UNIVERSITY OF CALIFORNIA
Santa Barbara

Analysis, Detection, and Exploitation of Phase Behavior in Java Programs

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

Doctor of Philosophy

in

Computer Science

by

Priya Nagpurkar

Committee in Charge:

Professor Chandra Krintz, Chair

Professor Timothy Sherwood

Professor Tobias Hollerer

September 2007

The Dissertation of
Priya Nagpurkar is approved:

Professor Timothy Sherwood

Professor Tobias Hollerer

Professor Chandra Krintz, Committee Chairperson

August 2007

Analysis, Detection, and Exploitation of Phase Behavior in Java Programs

Copyright © 2007

by

Priya Nagpurkar

Dedication and Gratitude

I dedicate this dissertation to my parents, Ashok and Rekha Nagpurkar, who never cease to amaze me with the perfect balance between independence and guidance that they have always struck in influencing my life. Their faith, encouragement and support, further reinforced by that of my brother, Ravindra, has played an important role in my decision to pursue, and in successfully completing this work.

My advisor, Chandra Krintz, played an equally important role, by recognizing and bringing out my potential for research. Her energy and enthusiasm for both research, and teaching were, and will continue to be, a constant inspiration. She has been a model *guru*, by being a good friend, philosopher, and guide. Many thanks also to Tim Sherwood, and Tobias Hoellerer for their guidance as members of my dissertation committee, and to our collaborators from the I.B.M. T.J. Watson Research Center for their valuable advice. Michael Hind, Peter Sweeney, Trey Cain, Mauricio Serrano, and Jong-Deok Choi were all excellent mentors.

I would like to express deep gratitude towards all my friends, old and new. Special thanks to my close friends, Rekha, Shilpa, Puja, Nicole, and Martina for being my extremely reliable support structure in times of need; to Lingli and Ye, for being great colleagues and neighbors; to Selim for great evenings in the climbing gym; and to Hussam for all the cups of tea; These and other members of the RACE lab made it a great place to work or to hang out. Visits to the CS office were always pleasant, thanks to Amanda, Greta, Julia, Beejay, and the rest of our very cheerful office staff. Finally, playing ultimate with the CS team was always something to look forward to – thanks to all of you on the ultimate team for your camaraderie!

Acknowledgements

The text of Chapter 3 is in part a reprint of the material as it appears in the proceedings of Elsevier Science of Computer Programming – Special Issue on Principles Practices and Programming in Java, Vol. 59. The dissertation author was the primary researcher and author and the co-author listed on this publication ([82]) directed and supervised the research which forms the basis for Chapter 3.

The text of Chapter 4 is in part a reprint of the material as it appears in the proceedings of the Fourth Annual International Symposium on Code Generation and Optimization (CGO). The dissertation author was the primary researcher and the co-authors listed on this publication ([80]) directed and supervised the research which forms the basis for Chapter 4.

The text of Chapter 6 is in part a reprint of the material as it appears in the proceedings of ACM Transactions on Architecture and Code Optimization (TACO),Vol. 3, Number 1. The dissertation author was the primary researcher and author with significant contribution from one of the co-authors, Hussam Mousa. The remaining co-authors listed on this publication ([84]) directed and supervised the research which forms the basis for Chapter 6.

The text of Chapter 7 is in part a reprint of the material as it appears in the proceedings of the Sixteenth International Conference on Parallel Architectures and Compilation Techniques (PACT). The dissertation author was the primary researcher and author and the co-authors listed on this publication ([79]) directed and supervised the research which forms the basis for Chapter 7.

Curriculum Vitæ

Priya Nagpurkar

Education

2007	**Doctor of Philosophy in Computer Science,** University of California, Santa Barbara.
2007	**Master of Science in Computer Science,** University of California, Santa Barbara.
2001	**Bachelor of Engineering in Computer Engineering,** Pune University.

Professional Experience

2006	**Summer Intern,** I.B.M. T.J. Watson Research Center.
2003 – 2007	**Graduate Research Assistant,** University of California, Santa Barbara.
2001 – 2003	**Graduate Teaching Assistant,** University of California, Santa Barbara.
2000	**Intern,** VERITAS Software India Ltd. (now Symantec), Pune, India.

Professional Activities

2007	**Program Committee Member**, International Conference on Principles and Practices of Programming in Java
2006	**Program Committee Member**, UCSB Graduate Student Research Conference
2006	**Submissions Chair**, International Conference on Principles and Practices of Programming in Java
2006-2007	**Graduate Student Representative**, Colloquium Committee
2005-2006	**Graduate Student Representative**, Graduate Admissions Committee

Publications

Priya Nagpurkar, Harold W. Cain, Mauricio Serrano, Jong-Deok Choi and Chandra Krintz: "Call-chain Software Instruction Prefetching in J2EE Server Applications," *In the Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT07)*

Lingli Zhang, Chandra Krintz, and Priya Nagpurkar: "Language and Virtual Machine Support for Efficient Fine-Grained Futures in Java," *In the Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT07)*

Lingli Zhang, Chandra Krintz, and Priya Nagpurkar: "Supporting Exception Handling for Futures in Java," *In the Proceedings of the International Conference on the Principles and Practice on Programming in Java (PPPJ07)*

Priya Nagpurkar, Harold W. Cain, Mauricio Serrano, Jong-Deok Choi and Chandra Krintz: "A Study of Instruction Cache Performance and the Potential for Instruction Prefetching in J2EE Server Applications," *Tenth Workshop on Computer Architecture Evaluation Using Commercial Workloads (CAECW-10)*

Priya Nagpurkar, Michael Hind, Chandra Krintz, Peter Sweeney, and V.T. Rajan: "Online Phase Detection Algorithms," *In the Proceedings of the International Symposium on Code Generation and Optimization (CGO06)*

Priya Nagpurkar, Chandra Krintz, and Timothy Sherwood: "Phase-aware Remote Profiling," *In the Proceedings of the International Symposium on Code Generation and Optimization (CGO05)*

Priya Nagpurkar and Chandra Krintz: "Visualization and Analysis of Phased Behavior in Java Programs," *In the Proceedings of the International Conference on the Principles and Practice of Programming in Java (PPPJ04)*

Priya Nagpurkar, Hussam Mousa, Chandra Krintz, and Timothy Sherwood: "Efficient Remote Profiling for Resource-Constrained Devices," *In the Proceedings of ACM Transactions on Architecture and Code Optimization (TACO),Vol. 3, Number 1, March, 2006, pages 1-32*

Priya Nagpurkar, and Chandra Krintz: "Phase-Based Visualization and Analysis of Java Programs," *In the Proceedings of Elsevier Science of Computer Programming – Special Issue on Principles Practices and Programming in Java, Vol. 59, Number 1-2, January 2006, pages 64-81*

Selim Gurun, Priya Nagpurkar and Ben Zhao: "Energy Consumption and Conservation in Mobile Peer-to-peer Systems," *The first International Workshop on Decentralized Resource Sharing in Mobile Computing and Networking (MobiShare06)*

Field of Study: Computer Science

Abstract

Analysis, Detection, and Exploitation of Phase Behavior in Java Programs

Priya Nagpurkar

The Java programming language offers developers many productivity enhancing features, including high-level abstractions, extensive libraries, architecture-independent execution, and type safety. These features are enabled by an intelligent execution environment that, incrementally and dynamically, compiles and executes compact representations of Java programs encoded for a virtual machine. While this necessarily adds overhead, the ability to compile (and recompile) code at runtime also enables the execution environment to perform dynamic, performance-enhancing optimizations based on the runtime behavior of the executing program. There are three primary steps in developing effective adaptive optimizations for these systems: (1) Development of a thorough analysis, understanding, and characterization of the performance of Java programs; (2) Extracting accurate data from programs efficiently at runtime; and (3) Guiding optimizations using feedback from the extracted performance data.

We address each of these steps in our research by focusing on techniques that capture and exploit the repeating patterns in program behavior (phases) within virtual execution environments, and in particular, those for Java programs. This dissertation can

be decomposed into two foci: phase analysis and detection tools and techniques and phase-aware techniques for efficient program analysis and optimization. We first study the time varying behavior of Java programs, show that Java programs do exhibit phase behavior, and present tools to extract and analyze this phase behavior. We then investigate the problem of accurate online phase detection for Java programs, within a Java virtual machine, the parameters that impact doing so effectively, and evaluate numerous online phase detectors. Finally we demonstrate the potential of phase-based optimizations by designing and evaluating two phase-based runtime techniques. The first technique is an accurate, low-overhead profiling scheme for resource-constrained devices that uses phases to drive when to sample the execution of a program. The second technique is a software instruction prefetching mechanism that uses method-level phase behavior to identify, predict, and prefetch methods that incur a large number of instruction cache misses for emerging Java workloads like database- and application servers. These two techniques span two extremes of execution environments used for Java applications: software for resource-constrained devices at the low end and application servers at the high-end.

Contents

Acknowledgements **v**

Curriculum Vitæ **vi**

Abstract **ix**

List of Figures **xiv**

List of Tables **xvi**

1 Introduction **1**

2 Background **6**
2.1 Phase Characterization, Detection, and Prediction Techniques 7
2.2 Applications of Phase Analysis 16
2.3 Dynamic Compilation and Adaptive Optimization in Java 20

3 Phase Behavior in Java Programs **23**
3.1 Phase Analysis Framework 25
3.1.1 Data Generation 26
3.1.2 Data Processing 29
3.2 Phase Analysis Toolkit 31
3.2.1 Phase Visualizer 31
3.2.2 Phase Finder 33
3.2.3 Phase Analyzer and Code Extractor 36
3.3 Analysis 37
3.3.1 Visual Analysis 38
3.3.2 Efficient Identification of Optimization Opportunities 44

3.3.3 Cross-Input Analysis 49
3.3.4 Other Opportunities for Exploiting Phase Behavior 51
3.4 Summary 53

4 Phase Detection for Java Programs 55
4.1 Online Phase Detection Framework 56
4.1.1 Window Policy 61
4.1.2 Model Policy 63
4.1.3 Analyzer Policy 64
4.2 Evaluating Phase Detectors 65
4.2.1 Phase Detection Baseline 66
4.2.2 Accuracy Scoring Metric 69
4.3 Analysis 71
4.3.1 Methodology 73
4.3.2 Window Policy 77
4.3.3 Model Policy 81
4.3.4 Analyzer Policy 84
4.3.5 Additional Analysis 85
4.4 Summary 90

5 Phase-based Runtime Techniques 91
5.1 Phase-aware Profiling 92
5.2 Instruction Prefetching 94

6 Phase-aware Remote Profiling 97
6.1 Phase-aware Sampling: Deciding When to Sample 101
6.2 Profiling Support for Toggling Profile Collection 106
6.2.1 Dynamic Instruction Stream Editing (DISE) 108
6.2.2 Hybrid Profiling Support using DISE 110
6.3 Evaluation 118
6.3.1 Phase-aware Profiling for General-purpose Programs 119
6.3.2 Phase-aware Profiling for Embedded Devices 133
6.4 Extending Phase-aware Profiling to Multiple Users 139
6.5 Related Work 143
6.5.1 Efficient Profiling 143
6.5.2 Monitoring Program Behavior for Bug Isolation and Test Coverage 145
6.6 Summary 147

7 Phase-based Instruction Prefetching **149**
7.1 Characterization of Instruction Cache Behavior 150
7.1.1 Methodology 151
7.1.2 Stall Cycles 152
7.1.3 Method-level Analysis 154
7.2 Method-level Phase Behavior 158
7.3 Call-chain Instruction Prefetching 161
7.3.1 Design and Implementation 161
7.3.2 Experimental Methodology 165
7.3.3 Evaluation 167
7.3.4 Discussion: Potential Improvements 174
7.4 Related Work 175
7.5 Summary 178

8 Conclusion **179**
8.1 Dissertation Summary 180
8.2 Impact and Future Directions 187

Bibliography **191**

List of Figures

1.1 Programming Language Usage Trends. 2

3.1 JVM phase analysis framework and toolkit 25
3.2 Architecture of the data generation framework 27
3.3 Phase Visualizer 32
3.4 Similarity graph for Mtrt input size 10. 39
3.5 Phases for Mtrt with similarity threshold 0.8 40
3.6 Similarity graphs for the SpecJVM benchmarks with input size 100 . . 42
3.7 Similarity graphs for the SpecJVM benchmarks with input size 10 . . 43
3.8 Code extracted using the phase framework and toolkit 47
3.9 Hand-optimized basic block exposed via phase analysis 48
3.10 Analysis of cross-input similarity 50

4.1 Illustrated view of the phase detection framework 57
4.2 Basic operation of the phase detection framework 58
4.3 Online phase detection framework 62
4.4 Evaluation of skip factor and Fixed versus Adaptive windowing 80
4.5 Unweighted vs. Weighted similarity models 82
4.6 Constant vs. Adaptive window policy 84
4.7 Slide vs. Move resizing 88
4.8 Accurate detection of phase boundaries 89

6.1 Run-time power usage 98
6.2 System overview 100
6.3 Overview of the phase-aware profiling scheme 102
6.4 The Hybrid Profiling Support (HPS) system 107
6.5 HPS extensions to DISE 112
6.6 HPS pattern and replacement specification grammar 113

6.7 Pattern and replacement productions for different profile types 118
6.8 DISE vs. HPS for performance sampling 122
6.9 Evaluation of representative selection policies 125
6.10 Average error in code region profiling 127
6.11 Efficacy across profile types 130
6.12 Evaluation of phase-aware sampling using the StrongARM environment and benchmarks 136
6.13 Distributed profiling across multiple executions 140

7.1 Commit Stall Cycle Categorization 152
7.2 Icache misses per 100 committed instructions 153
7.3 Per-method Contribution to Total icache Misses (cumulative distribution) 154
7.4 Method-level phases in WebSphere (running specjAppServer2001) . . 157
7.5 Correlation between method-level phases and phases in icache misses. 159
7.6 Overview of phase-based prefetching in a JVM. 160
7.7 Example call chain 162
7.8 Trace-based Analysis Methodology 166
7.9 Prefetch accuracy 170
7.10 Effect of miss distance on coverage and interference 172

List of Tables

3.1 Description of the benchmarks used. 38

4.1 Benchmark Characteristics 72
4.2 Window size comparison 74

6.1 Select benchmark statistics relevant to the profiles collected 120
6.2 Sampling overhead at 5% error 132
6.3 StrongARM methodology 134

7.1 Prefetch Target Characteristics. 156
7.2 Coverage achieved 171
7.3 Improvement in IPC 173

Chapter 1

Introduction

The greatest happiness for the thinking person is to have explored the explorable and to venerate in equanimity that which cannot be explored.

Johann Wolfgang von Goethe (1749-1832)

The Java programming language, and similarly C# and the Microsoft .Net languages, offer many benefits to programmers such as portability, programmer productivity through high-level abstractions and extensive libraries, type and memory safety, and dynamic loading. These features make it easier, not only to develop software, but also to debug and maintain it. As a result, these languages are very popular with software developers. Java, in particular, has seen tremendous growth since its inception, a little over a decade ago, and the trend is predicted to continue [27]. It is estimated that Java today drives a $100 billion a year software industry, and is deployed on a wide variety of devices, including millions of desktops, billions of embedded devices (from smart phones to car navigation systems), and enterprise servers [101, 1].

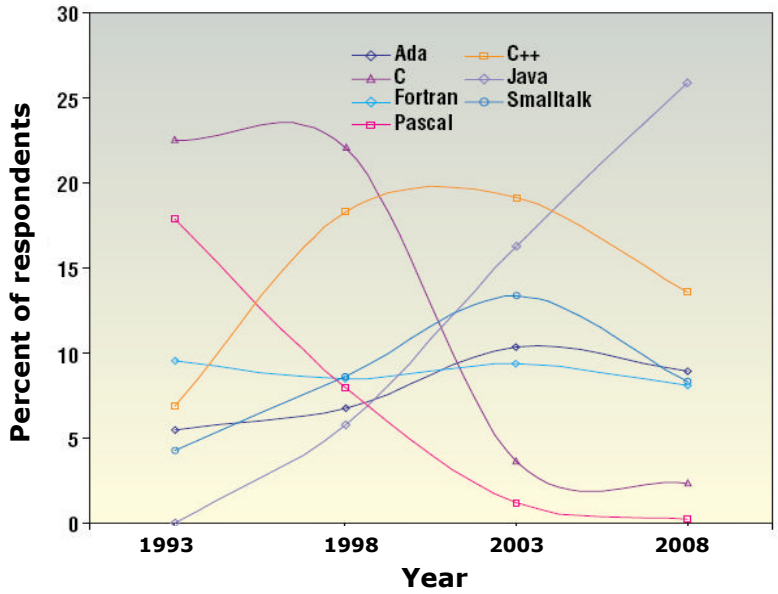


Figure 1.1: Programming Language Usage Trends. This figure from [27] uses historic data, gathered from real users, as well as prediction based on this data to show the long term trend in programming language popularity.

To enable these features, especially portability (the write-once, run-anywhere model), these programs are compiled into an architecture-independent intermediate format and executed within a virtual execution environment on the target host. The execution environment, a Java virtual machine or .Net runtime, implements a compilation system that converts the intermediate code to the native format of the underlying machine. This dynamic compilation necessarily introduces runtime overhead, but at the same time also exposes opportunities for adaptation – optimizations that we can customize according to the behavior of the executing program. State of the art Java Virtual Machines (JVMs) [71, 28, 102, 5, 53] employ adaptive optimization techniques based on

information gathered while the program is executing (feedback-directed optimization). Monitoring, analyzing, and predicting runtime program behavior are vital to feedback-directed optimization.

Recent research has shown, for non-Java programs, that program behavior varies over time, and exhibits repeating patterns [93, 41], and has focused on automatically characterizing this behavior [94, 95, 66, 35]. Phase analysis of programs is one such characterization, which isolates distinct behaviors in a program's execution by grouping periods of execution that are similar together in a *phase*. A program's execution can then be seen as a series of phases that might repeat themselves several times. Researchers have used phase behavior to improve program performance via hardware and software optimization [38, 96, 92, 72] and to reduce simulation time [94, 95] and generate cycle-close traces [88]. The ability to detect and predict phases at runtime has the potential of uncovering new opportunities in performing proactive adaptive optimizations for Java programs. With the ultimate aim of enabling better performance for Java programs, the thesis question that we explore is this work is:

> *How can we efficiently detect, track, predict, and exploit repeating patterns in program behavior (phases) in a Java virtual machine to facilitate runtime analysis and feedback-directed optimization in Java programs?*

To answer this question, we focus our efforts on phase characterization and detection in Java programs, and exploiting repetitions that phases manifest, using optimiza-

tion for a range of devices for which Java Virtual Machines are available. In particular, we

- investigate and develop offline mechanisms and tools that enable the characterization, visualization, and manipulation of phase behavior in Java programs,
- develop a modular, pluggable framework for implementing and investigating online phase detection algorithms within a Java Virtual Machine,
- devise and investigate a phase-aware approach to collecting accurate online execution profiles
- devise and investigate a prefetching scheme (a dynamic optimization) that exploits repeating patterns in Java server execution.

We take an empirical and implementation-oriented approach to developing phase detection and exploitation techniques in this thesis. We implement and empirically evaluate our techniques using a wide range of real programs and open-source Java virtual machine technologies. From this effort, we have produced a set of tools that significantly facilitate analysis of repeating patterns in the behavior of Java programs, and we have designed and evaluated novel techniques for exploiting phases to improve program profiling and execution performance.

The dissertation is organized as follows. We begin with a discussion of the relevant background in Chapter 2. This chapter includes techniques that we use in our work, as

well as the state of the art for phase analysis and its uses. We present our framework and toolkit for understanding and analyzing phase behavior in Java programs in Chapter 3, followed by our framework for the design and analysis of online phase detection algorithms within a JVM, in Chapter 4. Chapters 5, 6, and 7 focus on the use of phase behavior to enable two phase-based runtime techniques; Chapter 5 introduces these techniques and provides an overview, while the following Chapters provide details of each technique. We end with a summary of the dissertation and a discussion of future directions in Chapter 8.

Chapter 2

Background

The focus of this dissertation is understanding, analyzing, and exploiting the repeating patterns, or phases, in the time varying behavior of Java programs. Of particular interest to us, is the possibility of incorporating phase-awareness in virtual execution environments, like Java virtual machines, to drive adaptive, feedback-directed optimization of dynamically compiled programs.

Much research has already gone into the characterization, detection, prediction, and exploitation of phase behavior, especially in the area of computer architecture. In this Chapter, we present an overview of extant work on phase behavior, and also briefly describe the process of dynamic compilation for Java.

2.1 Phase Characterization, Detection, and Prediction Techniques

Program behavior has commonly been abstracted in the form of profiles gathered over the program's execution. Recently, there has been a lot of interest in studying program behavior during different parts of execution. Many researchers have observed, through detailed simulations and temporal profiles, that programs exhibit widely varying behavior during different parts of execution [93, 94, 41]. Program behavior, however, is not entirely random and often shows significant structure. In [94] and [41], the authors periodically gathered various hardware metrics, like IPC, cache misses, branch misprediction rate with the aim of studying low-level program behavior over time and finding any possible correlation between the metrics. Their findings indicate that, not only does program behavior change, but it also has periods of stable execution interspersed with transitions. During periods of stable execution, the architectural metrics measured are relatively stable. What is more interesting is the fact that the metrics transition in unison, though the nature of the transition might be different (that is the instruction cache miss rate might go up, whereas the IPC might go down). Recognizing the importance of automatically characterizing this behavior in order to exploit it for various purposes (like reducing simulation time, aiding prediction and proactive optimization), various techniques were developed at different levels in the system stack –

techniques using hardware performance counters, instruction working set signatures, basic blocks, and program structure (calls and loops) have been proposed to characterize, detect, and predict program phases. This section describes these techniques. Applications of phase behavior are discussed in Section 2.2.

Basic Block Distribution Analysis (BBDA)

Sherwood et al [94, 95] observe that program behavior is directly related to the code that is being executed. They therefore use profiles of a program's code structure, basic block frequencies, to capture phases and show that the periodicity captured by basic block profiles is shared by low-level architectural metrics. Basic block profiles gather frequencies of individual basic blocks based on how often they were executed. (a basic block is a straight-line piece of code with a single entry and exit point) Basic block profiles thus present an architecture-independent and relatively easy to generate method of capturing program behavior.

To extract time varying behavior, Sherwood et al. collect basic block footprints over fixed intervals, measured in terms of number of instructions executed, and compare footprints for different intervals. The footprint consists of a single n-dimensional vector with n counts of individual basic block frequencies. The dimensionality of the vector, n, is equal to the number of static basic blocks in the program. To accommodate the fact that larger basic blocks account for larger parts of the execution, the frequencies are

weighted by number of instructions in the basic block. Further, since absolute values are not necessary (proportions are), each value in the vector is normalized by dividing it with the sum of all elements in the vector.

To catch changes and repeating patterns, basic block vectors are compared using the basic block vector distance similarity measure. More specifically, the Manhattan distance is used. The Manhattan distance is calculated as the sum of absolute values of elementwise differences and represents the distance between the two compared vectors if the only paths you can take are parallel to the x and y axes. The calculation generates a value between 0 and 2, where 0 indicates complete similarity and 2 indicates complete dissimilarity. The authors use *clustering* to group similar intervals together to break the program's execution into phases. A phase thus consists of intervals of similar behavior, irrespective of temporal adjacency. Note here that the interval size controls the granularity at which phases can be detected.

Sherwood et al also introduce similarity matrices as a means of graphically representing phase behavior as captured by basic block distribution analysis [94]. A similarity matrix is the upper triangular N X N matrix, where a point (x,y) represents the similarity between intervals x and y, and N indicates the number of intervals of fixed duration generated.

Basic block distribution analysis thus presents an architecture-independent and effective way of capturing phase behavior. It is intended to be used offline and was first

introduced to find simulation points. In Chapter 3, we employ BBDA to analyze phase behavior in Java programs.

Following earlier work on time varying behavior of programs [93] and its characterization using basic block distribution analysis [94], Sherwood et al presented an efficient run-time phase tracking and prediction mechanism. Since tracking and comparing basic block vectors imposes a high overhead, an approximation of basic block vectors is used to track and detect changes in the *proportions* of code being executed. To approximate basic block vectors, branch PCs and the number of instructions executed between branches is captured. (Note that this can be done entirely in hardware and does not need any compiler support). As in BBDA, the program's execution is broken up into intervals of fixed size. During every interval, each branch PC is hashed and the corresponding counter is incremented with the number of instructions since the last branch. At the end of an interval, the table of counters is compressed to form a smaller footprint, which then represents program execution during the last interval. This footprint is compared with previously generated footprints and stored if it is found to be unique.(that is, if this interval is similar to a previous one, it belongs to that phase and its footprint need not be stored) Thus only one footprint per phase is stored.

To exploit repeating patterns in the program's behavior, the authors present a prediction scheme that predicts what phase the next interval will belong to. They observe that the set of phases seen recently and the duration of those phases are important in-

dicators of the next phase. The authors use a run length encoding Markov predictor to predict the phase of the next interval. The predictor uses a run-length encoded version of the history to index into a prediction table and is able to predict with a misprediction rate of 14% on average. Both tracking and prediction can be completely implemented in hardware and require less than 500 bytes of memory. While the authors used this dedicated, phase tracking and prediction hardware to evaluate dynamic architectural optimizations described briefly in Section 2.2, we used it in our efficient phase-based profiling scheme described in Chapter 6.

Instruction Working Set Signatures

Working sets were first introduced in the context of virtual memory pages [37] and represent the collection of pages that the process is working with. Dhodapkar et al. state that phase changes are manifestations of changes in working sets [38]. They keep track of the executing program using instruction working set signatures defined over a fixed window size. The working set signature is a lossy compressed representation of the working set, and consists of an n-bit vector formed by mapping working set elements into buckets using a random hash function. To detect a phase change, working set signatures defined over non-overlapping windows, are compared using the relative signature distance. A value of 1 indicates similarity and 0 indicates dissimilarity. A phase change is said to be detected if the distance is more than a certain threshold.

A phase is then defined as the maximum interval over which the working set remains constant (within the defined threshold). The size of the window controls the granularity at which phases can be detected.

Hardware Performance Counters

Most modern processor architectures include hardware performance counters that efficiently record various microarchitectural events, without interfering with the executing programs. Information gathered from these counters provides easy access to dynamic program execution characteristics.

Duesterwald et al. use hardware performance counter measurements to characterize the time varying behavior of programs, and to design online predictors [41]. They periodically gather various architectural metrics for the IBM Power3 and Power4 microarchitectures, using a timer-based interrupt to enable the recording of metrics after every 10ms. The characterization obtained is in the form of the low-level behavior of the program, as captured by hardware performance counters. Variability over time is measured in terms of average absolute distance between metric values at adjacent points in the time series. The authors make the following three observations:

- Program behavior varies significantly over time
- Program behavior is periodic

- Periodicity of metric behavior is shared across all metrics

As seen earlier in this Section, Sherwood et al also arrive at the same conclusions through detailed simulations, and design an architecture-independent way of characterizing program behavior. Duesterwald et al, on the other hand, use low-level metrics to characterize behavior, but exploit the fact that the periodicity is shared across metrics to design cross-metric predictors. The idea behind cross-metric predictors is that the history of a single metric can be used to predict one or more other metrics. The authors design and evaluate both statistical and table-based history predictors based on the accuracy with which values of metrics are predicted. Prediction accuracy is measured in terms of mean absolute prediction error, that is the mean absolute distance between the predicted and actual value. The conclusion drawn is that table-based predictors outperform statistical predictors.

Isci et al. combine several performance counter measurements to form a power vector that is a fine grained representation of the runtime power behavior of the processor [56, 57]. They then use a methodology similar to BBDA, using power vectors instead of basic block vectors, to identify power phases in programs. Power phases group together execution intervals with similar total, and ratio-wise power dissipation of processor components. In more recent work [55], the authors describe a runtime phase characterization and prediction methodology using only two hardware performance counters (since only two counters can be simultaneously monitored). The phase

classification is based on the ”memory-boundedness” of the program, which is computed by tracking the number of memory bus transactions and micro-ops retired in a particular execution interval. Thus a phase groups together intervals with similar memory-boundedness characteristics. Their prediction mechanism uses a global history based phase predictor, similar to a global history based branch predictor, to predict the next phase.

Program Structure

Recent approaches have used a coarser granularity of program elements than basic blocks, namely function and loop boundaries, to characterize phase behavior [66, 46]. Moreover, these approaches align phase boundaries with function or loop boundaries, rather than using an interval-based approach that techniques described above have used; in doing so, these approaches eliminate alignment problems associated with fixed length intervals, and simplify parameters associated with phase behavior characterization. Using high-level program elements also provides more semantic information that can simplify analysis and program-level optimization.

Lau et al. build a hierarchical function call- and loop graph from the execution trace of a program, and find function or loop boundaries that identify phases by tracking the execution variability along paths in the graph [66]. Execution variability is quantified by computing the variance in number of instructions executed (hierarchical) along a cer-

tain path across different invocations. Edges with variance lower than a certain threshold are chosen as phase identifiers. This is an offline approach, and the results of the analysis are used to insert phase markers into the program. The instrumented program can then exploit phase information as it executes. A somewhat similar approach was proposed by Shen at al. to identify and exploit phases in data reuse patterns [92]. The similarity is not in the use of program structure to perform phase characterization, but in the use of software phase markers. Phase characterization in this case is performed by using complex wavelet analysis to isolate data reuse patterns from a sampled data access trace.

Georges et al. study method-level behavior of Java programs [46]. They define methods, in which a substantial amount of time is spent, as interesting phases and gather metrics using hardware performance counters for these methods. They compute the coefficient of variance for each metric, to show that variance within a phase is low, and use the ANOVA technique to show that the variance across different phases is relatively higher. This, in turn, shows that method-level phases are correlated with low-level behavior.

2.2 Applications of Phase Analysis

In the previous section, we saw that it it possible to capture, characterize, and predict program phase behavior [94, 95, 96, 42, 92, 39, 81]. Moreover, existing systems use phase behavior to guide effective hardware reconfiguration [38, 96, 92], hardware-based value profiling [96], program and system analysis [74], efficient simulation [94], and cycle-close trace generation [88]. This section describes the prominent uses of phase behavior so far.

Architectural Simulation

One of the very first uses of phase behavior was in reducing the time and effort required for detailed and accurate architectural simulations. Sherwood et al. used BBDA based phase classification, described in the previous section, to find simulation points for a program. These simulation points correspond to different phases exhibited by the program, and as such mark distinct behaviors in the programs execution, which together generate an accurate representation of the whole program. The amount of detailed simulation required is far less, compared to simulating the whole program, since only sections of execution following the chosen simulation points need to be simulated. The authors report that using BBDA to choose a single simulation point reduces the average IPC (instructions per cycle) error to 18% as compared to 80% error obtained

for an arbitrarily chosen point (blind fast forwarding). This error is further reduced to 3% by choosing multiple simulation points.

Tuning Reconfigurable Hardware

For reconfigurable hardware components, like reconfigurable caches, one of multiple configurations can be chosen at runtime to optimize performance, either in terms of execution time or power consumption. Phase information can be used both in making reconfiguration decisions, and in choosing configurations.

Instruction working set analysis, described in the previous section, was used to enable dynamic reconfiguration of a multi-configuration instruction cache [38]. In addition to using working set signatures to track phase changes (and to trigger reconfiguration), the authors also used repeating working sets to reduce retuning time. They enhanced an existing dynamic reconfiguration algorithm [15] with working set analysis. The algorithm computes the relative signature difference with respect to the previous signature at the end of each window. On detecting a phase change, it waits till the phase stabilizes before retuning and installing an optimal configuration. Finding an optimal configuration is easy if the working set signature has been seen before, since a table of such signatures and optimal configurations is maintained.

