system requires a minimum number of samples (10,000) before confidence analysis is performed, to ensure that a small initial sampling bias does not lead to incorrect conclusions. Once a sufficient number of data points are collected, the confidence function described in Section III.B.2 is invoked periodically to determine if the difference between the means is statistically meaningful. If a confident conclusion is reached, the bakeoff ends and the winner is declared; otherwise, the bakeoff continues. To ensure that a bakeoff does not run indefinitely, the system may end a bakeoff after a fixed amount of wall clock time, or execution time, has expired.

Overhead

There are three primary sources of overhead: 1) executing the sampling condition on the fast path of the dispatcher, 2) reading and storing the processor's cycle counter at method entry and exit, and 3) processing the timing samples with the background thread. This overhead is for the infrastructure itself, and is independent of the optimizations being compared.

To quantify the overhead, the online system was configured so that it constantly performs a bakeoff. The online system will perform one bakeoff at a time, so the overhead is evaluated independently for each of the hot methods in the data set (as described in Section III.B.3).

Figure III.10 presents the overhead incurred when a large sample interval is used so that effectively no samples are taken. This data primarily represents the overhead of the fast-path sampling check in the dispatcher. This overhead is quite low, averaging 0.4%, with all but three methods less than 2%. This result is important because it shows that the overall overhead of the auditor can be reduced to this value by lowering the sample rate. Negative overhead is most

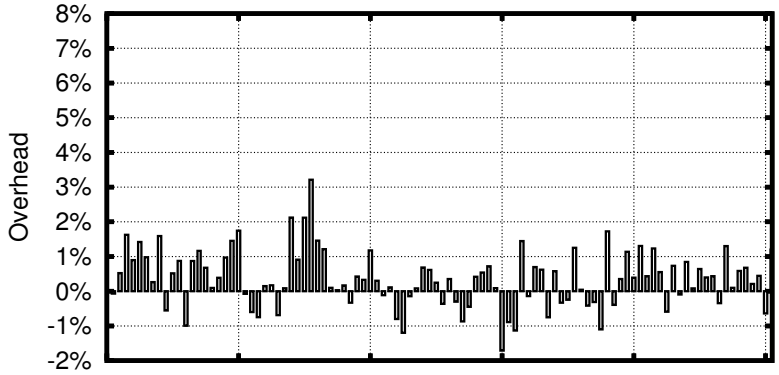


Figure III.10: Per-method overhead of the dispatcher fast-path. No timing samples are being taken.

likely noise, which is expected when measuring overheads in the range of 1%.

Figure III.11 presents the overhead when the dispatcher samples 1 out of 20 invocations of the instrumented method. This sample rate is fairly aggressive to allow quick convergence. It is the sample rate used in our full online system. The overhead increases when samples are taken, with an average of 1.5%, with some methods up to 5%. This amount of overhead is likely to be acceptable as it is incurred only during the bakeoff period. If lower overhead is desired, the sample rate can be reduced and the bakeoff can be performed over a longer period of time.

Online Performance Auditing with an Inlining Client

This section describes a fully automatic online performance auditing system that uses the proposed timing and analysis framework to improve performance. As described in Section III.A.1, the J9 VM already performs an additional compilation of the hottest methods to insert profiling instrumentation. This recompilation logic was modified to perform a bakeoff for these hot methods after they have been instrumented. When the bakeoff has completed, if a winner is

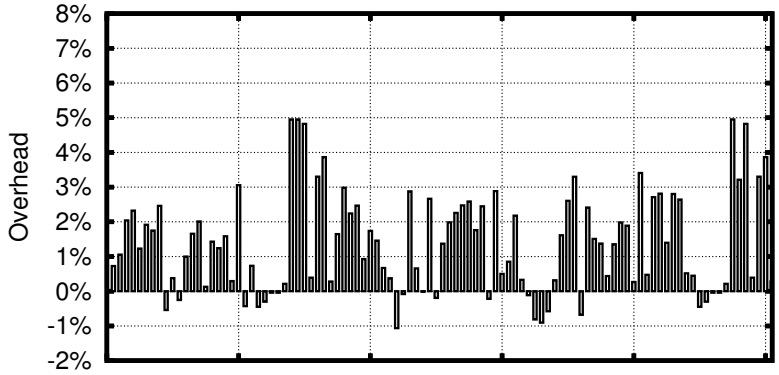


Figure III.11: Per-method overhead of the Performance Auditor when sampling 1 of 20 executions. Overhead includes recording and processing the timing samples.

confidently identified, the winner's optimization parameters are used for the final compilation of the method; otherwise, the VM's default behavior is used. The inlining example presented in Section III.B.1 is used as the optimization client, which compares 1) the original inlining heuristics, and 2) an inliner with quadrupled size thresholds.

The current system performs only one bakeoff at a time, in the order that methods are selected for instrumentation by J9. If a method is selected for a bakeoff while another bakeoff is already in process, it is added to a queue of pending bakeoffs. Multiple bakeoffs could potentially be performed simultaneously, but we did not experiment with this strategy.

Figure III.12 presents the steady-state performance achieved by our system. The x-axis presents the benchmarks, and the y-axis represents performance improvement relative to the default J9 VM. The black bar shows the performance when using the expanded inlining heuristic for *all* hot methods; the white bar shows the performance of the performance auditing system that runs bakeoffs to choose the default or expanded inlining heuristics.

Using the expanded heuristics without auditing resulted in performance

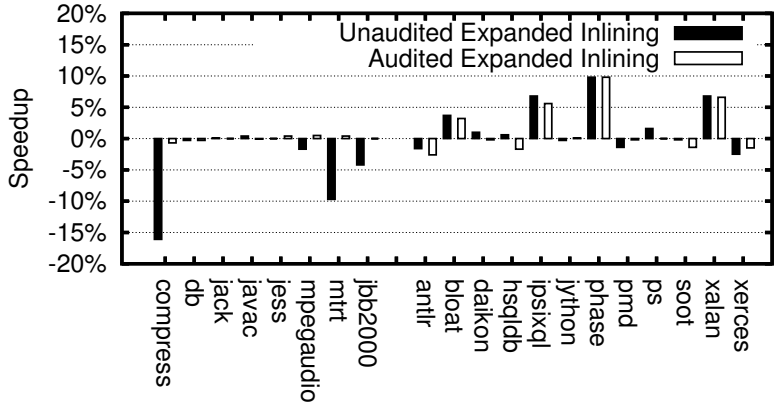


Figure III.12: Performance of the online system using the proposed statistical technique to guide inlining heuristic selection.

improvements between 5–10% for three of the benchmarks in our suite (ipsixql, phase, and xalan), but large degradations for two benchmarks (compress and mtrt). The online performance auditing system is able to achieve most of the performance gains of the expanded inlining heuristics, while avoiding the significant degradations.

The primary contribution of this work is not the speedup produced by this particular optimization client, but the success of the performance auditing infrastructure in identifying the better-performing version. The most important aspect of this performance result is that the system did *not* degrade performance measurably for any of the benchmarks in our suite, which demonstrates the viability of this technique to exploit high-risk/high-reward optimizations.

The left group of 8 benchmarks in Figure III.12 are from the Standard Performance Evaluation Corporation (SPEC). These benchmarks are used widely within the industry for performance benchmarking, and most commercial JVMs have been heavily tuned for these benchmarks. It is therefore not surprising that our simple inlining heuristic adjustment did not improve their performance. However, when executing new benchmarks, such as the 12 benchmarks on the right,

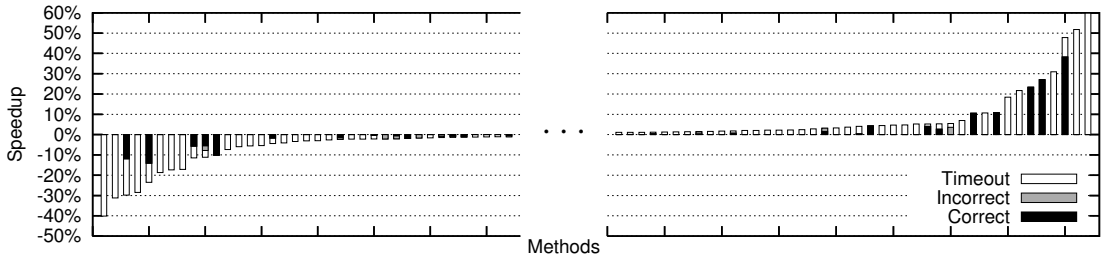


Figure III.13: Accuracy of the online system using the proposed statistical technique to guide inlining heuristic selection.

some substantial performance opportunities were discovered. We do not believe this to be an anomaly, but another example of the fundamental issue addressed by this work: predicting performance for unseen programs is difficult, and there is tremendous opportunity available if systems can automatically identify and correct performance anomalies.

Our auditing system has little impact on initial program startup behavior, because the performance auditing system only operates on methods that reach the highest levels of optimization. When a bakeoff is performed, the method is compiled 3 times, then once again after the bakeoff completes. Our current prototype makes no effort to distribute these compilations over time to avoid overhead, and thus, may introduce overhead relative to the original system before reaching steady state. This source of overhead can be avoided fairly easily by using known techniques, such as distributing the compilations over time using a low priority background compilation thread, or moving compilation to free processors. These techniques are not investigated in this work, because the primary goal of this work is to evaluate the potential of performance auditing by evaluating the accuracy of the bakeoff mechanism, and its potential impact on steady state performance.

To evaluate the accuracy of our technique, the bakeoff decisions from the online auditing system were compared to the performance results from offline

measurements. When an online bakeoff is performed, there are three possible outcomes:

1. *Timeout:* The bakeoff was terminated because it executed for too long without reaching a conclusion.
2. *Incorrect:* The system made a performance prediction that was inconsistent with offline measurements.
3. *Correct:* The system made a performance prediction that was consistent with offline measurements.

There are many sources of potential nondeterminism in the proposed performance auditing mechanism, as well as in the underlying VM itself, so the online auditing system was run 10 times for each benchmark, and the results for all bakeoffs were recorded.

Figure III.13 presents the accuracy of the decisions made by the online system. The graph is in the same format as Figure III.2, where each bar represents a hot method from the benchmark suite, and the total height of the bar represents the performance improvement (or degradation) caused by the expanded inlining heuristics for that method, as reported by offline measurements.

Each bar is broken down by shades of gray to show the results of the 10 online bakeoffs for that method. A solid black bar indicates that the correct conclusion was reached every time a bakeoff was performed. A bar that is 50% white means that 50% of the time the bakeoff ran too long and timed out. Methods where expanded inlining impacted performance by less than 1% were excluded from the graph to improve legibility. Bakeoffs were conducted for 152 methods; 73 methods impacted performance by less than 1%, and the remaining 79 are presented in Figure III.13.

The performance of the excluded methods is small enough that a) the

accuracy of the decision is irrelevant to overall performance, and b) the accuracy breakdown was not visibly discernible from the figure.

The most important aspect of the proposed system is its accuracy on the methods for which the expanded heuristics have a large positive or negative impact. Of the methods that show greater than 10% speedup from the expanded heuristics (rightmost bars), the online system makes the correct decision for around half of these methods (5/11) almost every time; the remaining 6 methods result in a timeout on every bakeoff. The system makes no wrong decisions for these methods.

Similarly, for the methods where expanded heuristics result in degradation of 10% or more (the leftmost bars), the system often makes correct decisions for 5 of the 11 methods, and consistently times out on the remaining methods. For one method, wrong decisions were made in 2 of the 10 bakeoffs.

The most important point from this data is that incorrect decisions are avoided in almost all cases for the methods with the largest potential gains or loses from the optimization. As shown by the performance results earlier in this section, making correct conclusions on a subset of the hot methods can result in substantial performance improvements at the program level.

The large number of timeouts shown in Figure III.13 are due to the time-to-convergence problems described in more detail in Sections III.D and III.C. The online system collects only one timing sample each time the profiled method is invoked, so many methods do not generate timing samples fast enough to make a confident performance prediction before the bakeoff times out. These methods (approximately 50% of the bakeoffs) are lost optimization opportunities. However, because the online system uses the default optimization strategy in these cases, the resulting steady state performance will be the same as if no bakeoff was performed.

The results show that the biggest challenge to the performance auditing approach is time to convergence, i.e., the number of data points that must be collected before an accurate performance comparison can be made. Regions of code with high timing variance will eventually converge if the variance is finite, but the number of samples required may be impractical. Terminating the bakeoff early due to a timeout is not a serious problem in the online system because the default optimization can be used; however, this means the resources used to perform the bakeoff were wasted, and an optimization opportunity may have been lost. The following section will discuss the issue of convergence time in more detail, and a solution will be presented that collects timing data at a finer granularity to reduce convergence time.

III.C Lightweight Code Markers

Insufficient timing data results in poor convergence for infrequently invoked methods in performance auditing systems. To address this issue, more timing data must be collected. The baseline performance auditing system collects method invocation timings, which means that infrequently invoked methods will not produce very many method invocation timings. Performance auditing is applied to a program's hottest methods, so if an infrequently invoked method is selected for performance auditing, the program must be spending a significant amount of time executing instructions in the method.

A method that is both hot and infrequently invoked must contain loops, because a method that does not contain loops is either cold or frequently invoked. So the obvious approach to collect more timing data from these infrequently invoked methods selected for performance auditing is to collect *loop iteration timings* instead of method invocation timings.

It is easy to collect loop iteration timings instead of method invocation

timings — it is a simple matter of moving the instructions that read the cycle counter. The challenge is to collect loop iteration timings that are *comparable across compilations*. This is difficult because compilers can generate very different optimized code from the same source.

Consider the following example: an infrequently invoked method contains a loop. Loop iteration timings are required. Two compiled versions of the method are generated by the performance auditor, versions V_1 and V_2. In V_1 the optimizer produces two specialized versions of the loop: one version of the loop is unrolled four times, and the other version of the loop is not unrolled. In V_2, the loop is unrolled twice and no specialization is performed.

The compiler returns these two versions of the code V_1 and V_2 to the performance auditor, and the performance auditor must determine how to instrument these optimized binaries to collect loop timings. The first problem is determining which loops in the optimized binaries correspond to the original loop in the method. Identifying the original loop in V_1 is especially challenging because the optimizer has produced two specialized versions of the loop, so the original loop in V_1 has actually become two loops in the optimized binary. As another example, inlining can introduce new loops into an optimized binary if callee methods that contains loops are inlined.

Even if this first problem of matching loops across binaries is solved, a second problem remains: loop timings collected in the obvious way may not be comparable due to optimizations such as loop unrolling. Different unroll factors are used in V_1 and V_2, which means that meaningful conclusions can not be drawn from binary-level loop timing data without first compensating for the unroll factors.

To solve these problems, a mapping between the binaries is required. This is the same problem addressed by cross binary simulation points in Chap-

ter II, but the circumstances are different in this chapter. The work in Chapter II presented an approach to build a mapping between binaries by profiling and analyzing binaries. A different approach is used in this chapter because this work is done in a Java VM, which offers unique opportunities for better solutions.

In particular, the cross binary simulation point approach requires full profiling runs for all binaries involved in the comparison, while the approach proposed in this section requires recompilation of all binaries involved in the comparison. Java VMs already contain infrastructure for recompilation, making it easy to implement the proposed recompilation-based solution in this context.

This section presents *lightweight code markers*, which are a new solution to the problem of mapping between a program's source and its optimized binary. Lightweight code markers are injected into a method before optimization. Lightweight code markers have the semantics of an arithmetic operation on a specially allocated global variable. The optimizer is allowed to freely manipulate these markers, and the markers are removed after optimization, so no code is generated for lightweight code markers. The optimizer's semantics-preserving property automatically maintains the mapping between source and binary. Because lightweight code markers are operations on global variables, the optimizer may not remove the markers, although it may move, copy, or combine markers.

This section shows how lightweight code markers are used in a performance auditing system to collect comparable loop timings for infrequently invoked methods, which results in fewer inconclusive statistical analyses.

III.C.1 Inserting Lightweight Code Markers

Lightweight code markers are simple arithmetic instructions that modify a specially allocated global variable. Lightweight code markers are added to the intermediate representation (IR) at the beginning of optimization, and re-

main in the code during optimization. The optimizer is free to copy, move, and merge lightweight code markers. After, or during, optimization, lightweight code markers can be used to map their current code locations back to unoptimized code by examining the marker's current location and the optimizer's changes to the arithmetic performed by the marker instruction. Lightweight code markers are removed before register allocation and final code generation, so they do not result in code that actually executes at runtime.

Lightweight code markers must satisfy three conflicting goals to be effective:

1. Markers should be lightweight; they should have little or no impact on optimization. Inserting instructions that block optimization would allow mapping from source to binary, but it would do so by disabling optimizations. Markers that disable optimization are not effective because the goal is to map from source to binary without changing optimization decisions.

2. Markers must maintain the desired mapping information from source to optimized code, thus they must be respected by the optimizer sufficiently to maintain the desired information. If the optimizer deletes markers, or moves a marker far from the code it tracks, mapping information is lost.

3. Marker insertion and removal should be easy to implement. No changes to the optimizer should be required.

To achieve these goals, we use markers that are simple arithmetic instructions that increment a global counter by one. This gives the optimizer significant freedom to optimize the markers. For example, if a marker is placed in a loop (`X++`) and the optimizer unrolls the loop four times, the optimizer will produce four copies of the original marker (`X++;X++;X++;X++`). The optimizer should then combine the four copies of the marker into a single merged marker,

which increments by four (`X += 4`). Because each marker increments a different global variable, optimized markers can be mapped back to source locations based on the variable that they increment.

Although the optimizer is free to optimize the markers, the semantics-preserving nature of the compiler guarantees that certain properties are maintained. Markers are never deleted because they modify a global variable. The markers may be duplicated, moved, or combined, but at all times markers that reference a global variable X map to source-level marker X. Merged markers can be identified by their increment value being greater than 1.

The optimizer also guarantees that each marker's total dynamic count can not change. If code generation is allowed for markers, and the program is run deterministically, the value of each marker's global variable will always be the same at the end of the run, regardless of the optimization settings used. Markers are normally removed after optimization, so they are not normally present at runtime.

This means that every marker's count always reflects the ratio of its current block's execution frequency and its original block's execution frequency. For example, if a marker that increments by 1 is placed in block A, and the optimizer moves the marker to block B and modifies the marker so it increments by 5, the marker indicates that every execution of block B corresponds to 5 executions of block A.

So even though the optimizer may move code markers from their original location, markers still provide information on how their new location relates to their original location. This information is very useful for some applications, such as collecting a source-level block profile from an optimized binary as in [4].

Figure III.14 shows the overall impact on performance of inserting and removing a marker in every loop in every method, or every block in every method.

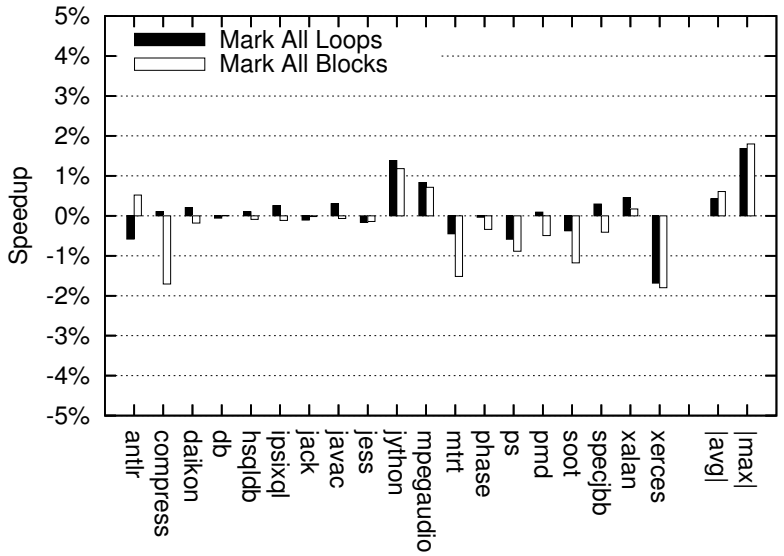


Figure III.14: Speedup/slowdown observed when a marker is inserted and removed in every loop or every block in every method.

Markers are inserted before the optimizer is run and are detected and removed when the optimizer completes, just before register allocation and code generation. When a marker is inserted and removed in every loop in every method, the average performance difference is 0.4%, and the worst-case performance difference is 1.7%. This demonstrates that the insertion of markers does not significantly change optimization decisions, and are indeed lightweight. Inserting and removing a marker in every basic block is significantly more invasive than marking every loop. This would be necessary if a full block mapping from source to binary was needed. Figure III.14 shows that even when markers are inserted and removed in every block in every method, the behavior of the optimizer still remains largely the same — the average performance difference is 0.6%, and the worst-case performance difference is 1.8%.