Runtime phase detection coupled with prediction is a powerful combination that enables proactive optimization decisions. Sherwood et al. used their phase tracking and

prediction hardware, described earlier, to drive two reconfiguration decisions: setting the size and associativity of a resizable cache, and adjusting processor width (number of instructions entering the processor pipeline every cycle). Both optimizations aim to reduce energy consumption while still maintaining performance. On seeing a new phase, some time is spent in finding out whether a particular configuration will be beneficial during this phase, so that this configuration can be chosen when the predictor next predicts this phase. For example, in their evaluation, Sherwood et al. experiment with reducing the processor width for phases with low IPC (Instructions per cycle). The authors report that their adaptive, phase-guided cache resizing optimization yields results at least as good as, or better than, voltage scaling. For the processor width optimization, the authors report average energy savings of 19.6% at the cost of 4% slowdown. Isci et al. apply their runtime phase characterization and prediction methodology to dynamic voltage and frequency scaling. Different phases are mapped to different voltage and frequency levels, depending on how memory-bound they are, and depending on the potential for concurrent execution.

Dynamic Optimization Software Systems

Papers on runtime optimizers in execution environments [5, 53, 3] and binary translators [14, 43] have also discussed the benefit of considering phase behavior [29]. Arnold et al. mention the use of phase-shift detection to trigger re-gathering of profiles

that drive dynamic optimizations in the JikesRVM [11]. Dynamo [14] uses phase-shift detection in its code cache policy. However, none of these systems currently use the various characteristics of phase behavior, namely periodicity and repetition, to drive dynamic optimizations.

In [60], Kistler and Franz use online phase-shift detection to trigger re-optimization in their continuous optimizing system for Oberon System 3. Change in program behavior is detected by observing whether the footprint of the profile has changed significantly in the last two consecutive time intervals. They use a similarity measure based on the geometric angle between the two profile-vectors.

Lu et al. [72] employ online phase detection to drive data cache prefetching in a dynamic binary optimization system. Specifically, they compare the average PC address from the most recent 4K samples to a range of values, which is created from the average and standard deviation of the previous seven 4K samples of the PC address. If the new average is sufficiently outside this interval for two consecutive 4K sample windows, a phase has ended.

2.3 Dynamic Compilation and Adaptive Optimization in Java

To enable productivity enhancing features like portability, type- and memory-safety, Java programs are first compiled into an architecture- independent format, called bytecode, transferred to the target machine, and executed inside a virtual machine. The Virtual Machine often employs dynamic program analysis and optimization, in the process of converting architecture-independent bytecode into native code, to improve performance. This process takes place as the program is executing, and is called dynamic compilation. Although dynamic compilation imposes a runtime overhead, the dynamic compilation system can identify and implement profitable optimizations that cannot be determined clearly to be beneficial statically, using information gathered at run time. In particular, input-dependent behavior can be optimized after a program is initiated if the execution delay associated with running the optimizer can be amortized by faster overall execution. Due to the wide spread use of Java for Internet-computing applications and to the portability and dynamism enabled by the language, such techniques are becoming increasingly important to enable the high-performance that users have come to expect.

Java Virtual Machines usually employ a light-weight interpreter along with an optimizing compiler, which is invoked only for frequently executing portions of the pro-

gram, so as to ensure that optimization effort is invested profitably. The optimizing compiler often distributes optimizations across different optimization levels, which are progressively applied. For example, when the JVM learns that a method is hot (because it has been invoked repeatedly), it is optimized. If it continues to be hot, it is optimized further. Similarly, some JVMs can further exploit repetition in the behavioral patterns that they learn via profiling to specialize a method for a specific behavior [45, 106]. To make their optimization decisions, current adaptive techniques observe potentially optimizable behavior and then optimize under the assumption that the behavior will continue. That is, they optimize future execution based on past observation. Past observations are made based on profiles gathered by the runtime measurement system. Examples of profiles commonly gathered include: time spent in methods, edge invocation counts, method call-pair counts. Recent efforts have also focused on efficiently gathering more complex profiles like call-chains and execution paths [40, 10, 110] within Java Virtual Machines. Both, efficiently gathering information (profiling), and making use of it (optimization), are important aspects of feedback-directed optimization.

Phase behavior, if present in Java programs, has great potential for improving estimates about future program behavior, and in performing phase-aware, specializing optimizations. However, as seen earlier in this chapter, there has not been much research on the analysis, detection, and use of phase behavior in Java programs. As a result, no extant VM for Java uses phase behavior to drive adaptive optimization. In the

following chapters, we address each of these issues, with the aim of enabling phase-aware, adaptive optimizations for Java programs.

Chapter 3

Phase Behavior in Java Programs

To enable the study of phase behavior in Java programs, and the potential for phase-based adaptive optimizations, we developed a framework for automatic phase identification and analysis. Specifically, we implemented a set of tools (both JVM-internal and external) that can be used offline to collect and analyze dynamic phase behavior in Java programs. The framework unifies all of the steps necessary for dynamic phase analysis. To enable this, we couple existing techniques into a single system.

To perform phase analysis within a JVM context, an effective system must

- Collect time-varying behavior during program execution (profiling)
- Take snapshots of this behavior over time (interval collection)
- Compare intervals to identify similarities and form (possibly non-contiguous) phases from interval similarity data

- Provide a visualization capability to lend insight to researchers developing new optimization techniques

- Extract important code regions from phase (or interval) data, e.g., frequently executed instructions, basic blocks, methods, etc.

The framework we describe in this chapter unifies these functionalities into a single tool for adaptive optimization development. It simplifies dynamic analysis since both phases and their relative similarities are automatically identified. Such a framework is necessary to enable new adaptive optimizations not currently available and to understand how current optimizations may be improved.

To demonstrate our approach, we use the framework to visualize and analyze phase behavior in SpecJVM Java benchmark suite [98]. Based on the identified phases, we describe new optimization opportunities for Java programs. We also use the phase information to analyze program behavior across inputs, thereby improving the value of online and offline profiling techniques [64]. Finally, we show how using phase behavior reveals optimization opportunities that are not apparent without considering time-varying behavior. By unifying the necessary phase identification, analysis, and visualization capabilities, our framework enables these results.

In the following section, we describe our JVM framework for the analysis of dynamic phase behavior in Java programs. In Sections 3.1 and 3.2 , we provide imple-

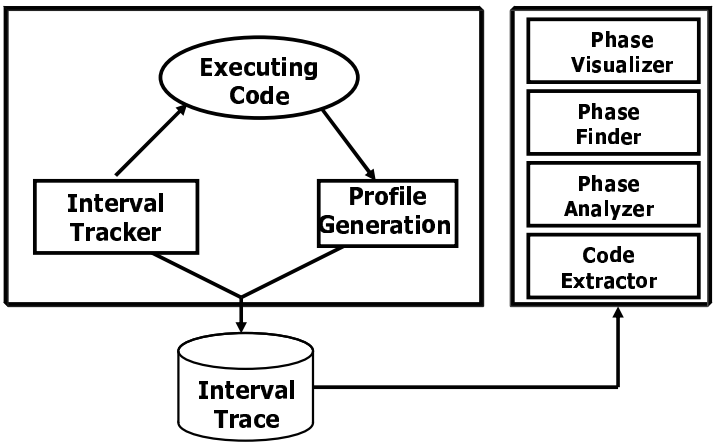


Figure 3.1: JVM framework and toolkit for the analysis of phase behavior in Java programs.

mentation details about the framework and the phase analysis toolkit. In Section 3.3, we discuss how the framework and toolkit can be employed to simplify analysis, to enable optimization, and to evaluate the efficacy of using phase behavior to guide adaptive optimization.

3.1 Phase Analysis Framework

We designed and implemented the phase analysis framework shown in Figure 3.1, to enable analysis and visualization of phase behavior in Java programs, and to facilitate optimization development. The framework consists of a Data Generation Framework and a Data Processing Toolkit described next.

3.1.1 Data Generation

The data generation framework is part of the Java Virtual Machine and generates a temporal trace of the Java application running in the virtual machine. This section describes the data generation framework and its implementation.

Overview

The goal of our work is to characterize the time-varying behavior of programs in terms of phases. As such, we must relate two points in a program's execution with respect to the execution characteristics of each. To enable this, we use basic block vectors [94] to capture a program execution characteristics. A basic block vector is an array of counters; its length is equal to the number of static basic blocks in the program. Each time a basic block is executed, its entry in the vector is incremented. As such, basic block vectors efficiently and effectively capture the behavior of executing code.

To study phase behavior, we need *snapshots* of basic block vectors, at different points during the program's execution. To enable this, we break the program's execution up into intervals of fixed length and dump the basic block vectors to disk at every interval. This snapshot represents the execution behavior of the program during that interval. The data generation framework produces an interval trace basic block vectors for every interval of execution.

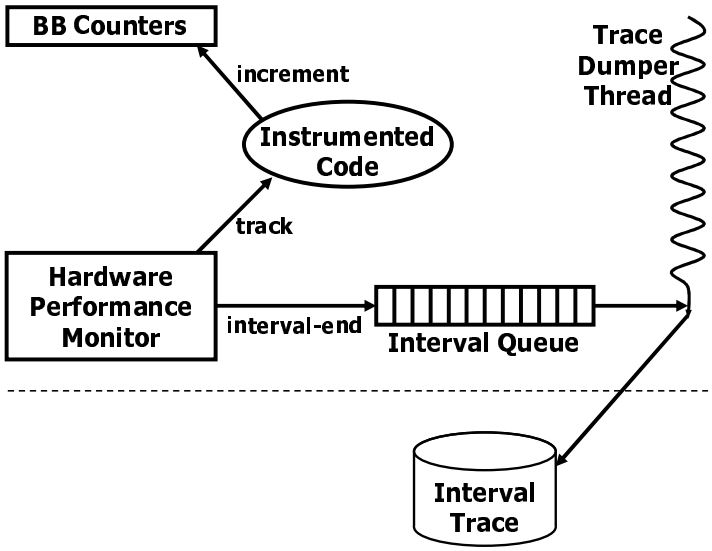


Figure 3.2: Architecture of the data generation framework. The instrumented application updates counters for the current snapshot (profile) interval in an interval queue. When the hardware performance monitors (HPMs) indicate that the interval granularity has been reached, a background dumper thread is signalled. The dumper thread expels all past snapshots to an interval trace on disk. The HPMs collect hardware data on a per-thread data; we monitor only application threads.

Implementation

We implemented the data generation framework within the optimizing compiler system in JikesRVM, an adaptive optimization Java Virtual Machine from IBM T.J. Watson Research Center [9]. As shown in Figure 3.2, we extended this VM to generate per-interval profiles, track intervals, and expel an execution snapshot at regular intervals.

To generate the profile, we extended JikesRVM to insert instrumentation into every basic block. We instrument only application methods and library methods used by the application. However, this can be easily extended to include VM methods (since this

VM is written in Java). The instrumentation consists of a counter identified by a unique identifier consisting of a method and basic block id [1]. To ensure that every method is instrumented, we force the JikesRVM to use the optimizing compiler (with the highest level of optimization – L2) to compile every method. The counters then collect basic block frequencies as the instrumented code executes. The basic block vector consisting of frequencies of all instrumented basic blocks forms the profile.

Periodically, we output a snapshot of the basic block counters into an interval trace on disk. The period with which we perform the dump is dictated by a granularity parameter, i.e., interval size. We identify interval boundaries (in terms of number of dynamic instructions specified by the interval size) using Hardware Performance Monitors (HPMs). Our version of JikesRVM is equipped with an interface to Papi [22] a C-interface to the underlying Pentium HPM registers. JikesRVM communicates with Papi via an interface developed by Thu Nguyen and the PANIC Laboratory at Rutgers University group at Rutgers University [85]. This interface enables us to collect *per-thread* HPM data to track hardware events for each thread in the system. As such, we monitor only those HPM events for application threads.

Each time a (timer-based) thread switch occurs for an application thread, compute the total number of instructions (reported by the HPMs) executed by all application threads. When this value exceeds the specified interval granularity, the counter values

[1] We must include the method identifier since basic block ids are not unique across methods.

are placed in an *Interval Queue*. When the interval queue exceeds a certain threshold, the application thread signals a background trace dumper thread. This thread copies all past intervals (snapshots) from the queue to a disk file. Once the dumper thread is scheduled, it remains scheduled until it completes the dump of all past intervals.

We currently use the framework offline since the overhead required for block-level profiling and trace dumping is large. For each of the benchmarks we evaluated using the framework, we experienced a 10-20x slowdown in execution performance, depending on interval size. However, we can extend the framework to reduce this overhead. For example, we can reduce profiling overhead by instrumenting only branching blocks [96, 50] or by using efficient sample-based profiling [11]. In addition, for an online system, we can avoid trace dumping and instead efficiently encode and cache phase information in a way that is similar to that used in online hardware-based systems [96]. We plan to evaluate the tradeoffs of such mechanisms as part of future work.

3.1.2 Data Processing

Given profiles for every interval in the program's execution, generated by the data generation framework, we need to compare them to see how execution in one interval relates to that in another. The similarity measure we use to compare two basic block vectors is the vector distance. Basic block vectors can be considered to be points in d-dimensional space, where d is the length of the vector. The more similar the basic

block vectors, the smaller the distance is between them. Basic block vectors that are similar represent periods of repeated execution behavior, as such, we group them into the same phase. Thus, a phase consists of all snapshots during program execution with similar execution characteristics. Note also that a phase does not necessarily consist of temporally contiguous intervals of execution.

To generate phases from the data produced by the data generation framework, we classify each interval as belonging to a certain phase depending on the relative distances (similarity) between basic block vectors. Thus, a phase can be as short as the size of an interval and as long as the entire execution. This also means that it is not possible to capture phase behavior within an interval. As such, the *granularity* of the interval affects observed phase behavior. In addition, the degree to which two intervals are similar, i.e., the *similarity threshold*, also affects observed phase behavior.

The framework can be used to evaluate both granularity and similarity thresholds. It is important that users be able to evaluate the effects of these parameters as it has been shown by other researchers that these parameters significantly impact phase-shift detection [50]. The choice of both interval size and similarity threshold can be governed by several factors, e.g., the application being analyzed, the intended use of phase information, etc. For example, if the phase information is to be used in dynamic optimization by the JVM, the similarity threshold ensures that the execution characteristics

within that phase are similar enough for the intended optimization to be applicable to the entire phase.

The data analysis toolkit consists of four primary tools: an image generator and visualizer, a phase finder, a phase analyzer, and a code extractor. The tools are written in either Perl or Java, available as open source, and easily extensible. These tools together constitute the Phase Analysis Toolkit that we have built, and are described in the next section.

3.2 Phase Analysis Toolkit

We next describe the various tools that we have developed to aid researchers in the analysis of phase data generated by our framework. Our toolkit consists of four primary tools: an image generator and visualizer, a phase finder, a phase analyzer, and a code extractor. The tools are written in either Perl or Java, available as open source, and easily extensible.

3.2.1 Phase Visualizer

The phase visualizer consumes the phase trace and from it generates a portable gray-map image. This image can be viewed using any image viewer; however, we developed our own Java-based viewer that enables users to point (using the mouse) to a

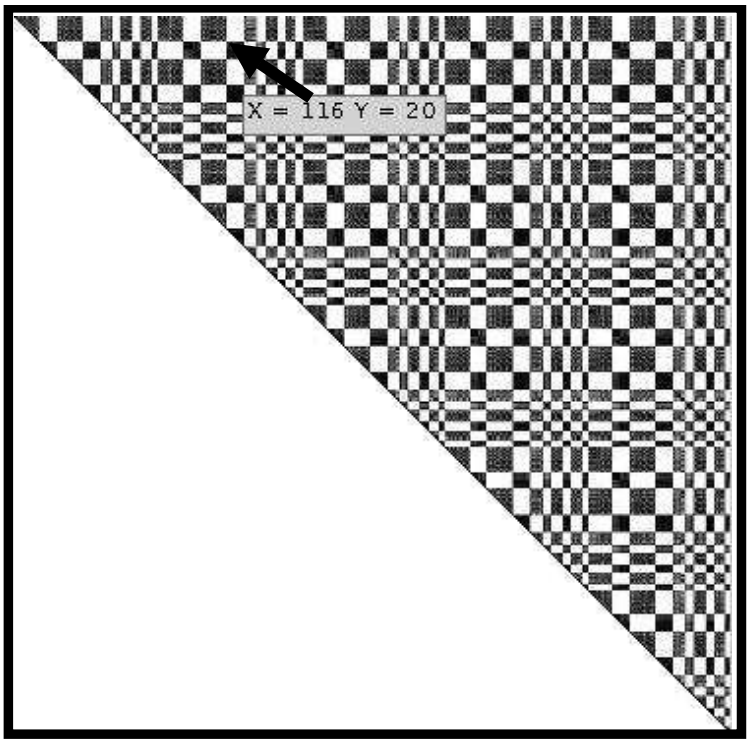


Figure 3.3: Phase Visualizer. The visualizers is a Java program the displays interval data in the portable gray-map format. The axes are interval id; intervals for this run are 5 million instructions. There are a total of 1312 intervals (x- and y- axis entries). The program is the compress SpecJVM benchmark executed with input size 100.

pixel on the image and view the interval coordinates. These coordinates allow the user to identify intervals within a visualized phase.

An image produced by the phase analyzer is shown in Figure 3.3. Each image is a *similarity matrix* [95]; the x-axis and y-axis are increasing interval id's. An interval is a period in time in the programs execution – in this figure we use an interval period of 5 million instructions. The visualizer omits data in the lower triangle since it is symmetric with the upper triangle. One reads the figure by selecting a point on the the diagonal;

each point on the diagonal is an interval. By then traversing the row, one can visualize how similar the row interval is compared to all others that follow it during execution.

The box at the mouse arrow in the figure identifies the interval coordinates that are represented by the pixel being pointed to. Black pixels indicate that the column interval is very similar to the row interval; white intervals indicate no similarity between intervals. The use of grayscale enables one to visualize the degree to which two intervals are similar. We discuss how we compute interval similarity in the next section.

In addition, the grayscale depiction of similarity enables identification of phases, phases boundaries, and repeating phases over time. The data in the figure was taken from the phase trace of SpecJVM benchmark _201_compress using input size 100. There are 1312 intervals for this program. There are many repeating blocks of black pixels; this indicates that there is phase behavior in compress and that the same phase is repeated over time. In addition, there are clear phase boundaries.

3.2.2 Phase Finder

To determine which intervals are similar (and thus, the pixel color displayed by the visualizer), the second tool that we developed is the *Phase Finder*. The tool consists of two components that compute the similarity between intervals and cluster intervals into phases.

The similarity component is a pluggable component that compares two intervals and generates a value that indicates how similar the two intervals are in terms of their execution behavior. As mentioned before, the similarity measure we use is the vector distance between the basic block vectors representing the two intervals. Specifically, we use the Manhattan distance to compute similarity. The Manhattan distance, also called the city-block distance, is the distance between two points measured along axes at right angles as against the Euclidean or straight-line distance. The Manhattan distance weighs differences in each dimension more heavily than the straight line distance and is therefore more suitable for data with high dimensionality (which in our case is the number of static basic blocks). The Manhattan distance is computed as the sum of the element-wise absolute differences between two vectors and is a value between 0 and 2. A difference value of zero implies that the two vectors are entirely similar and 2 denotes complete dissimilarity. Before computing the Manhattan distance, the basic block vectors are weighted with instruction counts by multiplying the frequency of each basic block with number of instructions in the basic block, since larger basic blocks account for more execution time. The weighted frequencies are then normalized by dividing them by the sum of all weighted frequencies in the vector – we perform this step since we are not interested in absolute values.

Other similarity metrics can be plugged into this component to enable evaluation of the efficacy of different techniques. For example, we could use other distance metrics

or use the vector angle instead [61]. Once we compute interval similarity, we map the similarity value to one of 65536 different grayscale values to generate the portable gray-map image.

The clustering component, uses interval similarity to determine which intervals should be included in a phase. This component is also pluggable, i.e., any clustering algorithm can be inserted, experimented with, and evaluated in terms of its efficacy for phase discovery.

For this component, we implemented a simple threshold-based mechanism that identifies similar intervals. The user provides a Manhattan distance threshold (between 0 and 2 in which 0 means perfectly similar and 2 means completely dissimilar). We then find intervals with Manhattan distances below the threshold given. This simple mechanism enables users to adjust the threshold value to vary the number of intervals in each phase. As such, it enables researchers to evaluate the degree to which similarity is important in exploiting relationships between intervals.

Another way in which phases can be generated is by using a conventional clustering algorithm like k-means clustering or minimum spanning tree clustering. A variation of k-means is used successfully in [95] to cluster intervals into phases using hardware. Under this scheme, the dimensionality of the data (basic block vectors) is first reduced using random linear projection, then the k-means clustering algorithm is applied. This

and similar techniques can be implemented as a clustering component and plugged into our infrastructure.

3.2.3 Phase Analyzer and Code Extractor

We also developed two tools enable users to extract statistics as well as code from each phase or interval: The *Phase Analyzer* and the *Code Extractor*. The phase analyzer generates and filters data to aid in the analysis of phases and individual intervals. This tool lists the intervals in each phase as well as how often the phase occurs and in what durations over the execution of the program.

The phase analyzer extracts details about the behavior of individual intervals or entire phases. For example, it reports the number of phases found, the number of instructions in each phase (over time), and how many instructions occur in dissimilar intervals that interrupt the different phases. Moreover, it lists sorted basic block and method frequencies. This data can be reported as weighted or unweighted counts. An unweighted count is the number of times a basic block or method executes during the phase or interval. A weighted count is this same number multiplied by the number of instructions in each block.

For all the data reported by the phase analyzer, we include a number of filters that significantly simplify analysis of the possibly vast amounts of data generated for a program. For example, a user can specify a threshold count below which data is not re-

ported. This enables users to analyze only the most frequent data. In addition, data can be combined into cumulative counts or into a number of categories, e.g., instructions, basic blocks, methods, and types of instructions.

Finally, to analyze the program code that makes up a phase, we developed a code extraction tool. By inputting intervals identified by the visualizer and statistics generated by the phase analyzer, users can use the code extractor to dump code blocks of interest. The granularity of the dump can be specified to be a single basic block, a series of basic blocks, or an entire method. We show how we employ all of the tools in the toolkit in the next section.

3.3 Analysis

In this section, we describe different ways in which our JVM phase framework can be used to guide adaptive optimization. The data we present in this section was gathered by executing Java benchmarks on a 1.13Ghz x86-based single-processor Pentium III machine running Redhat Linux v2.4.5 patched with perfctr for performance monitoring counters support. We used JikesRVM version 2.2.2 build 4-22-03 with an extension for Hardware Performance Monitoring support for x86 provided by the PANIC group at Rutgers University [85]. The benchmarks we examined are described in Table 3.1.

Program	Description
Compress	SpecJVM (201) Compression utility
DB	SpecJVM (209) Database access program
Jack	SpecJVM (228) Java parser generator based on the Purdue Compiler Construction Tool set
Javac	SpecJVM (213) Java to bytecode compiler
Jess	SpecJVM (202) Expert system shell: Computes solutions to rule based puzzles
Mpegaudio	SpecJVM (222) Audio file decompressor Conforms to ISO MPEG Layer-3 spec.
Mtrt	SpecJVM (227) Multi-threaded ray tracing implementation

Table 3.1: Description of the benchmarks used.

3.3.1 Visual Analysis

As mentioned previously, we visualized program phase behavior using an N x N similarity matrix, where N is the number of intervals in the program's execution. Each entry in a row or column represents an interval. Intervals are listed in each row or column in the order in which they occur in the program. An entry in the matrix at position (x,y) is a pixel colored to represent the similarity between interval x and interval y. The diagonal is black since an interval is entirely similar to itself. In addition, we color the lower triangle of the matrix white to avoid confusion since it is symmetric to the upper triangle. To see how an interval relates to the remaining program execution, we locate the interval of interest, say x, on the diagonal and move right along the row. Dark areas in the row identify intervals that are similar in behavior to interval x.

For example consider the similarity matrix for the benchmark Mtrt when we execute it with input size 10 (Figure 3.4).

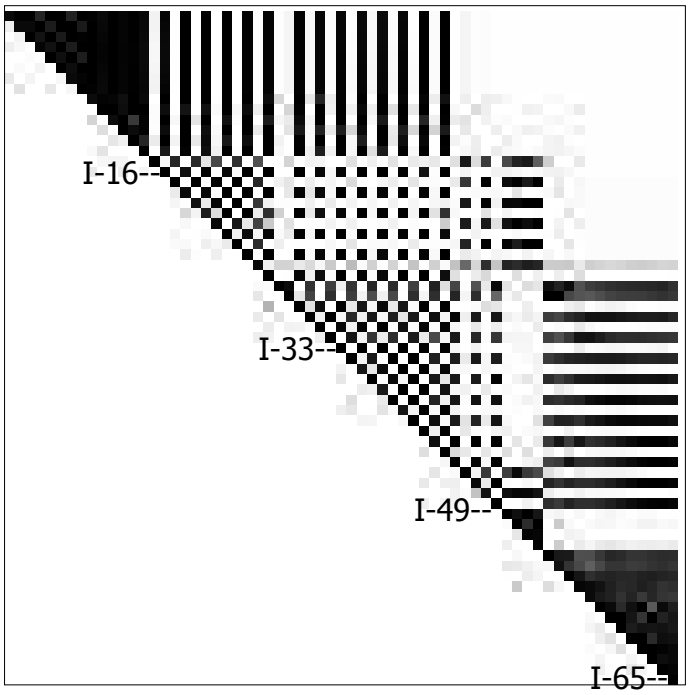


Figure 3.4: Similarity graph for Mtrt input size 10.

Mtrt executes for 65 intervals of approximately 5 million instructions each. We start at the top left corner of the matrix and move right along the x-axis. As we move right, we encounter dark pixels till we reach interval 15. That is, the initial phase, phase-1, begins at interval 0 and continues through interval 15. Interval 15 is entirely dissimilar and therefore belongs to another phase, which we call phase-2. After interval 15, the intervals alternate between phase-1 and phase-2 until we reach interval 44. From

interval 44 until the end of the execution, the intervals are completely dissimilar to phase-1.

To this point, we have visually discerned two phases. We have concluded that the intervals in phase-2 and intervals 44 through 65 are completely different from the intervals in phase-1. Now, we must investigate how intervals from interval 44 through 64 relate to each other. We do this by locating interval 44 on the diagonal and evaluating its row in the same way. We can observe two different phases in this row. It is important to note here that the dark intervals we encounter in row 44 are in no way related to the dark intervals in phase-1 even though the color may be the same. That is, a row in a similarity matrix identifies the similarity between the row interval and all future intervals.

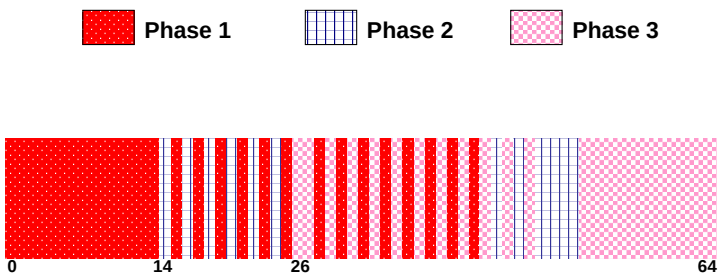


Figure 3.5: Phases for Mtrt with similarity threshold 0.8. The x-axis represents the interval identifier. The program's execution has been broken down into 65 intervals of 5 million instructions each. The pattern of each interval indicates the phase it belongs to.

Figure 3.5 shows the phases found by our phase-finder using a similarity threshold of 0.8 for Mtrt. We use a figure to depict the output of the phase finder. Each pattern

indicates a different interval; there were 3 phase detected. Using the phase analyzer, we can further evaluate each phase by analyzing the commonly executing methods. The most frequently executed methods in phase-1 are ReadPoly in class Scene and <init> in class PolyTypeObj. In phase-2 the most frequently executed methods are CreateChildren and CreateFaces in class OctNode. phase-3 then renders the scene by frequently executing Intersect from class OctNode, Combine from class Point, and RenderScene from class Scene.

Figures 3.6 and 3.7 show the interval similarity matrices for all of the benchmarks; we have omitted Compress since we include it in a later discussion. The first page matrices show the phase behavior the the programs when they are executed using input size 100; the second page shows the matrices for input size 10. Visual analysis of the benchmarks provides insight into the phase behavior in the programs and also enables us to target our efforts for further analysis using the other tools as we did above for Mtrt. We can see that each of the benchmark programs exhibits very different patterns.

In DB, Javac, Jess, and Mtrt there is a clear startup phase. Existing adaptive systems have shown that it can be profitable to consider startup behavior separately from the remaining execution [109, 106]. This phase in these three benchmarks is noticeably different from the rest of the execution. This is particularly evident in case of Javac input size 100. Using further analysis of the number of instructions executed by different methods (using the phase analyzer), we find that the most popular methods are read,

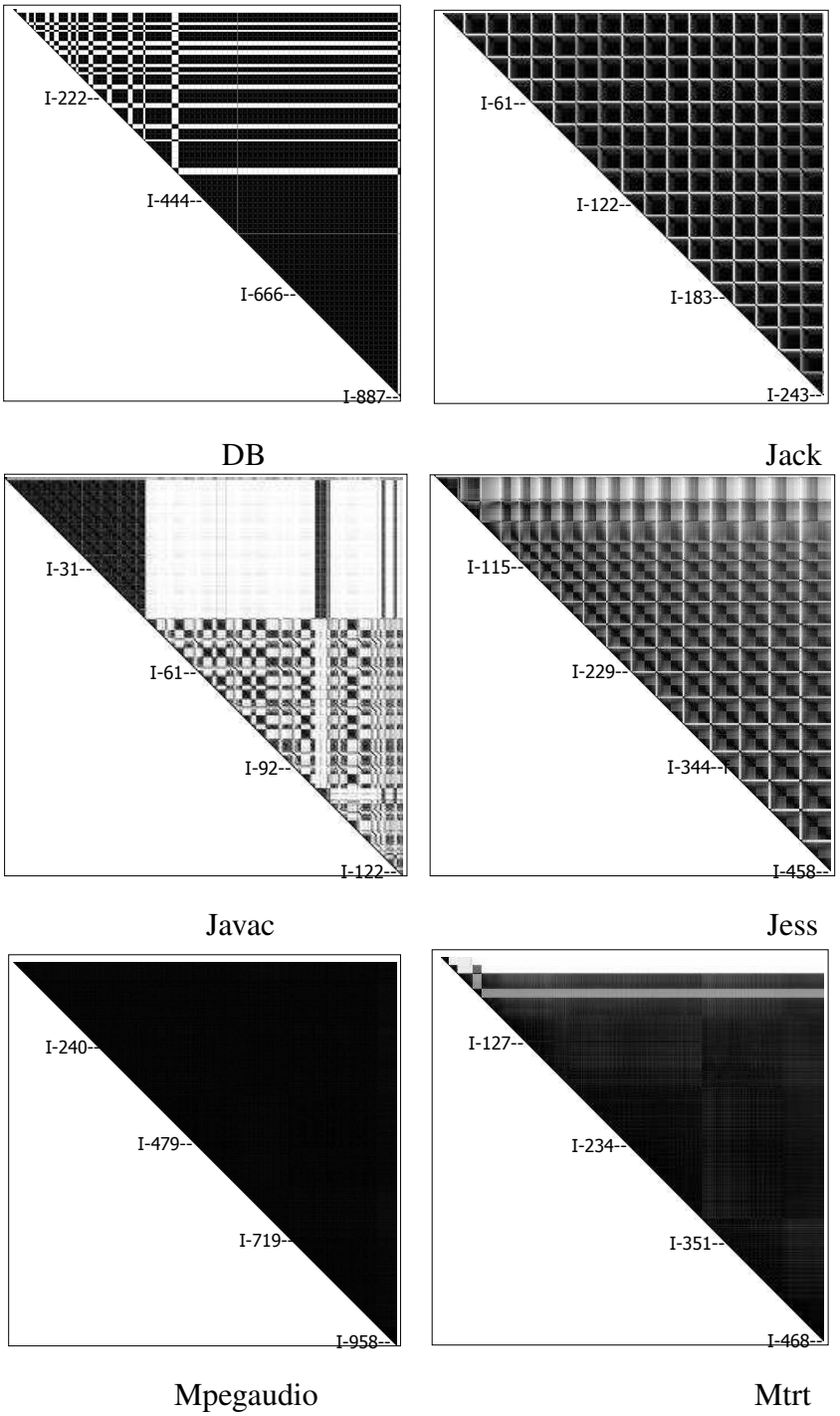


Figure 3.6: Similarity graphs for the SpecJVM benchmarks with input size 100.

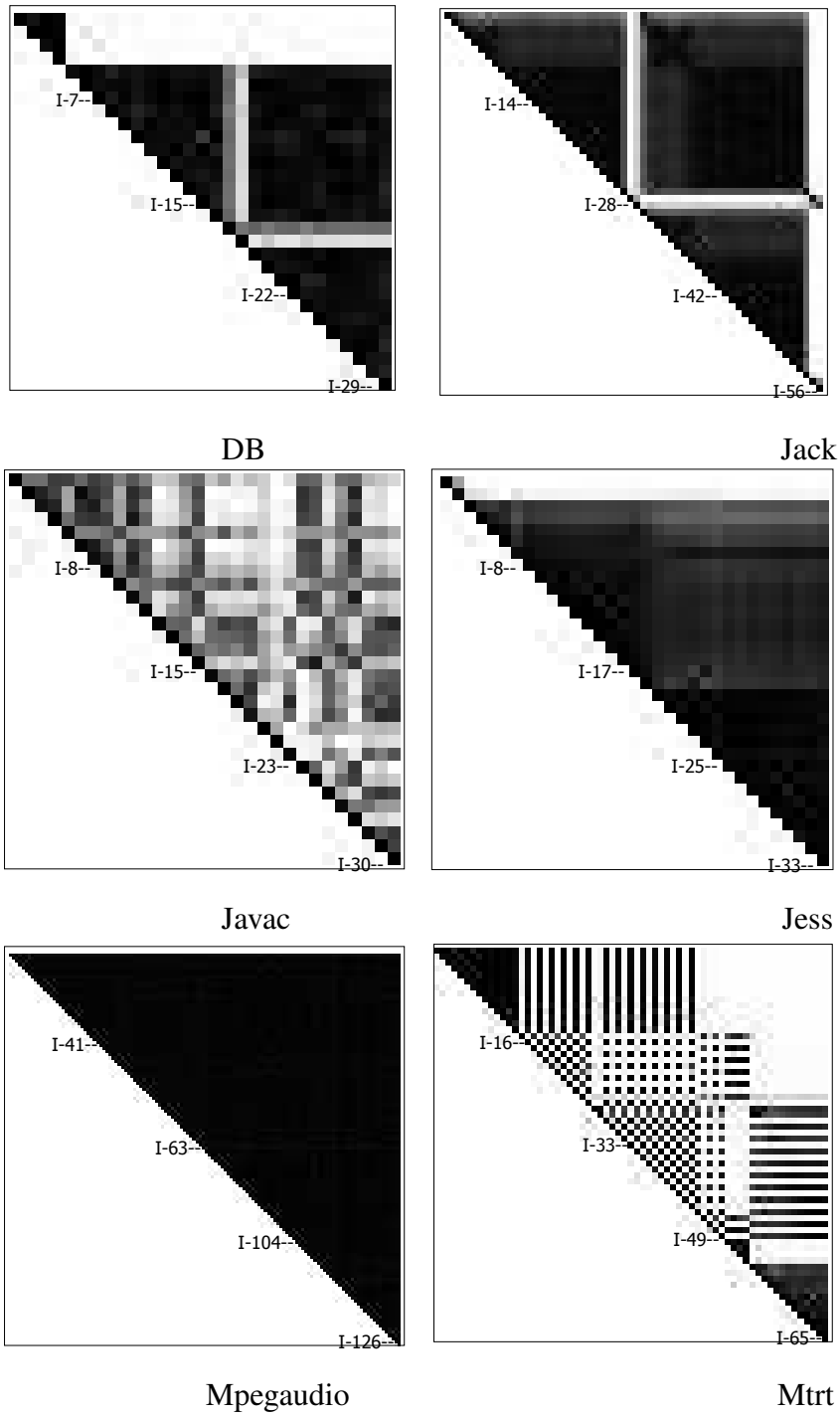


Figure 3.7: Similarity graphs for the SpecJVM benchmarks with input size 10.

scanIdentifier, and xscan during the first 90 intervals. They are again the most popular methods during intervals 202-212, depicted by the dark vertical bar in the right part of the startup phase. They are rarely executed in all other phases. For input size 10, the same benchmarks exhibit startup phases in varying sizes. For example, for DB and Mtrt, both input size 10 and size 100 have the same length startup phase; The duration of the startup phase for Javac and Jess are different across inputs. All other benchmarks, exhibit no perceivable startup phase.

Other programs exhibit other interesting phase features. Mpegaudio for both input sizes shows no apparent phase behavior. That is, each intervals is very similar to every other. Mtrt for input size 100 shows very dark intervals also, however there is a perceivable pattern that traverses the matrix. Jack exhibits a very regular pattern: 16 rows of almost perfect squares. Output from our phase analyzer for Jack reveals that the code does repeat itself 16 times for input size 100 and twice for input size 10. The reason for this is because this benchmark is a parser generator that generates the same parser 16 times for input size 100 and twice for input size 10. Our framework correctly identifies this repeating phase behavior.

3.3.2 Efficient Identification of Optimization Opportunities

Automatic visualization and extraction of phase behavior from programs greatly simplifies dynamic program analysis since the analysis of a single interval is the same as

that for all intervals in the phase. In addition, phase behavior can indicate the potential of optimizations. An optimization for an interval within a frequently occurring phase containing a large number of intervals has the potential for significantly improving execution performance.

To demonstrate our approach, we used the framework to analyze and optimize a frequently occurring phase in the SpecJVM Compress benchmark. For this experiment, we collected phase data using an interval size of 5 million instructions. We then employed the phase finder to extract highly similar intervals (using a Manhattan-distance difference threshold of 0.01). The phase we selected includes 11 intervals distributed across the execution of the program. We used the phase analyzer to filter weighted basic block counts so that we could immediately identify the most frequently executed basic block in the phase. Finally, we passed this basic block into the code extractor which dumped the register-based, low-level intermediate code (that uses the JikesRVM object model [5]) shown in Figure 3.8. We prefix this code in the figure with source code of equivalent semantics and include comments for clarity.

The basic block is the fourth block from the decompress()V virtual method in the class Decompressor. The block contains a single loop and the code has been fully optimized at the highest level available by the JikesRVM (L2) optimization system.

Despite the level of optimization, this block contains many optimization opportunities; three primary ones are (1) registers can be allocated more efficiently to avoid

spillage; (2) memory accesses can be reduced, and (3) invariant code can be removed from the block (and out of the loop). We performed the optimizations by hand to generate the code shown in Figure 3.9.

In the original code, BB4 has 19 instructions and 18 memory accesses. When hand-optimized, BB4 has 8 instructions and 4 memory accesses. We queried the phase analyzer to extract the number of times this basic block is executed during the phase. BB4 is executed 5.1 million times in the phase. By optimizing this single basic block, we eliminate over 56 million instructions and over 20 million memory accesses. BB4 is executed 46.7 million times throughout the program. As such, our total potential savings amount to over 513 million instructions and 186 million memory references. In addition, further optimization is possible. For example, the loop can be unrolled to eliminate compare, branch, and array bounds check instructions.

It is true that this type of analysis can be performed by simply collecting and analyzing basic block profiles. However, the framework enables us to significantly reduce the effort required for such analysis and to understand to potential impact of our optimization: We analyze and optimize a single block within interval and yet impact *all* intervals within the repeating phase. In addition, the framework significantly reduces analysis: this experiment required less than an 30 minutes to complete.

```
/*
do {
  a = y.obj2.ary[- - y.obj2.index];
  y.obj1.ary[y.obj1.counter++] = a;
} while (index != 0)
*/

LABEL BB4                          //start of BB4
R1 = R2(-52)                       //y.obj1
R4 = R2(-72)                       //y.obj2
R4(-20)- -                         //y.obj2.index- -
R3 = R4(-16)                       //y.obj2.ary[]
R6 = R4(-20)                       //y.obj2.index
array_bounds_check(R6,R3(-4))      //R6: y.obj2.index
                                   //R3(-4): y.obj2.ary.length
R5 = R1(-16)                       //y.obj1.counter (old)
FP(-20) = R5                       //spill R5
FP(-20)++                          //y.obj1.counter++ (new)
FP(-36) = R4                       //spill R4 (=y.obj2)
R4 = FP(-20)                       //new y.obj1.counter
R1(-16) = R4                       //y.obj1.counter = new
R1 = R1(-20)                       //y.obj1.ary[]
array_bounds_check(R5,R1(-4))      //R5: old y.obj1.counter
                                   // R1(-4): y.obj1.ary.length
R3 = R3(+R6)                       //y.obj2.ary[y.obj2.index]
R1(+R5) = R3                       //y.obj1.ary[old y.obj1.counter]
R4 = FP(-36)                       //restore R4 (=y.obj2)
cmp R4(-20), 0                     //compare y.obj2.index, 0
jne BB4                            //goto LABEL BB4 if !=
```

Figure 3.8: Code extracted using the phase framework and toolkit for SpecJVM benchmark executed using input size 100. The code is contained in the most frequently executed basic block in the phase being analyzed. We prefix the low-level intermediate code dumped by the code extractor with the source code equivalent.

```
LABEL BB3
...
R1 = R2(-52)                    //y.obj1
R2 = R1(-20)                    //y.obj1.ary[]
R4 = R2(-72)                    //y.obj2
R3 = R4(-16)                    //y.obj2.ary[]
R4(-20)- -                      //y.obj2.index- -
R6 = R4(-20)                    //y.obj2.index
array_bounds_check(R6,R3(-4))   //R6: y.obj2.index
                                //R3(-4): y.obj2.ary.length

LABEL BB4
R5 = R1(-16)                    //R1.counter
array_bounds_check(R5,R2(-4))   //R5: R1.counter
                                //R2(-4): y.obj1.ary.length
R4 = R3(+R6)                    //y.obj2.ary[y.obj2.index]
R2(+R5) = R4                    //y.obj1.ary[R1.counter] = R4
R1(-16) ++                      //R1.counter++
R6- -                           //y.obj2.index- -
cmp R6, 0                       //compare y.obj2.index, 0
jne BB4                         //goto LABEL BB4 if !=

LABEL BB5
R4 = R2(-72)                    //if necessary: restore R4
R4(-20) = R6                    //and store y.obj2.index
...
```

Figure 3.9: Hand-optimized basic block exposed via phase analysis (original code in Figure 3.8).

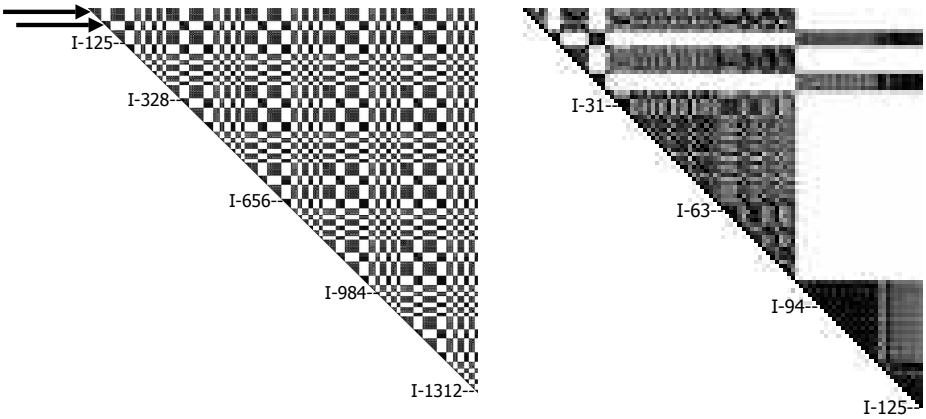
3.3.3 Cross-Input Analysis

Commonly, offline profiling techniques are limited since they are dependent upon the input used [64]. When a different input is used, the assumptions about the execution change and hence, optimizations performed based on a cross-input, offline profile may be incorrect and impose needless overhead. As such, we are interested in when offline profiling techniques are likely to be effective. We can use our analysis framework to perform such cross-input analysis.

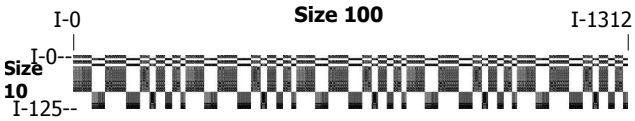
Figure 3.10, in graphs (a) and (b), shows the similarity matrices for the Compress SpecJVM benchmark for input size 100 and size 10, respectively. The total number of intervals is 1312 for size 100 and 125 for size 10. The interval size in both graphs is 5 million instructions. The two graphs appear very different at first glance.

Figure 3.10 (c) shows the similarity matrix *across* the two Compress inputs. We compute this matrix using the phase finder to identify similarities across to different runs of the same program. Since the static number of basic blocks are the same, we can compare the two vectors as if they were from the same program. Now, however, the similarity matrix is not square. The rows are from input 10 and the columns are from input 100.

The matrix indicates that there may be potential for cross-input optimization. Two different alternating (non-contiguous) patterns are visible through out the entire execution of input 10. This is much like the pattern that occurs for input 100 alone. We also



(a) Compress Size 100 (b) Compress Size 10



(c) Cross-Input Similarity for Compress

(d) Cross-Input Similarity for Mpegaudio

Figure 3.10: Analysis of cross-input similarity. Matrix (a) is for Compress and input size 100, matrix (b) is Compress and input size 10, matrix (c) is cross-input similarity for Compress, and matrix (d) is cross-input similarity for Mpegaudio.

analyzed Mpegaudio in a similar way to evaluate whether the lack of phase behavior that occurs in both inputs is the same behavior across inputs. The data is shown in Figure 3.10(d) and indicates that the same basic blocks are executing at the same frequency and duration for both input 10 (126 intervals) and input 100 (958 intervals). The matrix indicates that it is.

3.3.4 Other Opportunities for Exploiting Phase Behavior

There are many other ways in which we and other researchers can use this phase analysis framework to enable efficient analysis and optimization development. For example, we can use phases to identify opportunities to unload unneeded native code from the system to avoid unnecessary garbage collection [109], to perform dynamic voltage scaling to reduce power consumption [63, 107, 54], and to identify opportunities for code specialization.

Code specialization is the process of generating different versions of optimized code that each assume different behavioral patterns. For example, if we know that a particular value is frequently used as an argument to a method, we can generate a specialized version of the code that assumes that that value is a constant. Another form of specialization is deferred compilation [45, 106, 26] in which only part of a method is compiled and optimized. The remainder of the code is assumed to be unused and is omitted. If the code is executed in a different way from that assumed by the specializer, the system

must make amends via dynamic re-compilation and possibly on-stack-replacement, or invoke a different version of the code to handle the unexpected case.

We can use phase information to identify opportunities for these and other types of specialization. For example, consider the first two rows of repeating patterns, demarked with arrows, in Compress (input size 100), in Figure 3.10(a). We can consider each black region in the first row as a single, noncontiguous phase. Likewise, we can consider the set of black regions in the second row as a single, noncontiguous phase. Notice that these two phases are completely dissimilar; the black intervals in the first row map to white intervals in the second row and vice versa.

We used the framework to investigate the methods in these two phases. We employed the phase analyzer to identify the most popular methods in terms of instructions executed. In both cases (for both phases), they were the *same five methods*: compress()V, output(I)V, and cl_block()V from the class Compressor and decompress()V, and getcode()I from the class Decompressor (all are virtual methods). The JikesRVM adaptive optimization system correctly identifies these methods as hot and optimizes them.

However, using the framework, we are able to visualize and analyze the phase behavior in Compress. As can be seen in the figure and is described above, the execution behavior *alternates between two different phases*. That is, two completely different sets of basic blocks are being executed during each phase. By considering this phased be-

havior, we can specialize the code using deferred compilation in two different ways: In each version, we optimize only those blocks that are used during the phase. Alternately, we can reorganize the code across the 5 methods to exploit the phased behavior. We also can explore other types of specialization, e.g., based on parameter values, if inputs to the 5 methods dictate which phase the program is in. As such, by coupling the analysis available via our framework with an adaptive optimization system, we are able to identify and exploit an opportunity for specialization and code reorganization that is not apparent without considering time-varying behavior.

3.4 Summary

This chapter described the phase visualization and analysis framework we developed to enable the study of time-varying behavior, i.e., *phase* behavior in Java programs. The framework couples existing techniques from other research domains (architecture and binary optimization) into a unifying set of tools for data collection, processing, and analysis of dynamic phase behavior in Java programs. The framework is highly extensible and can be used by ourselves and others to investigate currently open questions about phase behavior analysis and exploitation in Java programs.

In summary, the contributions that we make with this work are

- The definition and implementation of a framework and toolkit for phase-aware dynamic analysis of Java programs.
- Visualization and analysis of phase behavior in the SpecJVM Java benchmark suite.
- Examples that demonstrate how the framework aids optimization researchers in the identification of optimization opportunities.
- A set of tools that are easily extensible that enable the experimentation with a number of phase parameters, e.g., interval size (granularity), interval similarity, and phase finding algorithms.

Chapter 4

Phase Detection for Java Programs

Using the phase analysis and visualization framework described in chapter 3, we were able to ascertain that Java programs do exhibit phase behavior, which could be exploited by dynamic optimization systems, like Java Virtual Machines, to perform specializing optimizations. Unfortunately, extant approaches for detection and prediction of phase behavior rely on either *offline* profiling [66, 92, 46, 88], hardware support [94, 95, 96, 42, 16, 76, 38], or are targeted toward a particular optimization (client), e.g., dynamic hardware reconfiguration. Programs that execute on virtual machines, such as programs written in the Java programming language, are compiled dynamically, executed on any hardware for which a VM is available, and optimized in a variety of different ways. As a result, it is desirable for a phase detection solution not to depend on 1) offline profile information, 2) specialized hardware, 3) architecture-specific metrics, or 4) a specific optimization client.

Vital to the efficacy of phase-guided optimizations is the accuracy of online phase detection algorithms [51]. By defining a large class of online phase detectors and evaluating their accuracy, we take a necessary initial step in the understanding of online phase detector accuracy for dynamic optimization systems.

To facilitate the design and implementation of online phase detection algorithms, we define a parameterizable framework; a phase detector is an instantiation of the framework. Section 4.1 describes this framework, its components, and parameters. To evaluate the accuracy of these algorithms, Section 4.2 defines a new client- and machine-independent empirical methodology. Section 4.3 employs this methodology to assess the accuracy of our online phase detectors.

4.1 Online Phase Detection Framework

An online phase detector accepts a continuous sequence of profile elements, such as methods, basic blocks, memory addresses, or values loaded from memory, and produces a state for each element. Each state signifies whether, after seeing the most recent profile element, the detector feels the application is in a phase, P , or transitioning between phases, T . The length of a phase is the number of consecutive P states. The detector can include optional features, such as a level of confidence in the current state, or whether a detected phase is similar to a previously known phase [96].

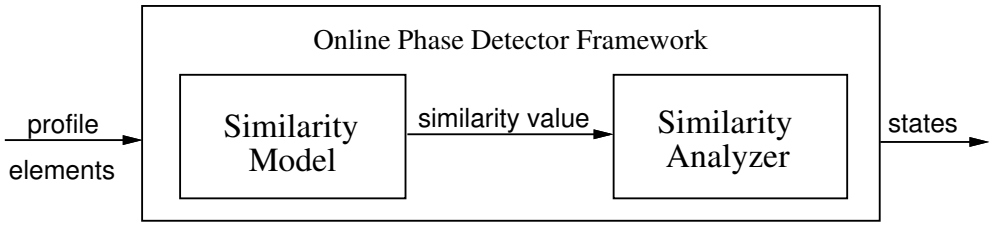
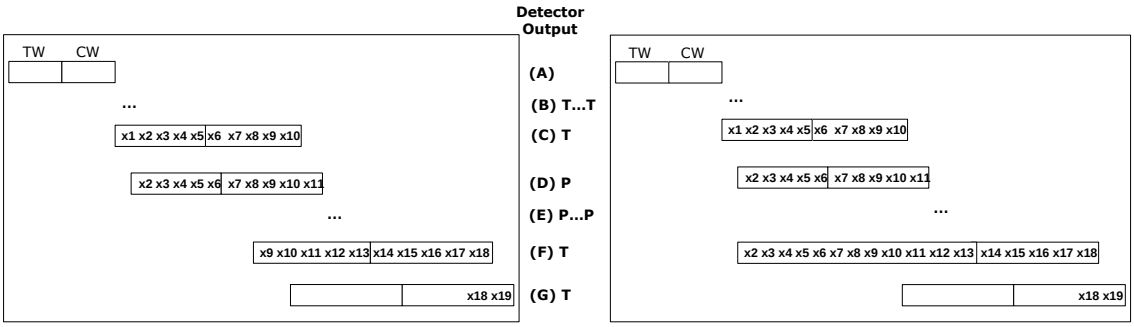


Figure 4.1: Illustrated view of the phase detection framework

The main difference between an online and offline phase detector is that an online detector does not have the complete profile available before identifying phases. Because an online detector will run alongside the program, it should be efficient in time and space. This work focuses on the *accuracy* of online phase detection algorithms; we defer investigation of the efficiency and use of these algorithms to future work.

This section presents our framework for online phase detection algorithms. Figure 4.1 presents a component view of this framework. The input to the framework is a sequence of profile elements, i.e., an execution profile. The first component, a *similarity model*, consumes the profile elements and transforms them into a sequence of similarity values that represents the degree of similarity between recent profile elements. The model passes the similarity value in an online manner to the second component, the *similarity analyzer*. The analyzer determines whether the similarity is sufficient to signify the execution is in phase, P , or in transition between phases, T . The output from the framework is a series of states, one per input element. From this output, we can



(a) (b)

Figure 4.2: Example of the basic operation of the framework using the Constant TW policy (a) and the Adaptive TW policy (b). Both policies use a *skipFactor* of 1.

identify phase boundaries at points in the output at which there is a T followed by a P state or a P followed by a T state.

The detector can include optional features, such as a level of confidence in the current state, or whether a detected phase is similar to a previously known phase [96]. Unlike an offline phase detector, our online detectors do not have the complete profile available from which it identifies phases. Because an online detector executes concurrently with the program, it must be efficient in both time and space. Moreover, because the clients of the framework make decisions based on phase boundaries, the algorithms that the framework instantiates must output phase boundaries accurately. This paper focuses on this latter constraint: phase detector accuracy.

The model and analyzer components can be implemented in many ways. For example, the model can differ in how it consumes, internally represents, and computes

the similarity of the profile. Many extant phase detection approaches compute similarity using unweighted sets [38] and weighted sets [62, 61, 94, 95, 96]. A simple analyzer reports a *P* state when the similarity value exceeds a predetermined fixed threshold [62, 61, 94, 95, 96, 38, 39]. By varying the implementation and parameterization of these components, the framework can be used to investigate, compare, and evaluate both extant and novel algorithms.

In our online phase detectors, a model represents the most recently consumed profile elements with a *current window* (CW), and represents the next most recently consumed profile elements with a *trailing window* (TW). A similarity value captures the similarity of the elements in the two windows. A window policy of the model determines, for example, the CW size, the TW size, and the number of profile elements consumed at a time, which we refer to as *skipFactor*. A significant amount of prior work [62, 61, 94, 95, 96, 38, 39] sets the size of the CW, TW, and *skipFactor* to the same value. We investigate the efficacy of such a parameterization as part of our analysis.

Figure 4.2 illustrates the basic operation of the framework using two different trailing window policies: Constant (a) and Adaptive (b). Each row illustrates a different point in time as reflected in the contents of the TW and CW. Profile elements are numbered in the order in which they are consumed. Initially, both the CW and TW are empty (row A). As the program executes, the windows fill *skipFactor* profile elements at a time (*skipFactor* equals 1 in this example). Until the windows fill (row B), the de-

tector outputs T. Once the windows are full, the model computes the similarity between the two windows and the analyzer produces a P or T state. At row C this computation results in a T state. The computation at row D results in a new phase being detected, which continues for a series of profile elements in row E.

When the phase ends at row F, we see the difference between the two policies. With the Constant TW, the TW size remains the same (length five in this example). The Adaptive TW policy grows the TW to include all elements in the phase. When the phase ends, the algorithm flushes the TW and initializes the CW with the last *skipFactor* profile elements. Row G illustrates the CW after it consumes the next profile element.

Figure 4.3 presents a high-level description of our framework's internal process. A detection client invokes `processProfile` with the most recent *skipFactor* profile elements. The model consumes the new profile elements, updates the CW and TW, and computes a similarity value for the updated windows. The analyzer uses this value to determine the new state, P or T. If the output state begins a new phase, the model can optionally anchor the TW at the start of the phase. While in phase, the analyzer tracks the statistics of the phase. If the output state ends a phase, the model clears the CW and TW and the analyzer can optionally reset any phase-specific statistics. Finally, the framework returns the output state to the detector client.

Our abstract representation of an input allows a wide variety of inputs, such as the methods invoked, basic blocks, branches, addresses loaded, or instructions executed

to be considered. This work considers dynamic branch traces. Prior work has shown that such control-flow based profiles can effectively summarize both control- and data-centric execution as well as micro-architectural behavior [96, 68].

In practice, the profile elements may form a hierarchy of phases [67], such as what one might expect from a nested-loop structure. Ideally, an online phase detector will find this hierarchy so that the detector's client can exploit it. However, because extant online clients currently do not make use of this phase hierarchy, we present phase detectors that produce flat (not nested) phase structures.

Three orthogonal design choices must be made to instantiate the framework into a concrete online phase detection algorithm. The choices are the window policy, the model policy, and the analyzer policy.

4.1.1 Window Policy

The window policy specifies the *skipFactor*, window sizes, and how to manage the TW. The value of *skipFactor* impacts both the overhead of the algorithm and its sensitivity to changes in the profile. A smaller *skipFactor* results in more frequent similarity computations. These comparisons may increase overhead, but result in a more accurate detector.

The size of the CW impacts the granularity at which the algorithm detects phases. A phase that is smaller than the CW may not be detected.

```
class PhaseDetector {
  Model model;
  Analyzer analyzer;
  PhaseState state, newState; // initialize to T

  public PhaseState processProfile(profileElements) {
    model.updateWindows(profileElements);
    similarityValue = model.computeSimilarity();
    newState = analyzer.processValue(similarityValue);
    if (state.isTrans() && newState.isPhase()){
      // start phase
      model.anchorTrailingWindow();
      analyzer.resetStats();
    } else if (state.isPhase() && newState.isTrans()) {
      // end phase
      model.clearWindows();
    } else if (state.isPhase()) {  // in phase
      analyzer.updateStats(similarityValue);
    }
    state = newState;
    return state;
  }
}
```

Figure 4.3: Online phase detection framework

The window policy also dictates the behavior of the TW. Many previous methodologies partition a profile into fixed intervals and then compute the similarity between intervals. In addition to modeling this approach online, i.e., TW size = CW size and computing the similarity between adjacent intervals, we also consider a novel adaptive alternative (that we describe above) for which the TW grows to accommodate the current phase once the algorithm detects that the program is in phase. Because a TW contains a representation of profile elements, such as a set that contains only unique, but not necessarily all, elements, we expect the size of the Adaptive TW to be manageable. As we describe for the example in Figure 4.2, when a phase ends, the model empties the TW and resets it to its original size.

4.1.2 Model Policy

The manner in which a phase detection algorithm models the similarity of profile elements impacts both the accuracy and efficiency of a phase detector [39, 96]. We investigate both unweighted set (also called working set) models and weighted set models.

For the unweighted set model, we consider asymmetric weighting, which computes the percentage of elements in the CW that are also in the TW. This model is biased toward the elements in the CW – which may be effective in combination with the Adaptive TW policy that we describe above. For example, if all elements in the

CW are present in the TW, regardless of their frequency, a similarity value of 1.0 results. Likewise, if the CW contains $\{a, b\}$ and the TW contains $\{a, c\}$, a score of 0.5 results regardless of how often a appears in the two windows.

For the weighted set model, we consider symmetric weighting, which treats both sets equally. It first computes the relative weight of each profile element in each set (TW and CW) independently. The relative weight is the percentage of a window for which a particular element accounts. The model then takes the sum of the minimum of the weights for each element in both windows, producing a number between 0 and 1. For example, assume CW contains $\{(a, 5), (b, 3), (c, 2)\}$ and the TW contains $\{(a, 25), (b, 15), (c, 10), (d, 50)\}$, then a accounts for 25% of TW (50% of CW); b accounts for 15% of TW (30% of CW); c accounts for 10% of TW (20% of CW); and d accounts for 50% of TW (0% of CW). By summing the minimum of these values across windows TW and CW, we produce a similarity value of 0.5 $(= .25 + .15 + .10)$.

4.1.3 Analyzer Policy

Given a similarity value, the analyzer determines whether this value represents sufficient similarity to indicate a *P* state. In addition to exploring a wide range of fixed thresholds, as other researchers have done, we explore analyzers that adapt their threshold based on past similarity values in this phase. The *average* analyzer computes a running average of the similarity values for the current phase, and uses a threshold that is a

delta below this average. For example, if the running average of the similarity values of the current phase is 0.88 and the delta parameter is 0.02, the analyzer reports a *P* state for values of 0.86 or higher.

The *standard deviation* analyzer computes a running average and standard deviation of the similarity values of the current phase and uses a threshold that is a multiple of the current standard deviation below the average. For example, if the running average of the similarity values of the current phase is .88, the standard deviation is .05, and the multiple is 1.0, a similarity value of $(.88 - (1.0 * .05) =)$ .83 or higher will be reported as a *P* state.

4.2 Evaluating Phase Detectors

We next focus on evaluating the accuracy of online phase detectors. Extant methodologies evaluate accuracy by using a particular phase detector client, such as a feedback-directed optimization, or by using an architecture-specific metric, such as variance in the number of cycles per instruction (CPI). Our methodology computes the accuracy of phase detection algorithms independent of the phase detection client and independent of any architecture-specific information. The methodology consists of two parts: a *baseline solution* (Section 4.2.1) and an *accuracy scoring metric* (Section 4.2.2).

4.2.1 Phase Detection Baseline

Our baseline solution implements an intuitively "correct" solution to phase boundary identification for a particular program's execution, that can be used to compare online phase detectors. The baseline solution is not an online detection algorithm. Instead, it employs a global view of a program's execution trace and makes multiple passes over the trace to identify periods of repetition. The baseline solution identifies periods of the execution as in phase and all other parts as in transition. We use the baseline solution as an oracle to evaluate online phase detection algorithms.

To identify periods of repetition, we consider two source constructs: loops and repeated method invocations, where repeated method invocations are recursive or temporally adjacent sequential invocations. We record the entrance and exit of each repetition construct with a unique identifier.

To determine the duration of a particular period of repetition, we correlate these events with the "time" of the latest dynamic branch, such as the loop was entered after the kth branch occurred. From the profile elements and source constructs, we construct a dynamic call-loop trace that we use to identify phase boundaries. Our approach is similar to the ones described by Lau et al. [66] to find software phase markers, and by Georges et al. [46] to find method-level phases. Their techniques summarize the execution of these repetitive events in a graph that is tied to the program's static structure and that is augmented with dynamic execution-time profile information. In contrast,

our approach tracks individual executions of such events in a trace that allows us to distinguish different executions of a loop body as being in phase or not if their execution lengths differ significantly.