Marker Identification and Removal

Markers must be identified and removed just before register allocation and final code generation. Identifying markers, even after they have been transformed by the optimizer, is easy because each marker refers to a global variable that was specially allocated just for the marker. Any instructions that reference these special global variables are markers. To remove a marker, instructions that reference the special global variables are removed, dead code is eliminated, and the corresponding global variable is deallocated.

III.C.2 Application to Performance Auditing

The previous subsection described a technique for correlating code between a program's source and its binary. This subsection describes how this technique is applied in the context of a performance auditing system.

The goal is to improve convergence time for a performance auditing system by collecting loop timings. To do this, methods with poor convergence are identified, and timing data is collected for a subset of the method's loops. This subsection presents a technique to predict if loop timings are needed, and an algorithm to select loops for timing instrumentation. Lightweight code markers, described in the previous subsection, are used to collect comparable loop timings across compilations.

The goals of the loop selection algorithm are:

1. Collect loop timings only when necessary. If method timings provide enough timing data, loop timings should not be collected.

2. Collect only as much loop timing data as necessary. Unnecessary timing instrumentation should not be performed.

3. Ensure that a sufficient amount of execution occurs between timings. The

```
void selectLoops(method) {
  if method.invocationsPerSec > invocationThreshhold
    return
  targetFrequency = initialTargetFrequency
  while targetFrequency > 0
    blockFrequency = findLoop(method, targetFrequency)
    if blockFrequency == -1
      return
    else
      targetFrequency -= blockFrequency
}
Frequency findLoop(method, targetFrequency) {
  foreach loop in method in depth first order
    if (loop.frequency > targetFrequency &&
         block = selectBlock(loop, targetFrequency))
      return block.frequency

  return -1
}
Block selectBlock(loop, targetFrequency) {
  sort loop.blocks by ascending |block - targetFrequency|
  foreach block in loop.blocks
    minDistance =
      min(distanceToNearestTimingPoint, distanceToSelf)
    if minDistance > minDistanceThreshold
      return block
  return nil
}
```

Figure III.15: Loop Selection Algorithm

timer has a minimum resolution, and tight loops may not contain enough dynamic instructions per iteration to be measurable.

This subsection first describes a technique to predict if loop timings, and thus, the correlation technique, are required. Then an algorithm is presented that selects a subset of a method's loops for timing instrumentation, and finally an algorithm is presented that selects specific basic blocks in loops for timing instrumentation.

The proposed algorithm makes decisions based on profile data. We are working in a Java virtual machine that interprets bytecodes before compiling them, and while a method is being interpreted, the virtual machine collects profile data for the method.

The raw block profile data collected by the interpreter indicates approximately how many times each block was executed. Because methods can be interpreted for variable amounts of time before being compiled, the basic block execution frequencies are normalized for each block B:

$$normalizedFrequency_B = \frac{frequency_B}{\sum_{block} frequency_{block}}$$

In other words, the normalized frequency for a block is calculated by summing the frequencies of all blocks in the method, and dividing the block's frequency by the sum. This calculates the fraction of the method's execution that is spent in each block.

Figure III.15 gives a high-level overview of the loop selection algorithm. The loop selection algorithm is run *before* optimization. The selection algorithm places lightweight code markers into loops selected for timing instrumentation, then the optimizer runs, and finally the lightweight code markers are identified and replaced with full timing instrumentation code.

Predicting if Loop Timings are Needed

The first step is to determine if loop timings are necessary. When compilation occurs for hot methods, the method's profile data is examined to estimate how frequently the method was invoked. This estimate is used to predict if loop timings are necessary.

If the method was frequently invoked, the profile data will indicate that the method entry block was executed frequently compared to other blocks in the method. If the method was frequently invoked, loop timings should not be necessary.

On the other hand, if the method was rarely invoked, the profile data will indicate that the method entry block was rarely executed relative to other blocks in the method. If the method was rarely invoked, but the method is still considered hot, method timings will not generate enough timing data, and loop timings are needed for the method.

Choosing Loops to Instrument

If loop timings are needed for a method, timing data must be collected for some subset of the method's loops — it is usually not a good idea to time *all* of a method's loops. Additionally, a specific basic block must be selected for timing instrumentation in each loop — it is not always a good idea to instrument the loop head block.

The algorithm works as follows. First a target frequency threshold is defined to ensure that the algorithm does not perform too much instrumentation or too little instrumentation. Ideally, the total frequency of instrumented blocks should just exceed the target frequency threshold.

The method's loops are examined in depth-first order, starting from outermost loops. If the current loop's normalized loop head block frequency

exceeds the instrumentation target threshold, the current loop is selected for instrumentation, and a subroutine is called to select a specific basic block in the loop for timing instrumentation. The implementation of this subroutine will be described in the following subsection. The subroutine may or may not find a satisfactory block for the instrumentation code. If the loop is very tight, for example, the subroutine will not be able to find a good block in the loop for the instrumentation code, because timing instrumentation can not be inserted without high overhead. If no satisfactory block can be found, the depth-first search continues. If a satisfactory block was found, the target frequency threshold is decreased by the satisfactory block's weight. If the target frequency threshold is less than or equal to zero, or if no satisfactory blocks are found, the algorithm terminates. Otherwise, the algorithm runs again.

Choosing a Block to Instrument

This subsection describes a subroutine which, given a loop, determines where to best place the timing instrumentation. Timing instrumentation should be placed such that 1) many timings will be collected and 2) the timing data will be meaningful — the timer has a minimum resolution, so very small code regions can not be timed effectively.

To collect useful timing data, the timing instrumentation must be executed often, and there must be enough dynamic instructions between any two timing points. There may not be a good place to put the timing instrumentation — in this case, the subroutine returns an error code and no timing instrumentation is attempted for the loop.

To find the best block to place timing instrumentation in a loop, the subroutine first collects a list of the blocks in the loop, and sorts them by $|frequency - targetFrequency|$. In other words, the subroutine sorts the blocks

by the distance between each block's frequency and the target frequency. Blocks that closely match the target frequency will be considered first.

Next, the algorithm iterates over the sorted list of blocks, and Dijkstra's algorithm is called to compute single source shortest paths from each block in the list to all other blocks. The results of Dijkstra's algorithm are examined to determine the minimum distance between each block and itself, and the minimum distance between each block and all timing points. A single-source shortest paths algorithm is used here, instead of an all-pairs shortest paths algorithm, because the distance between a block and its nearest timing point can change as additional timing points are selected.

If the minimum distance between a block and its nearest timing point exceeds the minimum distance threshold, the block can be instrumented safely, because there will always be enough dynamic instructions between instrumentation points.

The distance between a pair of blocks is measured in terms of instruction *cost* rather than instruction count. For example, a `load` instruction puts more distance between timing points than an `add` instruction. Instruction costs are static estimates based on instruction opcode. Instruction costs are commonly used in instruction scheduling. Additionally, edge weights are not considered. The minimum distance between two points is calculated without considering the frequency or even the feasibility of paths. This is done for safety, because interpreter profile data may not completely represent a method's future behavior.

Figure III.16 shows control flow graphs for three infrequently invoked methods that can not be safely instrumented to collect loop timing data. In these control flow graphs, circles represent basic blocks, rectangles indicate loops, and the number in each circle indicates the cost of each block, which is just the sum of the cost of the block's instructions.

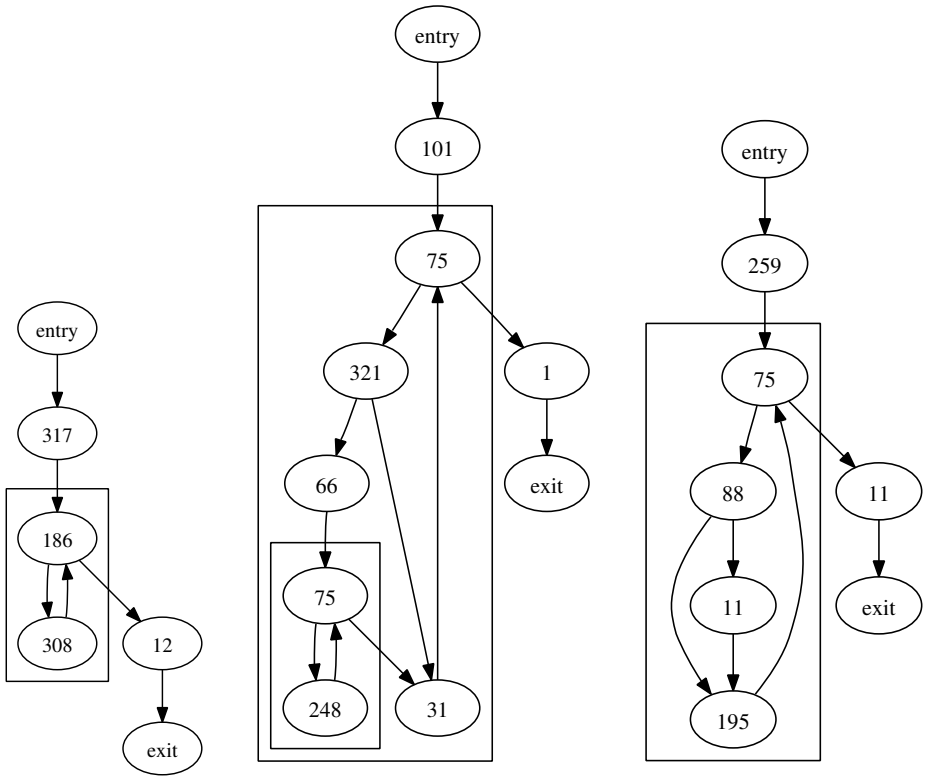


Figure III.16: Three CFGs selected for loop instrumentation, where the loop selection algorithm was unable to find a satisfactory timing point.

These three methods are infrequently invoked, yet loop timings cannot be collected from these methods, because all of their loops are tight. Inserting instrumentation code into these loops would not provide useful timing data — if timing instrumentation was inserted into these tight loops, the timing code would not actually measure anything, because these loops do not contain enough dynamic instructions per iteration to compensate for timing and instrumentation overheads.

The algorithms described in this section automatically identified the three methods shown in Figure III.16 as infrequently invoked methods where loop instrumentation could not be safely applied.

III.C.3 Methodology

The experiments in this section were performed with nearly the same methodology described in the section III.B.3, except that a newer version of IBM's J9 VM was used. Otherwise, the machine and benchmarks were identical.

Table III.2 shows the benchmarks used. The first column reports the number of methods executed by the JVM to load and run the benchmark. This counts all Java methods, including library methods. The second column lists the total size (in KB) of all bytecodes executed for each benchmark. These numbers report dynamic metrics, i.e., they are based on what is executed, not what could be executed. The third column lists the number of hot methods for each benchmark. Methods are hot if they consume enough cycles to be selected by J9 for aggressive feedback-directed optimizations. For these methods, J9 first performs an additional compilation to instrument the method for profiling, then optimizes the method at one of the two highest optimization levels (O4 or O5).

The last column in Table III.2 lists the number of hot methods that are infrequently invoked. These are hot methods, so the processor spends a significant

Table III.2: Benchmark suite

Program	Methods Executed	Bytecodes Exe (KB)	Hot Methods Total	Hot Methods Loopy
antlr [87]	1702	228	2	0
compress [84]	770	66	3	2
daikon [21]	2108	171	1	0
db [84]	782	67	3	2
hsqldb [87]	1416	147	2	1
ipsixql [18]	828	61	7	1
jack [84]	746	56	3	0
javac [84]	1467	133	2	0
jbb2000 [83]	1197	115	3	2
jess [84]	1140	86	4	1
jython [87]	1777	186	2	1
mpegaudio [84]	866	78	2	1
mtrt [84]	853	76	3	0
phase [65]	450	31	2	0
pmd [87]	2030	128	1	0
ps [87]	946	75	1	0
soot [80]	2061	235	4	2
xalan [87]	2108	171	1	0
xerces [94]	521	36	1	0

amount of time executing instructions in these methods, yet they are not invoked frequently. This implies that these methods spend most of their time in loops. These methods are discussed in detail in Section III.C.2.

The Read Time-Stamp Counter (`RDTSC`) instruction is used to collect timing data. The `RDTSC` instruction reads the processor's 64-bit cycle counter and stores its value in a register. To collect timings, a procedure `startTimer()` is defined, which reads the cycle counter and stores the count in the method's stack frame. Similarly, a procedure `stopTimer()` is defined which subtracts the current cycle count from the cycle count on the stack, and stores the difference in a circular buffer.

To collect method timings, the method's entry points are instrumented

with calls to `startTimer()`, and the method's exit points are instrumented with calls to `stopTimer()`. To collect loop timings, the method's entry and exit points are instrumented to collect method timings, but additional back-to-back calls to `stopTimer()` and `startTimer()` are inserted at each loop timing point. So, when a method that has been instrumented to collect loop timings is run, the dynamic sequence of calls to `startTimer()` and `stopTimer()` will be as follows:

```
startTimer() // method entry
stopTimer()  // loop timing point 1
startTimer() // loop timing point 1
stopTimer()  // loop timing point 2
startTimer() // loop timing point 2
stopTimer()  // method exit
```

All calls to `startTimer()` and `stopTimer()` are inlined.

This timing methodology measures the total time that the method was active on the stack to ensure fair comparisons in the presence of changing inlining decisions. Measuring only time spent in the instrumented method (excluding callees) would produce incorrect results in the presence of method inlining.

III.C.4 Evaluation: Performance Auditing with Lightweight Code Markers

This subsection evaluates the effectiveness of the proposed technique to collect comparable loop timings. Three sets of results are presented. First, a metric analysis, where the quality of loop timings are evaluated by examining their statistical properties. Next, an accuracy analysis, which examines the effectiveness of loop timings when used to predict speedups and slowdowns. Finally, a convergence study, which determines if loop timings will improve convergence time.

Metric Analysis

This section evaluates the proposed loop timing approach with metrics that measure their achievement of the primary objectives discussed in Section III.C.2: collecting enough, but not too many, data points. A sufficient number of timings are needed so the statistical analysis will converge quickly, yet the timer can not be started and stopped too frequently, because the timer has a minimum resolution.

Two metrics are defined to evaluate the effectiveness of the loop selection algorithm: *converge score* and *perturb score*. Converge score models how quickly the set of timing data is likely to converge when run through the statistical analysis. Higher is better. It is computed by dividing the number of timings collected by the variance in timings:

$$convergeScore = \frac{numTimings}{variance(timings)}$$

Converge score is defined in this way because the amount of time required for the statistical analysis to determine if one set of timings is faster than another is determined by the number of timing points available, and the variance in the timing data. When more timing points are available, or the variance in timing data decreases, the statistical analysis converges faster. Thus, the converge score metric is defined so that timing data with higher converge scores will converge more quickly.

Figure III.17 shows converge scores for hot methods. Results are shown for method timings, and method+loop timings (Section III.C.3 described how loop timings are always collected in addition to method timings). This figure shows that loop timings are only collected when necessary — methods with poor method timing convergence (the left 1/3 of the graph) are identified by the loop selection algorithm, and loop timings are collected for those methods, resulting

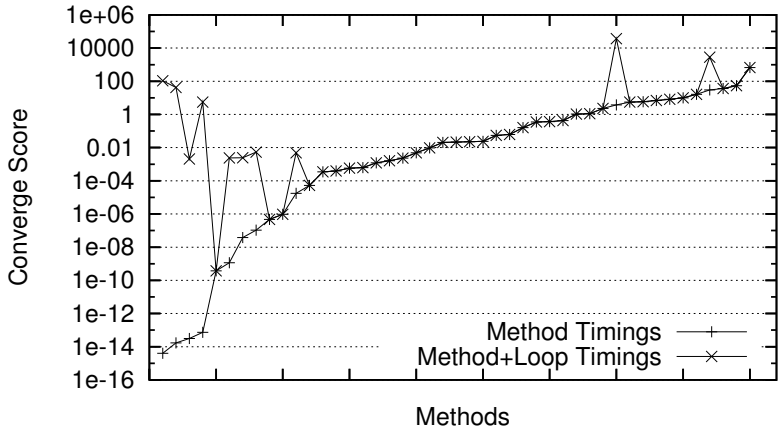


Figure III.17: Converge scores for hot methods, higher is better.

in improved converge scores. Most of the remaining methods (the right 2/3 of the graph) are generally left alone. There are two mispredictions on the right side of the graph, where loop timings are collected when they are not actually needed.

There are three methods on the left that could benefit from loop timings, yet loop timings are not collected for these methods. These three methods were discussed in Section III.C.2, and their control flow graphs were shown in Figure III.16. The proposed prediction scheme correctly predicts that loop timings are needed for these three methods, but the timing point selection algorithm is unable to find code locations where timing instrumentation can be safely inserted into these methods, because these three methods spend all their time in tight loops.

Figure III.17 shows that loop timings improve convergence time. Loop timings drastically increase the number of timings available without significantly increasing the variance in timings. This results in a large boost to the converge score for methods selected for loop instrumentation.

The second metric is *perturb score*. It models the perturbation intro-

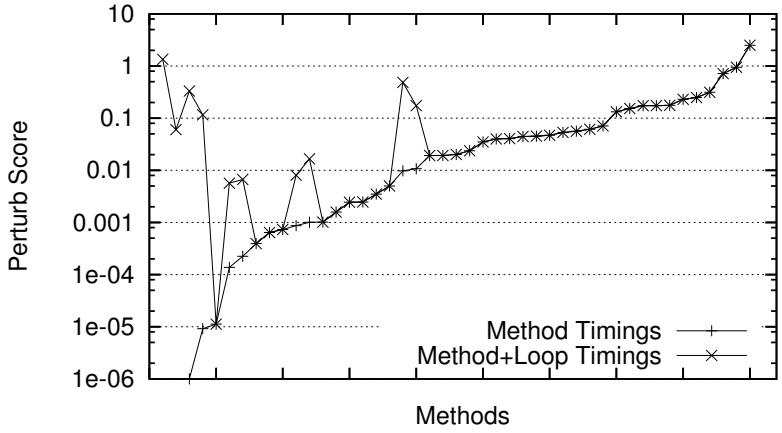


Figure III.18: Perturb scores for hot methods, lower is better.

duced by the increased frequency of timing collection. The minimum observable timing with the proposed cycle counting infrastructure (MIN_TIMING) is 88 cycles on our test machine, which is the number of elapsed cycles measured between two consecutive reads of the cycle counter. If the collected timing data contains many timings that are close to the minimum observable timing, then they are timing very little program execution and are most likely inaccurate.

Perturb score is defined as follows:

$$perturbScore = \frac{\sum \frac{100}{timing-(MIN_TIMING-0.1)}}{numTimings}$$

The perturb score metric is defined so that timing data that contains many timings close to the minimum possible timing will have a high perturb score. The maximum possible perturb score is 1000, which occurs when every timing collected is the minimum of 88 cycles. The constant 100 is used in the definition of perturb score to normalize the scores so that a perturb score of 1 or lower is likely to be acceptable — if the perturb score is 1, then the average timing collected is 187.9 cycles.

Figure III.18 shows perturb scores for hot methods. Increases in perturb score are inevitable when loop timings are used, because collection of loop tim-

ings necessitates starting and stopping the timer much more often. This figure shows that the loop selection algorithm increases the number of timings for the key methods without having a significant negative impact on perturbance. The perturb score for the leftmost loop timing data point is 1.34, which corresponds to an average cycle count of 168.5, which is still nearly double the minimum observable timing.

Accuracy Analysis

To evaluate the accuracy of the proposed approach, timing data was collected for various optimization settings (O1 and O2, for example) and the timing data was compared to determine which optimization setting produces faster code. For each hot method, its performance is first evaluated across different optimization settings with timings collected via method-only timing instrumentation. This result is considered the correct answer because the method-only instrumentation perturbs the execution the least, and is the most accurate timing mechanism.

The performance of hot methods across different optimization settings is also evaluated with method-and-loop timings collected via method+loop instrumentation, and the results of the two evaluations (method-only, method+loop) are checked to see if they agree on the performance difference between the differently compiled versions of each hot method.

If both techniques indicate a speedup, or both techniques indicate a slowdown, the method+loop timings are correct. But if one technique indicates a speedup and the other indicates a slowdown, or vice versa, the method+loop timings are incorrect. If method+loop timings indicate that the magnitude of the speedup or slowdown is less than 1%, the method+loop timings are not confident.