Our baseline solution requires a *minimum phase length* (MPL) parameter, which specifies the minimum length that a period of repetition must be before it will be considered a phase. A client would specify the MPL to ensure that the phases identified have sufficient duration to amortize the client's costs. For example, if a client's phase-based optimization requires an approximate cost of 100,000 branches, then employing this action for a phase that is only 50,000 branches long will result in a net loss. Section 4.3 shows how the MPL value used in our framework impacts accuracy.

All phases identified by our baseline solution are *complete repetitive instances* (CRI's), i.e., a set of profile elements within an entire loop execution (all iterations) or within a recursive execution. We consider a recursive execution to start upon invocation of a method (which the program later invokes recursively) for which there is no other execution instance on the stack. For example, if a program makes the following method calls without returning: `main` → `foo` → `bar` → `foo`, then the root of the recursive execution starts and ends at the invocation and return, respectively, of the `foo` instance called by `main`. Although it is possible for different iterations of a loop or different recursive executions of a method to have different branch behavior, we assume that these differences are small, and thus, we consider them part of the same phase.

If a CRI is smaller than the client-specified MPL, we attempt to combine the CRI with temporally adjacent CRI's with the same static identifier (e.g., method name or loop number) into a single phase. We do so if the distance (in terms of number of profile elements) between them is one. This enables us to combine perfectly nested loops and temporally adjacent, repeated invocations of the same method into a single phase.

We view nested loops either as one large phase consisting of the outer loop, or as smaller phases represented by executions of one or more nested loops. We employ MPL to decide between these two choices. If the number of profile elements (dynamic branches in our case) in an execution of a nested loop is at least MPL and there is more than one profile element between executions of the nested loop, we consider this execution of the nested loop a phase. If the number of profile elements in an execution of a nested loop is smaller than MPL or there is only one profile element between executions of the nested loop (as in a perfect loop nest), this execution of the nested loop is not viewed as a phase, and we consider the execution of the next outer loop. We repeat this process until the number of profiling elements exceeds MPL. When this occurs, we select the nest as the representative for the phase.

To validate this approach, we collected branch coverage data (percent of branches that are considered part of some "phase") in the baseline solutions. Our empirical study shows that MPL-based selection enables more control over phase size than specifying

a loop nest level. For example, using only outer loops to identify phases results in a very small number of large, coarse-grained phases that cannot be readily subdivided.

Each baseline solution identifies the state (P or T) of each profile element, from which we can extract the phase boundaries that represent the actual repeated execution of the program. We use the extracted phase boundaries to compare and evaluate online phase detection algorithms. We quantify this comparison using the accuracy scoring metric that we describe in the next subsection.

4.2.2 Accuracy Scoring Metric

To compare the efficacy of a phase detection algorithm against the baseline solution, we introduce a novel accuracy scoring metric that has two components. The first assesses how well the states identified (P or T) by the detector match those of the baseline solution. We refer to this property as *correlation* in the spirit of the work by Dhodapkar and Smith [38]. We define correlation as $\frac{\texttt{bothInPhase}+\texttt{bothInTransition}}{\texttt{totalEvents}}$, where `bothInPhase` is the total number of profile elements for which both the detector and the baseline solution output P . Similarly, `bothInTransition` is the number of events that the detector and baseline solution both output T . `totalEvents` is the total number of profile elements. This component of the score measures the extent to which the decisions of the detector and the baseline solution correlate.

The second component of the score measures how often the detected phase boundaries match those of the baseline solution, using two values: *sensitivity* and *false positives*. Sensitivity quantifies how often the detector and the baseline solution agree on phase boundaries. It is defined as $\frac{\texttt{numMatchedBoundaries}}{\texttt{numBaselineBoundaries}}$, where `numMatchedBoundaries` is the number of detected phase boundaries that match the baseline solution and `numBaselineBoundaries` is the number of phase boundaries identified by the baseline solution. The false positives value quantifies how often the detector identifies a phase boundary that the baseline solution does not. It is defined as $\frac{\texttt{numUnmatchedBoundaries}}{\texttt{numDetectedBoundaries}}$, where `numUnmatchedBoundaries` is the number of detected phases boundaries not identified by the baseline solution and `numDetectedBoundaries` is the number of phase boundaries identified by the detector.

Phase boundaries identified by the detector and baseline solution match when the following constraints are satisfied. First, the start of the detected phase must occur at, or after, the start and before the end of the identified phase in the baseline solution. Second, the end of the detected phase must occur at, or after, the end of the current phase and before the start of the next phase in the baseline solution. Third, the closest detected boundary to an identified boundary in the baseline solution that satisfies the first two constraints matches the identified boundary.

Correlation, sensitivity, and false positives are combined into a single weighted sum, called `score`, which we define as

$$\frac{\texttt{Correlation}}{2} + \left(\frac{\texttt{Sensitivity}}{4} + \frac{(1-\texttt{FalsePositives})}{4}\right).$$

We weigh Correlation and matching (Sensitivity and FalsePositives) equally and split the matching weight evenly between Sensitivity and FalsePositives. Thus, Correlation accounts for 50%, Sensitivity accounts for 25%, and FalsePositives accounts for 25% of the score. Scores fall into the range [0, 1] with higher scores signifying more accurate detectors. Achieving a perfect score in the correlation component, and thus, in the overall score, would require reporting a change in phase state as soon as it occurred in the baseline solution. This may be impossible for an online detector. For example, in our framework the windows must be full for the algorithm to make an evaluation (compare similarity) and to detect a state change. As a result, the algorithms will always detect a phase after it has started. The degree to which an algorithm is late depends on the window size and is reflected in the correlation portion of the score.

4.3 Analysis

This section presents the empirical evaluation of instantiations of the framework described in Section 4.1. After briefly describing our methodology, we present a detailed analysis of different dimensions of the framework.

Table 4.1: Benchmark Characteristics

(a)

Benchmark	Dynamic Branches	Loop Executions	Method Invocations	Recursion Roots
_201_compress	62,808,794	3,980,731	2,407,272	0
_202_jess	15,525,021	140,268	1,558,571	5,984
_205_raytrace	5,801,454	82,556	337,133	6,811
_209_db	3,374,648	317,397	13,621	0
_213_javac	2,770,921	200,121	995,992	10,786
_222_mpegaudio	37,099,265	1,906,483	2,831,987	0
_228_jack	5,926,061	593,135	514,923	4,471
Jlex	2,779,996	146,716	199,868	16

(b)

Benchmark	MPL=1k		MPL=5k		MPL=10k		MPL=25k		MPL=50k		MPL=100k	
	Num Phases	Pct in Phase	Num Phases	Pct in Phase	Num Phases	Pct in Phase	Num Phases	Pct in Phase	Num Phases	Pct in Phase	Num Phases	Pct in Phase
_201_compress	46	34.88	20	34.83	20	34.83	20	34.83	20	34.83	6	99.67
_202_jess	3250	91.44	1092	63.43	473	46.32	134	47.64	88	44.04	30	41.79
_205_raytrace	1448	88.34	198	55.80	84	71.38	41	63.08	25	52.75	17	43.37
_209_db	1152	88.84	303	92.25	147	89.43	51	83.66	13	93.82	5	97.26
_213_javac	665	49.60	149	45.49	76	56.69	29	50.11	15	66.21	9	55.29
_222_mpegaudio	7594	46.70	1968	28.12	894	52.85	894	98.13	22	3.20	2	99.75
_228_jack	1778	53.31	324	48.85	100	43.74	30	36.20	18	29.02	4	13.64
Jlex	102	97.10	53	94.74	49	94.40	39	88.61	32	78.76	2	92.85

4.3.1 Methodology

Our profile is a conditional branch trace of Java programs, which we obtained by modifying JikesRVM [58, 5] to produce a profile element for each branch executed. Each profile element represents a unique location in the source code as an integer value that encodes a unique method ID, a bytecode offset in the method where the branch is located, and a bit that represents whether the branch was taken. Our framework, however, is not Java or JikesRVM specific; it consumes profile elements generated by any toolset for profile extraction, e.g., we can also generate such profiles using the Phoenix instrumentation and compilation framework [90] from Microsoft Research.

We derived baseline solution phase structures from a call-loop trace by instrumenting loop and method entries and exits (both normal and exceptional). We record the unique loop or method identifier and the offset in the profile trace at that point. This allows us to correlate baseline and detected phase boundaries.

We evaluate our phase detection algorithms using eight Java benchmarks, seven from the SPECjvm98 [104] benchmark suite, and JLex [19] (a lexical analyzer generator for Java). We currently consider single-threaded applications only, though the framework can be extended to handle multi-threaded applications. We use input size 10 for the SPEC benchmarks and the default input for JLex. We optimize all application and library methods upon first invocation and extend the optimizing compiler of JikesRVM to add branch, method, and loop tracing instrumentation.

Table 4.2: Window size comparison. (a) shows average percent improvement in best score across all framework parameters when we use a CW size smaller or equal to the MPL as compared to a CW larger than MPL for three TW policies: Adaptive, Constant, and Fixed Interval. (b) is the average of best scores across all benchmarks when the size of CW is smaller than, equal to, and half of MPL.

(a)

Benchmark	Adaptive		Fixed		Base	
	Smaller	Equal	Smaller	Equal	Smaller	Equal
_201_compress	28.54	19.96	33.21	22.45	42.71	26.34
_202_jess	13.75	9.31	5.98	5.23	-2.88	-7.91
_205_raytrace	-6.25	-1.25	-0.56	5.26	-2.30	-2.30
_209_db	20.18	10.24	20.21	9.19	13.36	6.35
_213_javac	19.76	15.73	21.78	19.71	25.59	15.14
_222_mpegaudio	12.70	22.61	9.25	17.98	28.86	21.44
_228_jack	22.55	17.25	24.80	20.77	22.25	18.50
Jlex	13.75	9.31	8.93	10.09	3.30	1.72
Average	15.62	12.90	15.45	13.83	16.36	9.91

(b)

	Smaller	Equal	1/2 MPL
Base	0.601	0.570	0.610
Fixed	0.648	0.639	0.664
Adaptive	0.652	0.637	0.664

Table 4.1(a) lists each benchmark and its dynamic execution characteristics. Column 2 gives the number of dynamic branches in a trace. Column 3 gives the number of loops executed. Column 4 gives the number of method invocations; and column 5 is the number of method invocations that are the root of recursion. Both *Loop Executions* and *Recursion Roots* represent the frequency of code structures that can give rise to repetition of program behavior. Although loop executions dominate, we must also consider recursion when identifying phases.

For our baseline solutions, we consider the following MPL values: 1000, 5000, 10000, 25000, 50000, and 100000 (henceforth abbreviated to 1K, 5K, 10K, 25K, 50K, 100K). Table 4.1(b) provides information about the phases found by the baseline solution for different MPL values. For a given MPL value, the column to the left lists the number of phases found (*# Phases*) and the column to the right shows the percentage of profile elements (dynamic branches in this case) that are in phase (*% in Phase*). The number of phases found varies significantly across benchmarks and across MPL values. For example, with an MPL value of 1K, `compress` has only 46 phases whereas `mpegaudio` has 7,594. However, with an MPL value of 100K, `mpegaudio` has only two phases.

The table illustrates the trend that as MPL values increase the number of phases decreases. This is expected, since as the MPL value increases, our baseline solution identifies larger loops (and recursive chains) as phases.

Counter to intuition, the percentage of profile elements in phase does not correlate with the MPL value. This is an artifact of how the baseline solution selects which loop in a loop nest to identify as a phase (Section 4.2). With a small MPL value, an inner loop may be considered a phase while the containing loop is not. When the MPL is increased, the nested loop may no longer be bigger than the new MPL value, but the containing loop will be large enough to be a phase. When the containing loop becomes a phase, all the profile elements of the inner loop and containing loop are now part of the phase, and thus, increase the percentage in phase value compared to just the profile elements from the inner loop.

However, the percentage in phase value can also decrease when the MPL value is increased. For example, consider a simple loop that has sufficient profile elements to satisfy the MPL value, and thus, is identified as a phase. However, if the number of profile elements is not enough for a larger MPL value, none of these profile elements will be consider in phase with this larger MPL value.

We used our framework to instantiate a large number of phase detection algorithms. Given the various combinations of parameterizations possible (*skipFactor*, current window size, trailing window policy, model policy, and analyzer policy), we generated over 10,000 different algorithms, which we then compared against our baseline solutions. We computed a score for each detector using our accuracy scoring metric from Section 4.2. In the subsections that follow, we summarize and analyze this data in a way

that indicates the general trends in accuracy. In particular, we use the data to investigate the various framework parameters discussed in Section 4.1.

4.3.2 Window Policy

We first evaluate the impact of the current window (CW) size on detector accuracy. Intuitively, CW size should be related to the MPL parameter that is used by the baseline solution to find the actual phases in a program.

For our phase detection algorithms, we considered CW sizes of 500, 1K, 5K, 10K, 25K, 50K, and 100K. We computed scores for each CW size and MPL value combination, across all other parameters that we considered: skip factor, TW size, and model and analyzer policies. We then extracted the best score across all combinations and evaluated, for each benchmark, the average when the CW size was smaller, equal to, and larger than the MPL value. Table 4.2(a) shows the results. We present three sets of data (pairs of columns) for each benchmark. The first set is data for detectors that use the Adaptive TW policy and a skip factor of 1.[1] The second set is data for the Constant TW policy and a skip factor of 1. The final set, *Fixed Interval*, is data for a Constant TW policy with a skip factor equal to the CW size. This last policy is the one most

[1]The other Adaptive TW policy parameters that we used for this data set and those that follow include an anchor policy of RN (rightmost noisy + 1) and the sliding window resizing policy. We define and support these choices empirically in Section 4.3.5.

commonly used by extant approaches to phase detection, in which the *skipFactor*, TW size, and CW size are all the same value [62, 61, 94, 95, 96, 38, 39, 72, 35].

Each pair of columns under each policy is the percent improvement in score when we use a CW that is smaller than (first column) or equal to (second column) the baseline's MPL, over using a CW size that is larger than the MPL. We cannot compare this data across sets (Adaptive, Constant, and Fixed Interval), because each column is relative to the base case of that set, i.e., the score when CW size is larger than the MPL value. We compare these configurations using other data in the next subsection.

The data shows that, on average, the highest accuracy occurs with detectors that employ a CW size that is smaller than the MPL value. Although using a CW size that is equal to the MPL value also enables higher accuracy than using one that is larger, the improvement is not as great as for a CW size smaller than the MPL. One reason for this is that the detectors employ two windows, the size of which totals at least twice the CW; this total size is similar to MPL.

Table 4.2(b) shows the average of best scores across all benchmarks and MPL values, for each TW policy (Adaptive, Constant, and Fixed Interval). We show data for a CW size that is smaller than an MPL value (column 2) for a CW size that is equal to an MPL value (column 3), and for a CW size that is 1/2 the MPL value or smaller (column 4).

The scores that result from using a CW size smaller than MPL are similar to those for a CW of 1/2 the MPL. If the CW is 1/2 MPL or smaller, then the size of TW and CW together is at least MPL; thus, the detector is able to accurately identify the same phases as our baseline solution (for that MPL value). The data also shows that a CW smaller than MPL in some cases outperforms a CW of 1/2 the MPL. The reason for this is that the particular CW size that produces the best score for the smaller CW case varies across benchmarks. There is no single CW size smaller than 1/2 MPL that outperforms a CW of 1/2 MPL across all benchmarks on average. We therefore, use 1/2 the MPL as our CW size for the remainder of the paper in an effort to focus our analysis of the remaining dimensions of our algorithms.

We next evaluate the impact of skip factor on detector accuracy for the three TW policies (Adaptive, Constant, and Fixed Interval) and consider all other parameterizations of the model and analyzer policies. We again consider the best score across these configurations.

Figure 4.4 compares the three TW policies. The x-axis is MPL and the y-axis is the average of best scores across all configurations and benchmarks. A higher score is better. We consider two values of skip factor: one and CW size. The former enables high responsiveness by the detector to detect fine-grain changes in phase behavior. We evaluate the accuracy enabled by two different skipFactor values by comparing the Fixed Interval bars (*skipFactor* = CW size) against the remaining two (skipFactor =

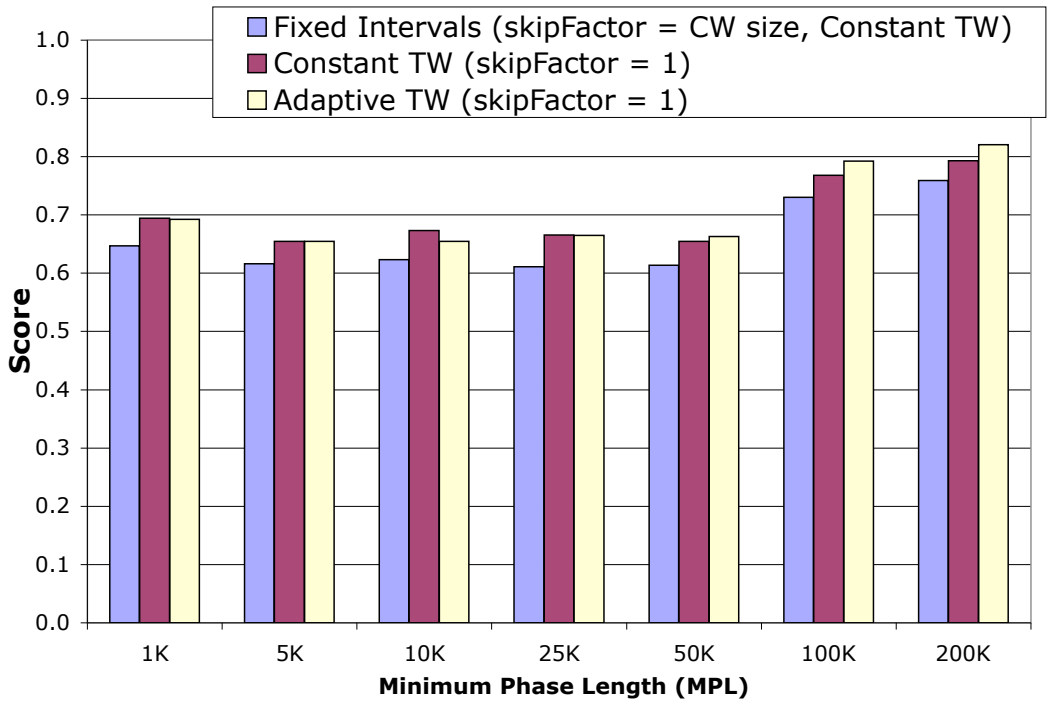


Figure 4.4: Evaluation of skip factor and Fixed versus Adaptive windowing. The data is the average of best scores across all benchmarks, models, and analyzers. The CW size is less than 1/2 the MPL.

1). The data shows that on average, the approach commonly used in existing systems (*skipFactor* = CW size) is significantly less accurate than both the Constant TW and Adaptive TW policies when *skipFactor* is one. Thus, the remaining evaluations use a skipFactor = 1.

When we compare Constant TW and Adaptive TW, the results are less clear. In general, our experiments show that for small MPLs, Constant TW does somewhat better than the Adaptive TW. However, this is not the case for all benchmarks when we consider them individually. For larger MPLs, Adaptive TW is consistently more accurate than a Constant TW. We added MPL 200K to this data set to evaluate whether the trend continues, and it does. For larger MPLs, some of the shorter running benchmarks exhibited a very small number (1 or 2) of very large phases, which were not useful or fair to include in a comparison (all detectors achieve very high scores since there are so few phases to match against).

The remainder of the paper presents results for MPL values of size 1K, 10K, 50K, and 100K. We continue to include data for both Constant TW and the Adaptive TW policies in our subsequent comparisons.

4.3.3 Model Policy

Figures 4.5 presents an empirical comparison between two models described in Section 4.1: weighted and unweighted. The x-axis shows MPL values and the y-axis

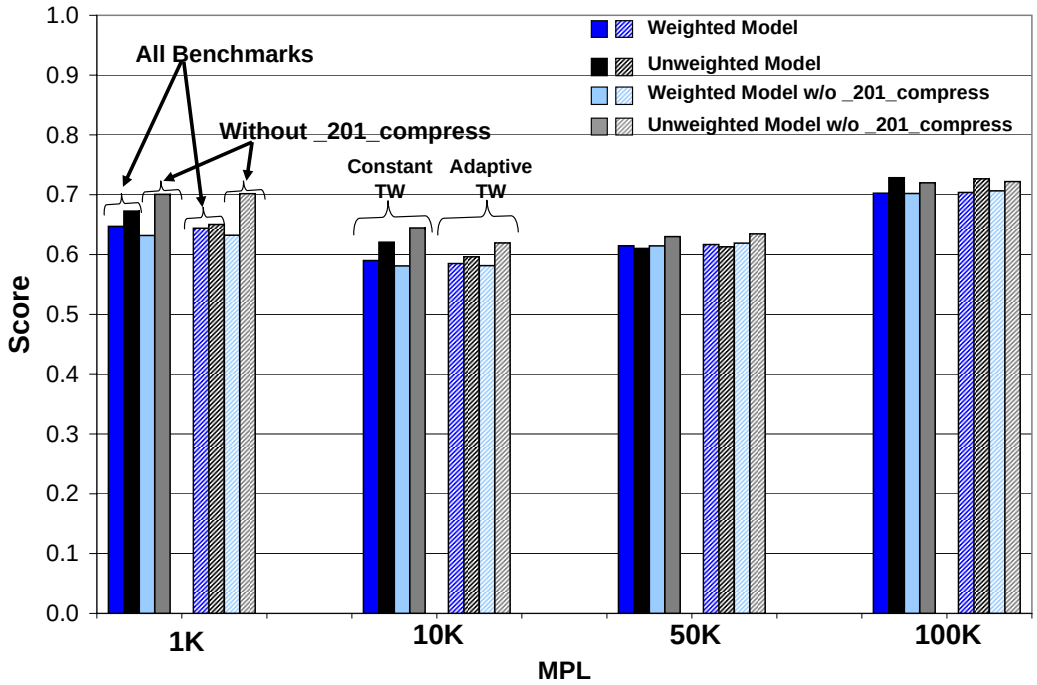
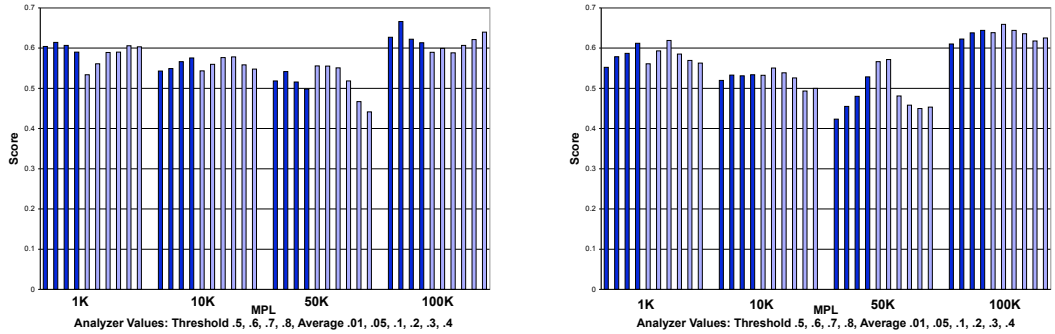


Figure 4.5: The average of best score across all benchmarks for two models. There are two sets of bars per MPL group for the Constant TW and Adaptive TW policies. For each policy, weighted and unweighted model scores are shown with and without the compress benchmark.

shows the average of best score across all benchmarks. For each MPL, there are two groups, each with four bars. The first group is for the Constant TW policy and the second group is for the Adaptive TW policy. Within each group, there are two pairs of bars. In each pair, the left bar is the weighted model results and the right bar is the unweighted model results.

The first pair of bars in each group shows the average of best score across all benchmarks. These results show that the unweighted model is more accurate than the weighted model in all but the 50K MPL case. When we consider the individual benchmark data, however, unweighted is significantly more accurate in a majority of cases for all benchmarks except one: _201_compress (compress hereafter). For this benchmark, the detectors that employ the weighted model are almost 50% better in many cases (we omit this data due to space constraints).

To show the average accuracy of detectors without considering the compress benchmark, we include a second pair of bars in each group of four in the graph. This pair shows the average of best scores for detectors that employ the weighted and unweighted models, respectively, on average across all of our benchmarks, except compress. From this data, we can conclude that, in general, the unweighted model is more accurate than weighted model for all MPLs and trailing window policies. As a result, we consider only the unweighted model for our analysis of similarity analyzers in the next section.



(a) Constant TW (b) Adaptive TW

Figure 4.6: Constant vs. Adaptive window policy. This Figure shows the average of best scores across all benchmarks for the Constant TW (a) and the Adaptive TW (b). Each chart is grouped into four sets of bars, one for each MPL value. Each MPL category has ten analyzers corresponding to (from left to right) the four Threshold analyzers (darker bars) with increasing threshold (0.5, 0.6, 0.7, 0.8) and the six Average analyzers (lighter bars) with increasing deltas (0.01, 0.05, 0.1, 0.2, 0.3, 0.4).

4.3.4 Analyzer Policy

Figure 4.6 shows a comparison between two categories of analyzers: Threshold and Average, each with different parameters. The figure contains two subgraphs. The left graph (a) presents the data for the Constant TW policy and the right graph (b) presents the data for the Adaptive TW policy. In each graph, the x-axis presents MPL values and the y-axis presents the average of best scores across all benchmarks. For each MPL, there are ten bars. Within the ten bars, the first four bars, which are darker, represent the Threshold analyzers with values of 0.5, 0.6, 0.7, and 0.8; and the last six bars represent the Average analyzers with values 0.01, 0.05, 0.1, 0.2, 0.3 and 0.4.

The data presents mixed results. Neither the Threshold nor the Average analyzers are clear winners for all MPL values and all benchmarks. However, if one were to pick a particular analyzer, certain values seem to be a better choice for a specific trailing window policy. In particular, if the Threshold analyzer is chosen, a threshold value of 0.6 wins in three out of four of the MPL values for the Constant TW policy, whereas the threshold value of 0.8 wins in three out of four of the MPL values for the Adaptive TW policy. If the Average analyzer is chosen, there is not a clear trend for the Constant TW policy; however, a value of 0.05 wins for three out of four of the MPL values for the Adaptive TW policy. A more comprehensive analysis of the data is required to better understand these trends.

4.3.5 Additional Analysis

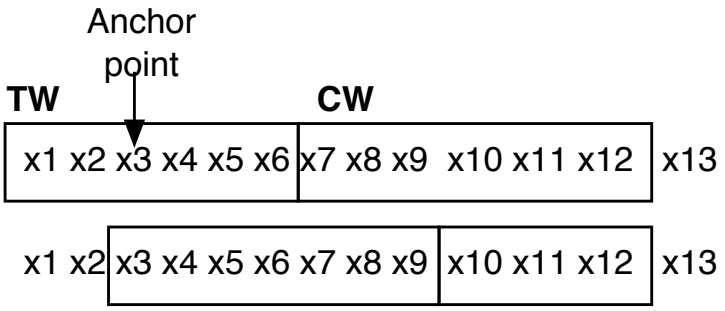
This section analyzes other parameters of our phase detection framework. The first parameter specifies how window resizing and anchoring is performed when an algorithm using the Adaptive TW policy detects the start of a phase. This parameter impacts the detection of phase-start boundaries, and therefore, can produce a more accurate representation of the phase. It is also important for an Adaptive TW policy because it serves as a signature of the entire phase.

Before discussing this parameter fully, we discuss other properties that can also impact the accurate identification of phase start boundaries. First, as mentioned in

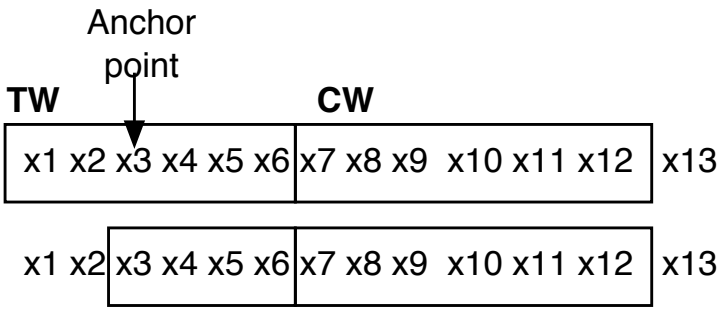
Section 4.2, an online algorithm will have a delay in profile elements before it can detect the beginning of a phase. Second, phase boundaries may not always align with skipFactor values. Third, phases often exhibit startup periods where the behavior is less stable (but is not considered a transition) than the steady state of a phase [83].

The anchor point is the position in the TW at which a new phase starts. We explore two options to determine where to place the anchor point. The first option places the anchor point one element to the right of the rightmost noisy element in the window (RN). Noisy elements are those that are in the TW and not in the CW. The second option places the anchor point at the leftmost non-noisy element (LNN). Both techniques attempt to eliminate instability during the start of a phase to enable more accurate detection of *T* or *P* states thereafter. RN is more aggressive at doing so. For example, if the TW contains elements a, b, and, c and the CW contains a, a, and c, then b is a noisy element. The RN policy selects the position of c in the TW and the LNN policy selects the position of a in the TW, as the start of the phase. Both policies attempt to eliminate profile elements that are part of the warm-up period [83] of the phase that may not be as stable as the steady state of the phase.

Once we identify the starting position of the new phase, we have two options for window resizing. We can *slide* the TW right, so that the left boundary of the TW is at the anchor point, thus reducing the size of the CW:

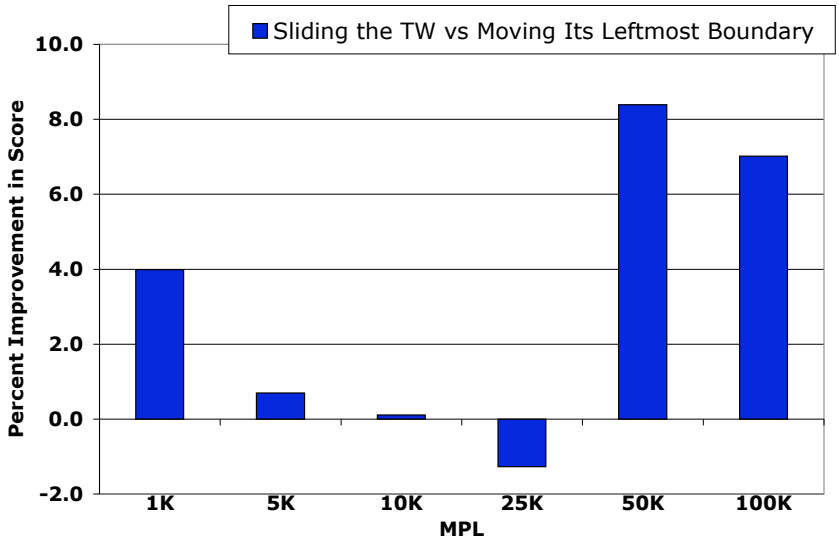


Alternatively, we can *move* the left boundary of the TW to the right and leave the CW unaffected:

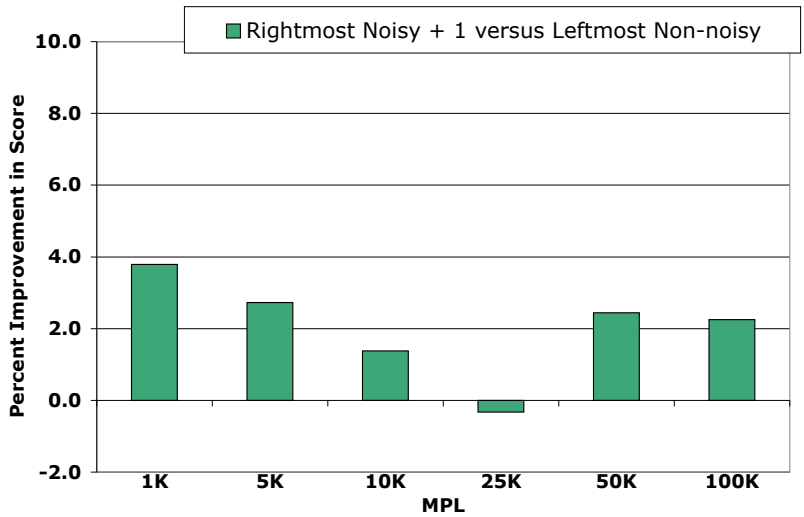


By sliding, we reduce the size of the CW; however, we continue to compare the two windows for similarity while the CW fills in this case. This enables the TW to hold as much of the phase as possible (our original goal with the trailing window policy). By moving the TW, we shrink its size as opposed to the CW.

Figure 4.7 evaluates these two policies across benchmarks for each of the MPLs (x-axis). Graph (a) shows percent improvement in score for Sliding over Moving of the TW (we use the RN anchoring strategy here). Graph (b) shows the percent improvement in score due to the use of RN over LNN to select an anchor point (for the Sliding resizing policy). On average, Sliding is more accurate than Moving and as such, is a



(a)



(b)

Figure 4.7: Percent improvement in score for Slide over Move resizing, using the RN anchoring strategy (a). Percent improvement in score for RN over LNN anchoring, using the Sliding resizing policy, (b).

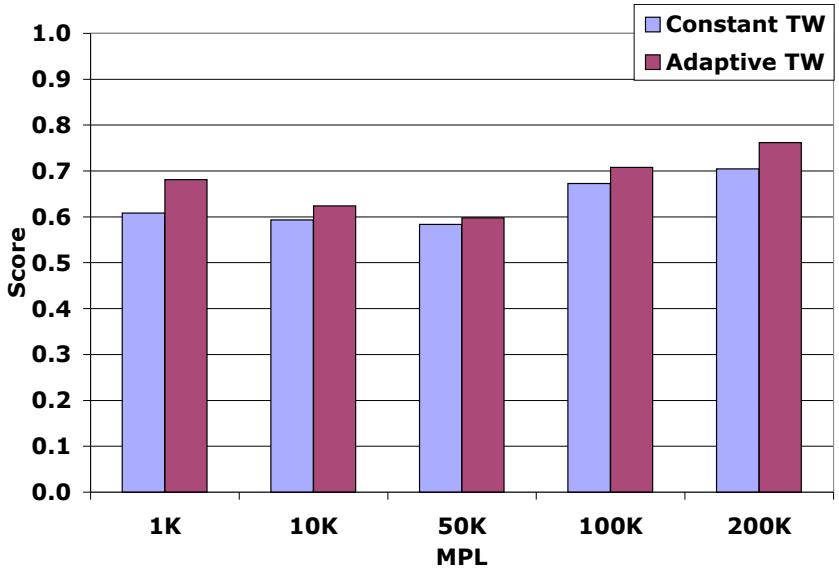


Figure 4.8: Accurate detection of phase boundaries. Average of best scores across all benchmarks, models, and analyzers for the Constant and Adaptive TW policies using the anchoring policy for detecting the beginning of a phase.

better resizing policy. It also seems intuitively correct for an Adaptive TW to include most of the recently detected phase before evaluating subsequent profile elements. In addition, RN is more accurate than LNN, on average. We use the Sliding and RN policies for the results in the previous sections and below.

Our last set of results compares the best scores for the Adaptive and Constant TW policies using a modified technique for finding the beginning of a phase. As discussed previously, an online algorithm detects a phase after some initial part of the phase has been seen. However, once a phase is detected, such an algorithm can identify where the phase began using the anchoring policy discussed above. This information can be used to accurately identify phase signatures [66, 92] and their repetition online. Figure 4.8

compare these new phase boundaries against those of the baseline solution. The results indicate that for every MPL, the Adaptive TW policy is significantly more accurate than the Constant TW policy, showing promise for its use in online detection of recurring phases.

4.4 Summary

In this chapter, we described our framework for developing a wide range of online phase detection algorithms. We also described a novel client- and machine-independent methodology for evaluating the accuracy of these algorithms, and performed a detailed empirical study of numerous algorithms using this methodology. Our conclusions are that the current window size should be smaller than the desired minimum phase length and that a skip factor of 1 has better accuracy than the standard practice of setting the skip factor to the current window size. We also find that an adaptive trailing window policy can be more accurate than a constant trailing window policy. Finally, our results for models tend to favor the unweighted model, although the results for analyzers are mixed.

Chapter 5

Phase-based Runtime Techniques

Chapters 3 and 4 dealt with analyzing, and detecting phase behavior in Java programs. The work presented in these Chapters addresses the question of whether Java programs exhibit phase behavior that can be captured and detected at runtime within a virtual execution environment. Another important question, and the focus of the remaining part of this thesis, is whether this phase behavior can be used to enable more efficient runtime techniques. From the discussion on extant, prominent applications in Section 2.2, it is evident that applications of phase behavior in the realm of software-level, feedback-directed optimization, are preliminary and limited. This Chapter presents an overview of the two phase-based runtime techniques that we investigated; the following two Chapters delve into the details of each. The first technique is an accurate, low-overhead profiling scheme that uses phases to drive when to sample the execution of a program. We employed this technique to perform efficient profiling on embedded devices. The second technique is a software instruction prefetching mecha-

nism that uses method-level phase behavior to identify, predict, and prefetch methods that incur a large number of instruction cache misses for emerging Java workloads like database- and application servers. These two techniques span two extremes of execution environments used for Java applications: software for resource-constrained devices at the low end and application servers at the high-end.

5.1 Phase-aware Profiling

With this work, we present a novel approach to program profiling that achieves efficiency and accuracy through the exploitation of program phases. Using program phase behavior, we can summarize a software system as a minimal but diverse set of program behaviors in a manner that is dynamic, efficient, and that accurately reflects overall program behavior. We propose a hardware-software method for general-purpose program profiling. The hardware efficiently monitors program execution behavior and makes predictions about what phase will occur next. The software system samples the program for only previously unseen phases, significantly reducing the overhead of program profiling.

As mentioned before, efficient profiling is one of the important aspects of feedback-directed optimization. Feedback-directed optimization can be either online, as in case of adaptive optimization in Java Virtual Machines, or offline, where information gath-

ered from one run of a program is used to optimize subsequent runs. As such, our phase-aware profiling mechanism can be extremely beneficial in any situation that demands efficient, but accurate profiles. We employ phase-aware profiling in performing remote, post-deployment profiling to enable feedback-directed performance optimization and software evolution via a distributed optimization system. The platforms that we are targeting are those that have emerged as new access points to the world's digital infrastructure: mobile, resource-constrained, battery-powered devices, e.g. personal digital assistants (PDA) and web-enabled cellular phones devices and their software continue to grow in complexity and capability, techniques are needed to ensure efficient execution, user satisfaction, and minimal power consumption. Feedback-based optimization and software evolution offers potential for such systems since such techniques gather information about a program *while* it is executing, once it has been deployed in the wild.

Since mobile devices typically have neither the extra space for compilers and optimizers, nor the resources to execute them on-the-fly, a distributed optimization system offers a good alternative. Such a system would gather information about a running program, transmit this information to an *optimization center* for analysis, possible recoding, and re-compilation using feedback-based optimization, and then update the code on the end-user system when the opportunity or need arises. Key to the success of such an approach, is a highly efficient remote performance profiling system that is

transparent and unobtrusive, i.e., that consumes only minimal device resources. This latter requirement is a significant challenge since profiles are commonly collected by executing instrumented versions of the software. Moreover, for deployed software, we must also communicate this information back to optimization center for analysis. This problem of overhead introduction is exacerbated for mobile devices with limited resources as this performance degradation can translate into significant battery drain.

We present the details of phase-aware profiling in Chapter 6, including the evaluation of its efficiency (in terms of computation, communication, and battery power) as well as the accuracy for a number of different profiles types that have been shown previously to be important for feedback-based optimization.

5.2 Instruction Prefetching

The goal of phase-based instruction prefetching is to use the ability of extracting and predicting repeating patterns in a programs behavior in tuning one aspect of the instruction cache – which instructions it contains. Modern processors demand a high-bandwidth supply of instructions from the pipeline, given current processor speeds, multi-core architectures, and techniques to exploit instruction level parallelism. If an instruction needed by the processor cannot be supplied in time (icache miss), the processor has to stall until the instruction can be made available. Higher memory access

latencies, together with the high capacity of executing instructions, exacerbates the impact of an icache miss. Prefetching is a technique that attempts to alleviate this problem by speculatively fetching instructions that will be required in the near future in advance.

Despite an abundance of research over the years, instruction cache (icache) miss stalls remain a source of performance degradation for many commercial applications [4, 17, 24, 25, 59, 75, 100]. Due to the relatively larger performance cost of data cache misses in most applications, research and development has largely focused on the data cache miss problem instead. As evidence, only a few architectures (IA-64, PA-RISC, and SPARC v9) include instruction cache prefetch instructions, while many architectures (e.g. IA-32, x86-64, and PowerPC), include no support for instruction prefetching. In contrast, all major architectures include support for software-directed data prefetching.

In theory, the icache miss problem is an easier problem to solve, because choosing blocks for instruction prefetching is solely a function of predicting control flow, while a data prefetching mechanism must also solve the considerably more difficult problem of predicting the data address that will be touched. In addition, control flow, through both branch prediction and phase behavior [96, 80], has been shown to be highly predictable. Consequently, given hardware support for software-directed instruction cache prefetching, we believe that it should be possible to significantly reduce instruction cache miss stalls for all applications (large and small).

We propose phase-based instruction prefetching as an adaptive optimization that can be performed within a Java Virtual Machine. Our target platform for this optimization is enterprise servers running commercial applications like database-, or application-servers. As a first step towards this end, we have developed a method-level instruction prefetching scheme that chooses prefetch points for methods with a high number of misses, from the call-chain for that method. This call-chain based prefetching scheme does not currently incorporate phase detection and prediction at runtime; however, we present an analysis of the method-level phase behavior in our target application, and discuss the feasibility of phase-based instruction prefetching. Chapter 7 presents the details.

Chapter 6

Phase-aware Remote Profiling

In this Chapter, we present phase-aware remote profiling, which employs program phase behavior in a novel way: to identify intelligently a representative set of profiling points. The advantage we gain by using phase information is that we only need to gather information about part of the phase and then use that information to approximate overall program behavior. By carefully selecting a representative from each phase, we can drastically reduce the number of times that we need to sample and the amount of total communication required for profile transmission to the optimization center. Since an interval will be similar to all other intervals in a phase, it can serve as a representative of the entire phase. As such, only select intervals of the program's phases need to be collected (instrumented, communicated, and analyzed) in order to capture the behavior of the entire program. This will make more efficient use of those limited resources available on mobile devices. Furthermore, these low-overhead profiles will be highly accurate (very similar to exhaustive profiles of the same program).

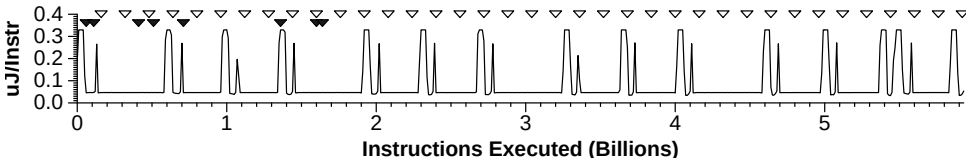


Figure 6.1: The figure shows the run-time power usage of the full execution of the program `mpeg encode`.The program exhibits different phases, marked by periods of high and low power. A random or periodic sampling method (the white triangles) will continue to take samples over the full execution of the program. A more intelligent sampling technique based on phase information (shown as black triangles) can achieve the same profile accuracy, by taking fewer key samples from each phase.

Figure 6.1 exemplifies our approach using actual energy data gathered from the execution of the mpeg encoding utility. The execution of mpeg exhibits a small number of distinct phases during execution that repeat multiple times. A random or periodic sampling method will continue to take samples over the full execution of the program regardless of any repeating behavior. In Figure 6.1, the white triangles show where samples would be taken if sampling is done periodically to achieve an accuracy error of 5% (i.e. the resulting basic block count profile is within 5% of the exhaustive profile). This has the unfortunate drawback that *most* of the samples will not provide any new information because they are so similar to samples seen in the past. A more intelligent sampling technique based on phase information (shown as black triangles) can achieve the same error rate with significantly fewer samples. This is done by taking only key samples from each phase.

We propose a hybrid method for general-purpose program profiling that consists of phase-aware sampler and a profiler. The phase-aware sampler is based on the phase-

tracking hardware proposed by Sherwood et al. described in Section 2.2 [96],and efficiently monitors program execution behavior and makes predictions about what phase will occur next. The profiling system is asked to sample the program for only previously unseen phases, significantly reducing the overhead. The profiling system is one, in which, profiling can be turned on and off. Such a system can be implemented in either hardware or software. [12, 52, 48] are examples of software profiling schemes that switch between instrumented and non-instrumented versions of the code depending on whether a particular part of the execution is to be profiled or not. Although our design does not depend on any particular profiling system, we incorporate a hardware based profiling approach based on the DISE dynamic expansion of microprocessor instructions [32]. More specifically, the profiling system that we propose consists of two main components: *Phase-Aware Sampling* and *Hybrid Profiling Support*(HPS). Phase-aware sampling uses phase information to guide its sampling decision, and HPS employs dynamic instruction stream editing [32] to toggle profile collection, i.e., to sample the executing instruction stream, whenever a decision to sample is made.

Figure 6.2 provides an overview of the interactions among the primary components of our system.

The phase-aware sampler consists of a hardware phase tracker that monitors the execution stream and predicts, for each interval in the program, whether the interval is in a previously seen or unseen phase. A software profiling daemon triggers profile

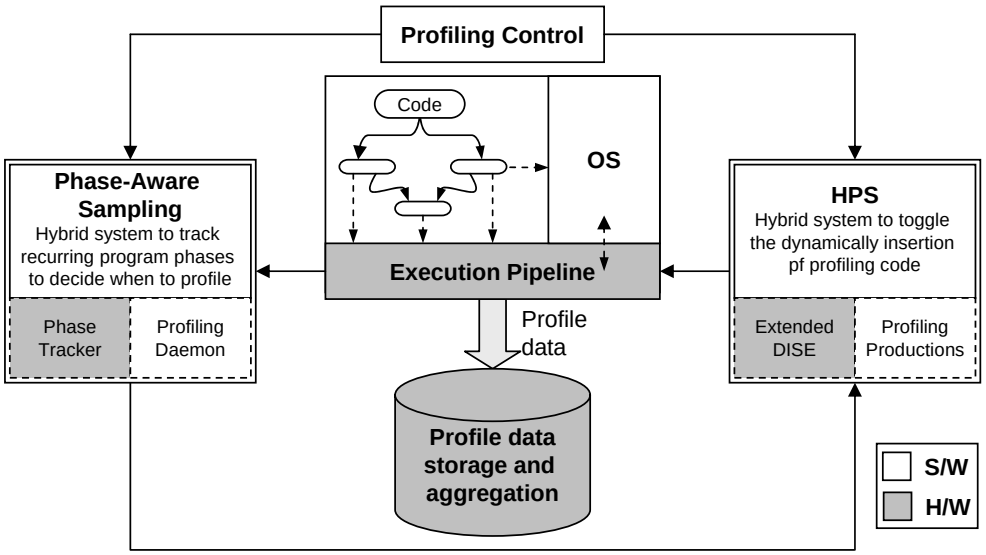


Figure 6.2: Overview of the main components of our efficient profiling system for remote embedded devices. The HPS system (section 6.2) is responsible for the dynamic insertion and toggling of profiling instructions. The phase tracking system (section 6.1) is responsible for deciding when to gather profile information.

collection by the hybrid profiling support (HPS) system for representative intervals from as-yet-unsampled phases. We can also trigger collection directly via a connection between the phase tracker and HPS; we avoid this approach since it restricts flexibility and system modularity.

HPS intercepts the instruction stream to dynamically insert instrumentation code for instructions of interest (those to be profiled) when its sampling flag is set by the phase-aware sampler. At the end of an interval, the profiling daemon clears the HPS sampling flag and all instructions execute unimpeded. The system stores samples locally for

aggregation and transmission by the profiling daemon to the optimization center, as needed.

In the sections that follow, we first present phase-aware sampling (section 6.1) and then detail the HPS system (section 6.2). We evaluate both the efficiency (in terms of computation, communication, and battery power) as well as the accuracy of our approach for a number of different profiles types that have been shown previously to be important for feedback-based optimization, e.g., hot methods, hot call pairs, and hot paths. We present results for a set of general-purpose benchmarks, as well as for a set of benchmarks for embedded devices in Section 6.3, followed by a discussion on extending phase-aware remote profiling to multiple users in Section 6.4.

6.1 Phase-aware Sampling: Deciding When to Sample

Figure 6.3 depicts our implementation of *Phase-Aware Profiling*. The system considers individual, fixed-length periods of program execution at at time. We use an interval length of 10 million instructions in this paper. To predict the phase in which a future interval will be, we employ the Phase Tracker hardware that we proposed in prior work [96]. The Phase Tracker is a small, low area, low overhead hardware resource that consumes approximately 4 picojoule of energy per dynamic branch (this means that on a high speed machine that executes 1 branch every 2ns, it will consume around 2 mW).

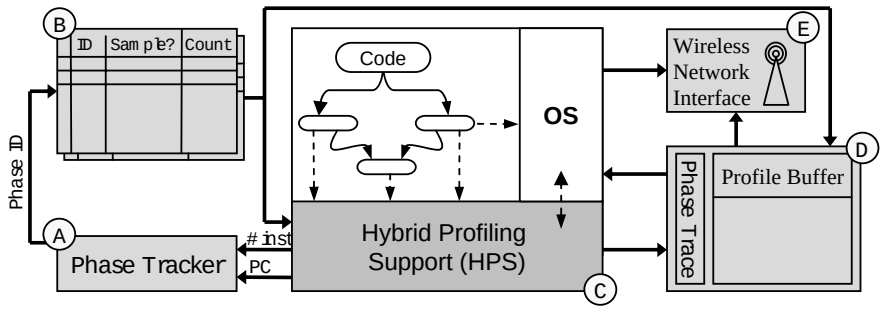


Figure 6.3: Overview of the phase-aware profiling scheme. Phases are tracked in hardware (A) and the results are fed to a small table that tracks the state of each phase (B). When a phase is deemed to be important, the profiler (HPS) is notified and a sample is taken (C). The sample is stored in a small profile buffer (D). This information is then transmitted back to a trace aggregation center (E).

The Phase Tracker (figure 6.3:A) collects the dynamic branch behavior of a program into intervals and segregates the intervals into phases according to a similarity threshold computed from the interval execution characteristics.

The Phase Tracker uses a similarity threshold to govern the number of phases generated from the set of intervals executed by the program. A higher threshold value will generate fewer phases, each consisting of more intervals, but with a higher similarity variance across the intervals of any single phase. The threshold can therefore be adjusted according to the desired sampling rate, and the resulting profile will represent the most diverse and relevant sets of dynamic behavior.

The Phase Tracker uses the program counter value of branch instructions, along with the number of executed instructions between branches, to produce a prediction for the phase of the next interval (the complete details on the prediction process can

be found in [96]). Prior work has shown the accuracy of the Phase Tracker phase prediction to be 85-90% [96]. We assume a prediction accuracy of 100% in this work; as such, our results indicate an upper bound on the potential of phase-aware profiling performance. [88] use a similar methodology to our phase-aware sampling system to significantly reduce the overhead of cycle-accurate architectural simulation. The phase tracker in this work identifies and simulates intervals that represent unique (per-phase) execution behaviors. This prior work does not assume perfect Phase Tracker prediction and achieves a 3.2% error in cycle count on average over exhaustive simulation.

Since the Phase Tracker is in hardware, it monitors the entire system, i.e. it is shared by multiple processes (much like hardware performance monitors (HPMs)). In our system we use the Phase Tracker to monitor a single process at a time. To distinguish per-process phase data, the operating system toggles phase tracking, via a register in the Phase Tracker hardware, upon a context switch. We only consider single-process phase tracking in this work; we plan to consider concurrent phase tracking as part of future work.

The Phase Tracker outputs a *phase ID*, which is a unique identifier for the behavior likely to be observed in the next interval. We store the phase IDs in a small table which tracks each phase and identifies when a sample should be taken. We have found that a fully associative table of size 20 is sufficient to minimize the number of misses using random replacement. In general, the worst case is one in which two similar behaviors

are sampled more than once as a result of a table miss. The performance effects from such misses, however, are negligible for tables of this size. The table tracks a list of phase IDs and stores a "sampled bit" that indicates if the phase has already been sampled. Additionally, we record a count of the number of times this phase has been seen in the past.

When the Phase Tracker predicts that the next interval is one from a previously unseen phase, it signals a lightweight profiling daemon (background thread). The daemon tracks the number of intervals encountered for newly executed phases and selects the most appropriate one for sampling.

Ideally, the most representative interval - i.e. most similar to the phase's other intervals - should be selected for profiling. Doing so online, however, is a challenge because we are not aware of the occurrence of future phase intervals. Therefore, we use a heuristic that quickly identifies a representative interval so that we do not miss the profiling of important phases. In our evaluation section, we consider using one of the first initial intervals encountered for a new phase and compare this policy to oracle-based approaches (centroid and random) that use future knowledge to identify the best representative interval of a phase. In general, we find that the first interval encountered is not a good representative since it commonly contains execution behaviors from the previous phase or a phase transition. The third interval is commonly sufficient to avoid this "warm-up" period.

Once the daemon identifies an interval to sample, it executes a special profile-toggling instruction. This instruction is equivalent to a no-op (it performs no work and imposes a single cycle pipeline performance penalty). Only authorized agents (processes that are granted such permission by the Operating System, e.g. the profiling daemon) may execute this instruction. Consuming this instruction causes HPS to set the sampling flag (figure 6.3:C). This is the external trigger scenario described in section 6.2.2. This simple mechanism is implemented using a single HPS-private counter and requires a single instruction to turn profiling on and off.

Once the profile has been generated, the profile daemon stores it in a specialized profile buffer and tags the profile with the phase identifier (figure 6.3:D). In addition to the profile data, the daemon records a trace of phase IDs from intervals in *previously seen* phases. This enables us to reconstruct accurate a time-varying behavior of the program if necessary.

Once the buffer fills, we must empty it via network transmission. At this point, the profile daemon transfers the profile over a wireless network back to some data center for further study and analysis (figure 6.3:E). In our embedded system study (section 6.3.2), we evaluate the efficacy of transferring data intermittently while the program is executing (e.g. when storage is limited) as well as transferring the complete phase trace once the program terminates. Since communication consumes significant battery power in mobile devices, we must ensure that we minimize the number of bytes

transfered. As such, in addition to using phase behavior to reduce profile size, we also incorporate compression of the trace prior to transmission. However, the application of compression consumes computational resources. We include the effect of this tradeoff (increased computation for decreased communication due to compression) as part of the empirical evaluation of our approach. In general, we found that the benefit of compression due to the reduced communication overhead far outweighs the computational overhead we introduce in terms of battery power.

6.2 Profiling Support for Toggling Profile Collection

To efficiently toggle profile collection, our system employs HPS, a hardware/software (hybrid) approach for dynamic profiling [77]. HPS is an application and extension of *Dynamic Instruction Stream Editing* (DISE) [32]. DISE is a hardware mechanism that dynamically and transparently inserts instructions into the execution stream thereby enabling semantics similar to macro expansion in the C programming language.

Figure 6.4 is an overview of HPS design. HPS consists of an extension to DISE (dark grey region in the hardware (H/W) section) that enables highly efficient, conditional, dynamic profile collection. HPS supports the toggling of profile collection via external triggers such as timer interrupts or periodic events. HPS also includes an internal flexible, program driven, triggering mechanism based on the sampling frame-

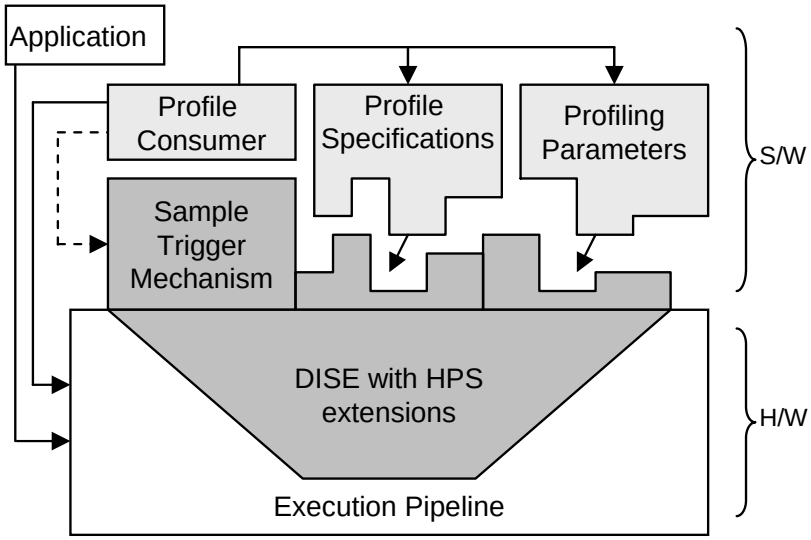


Figure 6.4: The Hybrid Profiling Support (HPS) system. HPS contains a hardware component which is an extension of Dynamic Instruction Stream Editing(DISE). Software level mechanisms control the profiling framework and provide facilities for the profile consumer to define their profiling specifications (including instructions of interest and actions to be taken). HPS allows the profile consumer to set the profiling strategy (e.g. triggering policy – exhaustive, periodic, program driven, ...) and control the profiling parameters.

work proposed by [12] and extended by [52]. Furthermore, HPS enables the profile consumer to specify arbitrary instructions of interest and associate them with profile collection actions via HPS and DISE based semantics. We next provide an overview of the DISE system and then describe how we extend DISE and apply it for efficient profiling in HPS.

6.2.1 Dynamic Instruction Stream Editing (DISE)

Hardware architects implement dynamic instruction stream editing by extending the *decode* stage of a processor pipeline. The extensions identify interesting instructions and replace them with an alternate stream of instructions.

To enable this, DISE stores encoded instruction patterns and pre-decoded replacement sequences in individual DISE-private SRAMs called `the pattern table (PT)` and `the replacement table (RT)` respectively. The hardware compares (in parallel) each instruction that enters the decode stage against the entries in the PT. On a match, DISE replaces the original instruction with an alternate stream of instructions that commonly includes the original instruction. DISE retrieves this stream of instructions from the RT according to an index generated by the PT.

An `Instantiation Logic` (IL) unit is employed by DISE to parameterize the replacement sequence according to specific information extracted from the original matched instruction. DISE also supports a small number of DISE-private hardware registers to enable efficient and transparent execution of the replacement sequence instructions without impacting the registers that an application uses.

The DISE mechanism operates within the single cycle bounds of the decode stage and imposes no overhead on instructions that are not replaced. For instructions that are replaced, DISE imposes a 1 cycle delay. DISE utilities operate concurrently with and transparently to the executing program and avoid polluting the memory hierarchy in

use by the program for instructions and data. Moreover, DISE is an abstract hardware mechanism for general purpose instruction stream editing. Users of this mechanism can customize the operational design parameters and components of DISE to fit the needs and limitations of a particular architecture.

DISE users build utilities called *Application Customization Functions*(ACFs). Users define ACFs by writing a set of DISE productions each of which consists of a *pattern specification* and a *parameterized replacement sequence*. The pattern specification includes a binary function computed from the various instruction fields. The parameterized replacement sequence is a list of instructions with fields that may contain either precise values or directives for dynamically computing the value based on the matched instruction. The IL unit processes the directives dynamically as it generates the replacement sequence.

The progenitors of DISE have built DISE utilities for software fault isolation [32], dynamic debugging [32, 33], and dynamic code decompression [31, 32]. The authors also suggest and evaluate a preliminary utility that uses DISE for exhaustive path profiling and code generation [30, 32].

The wide range of supported utilities makes DISE a prime candidate for consideration in embedded architectures. By including a DISE mechanism, embedded system designers can support a wide range of common runtime utilities simultaneously. This can reduce the memory footprint of applications, since many of these utilities are other-

wise supported through static software instrumentation. Such instrumentation increases the code size of the application and can degrade performance. By building on the general purpose framework of DISE, HPS brings efficient profiling support to embedded systems with only incremental hardware requirements to the general purpose mechanism provided by DISE.

6.2.2 Hybrid Profiling Support using DISE

To profile exhaustively, HPS dynamically replaces instructions to be profiled with a replacement sequence consisting of the instrumentation, i.e., the profile collection instructions (or procedure call that implements them), followed by the original instruction. To enable this, we define a DISE pattern/replacement production pair for each profile type: The pattern is the instruction type of interest and the replacement is the profiling instrumentation and original instruction.

We perform sample-based profiling by guarding the instrumentation in the replacement sequence with a check that first verifies whether sampling is turned on or off, i.e., whether a sampling flag is set or not. The instrumentation is only executed when the sampling flag is set.

A limitation of this implementation is that each time an instruction of interest is executed, the instruction is replaced regardless of whether the sampling flag is set. This is due to the fact that the replacement sequence itself checks the flag following replace-

ment. This can result in a very large number of replacements being performed which can introduce significant runtime overhead: a single cycle penalty for each replacement and 2-5 additional instructions (and potentially a pipeline flush) per profiled instruction. To reduce this overhead, we optimize and extend the DISE design.

Optimizing DISE for Efficient Sampling

To reduce the number of replacements that DISE performs, we extend DISE with a new functionality called *conditional control*. Conditional control, in essence, moves the check of the sampling flag out of the replacement sequence (software) and into the DISE engine (hardware). Conditional control enables HPS to instrument profiled instructions only while the sampling flag is set. To enable conditional control, we modify the DISE engine and production specification.

Figure 6.5 shows the DISE extensions that enable our efficient implementation of HPS. Instead of requiring that pattern specifications in the DISE ACF be implemented using unconditional matches, we allow *conditional pattern specification*. We add a masking layer which implements conditional matching based on the status flags of internal counters (we currently consider only overflow and zero status flags).

HPS manages internal counters using micro-instructions that are defined as part of the DISE productions. HPS stores these microinstructions in an associative table that we have defined called the Conditionals Table (CT). The CT is similar in structure (but

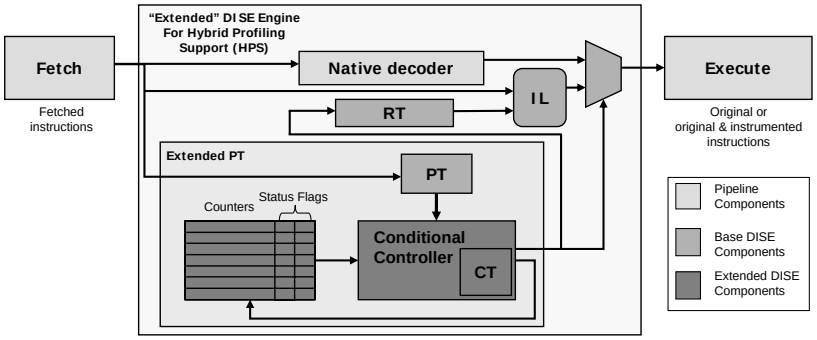


Figure 6.5: HPS extensions to DISE. HPS moves conditional control of instrumentation out of the critical path and into a dedicated controller.

smaller in size) to the pattern table (PT) defined as part of the original DISE infrastructure. The RT is part of the original DISE implementation and holds the replacement instruction sequences.