Figure III.19 presents the results of this experiment. This figure shows the overall accuracy of the proposed loop timing approach compared to the

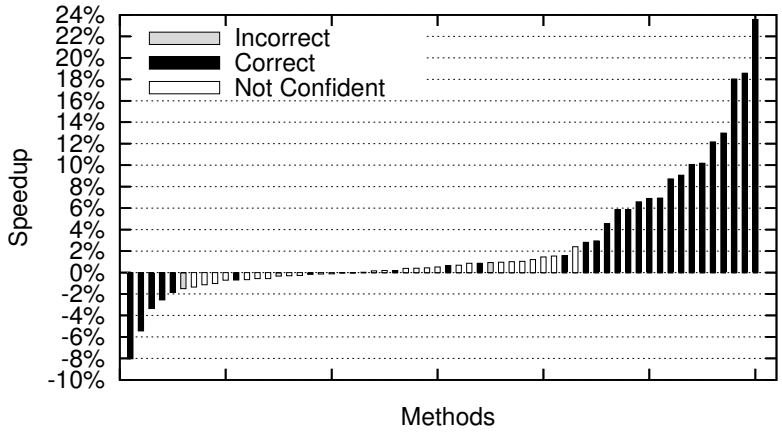


Figure III.19: Accuracy of our approach, where markers are inserted before optimization, and markers are replaced with full timing instrumentation after optimization.

method-only instrumentation used in Section III.B. This figure shows the overall accuracy of the loop timing approach, including the effects of loop timing point selection, code marker insertion, and replacing code markers with loop timing instrumentation. Hot methods are on the x-axis, and the y-axis shows the speedup detected by the method timings (i.e., the true speedup). The bars are solid black if the method+loop timings are correct, gray if they are incorrect, and white if they are not confident. This figure shows that, overall, the loop timing approach is highly accurate. The worst-case error is a failure to detect a 1% speedup.

To demonstrate the importance of using lightweight markers, Figure III.20 shows the accuracy of the naïve approach — inserting full timing instrumentation before optimization, then running the optimizer.

The naïve approach results in significant changes to optimization decisions, resulting in incorrect performance evaluations. With the naïve approach, the worst case error is failure to detect an 8% slowdown. Lightweight markers do not significantly change optimization decisions, resulting in significantly more accurate speedup detection.

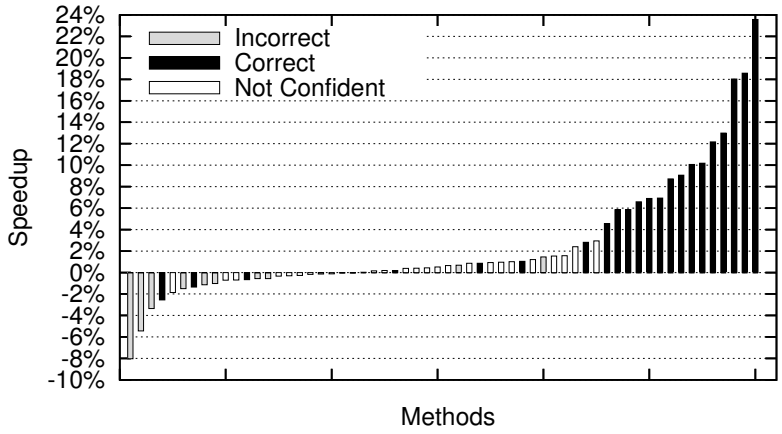


Figure III.20: Accuracy of the naïve approach, where full timing instrumentation is inserted before optimization.

Convergence Study

To demonstrate the importance of increasing the number of timings, the amount of time needed for the statistical analysis to detect a range of speedups is calculated. This is very similar to the offline convergence study presented in Section III.B.4.

For each hot method, method and method+loop timing data is collected. Each set of timing data is randomly partitioned into two sets, A and B. Because A and B came from the same data set, the average timing value in each set should be very close. Next an artificial speedup of $X\%$ is applied to the timings in set B. At this point, the timings in set B represent an optimized version of the method that runs exactly $X\%$ faster than the original.

A confidence threshold $Z\%$ is set (80% and 99.99% in this study). The experiment adds 100 samples at a time to A' from A, and to B' from B until $Z\%$ confidence is reached. Initially A' and B' each contain 100 samples. Confidence evaluation is performed, and if the confidence is not above the confidence threshold $Z\%$, 100 more samples are added to both A' and B', and the experiment

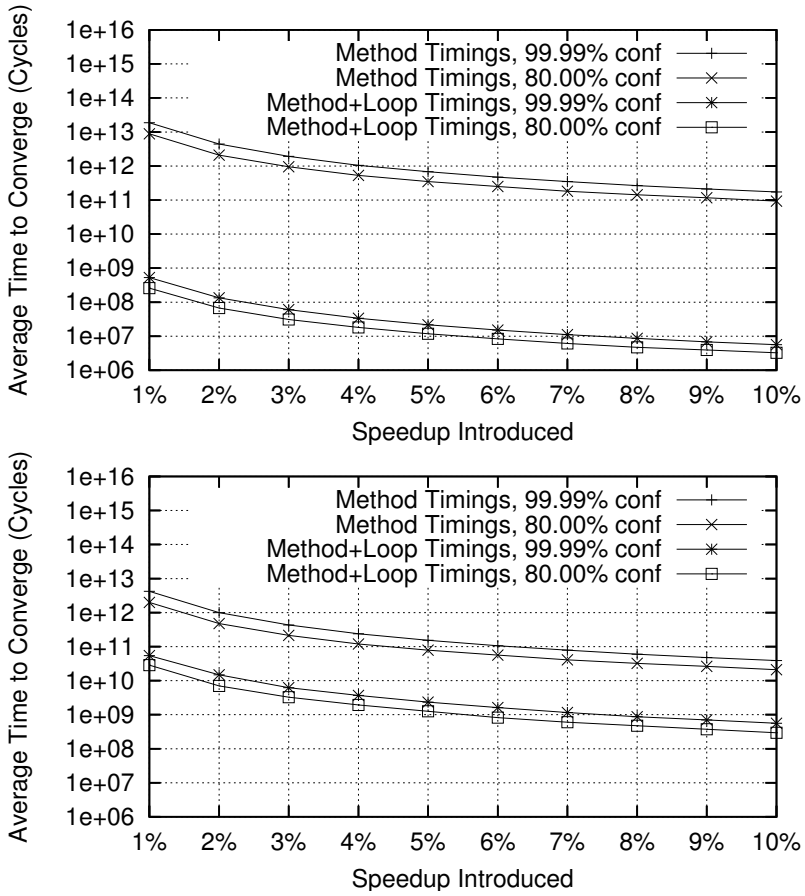


Figure III.21: Convergence time, comparing method+loop timings to method timings. Top figure: convergence time for hot methods selected for loop instrumentation. Bottom figure: convergence time for all hot methods.

repeats. More samples are added to A' and B' until confidence is above $Z\%$.

Unlike the offline convergence study presented in Section III.B.4, in this experiment there is always enough timing data available to reach the desired confidence threshold.

When the statistical analysis confidently detects a performance difference, the time to converge is noted, which is just the sum of the cycle counts in A' and B'. The convergence experiment is repeated 100 times with different random seeds, and the results are averaged over the 100 trials.

The top figure in Figure III.21 shows convergence time only for methods that were selected for loop instrumentation. This graph shows that loop timings improves convergence rate for infrequently invoked methods by about four orders of magnitude on average, which is consistent with the results shown in Figure III.17.

The bottom figure in Figure III.21 shows convergence time for all hot methods, including methods that were not selected for loop instrumentation. A relatively small number of hot methods require loop instrumentation, so when methods that were not selected for loop instrumentation are included in the study, the results are more modest. Still, loop timings improve the overall average convergence rate of all hot methods by about two orders of magnitude on average.

III.C.5 Other Potential Uses for Lightweight Code Markers

Lightweight code markers should prove to be useful for much more than just improved performance auditing.

Albert [4] modifies the optimizer to track basic block operations to collect a source-level block profile from an optimized binary. This should be easy to do with code markers: Insert a code marker into every basic block, and allow the optimizer to freely manipulate the markers. Regardless of where the markers may be at the end of optimization, when a marker is found that increments its global variable by N, that indicates that the marker's new block executes once for every N executions of the marker's original block, as described in Section III.C.1. So, each marker can be trivially replaced with instrumentation to collect a source-level block profile.

BMAT [91] uses binary matching to reuse stale profile data with new versions of a program. The matching approach is key because it works when there are source-level changes, unlike the bookkeeping approach and our ap-

proach, although these two approaches are tend to be more accurate than matching approaches when different compilers and optimizers are involved. But with code markers' ability to map between source and binary, *source* matching can be done instead of binary matching. Source matching is much easier than binary matching, because the matching algorithm only needs to worry about source-level changes at source-level — the effects of compilers and optimizers do not have to be considered at all.

In Chapter II, this dissertation proposed a system to meaningfully compare the results of accelerated architectural simulations when differently-compiled binaries are used for the same program. Architectural simulators are very slow, so accelerated architectural simulation methodologies are used, which simulate many small representative samples of program behavior to approximate whole-program simulation. In Chapter II the challenge was to find the same sample of program behavior in differently-compiled binaries. A matching system driven primarily by profile data (annotations in call-loop graphs) was presented. High accuracy was achieved, but special profiling runs of each binary were required. A code marker-based approach to this problem should allow for high accuracy without requiring special profiling runs — recompilation of the benchmarks would be required, however.

Lightweight code markers can be used to build a mapping between binaries by first mapping each binary to source. So, to use lightweight code markers for cross binary architectural simulation as described in Chapter II, lightweight code markers could be inserted at any point where a mapping between binaries is desired - for example, at procedure entry points and before loop branches. Then, when the multiple binary versions are being compiled, lightweight code markers are inserted before optimization. The optimizer is run, and finally the mutated lightweight code markers are examined.

By examining the optimizer's effects on the lightweight code markers across compilations, a mapping can be built across binaries. Furthermore, because lightweight code markers carry information about how the optimizer has changed the lightweight code markers, the mapping information provided by lightweight code markers can be used to compensate for differences in optimization across binaries, which was an area of concern described in Section II.D.1. For example, if the optimizer produces two specialized versions of a loop in one binary version, lightweight code markers can be used to correctly map the two copies of the loop in the specialized binary to single copies of the loop in other binaries. It would be difficult to compensate for optimization scenarios like this with the profile guided binary matching approach presented in Section II.D.1.