We define a set of lightweight micro-instructions to manage the internal counters and extend the pattern specification language to allow for checking of the status bits. The operations include only those that can be implemented with simple logic and executed within the cycle bounds of the decode stage. These include increment, decrement, checking of status (zero and overflow) flags, and copying a value from a HPS-internal register to a counter.

Figure 6.6 shows the extended DISE production specification that enables conditional control within a DISE ACF. We extend the root production, (DISE_Production) and the Pattern production to implement conditional control. We added Conditional_Control and Conditional productions that allow the ACF writer to specify simple conditional ex-

Extended DISE Production Specifications

DISE_Production	:= **Pattern ⇒ Replacement , Conditional_Control**
Pattern	:= **Instruction_Pattern && Conditional**
Instruction_Pattern	:= [Original DISE Pattern Specifications]
Conditional	:= **([!]** (overflow_N | *zero_N*)**) [(** || **|** && **) Conditional]**
Replacement	:= null **|** [Original DISE Replacement Specifications]
Conditional_Control	:= **(**null | *inc_N* | *dec_N* | *set_N(dise_reg)***) [,Conditional_Control]**

Figure 6.6: HPS pattern and replacement specification grammar (extended from the original DISE production specification grammar).

pressions. These expressions check the status flags for a particular counter for overflow and zero. The microinstructions that we defined are inc_N and dec_N which increment and decrement DISE counter N, respectively (where N varies between 0 and the number of DISE-private counters). The microinstruction set_N (dise_reg) sets counter N with a value retrieved from the DISE-private register dise_reg. A production can set the Conditional_Control section to null if no conditional microinstruction is required.

Although we allow for arbitrary logic expression on the status bits associated with each counter, in actuality, the number of possible logic functions for a finite set of counters is limited to:

$$(numberOfCounters * numberOfStatusBits)^2$$

We only consider two status bits: overflow and zero. The number of counters is a design parameter, and can be as small as 2 (e.g., to implement bursty tracing), or some higher number to allow for more elaborate profiling schemes or functionality. HPS designs with a small number of counters can be efficiently implemented using a simple truth-

table stored in a small $n \times 1$ memory (n is 16 for our particular design instance). This small lookup table implements the functionality of the CT and allows us to constrain the processing delay to within the cycle bounds of the decode stage. The PT processing delay is also within the single cycle bounds of the decode stage as described in the original DISE architecture [32].

Another DISE extension that we make for HPS is to enable specification of pattern productions without replacement sequences. This change is reflected in the optional Replacement production: null. When HPS encounters a pattern production rule with a null replacement specification, the pattern match fails upon completion. As a result, HPS simply forwards the current instruction through the decode stage unimpeded, while still permitting local conditional microinstructions. This implementation (match failure upon encountering a null replacement) is key to enabling low overhead in HPS, since the 1-cycle penalty is imposed only when an instruction is replaced.

Taking Samples

HPS is flexible in that it can implement a number of different, existing or novel, simple or complex, sample triggering strategies.

One of the two ways to trigger sampling is by using an external signal such as a timer or randomly generated interrupt, or, in our case, an intelligent, phase-guided trigger. To do so requires that authorized software be able to set and unset the sampling

flag. To enable this, we use a variation of the *Aware ACF* described in [32]. We define an HPS production that is matched by a special purpose no-op instruction (usually one of the reserved instructions in the instruction set architecture (ISA)). We refer to this instruction as on_trig_inst. The execution of this special instruction will cause the sampling flag (in this instance the zero status flag of counter number 0) to be turned on.

$$P0 : T.OP == on_trig_inst \rightarrow set_0(HPS_R0), null \tag{6.1}$$

This production specifies the pattern $T.OP == trig_inst$ as a successful match. $T.OP$ is an attribute of the instruction being decoded that identifies the operation code of the instruction. On a match, HPS attempts to use the replacement $(set_0(HPS_R0), null)$. $set_0(HPS_R0)$ is a conditional microinstruction which causes HPS to copy the contents of the DISE-private register HPS-R0 to counter 0. HPS_R0 holds the constant 0. Since the replacement also contains $null$, the production fails and the original instruction executes unimpeded. We can use a different special instruction to turn off the sampling flag (in this case the HPS register HPS-R1 holds the value 1:

$$P0 : T.OP == off_trig_inst \rightarrow set_0(HPS_R1), null \tag{6.2}$$

(We can also use the microinstruction inc_0 to increment the counter and implicitly clear the zero flag causing sampling to stop).

Profile consumers make use of these status bits as part of their profiling (instrumentation) productions. For example, if we are interested in profiling method invocations,

we use the following production:

$$P0 \quad : \quad T.OPCLASS == proc_call \&\& zero_0 \rightarrow null, RO$$

$$R0 \quad : \quad call(HotMethod_Handler); T.INSN$$

In the above example, whenever the instruction of interest is encountered (procedure call) *and* if the zero flag of HPS counter number 0 (also referred to as the sampling flag) is true (on), the replacement sequence R0 is streamed into the executions stream. R0 is simply a call to the profile handler and the original matched instruction (T.INSN). When the sampling flag is off (i.e. zero_0 is false), the pattern of P0 will fail and the original instruction is forwarded through the pipeline unimpeded.

HPS imposes no profiling overhead on the executing program, weather direct or indirect, when the sampling flag is unset (i.e. sampling is turned off). The only source of overhead HPS imposes on program execution is the *profiling overhead*. Profiling overhead is the cost of executing additional instructions for profile collection. The amount of profile overhead imposed by HPS depends on the duration of the sampling period and the profile type. The profile type dictates which instructions are inserted into the code and the points at which this code is inserted. We next describe how to specify productions for different profile types.

Specifying Profile Types in HPS

HPS is flexible in that it can implement any instruction-based profiling technique by specifying an ACF for each profile type. We use HPS to implement three different profile types: hot code region, hot call pair, and hot method. These profile types are widely used in dynamic and adaptive optimization systems [9, 28, 53].

Code regions are profiles that estimate basic block behavior and are generally efficient to collect. Each event in the profile is a dynamic branch; the data value for which is the cumulative number of instructions since the previous dynamic branch. Method profiles estimate the time spent in a method; for each method, we record the number of invocations as well as the number of backward branches taken. Call pair profiles are invocation counts for each caller-callee pair executed. We use the term "hot" to indicate that we are interested in the events with the highest values (i.e. that are most frequently occurring).

We show the HPS productions for each of these profile types in Figure 6.7. As before, P identifies a pattern production and R identifies a replacement production. These ACFs execute no conditional microinstructions. They do employ conditional replacement however – only when the zero status register of HPS internal counter 0 is set, will HPS replace a matched instruction. These productions are controlled by the external trigger sampling strategy described above.

Profile Type Productions

Hot Code Region Analysis
P1: T.OPCLASS == branch && zero_0 ⇒ R1, null
R1: call(Branch_handler,T.INSN);
T.INSN;

Hot Method Analysis
P2: T.OPCLASS == proc_call || (T.OPCLASS ==branch && T.PC < T.Target) && zero_0 ⇒ R2, null
R2: call(HotMethod_handler,T.INSN);
T.INSN;

Hot Call Pair Analysis
P3: T.OPCLASS == proc_call && zero_0 ⇒ R3, null
R3: call(CallPair_handler,T.INSN);
T.INSN;

Figure 6.7: Pattern and replacement productions for the three different profile types that we investigated for remote performance profiling of deployed, embedded device software: hot code region, hot method, and hot call pair profiling.

Currently we insert a call to the profile collection routine for each profile type. The typical size of a profile handler is a few hundred bytes. As such, several profile types can be implemented at once. We consider only a single profile type at a time in our evaluation of HPS. The only additional change required to enable collection of multiple profile types at once is to have multiple profile productions active simultaneously while merging the pattern specifications and replacement sequences of overlapping profiles.

6.3 Evaluation

In this section, we evaluate the efficacy of our remote profiling system. We first present an evaluation of our DISE extensions that enable HPS, and of the accuracy and overhead of our system using a general-purpose benchmark suite and processor. In

Section 6.3.2 we evaluate our system in the context of a resource-constrained device, its applications, and its power consumption.

6.3.1 Phase-aware Profiling for General-purpose Programs

Methodology

We employ a cycle-accurate simulation platform and simulator parameterization identical to that used in the original DISE studies [30, 31, 32, 33]. The platform is an extension to SimpleScalar [23] for the Alpha processor instruction set and system call definitions.

Our simulation environment models a 4-way superscalar MIPS R10000-like processor. It simulates a 12 stage pipeline with 128 entry reorder buffers and 80 reservation stations. Aggressive branch and load speculation is performed and an on-chip memory with 32KB instruction and data caches and a unified 1MB L2 cache is modeled. The DISE mechanism (and HPS system) is configured with 32 PT entries and 2K RT entries each occupying 8 bytes. We also modified the simulator to emulate the capture of phase information as described in [96] with an interval size of 10 million instructions. We extend the DISE simulation engine to export the semantics of the conditional controls that we define in Section 6.2.

To generate the instructions for profile collection using each of our three profile types (section 6.2.2), we write the code using the C language and compile it for our

Table 6.1: Select benchmark statistics relevant to the profiles collected

	method Count	Call Sites	Call Pairs	Call Count (millions)	Dynamic Branch Cnt (millions)	Dynamic Instructions (millions)
bzip2	106	245	245	44.6	1,006	4,546
crafty	165	792	792	43.4	496	2,542
eon.cook	600	2,032	2,068	36.9	198	1,720
eon.kajiya	603	2,039	2,076	183.3	998	8,564
eon.rushmeier	603	2,049	2,086	54.3	296	2,497
gap	487	1,779	2,672	18.8	167	723
gcc	1,234	7,565	7,671	22.1	317	1,199
gzip	113	218	218	25.5	351	1,531
mcf	118	218	218	238.2	2,040	8,829
parser	338	1,149	1,151	82.0	690	2,798
twolf	238	1,120	1,120	104.3	1,663	12,442
vpr.place	180	627	627	13.9	169	1,466
vpr.route	264	1,055	1,055	84.5	1,152	10,238
Average	388	1,607	1,692	73	734	4,546

target platform (Alpha EV6). We hand-optimize the generated assembly to ensure compactness. We also insert a no-op instruction to simulate the single cycle stall associated with each replacement. The no-op instruction produces the desired result since the stall due to DISE replacements only affects the decode stage (i.e. delays the propagation of the single instruction that is macro-replaced) and does not impact the later pipeline stages.

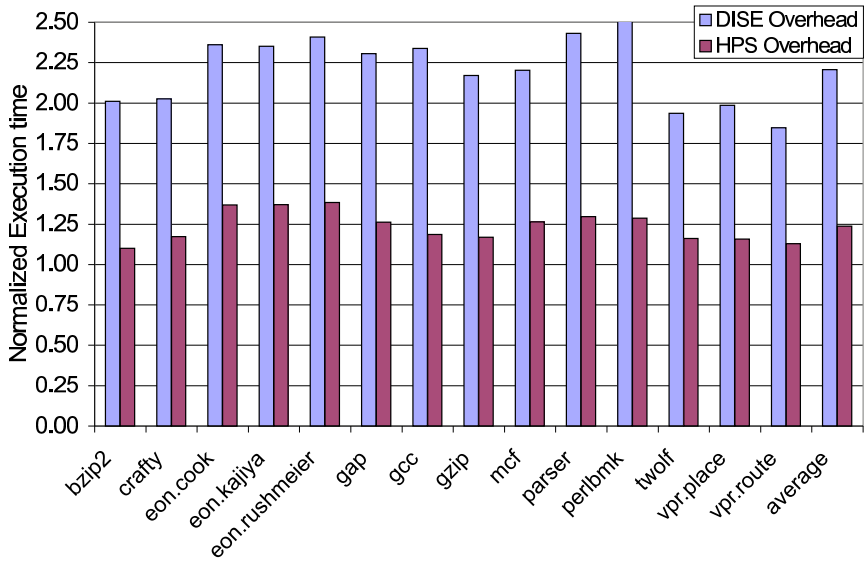
We evaluate the performance and profile quality of our system using the benchmarks of the SPECINT2000. We compile the benchmarks for the Alpha EV6 platform using GCC 3.2.2 with the -O4 optimization flag. We report results for complete runs on the train input set. We considered other inputs and found the trends in our data to be the same as that we present herein.

Figure 6.1 shows some of the dynamic behavior metrics for the SPECINT2000 benchmarks used in our empirical studies. *Method count* is the number of unique methods in the benchmark while *Call sites* is the number of static call operations. *Call Pairs* is the number of unique call site and target address pairing observed in the dynamic execution of the benchmark. The extra number of call pairs beyond the number of call sites indicates a number of indirect jumps. *Call count* is the number of dynamic calls made during the benchmark's execution. *Dynamic Branch Count* and *Dynamic Instructions* is the the number of branches and instructions executed, respectively. These dynamic metrics are relevant to the performance profiles we investigate: hot code region (branches), hot method, and hot call pair profiles.

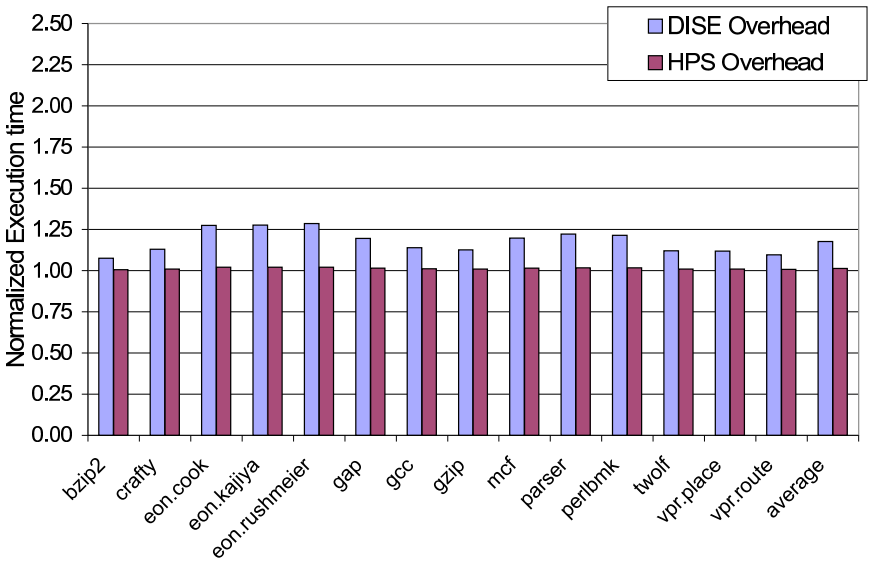
HPS Performance

We first evaluate the performance impact of our HPS extensions that enable conditional control within the DISE engine. Figure **??** shows the overhead of sample-based hot code region (top graph) and hot call pair (bottom graph)profiling using DISE without conditional controls (left bar) and with our HPS optimizations (right bar). Each bar is the execution time for each program normalized to execution without DISE and without profiling. For this data, we use periodic sampling at a frequency of 1/100 events.

The data shows that HPS (i.e. DISE with conditional controls) significantly reduces the overhead of a naive, DISE sampling strategy. The overhead that HPS introduces is



Hot Code Region (Branch) Profiling



Hot Call Pair Profiling

Figure 6.8: DISE vs. HPS for performance sampling: The overhead introduced by each of the individual profile types. The DISE data is the overhead of sampling without our HPS extensions (i.e. moving sampling flag manipulation and checking into the DISE engine).

profiling overhead alone – the cost of executing the extra instrumentation instructions for each event of interest. The DISE data includes this overhead as well as the basic overhead for manipulating the sampling flag and the cost of checking that flag as part of the unconditional replacement sequences.

On average, DISE sampling introduces a 120% increase in execution time for hot code region (branch) profiling, while HPS only introduces a 24% increase. For hot call pair profiling the overhead of DISE sampling is 18%, while HPS's overhead is only 1.3%. Hot method profiling exhibits performance characteristics that are similar to hot call pair profiling.

Phase-aware Profiling Efficacy

To evaluate the impact of remote profiling, we examine its accuracy versus its overhead using four different profile collection strategies:

- Exhaustive - gather an exact profile for each interval. We use this strategy to evaluate the accuracy of the other policies.
- Periodic Sampling - gather a profile every Nth interval, for N in [3,100].
- Random Sampling - gather a profile for interval i with a probability of 1/P for some P

- Phase-based - gather a profile for every interval that is dissimilar from all previously gathered intervals, given some threshold of similarity.

For the periodic and random strategies, we gather data for different sampling frequencies. The number of intervals we profile, and therefore the percentage of total execution that we profile, depends on the sampling period N. We perform experiments for a range of sampling frequencies which correspond to a range of overheads and accuracies. Because a truly random strategy is at the whim of chance as to whether or not it performs well, we characterize two aspects of random profiling for each of the different percent-sampled values: 1) we compute the average error across 10 runs (avg random), and 2) we compute the maximum error seen across 10 runs (max random). To get a range of accuracies and overheads, we adjust the parameter P and examine the effect.

Selection of Phase Representatives For phase-aware profiling, as we have described previously, we begin with an implementation of the phase prediction system that we have developed in prior work [96]. We identify empirically, the best interval from each phase to act as the representative from that phase. Figure 6.9 shows the percentage error at 1% sampled for four different representative selection policies for phase-based profiling. The y axis shows percentage error in basic block counts. The different polices for representative selection that we evaluate are (a) `first`: select the first interval

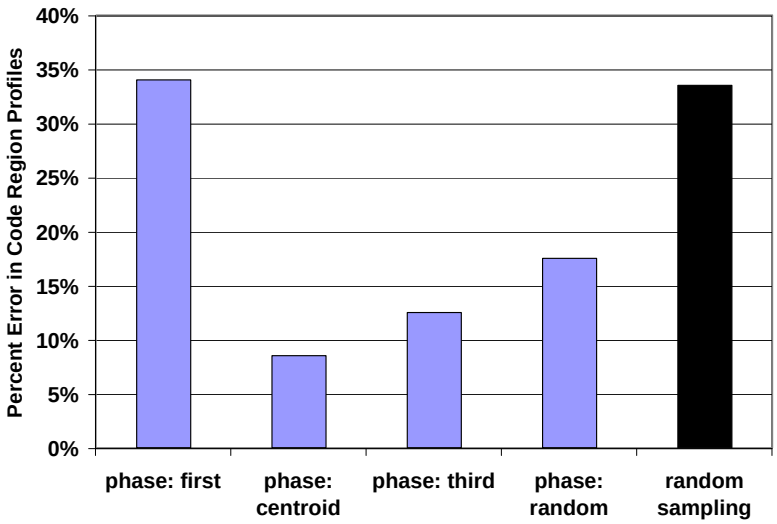


Figure 6.9: Evaluation of representative selection policies. The graph shows the average error in block counts at 1% sample rate for different representative selection schemes. The black bar shows the performance of average random sampling (non-phase-based).

as the representative, (b) `centroid`: select the centroid of the intervals in the phase as the representative, (c) `third`: select the third interval as the representative. (d) `random`: randomly select one representative from all intervals in the phase (we report performance for this strategy as the average performance of 5 selections). For comparison, we also show the error that the random sampling strategy, that we describe above, produces. This strategy chooses random samples from the entire program (black bar on far right).

As expected, the centroid method, `centroid`, performs the best: its error remains low even when we sample very little of the program. `First` and `random` perform the worst. This happens since the first is not representative of the steady state (the phase is

just warming up) and because selecting randomly can result in selection of a representative that is dissimilar to all others. `Third` enables accuracy that is between that of `best` and `first`/`random`. That is, `third` is able to select an interval that is more representative of the steady state of the phase than `first` and `random`. Moreover, `third` is simple and can be implemented without additional overhead. As such, we use `third` for the rest of the results in the paper.

Profile Accuracy and Overhead Unlike the random and periodic sampling approaches, there is no sampling frequency variable that we can vary to get different tradeoff points between accuracy and overhead. Instead, we achieve a similar effect by dynamically tuning the *similarity threshold*. The similarity threshold determines the cutoff point at which two intervals are said to be similar and hence are part of the same phase. As we lower the threshold, the system detects more unique phases, each with a fewer number of intervals. As this occurs, the system takes more samples (since there are more unique phases). This both increases the percentage of the program's execution that the system samples and improves the accuracy of the resulting profile.

To measure profile accuracy, we compare each sampled profile to the exhaustive profile. We compute the percentage error in the code region profiles as our accuracy metric. The code region profiles contain counts for each dynamic branch. We compute the element-wise difference in branch counts between a sampled profile and the ex-

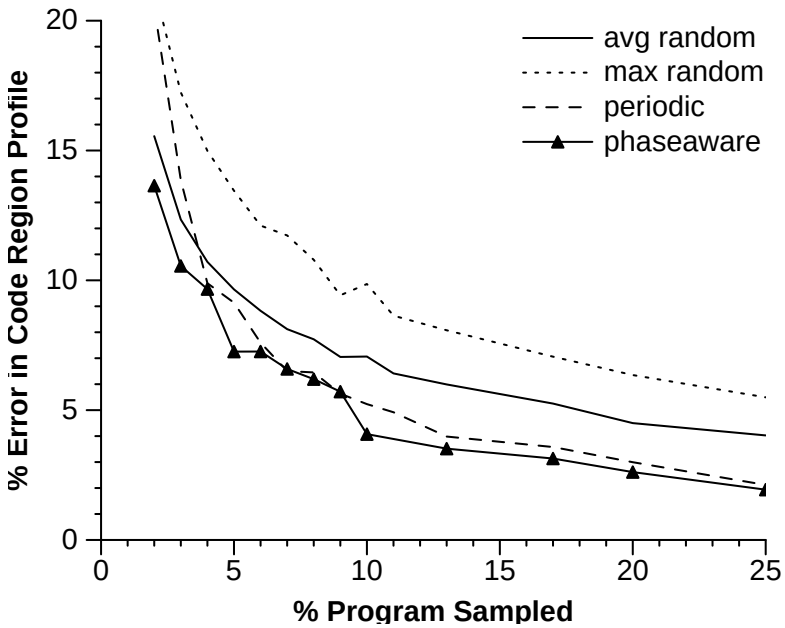


Figure 6.10: Average error in code region profile for various sampling percentages.

haustive profile. We then divide this value by the total counts in the exhaustive profile to produce the error percentage. The best sampling strategy is the one that produces the least amount of error for the smallest percentage of the program that is sampled.

Figure 6.10 shows the average error across benchmarks using the third interval of each phase as the phase representative. The graph compares the accuracy of each of the different sampling strategies, `avg random`, `max random`, `periodic`, and `phase-aware`. The y-axis is the error percentage in total branch counts (not just the hot branches which we study later) and the x-axis is the percentage of the program that was sampled for a given parameterization of each strategy.

The data indicates that on average, phase-aware profiling achieves lower error for small percent-sampled values than the other strategies. The errorfor periodic sampling approaches that of phase-aware sampling for some percentages sampled – both approaches require 10% of the program to be sampled to achieve 5% or less error. The random strategy requires that 20% of the program be sampled to achieve an error of less than 5%. For lower percent sampled values, the benefits from our system increase substantially.

Other Profile Types We next demonstrate that phase-aware sampling is not restricted to any single profile type by showing that it performs well for others. We evaluate the efficacy of each of the sampling strategies in identifying frequently executing parts of the program. Profiles that capture frequently executing parts are commonly used for feedback directed optimization, e.g., hot code regions, hot methods, and hot call-pairs, and as such are important profile types for our distributed optimization system. We measure the error produced by each of the profiling strategies for these profile types. We define "hot" as the top 15% of the most frequently executed events.

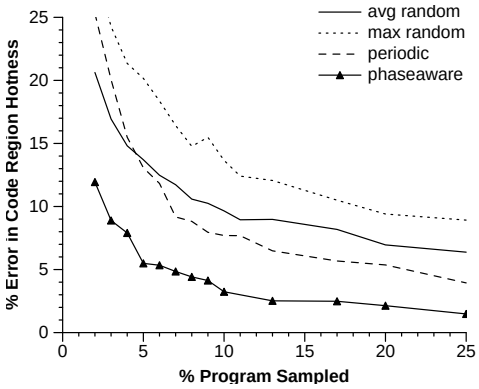
Figure 6.11 shows the results. The x axis is the percent of the program that was sampled, and the y axis is the percentage error in identifying hot branches, hot call-pairs and hot methods, on average across benchmarks. We omit max random and average random data from the hot call pair graph since both were in a range significantly larger

than the other strategies. Average random ranges from 33-53% error and max random ranges from 49-64% error.

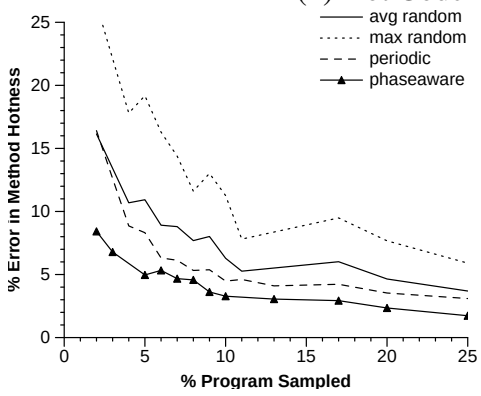
The graphs show that the phase-aware strategy performs considerably better than both random and periodic sampling for all three profile types. Assuming an error of 5%, the phase aware strategy needs only to sample 5% for hot methods and hot branches, and 20% for hot call pairs. Periodic sampling performs similarly to phase-aware sampling for call pairs, but requires that we sample 10% of the program for hot methods and 20% of the program for hot branches. Random sampling rarely achieves an error of 5% or less; however, it does so for hot methods for which it requires that we sample 20% of the program.

In Figure 6.2, we quantify these overhead percentages (assuming 5% error) for individual benchmarks using three tables. The top table is for hot code region (dynamic branch) profiling, the middle table is for hot method profiling, and the bottom table is for hot call-pairs. We chose 5% error as our cutoff arbitrarily; however, we selected a value that we believe to be tolerable and amortized by commonly used, profile-based, dynamic optimizations. For lower error values, the benefits of our system increase significantly since phase aware profiling is able to extract unique and important program behaviors by sampling only a very small portion of the execution.

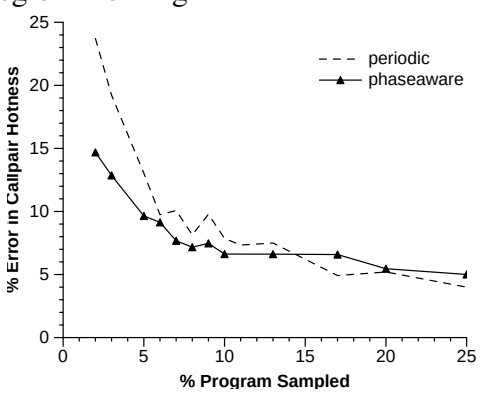
Each table reports data for the three profiling strategies: phase-based, periodic, and (averaged) random sampling. For average random, if the error does not reach 5% or



(1) Hot Code Region Profiling



(2) Hot Method Profiling



(3) Hot Call-pair Profiling

Figure 6.11: Efficacy of different sampling strategies for different profile types. We omit max random and average random data from the hot call pair graph since both were in a range significantly larger than the other strategies. Average random ranges from 33-53% error and max random ranges from 49-64% error.

less, we use a percent sampled value of 25%. The data in columns two, three, and five show profiling overhead in millions of cycles for each of the benchmarks. Columns four and six show the percent reduction in this overhead due to phase-aware sampling.

The final row in each table shows the average across benchmarks. For hot code regions, phase-aware profiling reduces the overhead of periodic and random sampling by 76% and 80%, respectively, on average. This reduction is 50% and 76%, respectively, for hot methods. The primary reasons for these benefits are two-fold: (1) Random and periodic sampling collect redundant information (i.e. profiles of the execution that they have already collected); and (2) random and periodic sampling techniques miss important behaviors (which degrades accuracy) which phase-aware sampling is able to capture.

For hot call pairs, phase-aware sampling reduces the overhead of random sampling by 22% on average. However, as visible in the graphs in Figure 6.11, phase and periodic sampling perform similarly for this profile type. For some benchmarks, periodic sampling for hot call pairs outperforms phase-based sampling. One reason for this is that the absolute number of hot call pairs is very small for most benchmarks. As a result, one or two missed pairs can result in a significant increase in percentage error. This is the case for bzip2.

Table 6.2: Overhead of different sampling strategies for different profile types, assuming a 5% error rate.

	Phase Cycles (Mil)	Periodic Cycles (Mil)	Phase % Red.	Random Cycles (Mil)	Phase % Red.
bzip2	1335.98	6906.88	81	8625.32	85
crafty	866.53	3459.94	75	4340.26	80
eon.cook	372.41	1400.25	73	1765.72	79
eon.kajiya	1790.06	6964.39	74	8763.70	80
eon.rushmeier	484.49	2082.44	77	2637.47	82
gap	204.89	2082.44	90	1448.54	86
gcc	604.27	2264.32	73	2849.29	79
gzip	628.00	2495.98	75	3130.52	80
mcf	3313.47	14226.94	77	18139.31	82
parser	1371.59	4885.99	72	6067.17	77
twolf	3303.95	11619.15	72	14629.74	77
vpr.place	308.05	1220.19	75	1524.14	80
vpr.route	2260.67	8066.98	72	10088.30	78
Average	1295.72	5205.84	76	6462.27	80

(1) Hot Code Region Profiling

	Phase Cycles (Mil)	Periodic Cycles (Mil)	Phase % Red.	Random Cycles (Mil)	Phase % Red.
bzip2	94.37	156.53	40	365.39	74
crafty	61.45	131.70	53	334.41	82
eon.cook	61.01	113.71	46	230.01	73
eon.kajiya	286.25	555.14	48	1114.64	74
eon.rushmeier	76.02	167.36	55	332.45	77
gap	17.18	61.27	72	118.11	85
gcc	41.97	75.64	45	148.63	72
gzip	45.67	96.56	53	207.54	78
mcf	260.21	787.47	67	1582.23	84
parser	160.04	268.31	40	549.37	71
twolf	211.17	344.37	39	693.21	70
vpr.place	20.67	42.84	52	86.63	76
vpr.route	160.51	259.28	38	523.59	69
Average	115.12	235.40	50	483.56	76

(2) Hot Method Profiling

	Phase Cycles (Mil)	Periodic Cycles (Mil)	Phase % Red.	Random Cycles (Mil)	Phase % Red.
bzip2	504.25	371.16	-36	444.06	-14
crafty	329.51	347.49	5	435.76	24
eon.cook	287.59	291.73	1	367.80	22
eon.kajiya	1517.56	1467.67	-3	1845.23	18
eon.rushmeier	437.67	432.63	-1	553.07	21
gap	124.44	432.63	71	185.91	33
gcc	166.84	179.51	7	225.19	26
gzip	172.60	199.69	14	247.96	30
mcf	1478.24	1865.42	21	2505.53	41
parser	669.34	648.16	-3	822.04	19
twolf	811.05	837.31	3	1057.49	23
vpr.place	114.44	115.26	1	142.37	20
vpr.route	678.25	676.85	0	850.45	20
Average	560.91	605.04	6	744.84	22

(3) Hot Call Pair Profiling

6.3.2 Phase-aware Profiling for Embedded Devices

To investigate the efficacy of phase-aware remote profiling for embedded devices, we also evaluate our approach for a popular hand-held device processor, the Intel StrongARM.

Methodology

We employ SimpleScalar to emulate a StrongARM processor. As in the prior section, we modify the simulator to emulate the capture of phase information as described in [96] with an interval size of 10 million instructions. The authors in [67] explore the overhead/accuracy tradeoffs and efficacy of using variable sized intervals. Our large, fixed-size intervals are practical for resource constrained systems since they minimize sample storage and maintenance overhead and simplify phase prediction. Since we are interested in the efficacy of remote profiling for mobile devices, we estimate the overhead of our system in terms of power consumption.