III.D Discussion

The approach taken by this chapter — comparing the performance of multiple versions of a region of code *without* attempting to hold inputs and program state constant — is often initially dismissed as infeasible. The statistical timing analysis presented in Section III.B.2 addresses this main concern with its ability to detect arbitrarily small performance differences with arbitrarily high accuracy, as long as the client is willing to wait for the collection of enough timing data. The accuracy of the approach is not an issue, because accuracy can be made arbitrarily high. Instead, the main issue is the *feasibility* of the approach: how much timing data is needed to make reasonably accurate performance comparisons? The results show that the proposed approach is indeed feasible. This section discusses some of the other challenges that were overcome in building the performance auditor.

One technical challenge was engineering the system to ensure that the collection and processing of data did not skew the results. Whenever possible, our

system randomizes the order in which data is collected and processed (e.g., which version of the code is timed first, which buffer is processed first, etc.). Adopting this approach whenever possible improves the accuracy of the decisions made by the online system.

Removing outliers from the timing sets was also a key to timely convergence. A VM environment has many sources of timing noise, and removing outliers from the timings was an effective solution. Removing outliers is not strictly necessary, because the noise would be evenly distributed across all optimized versions; however, removing the outliers improves convergence time substantially. Unfortunately, removing outliers also has a downside; some data points labeled as "outliers" could have been a legitimate effect of the optimization being evaluated. For example, an optimization that increases the cost of a rare path might not be detected if too many outliers are discarded.

A key to reducing the number of discarded outliers is to reduce the noise in the timing mechanism itself. One source of noise is when timings are polluted by VM activity, such as JIT compilation or garbage collection. Such timings can be identified and eliminated fairly easily. Another source of potential noise is that the cycle counter available on our platform provides wall clock time, rather than CPU time, so if the operating system switches out the VM process to run another process, the other process's time is included in our method timings. Operating system support to provide thread-specific measurements of CPU time would help reduce these outliers.

The overhead caused by compiling multiple versions of a method for the bakeoff cannot be completely ignored, but can be managed, and the additional overhead is defensible if the technique results in legitimate speedups. As described in Section III.A.1, some production VMs already perform additional compilation of hot methods to perform instrumentation for feedback-directed op-

timization. The overhead introduced by these additional compilations is reduced by performing compilation in the background, and moving compilation to free processors. Hardware trends, such as multi-core systems, are likely to help in this regard, especially if the additional parallel cycles provided by the hardware are not fully exploited by the application. These cycles can easily be used by a VM for additional optimizations and other runtime services.

Differences in code layout can introduce performance variations, which are not exploited by our current implementation. On many architectures, the position of code in memory can have a significant performance impact. Our current system does not try to place the code for the bakeoff in a good location. In addition, it recompiles each method after the bakeoff completes, which may place the final code in a non-optimal location. However, it is possible to use the performance auditor to help find a good code layout: after the bakeoff completes, the optimized version can be patched to remove the timing code, instead of recompiling. Doing so ensures that any benefits from code positioning identified during the bakeoff are not lost.

Program phase shifts provide both an opportunity and a challenge for any online system. Performing optimizations online allows for the detection of phase shifts, and optimizations can be targeted to maximize performance in each phase, which exposes potential performance gains not visible to less adaptive systems. However, a program phase shift can also reduce performance if the system does not re-evaluate optimization decisions when program behavior changes. Online performance auditing, like all feedback-directed optimizations, is susceptible to the potential benefits and degradations due to program phases. Managing the performance risks of performing FDO in the presence of program phases remains an open research area.

Finally, an open question regarding the future use of our work is how

to manage the exponential optimization search space. Not only are there dozens of tuning knobs and heuristics that could potentially be applied at runtime, but some of these heuristics have hundreds of potential values; testing all combinations is clearly intractable. However, this exponential space is not created by our work; it already exists, and is essentially ignored by today's systems; a large space does not imply that it is not worth searching, or that substantial performance improvements cannot be attained. An existing large body of work explores the optimization space with offline empirical search, and our framework enables a similar line of online research. We believe that the large search space can be best managed by using a combination of offline *and* online techniques. Extensive offline tuning can be performed to identify the most problematic optimizations and heuristics, and these can be explored by an online system.

III.E Summary

This chapter presented the performance auditor, which directly measures the performance of generated code to guide optimization decisions. The performance auditor collects timing samples for generated code online, as the application executes. Individual timing samples can not be directly compared, because programs change their state over time.

A statistical timing analysis was presented that examines pools of timing data collected from different implementations of a code region as the application executes. The statistical analysis determines how many timing samples are needed to confidently predict which implementation is the fastest by considering the variance in timings. This statistical technique is at the core of a performance auditing system.

This chapter also presented lightweight code markers, which map between a program's source and its optimized binary without using heuristic match-

ing, and without modifying the optimizer. Lightweight code markers were used to improve statistical confidence in a performance auditing system by collecting loop iteration timings in addition to method timings for infrequently invoked methods.

Acknowledgements

The performance auditor was the result of collaboration with Matthew Arnold and Michael Hind at IBM T.J. Watson, and Brad Calder at UC San Diego. I thank my co-authors for allowing me to present the results of our collaboration in my dissertation.

Section III.B contains material that appears in "Online Performance Auditing: Using Hot Optimizations Without Getting Burned", in *Conference on Programming Language Design and Implementation (PLDI)*, Jeremy Lau, Matthew Arnold, Michael Hind, Brad Calder. The dissertation author was the primary investigator and author of this paper. Portions of Section III.B are Copyright ©2006 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or `permissions@acm.org`.

Section III.C contains material in "A Loop Correlation Technique to Improve Performance Auditing", submitted to *Conference on Parallel Architectures and Compilation Techniques (PACT)*, Jeremy Lau, Matthew Arnold, Michael Hind, Brad Calder. The dissertation author was the primary investigator and author of this paper.

IV

Conclusion and Future Challenges

Performance comparisons are ubiquitous in computer science. For example, computer architects compare the performance of processors, and compiler writers compare the performance of generated code.

It is typically impossible to prove that one computer system is *always* faster than another, for all possible workloads, so these performance comparisons are used as predictors: performance is compared on several representative workloads, and the results are used to argue that one computer system is generally faster than another. Unfortunately, there are many scenarios where it is difficult to make a fair performance comparison. This dissertation focuses on two such scenarios, where performance must be predicted across compilations.

IV.A Predicting Performance Across Compilations

This dissertation focused on the problem of predicting performance across compilations, where differently-compiled binaries are examined, and the best-performing binary is identified. These types of performance comparisons are

common, especially in the fields of computer architecture and compilers. This dissertation discussed two specific instances of the problem of predicting performance across compilations: cross binary simulation points, which are used in accelerated architectural simulations, and performance audited dynamic optimization, which uses empirical performance measurements to guide dynamic optimizations.

IV.A.1 Cross Binary Simulation Points

Computer architects typically evaluate new processor designs through slow cycle-level simulation. But because of the poor performance of cycle-level simulators, accelerated simulation methodologies are very popular, where small samples of a program's behavior are simulated, and the results are extrapolated to predict the results of a whole-program simulation.

But these accelerated simulation techniques make it difficult to meaningfully compare the results of these accelerated simulation techniques, especially when multiple compilations of a program are involved. The main challenge is that samples must be selected consistently across compiled binaries, so a set of high-level program behaviors selected for simulation are equally represented in all binaries. To overcome this challenge, this dissertation described a technique to select simulation regions consistently across binaries with a profile guided binary matching approach.

IV.A.2 Performance Audited Dynamic Optimization

Dynamic optimizers must predict if their optimizations will actually improve performance before applying them — if an optimization is unlikely to improve performance, or if an optimization will degrade performance, the optimization should not be applied.

As computer systems become increasingly complex, it becomes increasingly difficult for dynamic optimization systems to predict if their optimizations will actually improve performance. To address this issue, this dissertation presented performance auditing, which guides dynamic optimization decisions by performing empirical performance evaluations as programs execute. The performance of differently-compiled versions of the same code were measured, and the results of the measurements were used to directly guide optimization decisions.

The main challenge in building a performance auditing system was that performance measurements were collected as programs execute. This meant that individual performance measurements were not directly comparable, because a difference in timings may have been due to the program doing fundamentally different work when one of the timings was collected. To overcome this challenge, this dissertation presented a statistical technique that analyzes pools of timing data.

IV.B Building a Mapping Across Binaries

In order to predict performance across compilations, a mapping between the differently-compiled binaries is often required. With cross binary simulation points, a mapping between binaries was required so semantically equivalent simulation points could be selected across binaries. It was important to select semantically equivalent simulation regions in order to ensure that performance analysis results for these simulation regions were comparable across binaries. Similarly, in performance auditing, a mapping between binaries was required so timing data collected from one binary could be meaningfully compared to timing data collected from another binary.

Building a mapping across binaries is fundamentally a matching problem, where semantically equivalent code regions must be identified across binaries.

Binary matching techniques fall into two general categories:

1. *Pre-Optimization Matching* — No work is required to match binaries before they are optimized, because the differently-compiled binaries are compiled from the same source. Nothing needs to be done to do source-level matching when the source is the same. The challenge then becomes building a mapping between the source and each optimized binary. The lightweight code markers presented in Section III.C.1 fall into this category, as well as the modify-the-optimizer "bookkeeping" approaches described in Section III.A.4.

2. *Post-Optimization Matching* — Binaries compiled from the same source can appear very different from each other after optimization: the output of `gcc -O1` is typically very different from the output of `gcc -O2`. These differences make it challenging to match binaries after optimization. The profile guided matching algorithm presented in Section II.D.1 falls into this category, as well as the binary matching techniques described in Section III.A.4.

These two general categories of solutions each have their advantages and disadvantages. Pre-optimization matching typically results in higher accuracy, but it requires recompiling binaries, and typically requires control over the compiler and optimizer. On the other hand, post-optimization matching tends to have more issues with accuracy, but it does not require recompiling binaries or control over the compiler or optimizer.

In light of these advantages and disadvantages, this dissertation presented two techniques to build mappings between binaries, each tuned for its corresponding use case.