We evaluate our system using six benchmarks from the MediaBench benchmark suite [69], a suite designed for the empirical evaluation of media applications. The benchmarks we use include encoding and decoding programs for mpeg (movie), jpeg (picture), and gsm (voice). We show the basic statistics for the programs and the inputs we used in this study in table (a) of Figure 6.3. The second column in the table is the number of static branches in the program, which correlates with the size of the

Table 6.3: StrongARM methodology. (a) shows the general MediaBench benchmark statistics. (b) shows the empirical data that we use to estimate energy consumption. The second column is Joules per second and the final column is instructions per second for the instruction types and bandwidth for wireless transmission.

		Dynamic Statistics				
Benchmark	Static Branches	Branches (Millions)	Instructions (Millions)	Cache Miss Rate	Energy (Joules)	Time (seconds)
gsmdecode	572	182.05	1610.05	0.000	29.93	16.35
gsmencode	748	79.42	2562.29	0.000	29.38	19.93
jpegdecode	930	111.76	1421.33	0.006	46.50	21.87
jpegencode	1175	433.70	4218.60	0.002	100.65	51.41
mpegdecode	1104	309.95	3007.85	0.001	65.03	900.63
mpegencode	2216	244.25	4196.19	0.001	52.63	282.40
Average	1124	226.86	2836.05	0.002	54.02	215.43

(a)

Instr Type	Average Joules / s	Instr / s (Millions)
IREG	0.865	204.790
IMEM-R	0.973	19.462
IMEM-Rcache	0.000	137.510
IMEM-W	1.340	11.625
FPREG	0.965	0.439
Wireless Card	Specification 5V*0.285A	Max Bandwidth

(b)

branch profiles generated. The next five columns show the dynamic statistics: number of branches executed (in millions), number of instructions executed (in millions), the cache miss rate assuming a 64K, 4-way set associative, instruction and data cache, the energy consumed by executing the program (in Joules), and the execution time (in seconds). Since the inputs that are provided with MediaBench are very short, and because these applications are typically used in a streaming fashion, it was necessary to find more substantial inputs to analyze the realistic long term effects of profiling. We plan to make these inputs available via our web page.

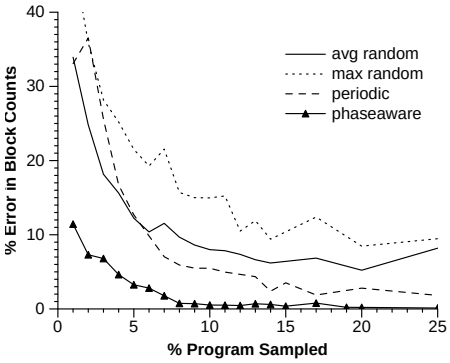
To compute energy consumption and execution time, we use a model that we generate from an actual hardware system. We compute the energy (Joules per second) consumed per-instruction (including events such as cache misses), per-byte-transmitted energy consumed, and instructions per second. We summarize the values in table (b) of Figure 6.3. We generate these values using an HP iPAQ H3835 running Familiar

Linux v0.6.1, a Lucent/Orinoco Gold wireless card, and hand-coded benchmarks. We periodically (every 10 seconds) measure battery voltage and current levels from those exported via the Linux /proc/battery interface. We calibrate our model and validate it using a variety of benchmarks in [65].

In the table ((b) in Figure 6.3), we report the average Joules per second consumed by each of these latter, single-instruction programs (IREG: integer register operations, IMEM-R: load operations that miss in the L1 cache, IMEM-W: store operations that miss in the L1 cache, and FPREG: floating point operations). We compute instructions per second of each benchmark in a similar fashion using the instructions per second measurement of each constituent instruction type (reported via simulation). To compute the power consumption for transfer, we compute the number of Joules per byte transfered (assuming 11Mb/s bandwidth) using the specifications of our wireless card. We show the Joules per second of transfer in the final row of the table.

Profile Accuracy and Overhead

We first evaluate the accuracy of the different sampling techniques for the StrongARM benchmarks and environment. Figure 6.12(a) shows the percentage error between the sample-based and exhaustive code region profiles for each of the profiling strategies. As before, we calculate the error in branch counts for the different sampled



(a)

Sampling Overhead For Accuracy Error of 5%				
% Sampled:	Periodic: 11%	AvgRand: 20%	Phase: 4%	
Energy				
	Joules		Percent Reduction	
Protocol	Periodic	AvgRandom	vs Periodic	vs ARand
At End	7.75	15.02	58.97	78.83
Interleaved	8.24	16.03	58.90	78.86
Computation Overhead				
	Instructions Executed (Millions)		Percent Reduction	
Protocol	Periodic	AvgRandom	vs Periodic	vs ARand
At End	265.61	514.20	58.84	78.74
Interleaved	280.48	545.46	57.58	78.19
Communication Overhead (Compressed)				
	Bytes Transfered		Percent Reduction	
Protocol	Periodic	AvgRandom	vs Periodic	vs ARand
At End	2217.83	2217.83	0.00	0.00
Interleaved	27095.50	51683.97	51.36	74.50

(b)

Figure 6.12: Evaluation of phase-aware sampling using the StrongARM environment and benchmarks. The graph in (a) shows the error in branch counts (code region profiling) over the percentage of the program sampled for each of the four sampling strategies. The table in (b) shows the impact of each of the sampling strategies (we omit max random) on energy, computation, and communication when we transmit the samples all at once at the end (At End) or intermittently during execution (Interleaved). On average, phase aware sampling eliminates 51 to 79% of the overhead imposed by the periodic and random strategies.

percentages of program execution. The y-axis is error and the x-axis is percent of the program that was sampled for a given parameterization of each technique.

The graph shows that on average, phase-aware profiling results in significantly lower error for a very small sampled percentage than both random and periodic profiling. The difference between the phase-aware and periodic strategies for this empirical setup (embedded device) is more pronounced than it was for the general purpose applications (section 6.3.1). Phase aware sampling produces very high accuracy profiles, e.g., less than 5% error, by sampling a very small amount of the program's execution (4%). To achieve the same accuracy, periodic sampling requires that 11% be sampled, average random sampling requires that 20% be sampled, and max random is never able to achieve an error of less than 5%.

Impact on Power

We next evaluate the overhead of our system in terms of power consumption for each of the sampling strategies, assuming a maximum error of 5%. We omit max random since it is unable to achieve an error of 5%. We calculate the overall power consumption for each profiling strategy for *all of the required remote profile collection functions*: computation overhead for instrumentation, communication overhead (using compression), and computation overhead for applying compression.

We present the results in Figure 6.12(b). We show how 5% error translates into energy, computation, and communication overhead in the three sections of the table. In each section, we show the average overhead for each metric across benchmarks for periodic and average random sampling in columns 2 and 3. In column 4 and 5, we show the percent reduction enabled by phase-based profiling over each of these techniques, respectively.

Each section in the table contains two rows of data for the two different communication protocols that we studied. For "At End", we combine the basic block vectors of each profiled interval into a single vector; upon program termination, we compress the vector and transmit it. Using this protocol, phase-aware profiling reduces energy consumption by 75% over random sampling. Phase-aware profiling reduces computation overhead by requiring 72% fewer instructions for instrumentation over random sampling.

Given this "At End" approach, the communication cost is the same across profiling techniques since we are communicating a single profile vector in either case (though the counts will be different). However, we investigated another protocol, one in which we compress and transmit the basic block vector after each interval. This protocol reduces the amount of device storage required (which may be highly constrained for real devices); as such, it is a realistic alternative that we should consider. Using this "Interleaved" protocol, phase-based remote profiling can also reduce communication

overhead since fewer intervals are communicated to achieve the same 5% accuracy. These results are shown in the second row of each section. The reductions in overhead for energy and computation are similar to the "At End" protocol. However, phase-based profiling requires less than $1/4$ of the number of bytes be transmitted *to communicate the same information* as the random approach.

In summary, our remote performance profiling achieves high accuracy with low overhead by using an efficient hybrid profiling support system that toggles profile collection and an intelligent phase-aware sampler that determines when samples should be taken. Our data indicates that our techniques are effective in a general purpose setting as well as for resource-constrained, battery-powered systems.

6.4 Extending Phase-aware Profiling to Multiple Users

Given a large connected user base, we can further reduce the overhead of phase-based remote profiling by providing feedback to users about phase discovery. If a user executes a program using the same input as that used by another user for which phase data already has been collected, the second execution will provide us with no new information, wasting resources needlessly. As part of our phase based remote profiling system, we performed a preliminary investigation into the use of phase IDs (those described in Section 6.1) as a *feedback mechanism to other users* so that they

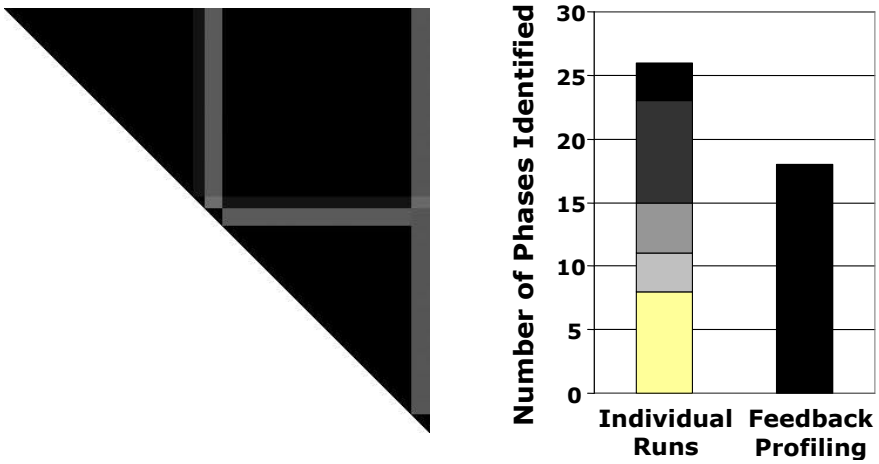


Figure 6.13: Distributed profiling across multiple executions. Similarity matrix (left) for the gsmencode MediaBench benchmark across 5 different executions using different inputs. The right graph shows the number of phases identified if we profile this benchmark with each input separately (left bar broken down by input). The right bar graph shows the number of *unique* phases. Only unique phases need to be sampled using our feedback-directed phase profiling technique.

may avoid unnecessary profiling. Such a technique requires that the Phase Tracker and device architecture be homogeneous so that a phase ID identified by the Phase Tracker for a particular program interval on one device is the same as that for the same interval on another device.

To provide *dynamic feedback* to users, we communicate phase IDs periodically, in the reverse direction. The remote profiling system on the user's device adds these phase IDs to the phase ID table in the Phase Tracker (if they are not already present) and sets their "sampled bit". As such, when a previously unseen phase is predicted, it is *only* sampled if it also has not been provided by the feedback mechanism.

By communicating phase IDs of known phases to users (so that their systems avoid sampling them), we can reduce the overhead of phase-aware sampling when users execute the same program with the same inputs. However, it may also be possible to use this technique when the same phase occurs *across inputs*. A more realistic scenario is one in which software is deployed and users execute it with a wide range of diverse inputs. Each input will cause the program to exhibit phased behavior; for some programs, some phases may be the same across inputs.

To evaluate the potential of feedback-directed remote phase profiling, we analyzed many different inputs for one of our benchmarks, `gsmdecode`. Figure 6.13 shows the similarity matrix for this benchmark. A similarity matrix is a 2-dimensional array all of the intervals in a program; each entry is the similarity value between two intervals encoded as a gray-scale value with dark values identifying similar intervals (in the same phase), e.g., the points on the diagonal are black since an interval is exactly the same as itself. The x-axis and y-axis of figure are increasing interval id's. We omit data in the lower triangle for clarity, since it is symmetric with the upper triangle. We read the figure by first selecting an interval on the the diagonal and then traversing the row. By doing so, we can visualize how similar the row interval is compared to all others that follow it during execution. By traversing the column above, we can visualize how similar the row interval is compared to all other intervals that came before it during execution.

Commonly, similarity matrices are used to analyze the execution of a benchmark running a single input [81, 96]. However, we use it here to visualize execution of *five* different inputs. We concatenate the intervals from each input and then compute the similarity between each interval in the entire set. The dark regions indicate that even across inputs there are many intervals that are very similar, i.e., there are phases that span inputs.

The graph on the right in the figure shows the number of total intervals in all five executions of `gsmdecode`. The total height of the left bar is 28, indicating the number of different phases identified if we were to execute the program with each input individually. The left bar is broken up into pieces, indicating the number of phases found for each input. The right bar shows the number of intervals that we must sample, across all five inputs, to gather all of the *unique* phase behavior: 18 phases.

The data shows that there are 10 phases (36%) that overlap across all of the inputs. This indicates that there is potential for reducing the overhead of phase-driven remote profiling further using feedback-directed profile collection. For this benchmark, we can communicate the phase IDs to the user base as each is discovered by individual users. For programs that execute a phase that has already been identified, we can avoid collection and communication of the profile.

6.5 Related Work

Our work builds upon and extends a body of related research on program phase behavior described in Chapter 2.1 [94, 95, 96, 38, 41, 39, 81, 50]. Our work is novel in that it is the first, to our knowledge, to investigate the efficacy of remote performance profiling. Moreover, we use program phase behavior to significantly improve the efficiency of remote profiling and as such, we make it feasible to gather performance characteristics about software for mobile devices post-deployment.

The other areas of research related to our work include sample-based profiling techniques, and post-deployment monitoring of software.

6.5.1 Efficient Profiling

Many other researchers have identified that an entire program need not be profiled to extract accurate execution behavior information from it. Instead, many sample-based approaches have been proposed [40, 105, 12, 9, 28]. Sample-based profiling is used to gather performance statistics about a program for use on the same device (as opposed to remotely), in compiler and runtime optimization.

Extant sample-based profiling techniques that couple hardware support for performance profiling include those that employ hardware performance counters [6, 36], and others that use special-purpose hardware to guide sampling [111, 91]. The work in [91]

is somewhat related to the research herein in that it describes a performance profiling approach that couples hardware and software in an attempt to reduce profiling overhead by using programmable hardware to capture and compress profile information before passing it on to software for analysis and exploitation. The generated profile is in the form of a single or multiple event streams. Dedicated hardware performs lossy compression on this stream, by using hardware-based low-cost sampling mechanisms, thereby reducing the amount of information that the software profiler has to process. This approach is completely orthogonal to ours, in that, it uses specialized hardware to capture and pre-process profile information as dictated by the software profiler. We are interested in using specialized hardware to drive our profiling policies.

There are many sample-based, software-only performance profiling techniques, e.g., [40, 105, 12, 9, 28]. These approaches are intended to be used within extant dynamic optimization systems. Duesterwald et al [40] present online path profiling to enable hot path prediction in dynamic optimization systems, and [105] examines several sampled based techniques to gather profiles within a Java Virtual Machine to enable feedback-directed dynamic optimization. In [12], the authors present an online, software only mechanism for sampling executing code. They use code duplication (methods both with and without instrumentation) and transfer execution between the two based on method invocation and taken backward branch (backedges) counts. They show that for sampling method call-pair frequencies and field accesses that their technique exhibits

very low overhead. In our work, we show that phase-aware profiling can capture a wide range of profile types, e.g., basic block frequencies, hot blocks, hot methods, hot call-pairs, and hot paths, with high accuracy.

6.5.2 Monitoring Program Behavior for Bug Isolation and Test Coverage

In addition to performance profiling, the focus of some related work has been on efficient and effective distributed monitoring of remote software for the purposes of error reporting, bug isolation, and code coverage. Using these techniques, information about an executing program is collected at a user site (the point of execution) and communicated to a centralized location for analysis, debugging, and further application development. Two extant techniques that perform such application monitoring include residual testing [87] and expectation-driven event monitoring (EDEM) [49]. Residual testing is the process of continual program monitoring for fulfillment of test obligations not satisfied prior to deployment. Residual testing does not address the issue of reducing the instrumentation required to monitor the residue. EDEM uses software agents deployed over the Internet to collect application-usage data. This approach addresses the problem of monitoring deployed software, but is limited in that it collects the same information from every execution at every site visited and only gathers information about certain events; it cannot collect general profile information.

Another framework that performs remote post-deployment program monitoring that attempts to reduce profiling overhead is software tomography in the GAMMA system [86, 21]. Software tomography is the process of dividing an application into subtasks then assigning instrumentation across subtasks in an effort to reduce the overhead imposed on any single task. The instrumented tasks are then distributed amongst a large number of users. The technique significantly reduces the overhead per user since only certain tasks are monitored. The process of assigning instrumented tasks to users is iterative to enable testing the application for code coverage. The authors show in simulation, that there is potential for reducing monitoring overhead for branch coverage.

The authors in [70] describe a similar sampling infrastructure for bug isolation (as opposed to code coverage) that distributes the overhead of remote profiling across many users of a single application. The system collects samples in a single program instance based on a geometric distribution that enables a statistically fair random sample. This ensures that all events (including rare ones) are accurately represented. The authors show that they are able to introduce a small amount of sampling overhead (instrumented computation and profile communication) per user yet gather enough information to aid in bug isolation.

6.6 Summary

This Chapter described phase-aware profiling and its application to the problem of efficient collection of remote profiles from resource-restricted devices. Our approach combined phase-aware sampling with hybrid profiling support (HPS).

Phase-aware sampling exploits repeating patterns in program behavior, i.e., program phase behavior, to intelligently identify execution intervals (fixed-length periods) that represent each phase. By only sampling one interval for each unique phase in the program, we can collect accurate sample-based profiles while introducing significantly less overhead. HPS is able to gather runtime profiles without code duplication or any overhead other than that of the profiling instrumentation.

We presented an extensive empirical evaluation of our approach using simulations for a general-purpose benchmark suite and execution environment, as well as for a popular embedded device. We evaluated the accuracy and the overhead of our system in terms of computation, communication, and battery power for a number of different profile types, including hot call-pairs, hot methods, and hot code regions. We compared our system to popular random and periodic sampling strategies and showed that we are able to reduce the overhead of these strategies in a general-purpose setting by 6-80% on average and in an embedded device setting by 50-75%, assuming an error rate of 5%.

The main contributions of the work presented in this Chapter are summarized below:

- New architectural features that are useful for profiling and optimizing remote connected devices such as IPAQs and cell phones. These hardware hooks, guide flexible software profiling in selecting the most important parts of program execution.

- We show that these hardware guides for sampling can be built by exploiting the concept of program phase behavior, and that profile communication can be reduced by up to a factor of 6 over random sampling (where both achieve an accuracy of 10%).

- A simple online policy for deciding when to profile that is almost as effective as one with full trace knowledge

- A demonstration that phases can be used to accurately guide the profiling of multiple different types of information (basic block profiles, hot methods, and call-pair tracing

- An empirical evaluation of these techniques for all of the overheads associated with remote profiling for resource-restricted devices (communication, computation, and power).

Chapter 7

Phase-based Instruction Prefetching

In this Chapter, we describe our work on instruction prefetching. We first present a detailed characterization of instruction cache performance across three Java server workloads running natively on an IBM Power5 multiprocessor, showing that instruction misses are indeed still a problem for large-scale server applications. We follow this characterization with a detailed study of the industry-standard SPECjAppServer2004 J2EE benchmark [34] running in a full-system simulator. We analyze the method-level behavior and potential for a phase-based instruction prefetching mechanism. As a first step towards phase-based prefetching in JVMs, we describe a simple algorithm for inserting software instruction prefetches at points in the call-chain leading to specific delinquent methods, that we call *call-chain based instruction prefetching*.

Prior work discussed a call-graph based software prefetch mechanism for instruction misses which uses caller-callee relationships, inserting prefetches for a callee at the entrance of a caller [8]. In object-oriented programs which are characterized by

small method sizes (such as WebSphere), this may result in late prefetches due to the short distance between the caller being invoked and its subsequent calls. We investigate moving prefetches up the call chain, inserting them at a greater distance from the callee. This leads to a trade off between prefetch coverage, accuracy, and timeliness. Using this simple scheme for software prefetching, we show that coverage improves as we move up the call chain, resulting in better performance improvements because of more timely prefetches. However, hoisting prefetches too far can result in a decrease in useful prefetches and increase cache pollution. Our results indicate that our call-chain prefetching enables a 31% reduction in icache misses for Java code. This reduction translates into a significant reduction of the stalls caused by instruction cache misses, resulting in a 5% improvement in overall application server performance – a significant portion of the estimated 12% performance penalty for instruction cache misses.

7.1 Characterization of Instruction Cache Behavior

This section shows that instruction cache misses are a significant source of stalls in our J2EE applications. We also perform method-level analysis of miss behavior, and examine the potential of method-level, phase-based prefetching algorithms.

7.1.1 Methodology

Our study uses a two-pronged experimental setup. We first experiment with WebSphere performance running natively on a 2-socket, 4-way Power5 multiprocessor running AIX 5.3, configured with 16GB of DRAM. We use SPECjAppServer2004 and Trade6 [18], a J2EE online brokerage benchmark internally developed by IBM. These heavily-multithreaded applications run in a three-tier configuration with a DB2 8.2 back-end tier and a client/driver front-end tier. We report results for the middle-tier only, which is running the WebSphere application server, running on top of IBM's Java virtual machine [47]. Performance counters are sampled after a 15 minute warmup period, with three 30-second sampling intervals for each set of counters. We solely focus on the L1 instruction cache in our study, which is 64KBytes, 2-way set-associative with 128-bytes cache lines.

For more detailed analysis (solely on SPECjAppServer2004), we use *Mambo* [20], a full system simulator developed at the IBM Austin Research Lab. Mambo simulates the underlying hardware in enough detail that it can run the entire software stack used in our native system (AIX 5.3, J9 v2.3, WebSphere 6.1). Figure 7.8 illustrates this experimental methodology. Using Mambo, we gather a method entry/exit trace augmented with information about cache misses, and analyze it to select relevant methods for which prefetching will be useful, and points at which we will insert prefetches (which we refer to as *prefetch points*) for these relevant methods. The trace is gath-

ered over the execution of 1 billion instructions (post-SPECjAppServer2004 warmup), which is also the length of our simulations.

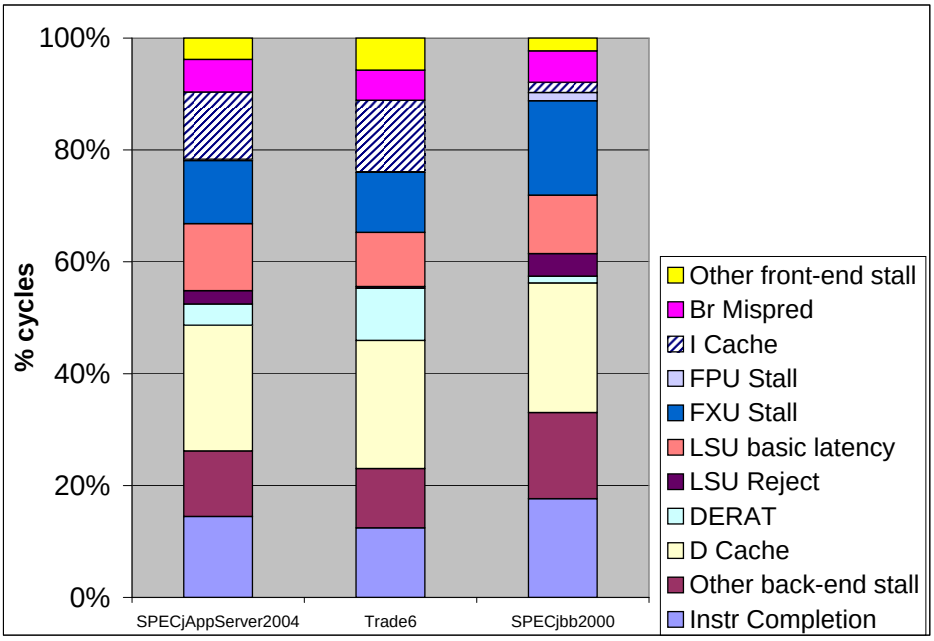


Figure 7.1: Commit Stall Cycle Categorization (icache miss stalls in striped bar).

7.1.2 Stall Cycles

The Power5 performance counter facility offers a set of counters that are incremented at each stall of the processor's commit stage, where each counter corresponds to the cause of the stall. Figure 7.1 shows a breakdown of stall cycles created using these counters, for SPECjAppServer2004 [34], Trade6 [18], and SPECjbb2000 [103]. Power5's 2-way SMT feature does a relatively good job of keeping the pipeline busy, however instruction cache misses still account for a significant fraction of stall cy-

cles (12%) for both of the WebSphere J2EE applications. It has been shown that the Power5 counter mechanism actually underestimates the performance penalty of icache misses [44]. Consequently, we consider this estimate a conservative lower bound. SPECjbb2000 exhibits only a small (2%) instruction miss penalty; we have observed similar results for SPECjbb2005.

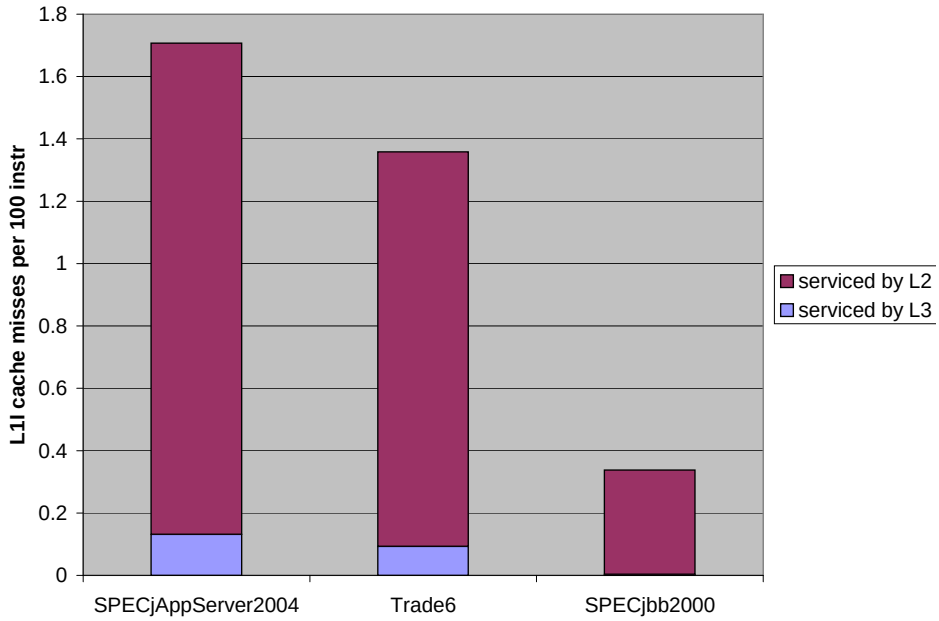


Figure 7.2: Icache misses per 100 committed instructions

Figure 7.2 shows the number of L1 icache misses per 100 committed instructions, broken down by the location from which they are serviced. The vast majority of icache misses (92% and 93% for SPECjAppServer2004 and Trade6, respectively) are satisfied from the 1.8MB L2, and nearly all of the remaining misses are satisfied from the 36MB

L3. An insignificant fraction of misses (less than 1%) are satisfied from memory or from another remote cache.

7.1.3 Method-level Analysis

Using our Mambo-based simulation methodology [20], we have also collected a profile of these instruction cache misses for WebSphere running SPECjAppServer2004, mapping each cache miss back to the individual method that caused it. In terms of high-level software components, JITTED java code accounts for the majority of icache misses (71%), followed by the AIX kernel (12%), and the J9 runtime system (7%). The remaining 10% of misses are distributed over a large set of system libraries.

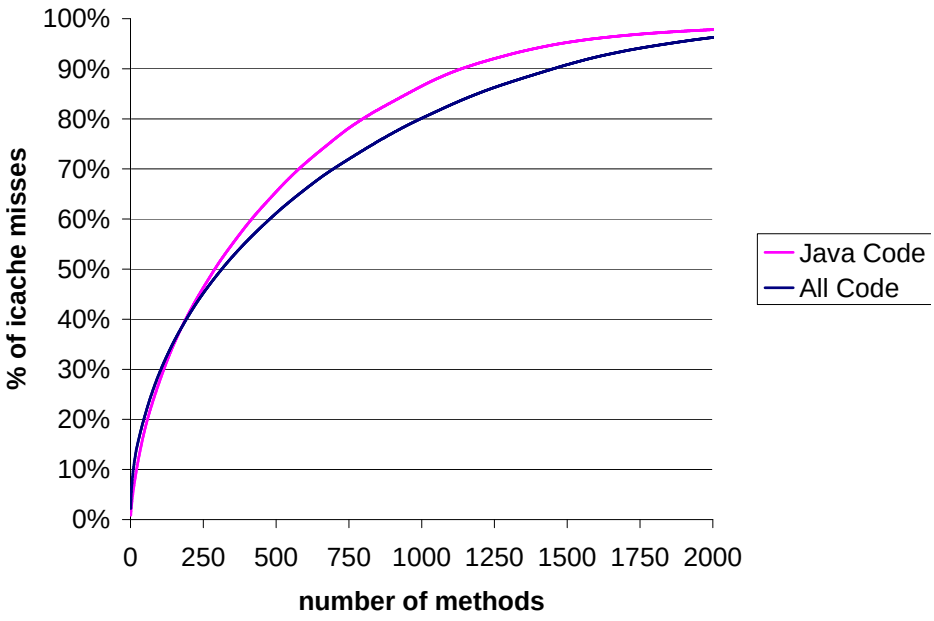


Figure 7.3: Per-method Contribution to Total icache Misses (cumulative distribution)

When taking a closer look at the individual methods in this profile, we find that the profile is extremely flat; no single method accounts for more than 2%, and the largest contributor from the Java portion of code is merely 0.525%. Figure 7.3 shows the contribution of total instruction cache misses by the number of methods causing those misses. This chart includes data for all misses, and data for misses that are caused by Java code. In both cases, 50% of all misses can be attributed to the 300 worst offending methods. In order to cover 75% of all misses, more than 700 methods must be considered. Obviously, in order for an instruction prefetching mechanism to be beneficial, it must target a large number of methods.

We analyzed the top ten Java methods with the highest number of misses in more detail. Table 7.1 summarizes this information. The columns list percentage icache misses, average per-invocation miss count, number instruction cache blocks that are ever actually touched, and the number of direct callers for each method. The last row shows the average values, averaged across all Java methods. Note that the fourth column refers to the static number of used blocks for a method and is an indication of method size. We observe that the number of callers for a method varies and is not always small. Consequently, code positioning schemes [89, 47] might not be effective in addressing the cache miss problem, because each method often has many callers. In addition, the IBM J9 JIT [47] already optimizes code layout by reordering basic blocks

Table 7.1: Prefetch Target Characteristics.

Top Methods	ICache Misses (% of Total)	Num. Invocations (across trace)	Avg. Per-invocation Misses	Num. Used Cache Blocks (Static)	Num. Direct Callers
1	0.52	15024	4.4	20	2
2	0.47	14577	2.8	14	94
3	0.45	661	54.9	64	9
4	0.44	10834	2.7	16	3
5	0.38	20740	61.2	64	1
6	0.35	37646	0.8	3	7
7	0.28	862	16.4	16	1
8	0.28	20794	7.1	9	1
9	0.27	4453	1.2	21	1
10	0.26	14577	13.0	14	2
Average (5068 total)	0.01	1439.6	3.1	4.2	2.2

such that the commonly executed code appears close in memory, thus we expect the benefits of further code reordering to be small.