IV.B.1 Cross Binary Simulation Points

In computer architecture research, source code is often not available. For example, suppose a processor vendor wishes to evaluate a set of instruction

set extensions on real-world applications. For a program to take advantage of instruction set extensions, recompilation is required. But application vendors are unlikely to provide source code for their products, even if it is only for the purposes of performance evaluation. More likely, the processor vendor will send their compilers to the application vendor, the application vendor will produce binaries, and then send the binaries back to the processor vendor for performance evaluation.

In these situations, it is necessary to predict performance across compilations. Because of the poor performance of architectural simulators, sampled architectural simulation is usually used. But Section II.D.3 showed that simulation samples must be selected consistently across compilations. To identify equivalent samples across binaries, a mapping between binaries is required. A pre-optimization matching solution is infeasible in these situations, because source code is not available, so the binaries must be compared directly. Cross binary simulation points, described in Section II.D, compare binaries directly to identify semantically equivalent code regions for sampled simulation[1].

A post-optimization binary matching approach was used for cross binary simulation points, but cross binary simulation points require very high accuracy. To achieve high accuracy with a binary matching approach, each binary was profiled to collect call-loop graphs, and the profile data in call-loop graphs was used to guide a binary matching algorithm. The benefit was that comparable simulation points were identified across binaries without requiring control over the compiler. The cost, however, was that special profiling runs of each binary were required to identify semantically equivalent code regions.

[1] In the specific case of the cross binary simulation point evaluation methodology, closed-source Intel compilers were used to generate the differently-compiled binaries, making a pre-optimization matching solution infeasible because control over the compiler was not available

IV.B.2 Performance Audited Dynamic Optimization

Dynamic compilation systems compile programs as they run, making them attractive targets for pre-optimization binary matching. This dissertation presented performance auditing in Section III.B, where empirical performance measurements of running code are used to guide optimization decisions. In a performance auditing system, timing samples are collected as a program runs to evaluate the performance of its generated code.

In order to meaningfully compare these timing samples, the timing samples must be collected consistently. This was not an issue in the baseline performance auditing system presented in Section III.B, because the baseline system only collected method timings, so the runtime of one implementation of `System.out.println()` would always be compared to another implementation of `System.out.println()`, ensuring comparable results.

But Section III.B.5 showed that method timings are ineffective for infrequently invoked methods, because a large number of timing samples are needed by the proposed statistical analysis technique to confidently determine which implementation of a method runs fastest. Only a program's hottest methods are selected for performance auditing, so if a method is selected for performance auditing, and it is infrequently invoked, that method must be spending most of its time in loops. Therefore, the obvious solution is to collect loop timings for such methods, instead of method timings.

But to collect comparable loop timings for infrequently invoked methods across differently-compiled binaries, a mapping between binaries is required, because the same loop may be optimized in different ways in different compilations. So once again, a technique is required to build a mapping between binaries. The same two options present themselves: pre-optimization matching, or post-optimization matching.

Collecting loop timings for performance auditing, like cross binary simulation points, requires very high match accuracy. The post-optimization profile guided matching approach used in cross binary simulation points could have been adapted for performance auditing, but a profiling phase would be necessary, which takes time, and time is a precious resource in dynamic optimization systems.

Performance auditing works in the context of a dynamic compilation system, which makes a pre-optimization matching approach very attractive. With a pre-optimization matching approach, the idea is to do the matching at source level, then map from source level to each binary. This typically requires bookkeeping — tracking the optimizer's changes to the code as it runs, as described in Section III.A.4. Implementation of these bookkeeping approaches tends to be labor intensive, because a fairly thorough understanding of the optimizer's operation is required. To address this issue, this dissertation proposed lightweight code markers, which allow the optimizer's changes to be tracked without modifying the optimizer. The benefit was that comparable loop timings were collected across binaries. The cost was that compilation was required — in order to track the optimizer's changes, the optimizer must be run. But because the performance auditor works in the context of a dynamic optimization system, the incremental cost of this solution was low, because dynamic optimization systems already compile programs as they execute.

IV.C Future Challenges

This dissertation presented ground breaking work in two problem areas: cross binary architectural simulation, and performance audited dynamic optimization for general purpose programs. Exploration of these problem areas has just begun, and naturally, there are many open questions.

IV.C.1 Cross Binary Architectural Simulation

This subsection examines several areas of future work in the area of cross binary architectural simulation.

Alternative Simulation Methodologies

This dissertation presented a technique to perform cross binary architectural simulation with the SimPoint [78] sampled simulation methodology. There are several other sampled simulation methodologies available, the most popular alternative being SMARTS [93]. The underlying philosophies of these two methodologies are very different: SimPoint examines the code executed by a program to identify representative samples of execution, while SMARTS relies on statistical sampling. SimPoint simulates a small number of large samples, while SMARTS simulates a large number of small samples. Both techniques are very accurate, yet they are very different in their approach. Does a statistical sampling approach like SMARTS require modifications to accurately compare sampled simulation results across binaries? In other words, will SMARTS apply consistent sampling bias across binaries?

Statistical sampling techniques like SMARTS may require modifications for accurate cross binary sampled simulation, because very similar issues were raised with SimPoint at the start of the work presented in this dissertation — because SimPoint's goal is to minimize sampling error, it was believed that cross binary results would automatically be comparable. This was shown not to be the case in Section II.D.3 for SimPoint, and the same may be true of SMARTS. If so, it will be interesting to see what modifications to the approach are needed.

Improved Binary Matching

A profile guided binary matching technique was presented in Section II.D.1. The proposed approach is highly accurate, meaning that the approach very rarely produces mismatches. But accuracy is not everything — an approach that produces no matching information will produce no mismatches. Coverage is very important, and there are some issues where the proposed approach fails to identify semantically equivalent procedures or loops across binaries, due to differences in optimization. For example, if a loop is unrolled in one binary version, that loop can not be matched by the proposed binary matching technique, because the loop iteration counts will not match in the profiled call-loop graphs. It does not seem possible to compensate for all possible optimizations with a post-optimization binary matching approach, but it will be very interesting to see where the limits are, and how many more correct matches can be produced with improved post-optimization binary matching techniques.

IV.C.2 Performance Audited Dynamic Optimization

This subsection examines some areas of future work in the area of performance audited dynamic optimization.

Performance Auditing Beyond Dynamic Optimization

The statistical analysis at the core of a performance auditing system is not tied to dynamic optimization. It may be possible, for example, to do performance audited garbage collection or performance audited processor scheduling. The idea is simply to extend the philosophy of online empirical performance measurements wherever possible: as computer systems become increasingly complex, the predictive power of performance models decreases, motivating the need for empirical measurements. The performance auditing approach provides a frame-

work for online empirical performance measurements, which should be useful in a variety of scenarios beyond dynamic optimization.

Whole Program Performance Auditing

Performance auditing, as described in this dissertation, is used to target specific code regions. It may be possible to slightly modify the approach so that whole program performance is evaluated, instead of the performance of a specific code region in a program. In the proposed performance auditing system, a timer is started on method entry, stopped on method exit, and reset on loop iterations. What if a timer is started at program startup, never stopped, and reset periodically? With such an approach, performance auditing could be used to monitor the overall performance of a program, rather than just a specific method.

With this type of timing analysis, a whole program performance auditor would collect periodic timing measurements to gauge a program's overall performance. For example, in the case of a block-based compression program, timing measurements would be collected once for each block of data compressed, to evaluate the program's overall performance.

The challenge is determining when these timing measurements should be collected so they can be meaningfully compared. The timer becomes like a heart monitor for the program: it measures the program's overall performance for a well-defined unit of work. If an appropriate timing point can be found, the performance auditing methodology can be applied to monitor whole program performance, instead of just the performance of a single method.

This type of timing analysis will be very useful for implementing alternative forms of performance auditing, such as those described above. It may be possible to use some of the ideas of software phase markers, described in

Section II.C, to address this challenge. Software phase markers identify stable patterns in program behavior, and that is exactly what is needed to determine where timing measurements should be collected for whole program performance auditing. Timing measurements should be collected periodically as the program completes units of work, and it may be possible to use the software phase marker methodology to identify the appropriate code structure to instrument, in order to collect comparable whole program timing measurements.

Bibliography

[1] A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Shpeisman. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal*, 7(1):19–31, Feb. 2003.

[2] A.-R. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. *ACM SIGPLAN Notices*, 39(6):267–276, June 2004. In *Conference on Programming Language Design and Implementation (PLDI)*.

[3] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *The International Symposium on Code Generation and Optimization*, 2006.

[4] E. Albert. A transparent method for correlating profiles with source programs. In *2nd ACM Workshop on Feedback-Directed Optimization (FDO-2)*, Nov. 1999.

[5] A. W. Appel. Simple generational garbage collection and fast allocation. *Software—Practice and Experience*, 19(2):171–183, Feb. 1989.

[6] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, Feb. 2005. Special issue on Program Generation, Optimization, and Adaptation.

[7] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. *ACM SIGPLAN Notices*, 36(5):168–179, May 2001. In *Conference on Programming Language Design and Implementation (PLDI)*.

[8] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. *ACM SIGPLAN Notices*, 33(5):258–268,

May 1998. In *Conference on Programming Language Design and Implementation (PLDI)*.

[9] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *33rd International Symposium on Microarchitecture*, pages 245–257, 2000.

[10] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W. W. Hwu. Vacuum packing: Extracting hardware-detected program phases for post-link optimization. In *35th International Symposium on Microarchitecture*, Dec. 2002.

[11] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *1997 International Conference on Supercomputing*, pages 340–347, 1997.

[12] J. Cavazos and J. E. B. Moss. Inducing heuristics to decide whether to schedule. *ACM SIGPLAN Notices*, 39(6):183–194, June 2004. In *Conference on Programming Language Design and Implementation (PLDI)*.

[13] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *The International Symposium on Code Generation and Optimization*, Mar. 2005.

[14] H. Chen, J. Lu, W.-C. Hsu, and P.-C. Yew. Continuous adaptive object-code re-optimization framework. In *Ninth Asia-Pacific Computer Systems Architecture*, Sept. 2004.

[15] B. Childers, J. Davidson, and M. L. Soffa. Continuous compilation: A new approach to aggressive and adaptive code transformation. In *International Symposium on Parallel and Distributed Processing Symposium*, Apr. 2003.

[16] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. *ACM SIGPLAN Notices*, 37(5):199–209, May 2002. In *Conference on Programming Language Design and Implementation (PLDI)*.

[17] A. Cohen and R. D. Ryan. *Wavelets and Multiscale Signal Processing*. Chapman & Hall, 1995.

[18] http://www-plan.cs.colorado.edu/henkel/projects/colorado_bench.

[19] K. Cooper, P. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, May 1999.

[20] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, May 2002.

[21] http://pag.csail.mit.edu/daikon.

[22] J. Dean and C. Chambers. Towards better inlining decisions using inlining trials. In *LISP and Functional Programming*, pages 273–282, 1994.

[23] P. Denning and S. C. Schwartz. Properties of the working-set model. *Communications of the ACM*, 15(3):191–198, Mar. 1972.

[24] A. Dhodapkar and J. Smith. Comparing program phase detection techniques. In *36th International Symposium on Microarchitecture*, Dec. 2003.

[25] A. Dhodapkar and J. E. Smith. Dynamic microarchitecture adaptation via co-designed virtual machines. In *International Solid State Circuits Conference*, Feb. 2002.

[26] A. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *29th Annual International Symposium on Computer Architecture*, May 2002.

[27] P. C. Diniz and M. C. Rinard. Dynamic feedback: An effective technique for adaptive computing. *ACM SIGPLAN Notices*, 32(5):71–84, May 1997. In *Conference on Programming Language Design and Implementation (PLDI)*.

[28] J. Engblom, A. Ermedahl, and P. Altenbernd. Facilitating worst-case execution time analysis for optimized code. In *EuroMicro Workshop on Real-Time Systems*, June 1998.

[29] S. J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *The International Symposium on Code Generation and Optimization with Special Emphasis on Feedback-Directed and Runtime Optimization*, pages 241–252, 2003.

[30] M. Frigo. A fast Fourier transform compiler. *ACM SIGPLAN Notices*, 34(5):169–180, May 1999. In *Conference on Programming Language Design and Implementation (PLDI)*.

[31] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *1998 IEEE International Conference on Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[32] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *Proceedings of the 1st International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2005)*, number 3793 in LNCS, pages 29–46. Springer Verlag, November 2005.

[33] A. Georges, D. Buytaert, L. Eeckhout, and K. D. Bosschere. Method-level phase behavior in Java workloads. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 2004.

[34] N. Grcevski, A. Kilstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *3rd Virtual Machine Research and Technology Symposium (VM)*, May 2004.

[35] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7, Sept. 2005.

[36] G. Hamerly, E. Perelman, J. Lau, T. Sherwood, and B. Calder. Using machine learning to guide architecture simulation. *Journal of Machine Learning Research*, 7:343–378, 2006.

[37] M. Hind, V. Rajan, and P. Sweeney. Phase shift detection: A problem classification. Technical report, IBM, Aug. 2003.

[38] M. Hirzel, A. Diwan, and M. Hind. Pointer analysis in the pressence of dynamic class loading. In *18th European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 96–122, June 2004.

[39] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *5th European Conference on Object-Oriented Programming (ECOOP)*, volume 512 of *LNCS*, pages 21–38, July 1991.

[40] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. *ACM SIGPLAN Notices*, 27(7):32–43, July 1992. In *Conference on Programming Language Design and Implementation (PLDI)*.

[41] M. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: Application to energy reduction. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[42] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving program locality. *ACM SIGPLAN Notices*, 39(10):69–80, Oct. 2004. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[43] E.-J. Im and K. Yelick. Optimizing sparse matrix-vector multiplication for register reuse in SPARSITY. In *International Conference on Computational Science*, May 2001.

[44] E.-J. Im, K. Yelick, and R. Vuduc. SPARSITY: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1), Jan. 2004.

[45] C. Isci and M. Martonosi. Identifying program power phase behavior using power vectors. In *Workshop on Workload Characterization*, Sept. 2003.

[46] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *36th International Symposium on Microarchitecture*, Dec. 2003.

[47] K. Ishizaki, M. Takeuchi, K. Kawachiya, T. Suganuma, O. Gohda, T. Inagaki, A. Koseki, K. Ogata, M. Kawahito, T. Yasue, T. Ogasawara, T. Onodera, H. Komatsu, and T. Nakatani. Effectiveness of cross-platform optimizations for a Java just-in-time compiler. *ACM SIGPLAN Notices*, 38(11):187–204, Nov. 2003.

[48] A. Jaleel, R. S. Cohn, C. Luk, and B. Jacob. Cmp$im: A binary instrumentation approach to modeling memory behavior of workloads on cmps. Technical Report UMDSCA-2006-01, Intel, Jan. 2006.

[49] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *International Workshop on Mining Software Repositories (MSR)*, May 2006.

[50] T. Kistler and M. Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *Transactions on Programming Languages and Systems (TOPLAS)*, 22(3):490–505, 2000.

[51] T. P. Kistler and M. Franz. Continuous program optimization: Design and evaluation. *IEEE Transactions on Computers*, 50(6):549–566, June 2001.

[52] P. Kulkarni, D. Whalley, G. Tyson, and J. Davidson. Exhaustive optimization phase order space exploration. In *The International Symposium on Code Generation and Optimization*, Mar. 2006.

[53] P. A. Kulkarni, S. R. Hines, D. B. Whalley, J. D. Hiser, J. W. Davidson, and D. L. Jones. Fast and efficient searches for effective optimization phase sequences. *ACM Transactions on Architecture and Code Optimization*, 2(2):165–198, June 2005.

[54] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Processor power reduction via single-ISA heterogeneous multi-core architectures. *Computer Architecture Letters*, 2, Apr. 2003.

[55] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2005.

[56] J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. In *IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2004.

[57] J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, Jan. 2005.

[58] X. Li, M. J. Garzarán, and D. Padua. Optimizing sorting with genetic algorithms. In *The International Symposium on Code Generation and Optimization*, pages 99–110, Mar. 2005.

[59] W. Liu and M. Huang. EXPERT: Expedited simulation exploiting program behavior repetition. In *International Conference on Supercomputing*, June 2004.

[60] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.

[61] J. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. LeCam and J. Neyman, editors, *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, Berkeley, CA, 1967. University of California Press.

[62] D. Maier, P. Ramarao, M. Stoodley, and V. Sundaresan. Experiences with multithreading and dynamic class loading in a Java just-in-time compiler. In *The International Symposium on Code Generation and Optimization*, Mar. 2006.

[63] M. Merten, A. Trick, R. Barnes, E. Nystrom, C. George, J. Gyllenhaal, and W. mei W. Hwu. An architectural framework for run-time optimization. *IEEE Transactions on Computers*, 50(6):567–589, June 2001.

[64] R. Muth, S. Debray, S. Watterson, and K. D. Bosschere. alto : A link-time optimizer for the DEC Alpha. In *Software—Practice and Experience*, pages 31:67–101, Jan. 2001.

[65] P. Nagpurkar, M. Hind, C. Krintz, P. F. Sweeney, and V. Rajan. Online phase detection algorithms. In *The International Symposium on Code Generation and Optimization*, Mar. 2006.

[66] P. Nagpurkar, C. Krintz, and T. Sherwood. Phase-aware remote profiling. In *International Symposium on Code Generation and Optimization*, Mar. 2005.

[67] N. Nethercote, D. Burger, and K. S. McKinley. Self-evaluating compilation applied to loop unrolling. Technical Report TR-06-12, The University of Texas at Austin, Feb. 2006.

[68] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. In *The Computer Journal vol. 40*, 1997.

[69] M. Paleczny, C. Vick, and C. Click. The Java Hotspot server compiler. In *Java Virtual Machine Research and Technology Symposium (JVM)*, pages 1–12, Apr. 2001.

[70] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *37th International Symposium on Microarchitecture*, Dec. 2004.

[71] D. Pelleg and A. Moore. X-means: Extending K-means with efficient estimation of the number of clusters. In *Proceedings of the 17th International Conf. on Machine Learning*, pages 727–734. Morgan Kaufmann, San Francisco, CA, 2000.

[72] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2003.

[73] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2), 2005. Special issue on Program Generation, Optimization, and Adaptation.

[74] R. M. Rabbah and K. V. Palem. Data remapping for design space optimization of embedded memory systems. *ACM Transactions on Embedded Computing Systems*, 2(2):1–32, May 2003.

[75] R. Saavedra and D. Park. Improving the effectiveness of software prefetching with adaptive execution. In *Conference on Parallel Architectures and Compilation Techniques*, 1996.

[76] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[77] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.

[78] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[79] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[80] http://www.sable.mcgill.ca/software/#soot.

[81] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[82] A. Srivastava and D. W. Wall. A practical system for intermodule code optimizations at link-time. *Journal of Programming Languages*, Mar. 1993.

[83] Standard Performance Evaluation Corporation. SPECjbb2000 Java Business Benchmark. http://www.spec.org/jbb2000.

[84] Standard Performance Evaluation Corporation. SPECjvm98 Benchmarks. http://www.spec.org/jvm98.

[85] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *The International Symposium on Code Generation and Optimization*, pages 123–134, 2005.

[86] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. *ACM SIGPLAN Notices*, 36(11):180–195, Nov. 2001. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[87] The DaCapo Project. DaCapo Benchmark Suite, version beta051009. http://www-ali.cs.umass.edu/DaCapo/gcbm.html.

[88] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 277–288, 2005.

[89] S. Triantafyllis, M. Vachharajani, and D. August. Compiler optimization-space exploration. *Journal of Instruction-Level Parallelism*, 7:1–25, Jan. 2005.

[90] M. J. Voss and R. Eigemann. High-level adaptive program optimization with ADAPT. *ACM SIGPLAN Notices*, 36(7):93–102, July 2001.

[91] Z. Wang, K. Pierce, and S. McFarling. BMAT - a binary matching tool for stale profile propagation. *Journal of Instruction-Level Parallelism*, Apr. 2000.

[92] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[93] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[94] http://xml.apache.org/xerces2-j/index.html.

[95] J. Yi, S. Kodakara, R. Sendag, D. Lilja, and D. Hawkins. Characterizing and comparing prevailing simulation techniques. In *International Symposium on High-Performance Computer Architecture*, Feb 2005.

[96] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2003.

[97] X. Zhang and R. Gupta. Matching execution histories of program versions. In *International Symposium on Foundations of Software Engineering (FSE)*, Sept. 2005.