These results reflect many of the facts of large-scale applications written in object-oriented styles:

- there are a large number of small methods
- each method contributes a small amount to the total execution time
- there is a substantial reuse of functionality, for example in the form of foundation libraries

We also observe that in most cases, excluding method numbers 3 and 5, the average per-invocation misses for a method is low. This bi-modal data suggests a bi-modal optimization: for methods with a small number of blocks and small number of misses per invocation, prefetches should be inserted for the entire method; for methods with a

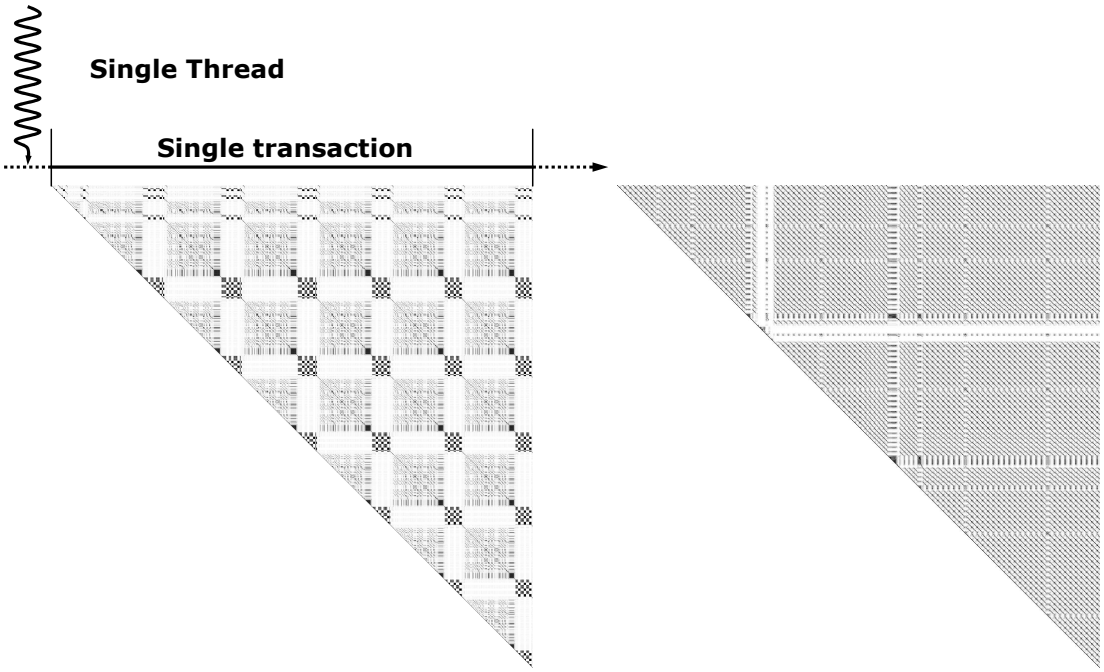


Figure 7.4: Method-level phases in WebSphere (running specjAppServer2001)

large number of blocks or a large number of misses per invocation, some of the blocks should be prefetched upon entry to the method, since there will be ample time to overlap the prefetch latency within the method. We briefly discuss such a bi-modal scheme in Section 7.3.4. For now, we adopt the all-or-nothing approach: at each prefetch point, we perform prefetching for all of the blocks that have been identified as useful (i.e. all of the method's blocks, excluding those blocks that are never used during our one billion instruction profile).

7.2 Method-level Phase Behavior

Our goal with phase-based instruction prefetching is to exploit the repeating patterns in method invocations in deciding what and when to prefetch. From column three of table 7.1, we can see that a single method is invoked numerous times, that is, it repeats numerous times, even within the short duration of our trace. To determine if this repetition exhibits patterns that could be exploited, we performed method-level phase analysis on parts of the program (we extracted transactions from the trace); we periodically gathered method vectors (similar to basic block vectors), consisting of method invocation frequencies within an interval, and generated similarity matrices to visualize the phase behavior. To compute similarity between method vectors for different intervals, we used a percentage overlap metric, which is computed as follows: given two method vectors, the percentage weight of each method is computed with respect to each of the two vectors, and the minimum of the two is added to a score, which is between 0 and 1. We chose an interval size of ten thousand instructions. All Java methods were profiled, including those from SPECjAppServer, WebSphere, and Java libraries. Figure 7.4 shows the similarity matrices for two different transactions. From this analysis, we can infer that the application under study exhibits predictable repeating patterns.

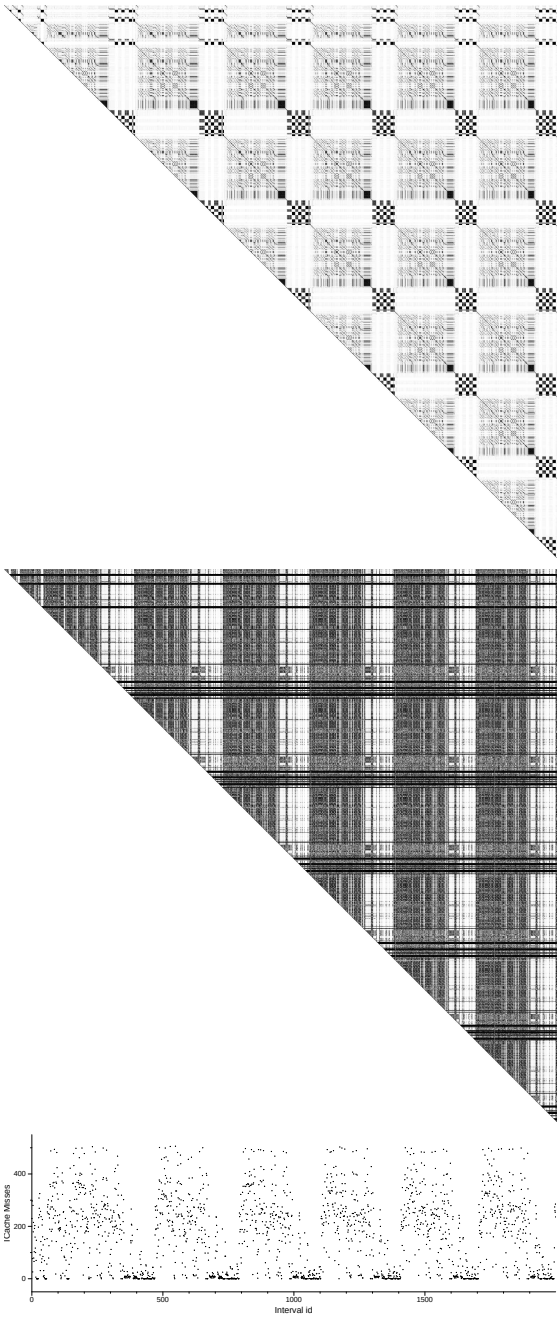


Figure 7.5: Correlation between method-level phases and phases in icache misses.

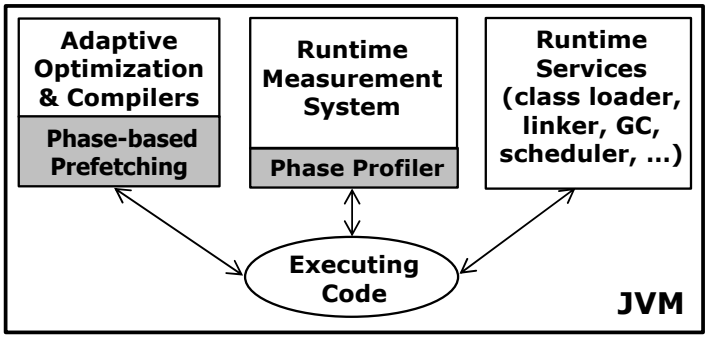


Figure 7.6: Overview of phase-based prefetching in a JVM.

An important aspect of the feasibility of a method-level scheme in driving an instruction cache optimization, is the correlation between method-level behavior and icache miss behavior. As shown in Figure 7.5, we studied this relationship by comparing the method-level similarity matrix with a similarity matrix generated using an icache miss trace for the same part of the program's execution. The icache-miss similarity matrix uses a single measurement: the number of icache misses in an interval, and computes similarity as the distance between the number of icache misses in two different intervals. In addition to the similarity matrix, the graph at the bottom of the figure plots the actual value of the number of misses. This gives us further insight into which phases suffered more icache misses.

Figure 7.6 illustrates our idea of phase-based prefetching as an adaptive optimization within a JVM. We plan to leverage existing profiling mechanisms like profiling for method invocation counts, or the recently proposed call-chain sampling technique [110] to track method-level phases, and build phase signatures. Phase signatures, along with

information about delinquent methods in that phase will be stored in a table, and used to predict, and initiate prefetching for, repeating instances of that phase.

7.3 Call-chain Instruction Prefetching

This Section describes the offline, trace-driven, method-level prefetching mechanism that we designed as a precursor to an online scheme that makes use of repeating patterns.

7.3.1 Design and Implementation

Call-chain based prefetching is an all-software profile-driven prefetching mechanism which individually targets methods that cause significant icache misses. It can be utilized in both dynamic compilation environments and statically compiled applications. Beginning with an instruction cache miss profile for a particular application, delinquent methods are first chosen as prefetch targets, based on applying a threshold value. For each target method, one or more predecessor methods in the call chain are chosen as *prefetch points*, where we define the call chain as the set of methods on the execution stack when the target method is invoked. The method prologue of each selected prefetch point is augmented with instruction cache prefetch instructions for

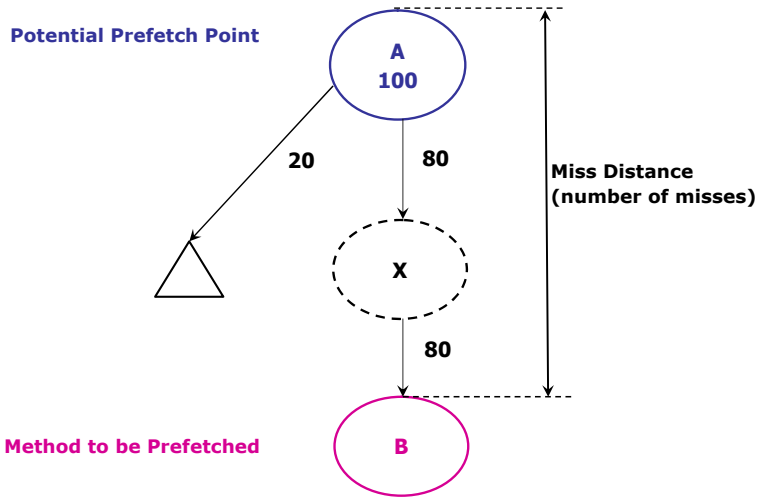


Figure 7.7: Example call chain. Edges are labeled with the number of calls from parent to child.

one or more target methods. These prefetch points are selected using the algorithm described here.

A prefetch must meet several criteria: it must be timely enough to overlap the latency of the prefetch request with other work before the prefetched block's use, but not so far in advance that the prefetch request is evicted from the cache prior to its use. The confidence of the request must also be high to reduce the risk of cache pollution with data that will never be used. In order to maximize confidence and timeliness, our profiling mechanism includes a means of approximating each.

Confidence

When a method is invoked, the control flow in its body governs which callsites are executed, and as a result, which methods are subsequently invoked. Consequently, different call-paths can be followed for different invocations of a method. Confidence of a prefetch point refers to the probability of reaching the target method on a call path beginning with the invocation of the prefetch point method. From the partial call graph shown in Figure 7.7, given a call-chain G containing a prefetch point at node A and prefetch target at node B, confidence is the probability of reaching B by following a call path starting from A. The confidence of prefetch point A in this case is 80%. When choosing prefetch points for a method, points with high confidence values will minimize the amount of cache pollution due to bad prefetches, because it likely that a target will be called if a prefetch point is reached.

Profile collection of the confidence metric is straightforward given a call graph profile [99, 10] or calling context profile [13, 110]. Because we must only estimate the set of methods that are on the stack when a prefetch target is called, a complete call graph or calling context tree is unnecessary; consequently the overhead of confidence profile collection should be more efficient.

Miss Distance

Miss distance is a parameter that we use to ensure prefetch timeliness. We define miss distance as the number of instruction cache misses on the path between the entry for the potential prefetch point and the entry for the target method. This metric is an indication of the amount of changing code and thus replacement demand on that path. When choosing prefetch points, we can selectively filter the set of candidates by measuring the average miss distance from the potential prefetch point to the prefetch target. If we require an average miss distance of at least one between the prefetch point and the target, we will usually be able to ensure that the prefetch will not be late (the miss latency can be used to fulfill the prefetch). A suitable upper bound is also chosen to avoid prefetches that are likely to be displaced prior to reaching the prefetch target.

Profile collection of the miss distance metric is slightly more difficult than the confidence metric, however there are at least two means by which it can be obtained. Using performance counter hardware that include an instruction cache miss counter (which most processors provide), one can instrument the application to read the counter value at the entry of a prefetch point, and read again at the entry of a prefetch target. Based on the difference of these values, miss distance is calculated. Alternatively, one can collect this profile for all points simultaneously using cache simulation, as described in Section 7.3.2.

We use these metrics to select prefetch points from predecessors in the call-chain for the method to be prefetched. Note that the prefetch points selected after applying the criteria described above form a subset of all possible prefetch points for the target method. Depending on the control flow and thresholds chosen, the selected prefetch points might not trigger prefetching for every instance of the prefetched method. For example, given a target method that is rarely called from any other methods, it is possible that no prefetch points will be chosen for the target. We next empirically evaluate the efficacy of our all-software approach to prefetching. We describe the methodology we used to collect the results in this section, and then present our performance data and analysis.

7.3.2 Experimental Methodology

For the performance data collected in this section, we utilize one of Mambo's timing simulators that models the processor and system microarchitecture of the IBM Power5. This timing model has been validated to be cycle-accurate with respect to Power5 hardware within a 2% margin of error averaged across a suite of benchmarks [108]. The publicly available attributes of the machine are detailed in prior work [97]. Most pertinent to this research is its 8-way set associative 64KB instruction cache, with 128 byte cache blocks. The instruction cache is limited to two outstanding misses, and uses a pseudo-LRU tree-based replacement algorithm. Instruction prefetch is modeled by in-

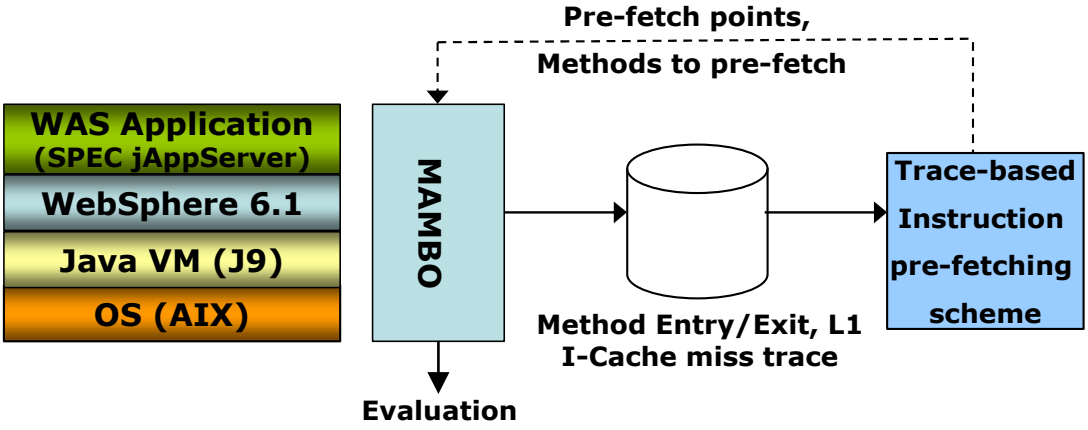


Figure 7.8: Trace-based Analysis Methodology

serting the prefetch address into a 512-entry prefetch FIFO when the trigger instruction commits. The FIFO arbitrates with the processor front-end for one of the icache's two cache ports.

Given the recent trend towards on-chip EDRAM caches, we increase the L2 cache latency from the Power5 model to 35 cycles, reflecting both EDRAM's increased latency as well as the increasing delays due to CMP arbitration logic in front of future shared L2 caches. All other simulation parameters are identical to the Power5. Due to workload setup issues, we only simulate a single-threaded uniprocessor for this evaluation. We expect the addition of multiple threads per core to only exacerbate the instruction cache problem, therefore these results should represent a conservative estimation of potential performance improvement. For each prefetch configuration, we simulate 500 million instructions

Due to our interest in building a dynamic instruction prefetching mechanism into a Java Virtual Machine, we only consider prefetching for Java methods that have been compiled. Java methods also constitute the majority of icache misses, at 71%. We have implemented a trace-based algorithm to select prefetch points and studied its effect in a timing simulator. Our objective is to observe the effect that an offline profile-based scheme could have on the real machine, so that future systems may consider it as an online optimization.

For the trace-based analysis, we gather traces with method entry and exit events, annotated with icache miss counts, using Mambo. We then process these traces to find both methods to prefetch and prefetch points for these methods, using the parameters described in Section 7.3. For the results presented, we target the top 750 Java methods with the most L1 icache misses. Since WebSphere has multiple threads, we gather per-thread traces and aggregate the results of our analysis across threads. Our implementation of the prefetching mechanism in Mambo uses a prefetch table generated by the analysis to issue prefetches for the specified addresses at the specified prefetch points.

7.3.3 Evaluation

In our evaluation, we focus on the timeliness of the prefetching mechanism, and analyze the effect of several different miss distance ranges. Unfortunately, due to large

number of combinations of parameters, we are unable to collect and present sensitivity data for many combinations. We consider a miss distance lower bound of two, and experiment with six different upper bounds. The confidence threshold is fixed at 90%, which means only prefetch points with at least 90% confidence are chosen. Note that with a fixed lower bound, miss distance ranges with increasing upper bounds will include all prefetch points from ranges with a smaller upper bound. Since the lower bound is fixed, we only use the upper bound to denote the miss distance range. We also include an additional *entry* case, for which entering the method to be prefetched triggers prefetching. This case is similar to the scheme described in [8]. We report improvements with respect to the baseline, no-prefetching case. We consider three aspects: prefetch accuracy, coverage, and impact on execution time. We next describe the metrics used and then present results for each of these aspects.

Prefetch Accuracy

While coverage and impact on execution time quantify the overall efficacy of the prefetching mechanism in terms of either reduction in the number of misses, or improvement in execution time, accuracy analyzes individual prefetch requests. More specifically, prefetches issued are categorized as follows:

- *Prefetch Hits* are prefetch requests that are already in the L1 icache.

- *Prefetch Misses* are prefetch requests that are not in the L1 icache and must be fulfilled.

- *Useful Prefetches* are prefetch requests that bring a block into the cache which is subsequently touched by an instruction fetch prior to its eviction.

- *Useless Prefetches* are prefetch requests that bring a block into the cache which is either unreferenced before eviction, or is requested by an instruction fetch while the prefetch miss is outstanding (untimely).

Accuracy is computed as the percentage of *Prefetch Misses* that are *Useful*. Using our current prefetching mechanism, we found approximately 20% of the total prefetches issued to be *Prefetch Misses*. In future work, we plan to reduce the number of unnecessary prefetch requests issued by eliminating redundant prefetch requests.

Figure 7.9 shows the percentage of *Useful* and *Useless* prefetches for different miss distance ranges. *Useless* prefetches include prefetches that are late. As expected, prefetch accuracy increases as the prefetches are hoisted farther from the method to be prefetched, owing to fewer late prefetches. However, once the miss distance reaches an upper bound of 256, accuracy begins to decline as a result of cache pollution from prefetches that are too early.

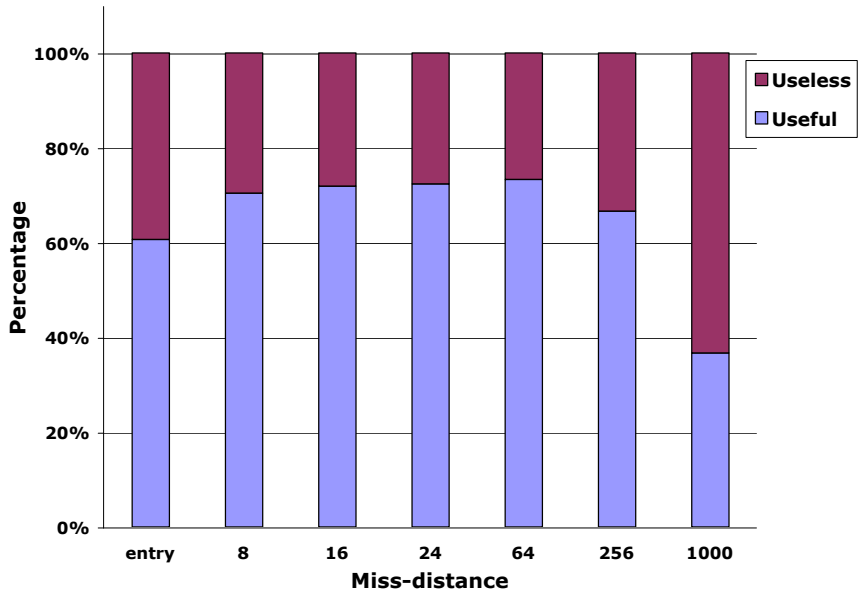


Figure 7.9: Prefetch accuracy. This graph shows the percentage of Useful and Useless prefetches for different miss distance ranges.

Coverage

Coverage is defined as the percentage reduction in misses as a result of prefetching, compared with the baseline, no-prefetching case. We track three categories of coverage: *Prefetched*, *Java*, and *All*. *Prefetched* coverage is the coverage for methods targeted by the prefetching mechanism, *Java* coverage is the coverage with respect to all Java methods, and *All* coverage refers to coverage with respect to all code, including non-Java methods.

As a result of carrying out prefetching, there is a possibility of increasing the number of misses for some methods, especially if the number of useless prefetches is high. These methods include non-Java methods, Java methods not targeted by the prefetching

Table 7.2: Coverage achieved for different miss distance ranges. Coverage is the percentage reduction in misses compared to the baseline, no-prefetching case. We show coverage for methods targeted by the prefetching mechanism (top 750), coverage for Java code, and coverage for all code.

	Coverage (% Reduction in Misses)						
	Entry	**8**	**16**	**24**	**64**	**256**	**1000**
Prefetched	30.0	28.9	36.1	40.0	48.7	53.3	48.2
Java	18.5	18.3	22.9	25.5	31.4	33.6	29.5
All	12.1	12.4	15.7	17.2	21.5	21.8	17.2

mechanism, and in some cases Java methods that are targeted by the prefetching mechanism. *Interference* measures the negative effects due to prefetching, and is measured as the percentage increase in misses excluding methods that are helped by prefetching, relative to the baseline case without prefetching.

The table in Figure 7.2 lists coverage for different values of miss distance, whereas the graph in Figure 7.10 shows trends for overall coverage and interference. The three separate interference curves represent interference for non-Java code, interference for Java code, and total interference, which is the sum of the first two. Overall coverage is the achieved coverage with respect to all code, and includes interference effects (that is it incorporates both positive and negative change in the number of misses due to prefetching) Achieved coverage depends on several factors, including potential coverage and interference. As mentioned in Section 7.3, the set of prefetch points chosen using a particular confidence threshold and miss distance range might only target a subset of the prefetched method's invocations, and therefore a subset of the misses incurred by that method. Potential coverage refers to the maximum number of misses

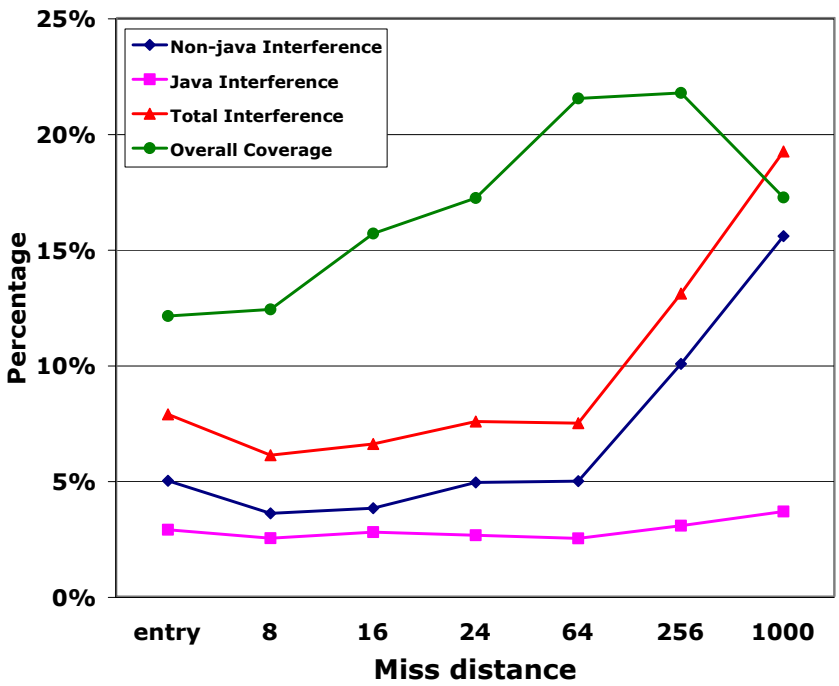


Figure 7.10: Effect of miss distance on coverage and interference. Coverage shown is overall coverage for all code.

in the prefetched method that could be reduced using the chosen prefetch points for it. Given a certain potential coverage, achieved coverage is further affected by interference, which in turn depends on prefetch accuracy.

From the table in Figure 7.2, we can see that coverage improves with increasing miss distance upper bound until the upper bound reaches the value of 1000. From Figure 7.10 we can see that interference is very high for this case. 64 is the best observed upper bound for miss distance, with a relatively high overall coverage of 21.5% and before the steep rise in interference. For the entry case, it is interesting to note that

Table 7.3: Improvement in IPC as a result of prefetching. Improvement is calculated with respect to the base, no-prefetching case. The simulation length is 500 million instructions

% Improvement in IPC						
Entry	**8**	**16**	**24**	**64**	**256**	**1000**
2.6	2.7	3.4	3.6	4.6	5.1	3.1

coverage is only 30%, in spite of the fact that potential coverage is high (since every instance of the targeted method is prefetched). This can be attributed to the large number of late or useless prefetches that this case generates.

Impact on Execution Time

Finally, we measure the net effect of our prefetching mechanism on execution time by computing percentage improvement in instructions per cycle (IPC). Table 7.3 lists these values for the miss distance ranges considered. With a miss distance range of [2,64], which is the best case from our results above, we achieve a 4.6% improvement in IPC. This is 2% better than the entry case. The IPC for the baseline, no-prefetching case was 0.518. Our results verify that timeliness is an important consideration for prefetching schemes targeting the L1 icache, given the trend of increasing L2 latencies for modern processors. We show that using a simple call-chain based mechanism to hoist prefetch requests at a suitable distance from the target method, we can increase the accuracy and efficacy of prefetching. For the benchmark we analyzed, we found

the miss distance upper bound of 64 to be best. We plan to explore a larger set of configurations in future versions of this work.

7.3.4 Discussion: Potential Improvements

In this section, we discuss two ways of increasing coverage that we have identified, with exploration of these observations planned in future work.

Callsite Interference and Incremental Prefetching

In addition to the factors affecting coverage that we discussed before (choice of prefetch points and prefetch accuracy), there is one more factor that can limit coverage. As mentioned earlier we currently issue prefetch requests for all used cache blocks for a method, when a prefetch point for that method is encountered. However, if there are callsites within the prefetched method that divert execution to other methods, the prefetched blocks might be replaced before they can be used. We call this *Callsite Interference*, and propose an incremental prefetching scheme to address it. Under this scheme, we first find callsites that cause significant interference (more than a certain number of misses before returning to the caller), and partition the method into blocks before and after the callsite. We then select additional prefetch points that issue prefetches for blocks after the callsite using the same analysis parameters described before. These prefetch points must be chosen so that they issue timely prefetches for

blocks that will be used after returning from the interfering callsite. For the top 750 methods targeted by our prefetching mechanism, we found that 159 of them had one or more interfering callsites using an interference threshold of 64.

Balancing Confidence and Potential Coverage

Confidence is one of the parameters we use in selecting prefetch points. For the results presented above, we use a high confidence threshold of 90% to minimize useless prefetches. In doing so, we lose some potential coverage, thereby reducing the coverage achieved. To minimize the useless prefetches due to prefetch points with low confidence, but at the same time not ignore prefetch points that have high potential coverage, we considered using a hybrid parameter composed of confidence and potential coverage, instead of considering confidence alone.

7.4 Related Work

A significant body of work has been proposed and implemented for instruction prefetching. In this section, we review the research most related to the approach that we present in this Chapter. The primary difference that sets our work apart is that it a software-only technique for reducing the overhead of instruction cache misses in server applications.

Annavaram et al. [7, 8] describe a profile-based software scheme for call graph prefetching for database applications, using both hardware and software approaches. Their software prefetching scheme [8] uses a labeled call graph to insert a prefetch instruction for the first callee; a prefetch for the second callee function is inserted immediately after the call to the first callee function, and so on. To reduce cache pollution, only the first n cache lines of a function are prefetched; the rest of the callee function is prefetched after entering the callee function. Their scheme may not be timely enough for some prefetches due to the distance between the prefetching point and the first miss, although it may achieve some partial overlapping of misses.

The authors of this work also conclude that code prefetch positioning alone is not effective enough to reduce instruction cache misses because of several factors, e.g., the number and size of small functions, the same functions invoked by multiple call sites. Regarding the latter, although code duplication or aggressive function inlining may be useful to eliminate some of the callers of a function so that code placement is more effective, the resulting code increase may result in an increase of the number of instruction cache misses. Based on the data presented in figure 7.1, we believe this will also be the case for WebSphere.

Spracklen et al. [100] characterize instruction cache prefetching in modern commercial applications, showing that these applications incur significant instruction cache misses. Their paper proposes a hardware scheme using both sequential and non-sequential

hardware prefetchers. Sequential prefetchers using next-N-line prefetching can cover small discontinuities in the fetch stream, however it is not effective at eliminating the misses resulting from transitions to distant lines. In their non-sequential prefetcher, a basic block predictor predicts a sequence of basic blocks to be prefetched, achieving a better coverage at the cost of substantial hardware investment. Our current scheme only considers software prefetches for the hot basic blocks of a method, so our scheme is more precise than sequential prefetching, but it may prefetch a basic block that may be later discarded. In future work, we plan to investigate a more fine-grain approach using control branches triggered from method calls and returns.

Luk et al. [73] present a cooperative, hardware-software approach to prefetching. The compiler aggressively inserts prefetch instructions to prefetch the targets of control transfers far enough in advance, often in multiple ways. To reduce cache pollution by the software prefetches, the hardware has a filtering mechanism to allow it to get far ahead without polluting the cache.

Other approaches to instruction prefetching include using helper prefetching threads whose only purposes is to run ahead to provide prefetching for the main thread. [2]. We do not explore this option since server applications are typically highly multithreaded and as such, the use of helper threads instead of worker threads may be a liability rather than an advantage. Another problem with this approach is the overhead of triggering

helper threads and that helper threads need to run ahead enough of the worker threads to be able to hide latency.

7.5 Summary

In this chapter, we first presented a characterization of instruction cache performance for IBM's WebSphere Application Server and proposed a call-chain based instruction prefetching mechanism as a first step towards evaluating the potential for phase-based instruction prefetching to improve cache performance of large scale server applications. We evaluated the potential of our mechanism and found a 31% reduction in icache misses for Java code (and 22% overall) by targeting only a subset of executed methods.

Chapter 8

Conclusion

Portability and productivity-enhancing features in programming languages such as Java, have fuelled their rapid growth and popularity in recent years. Although initially boosted by the proliferation of internet computing, Java today is widely used in a variety of domains and across a wide spectrum of devices. To enable portability (the write-once, run-anywhere model), these programs are compiled into an architecture-independent intermediate format and executed within a virtual execution environment on the target host. The execution environment, a Java virtual machine or .Net runtime, implements a compilation system that converts the intermediate code to the native format of the underlying machine. This dynamic compilation enables portability but necessarily introduces runtime overhead. However, dynamic compilation also exposes opportunities for adaptation – optimizations that we can customize according to the behavior of the executing program.

The end-goal of the research described in this dissertation is to extract and exploit the repeating patterns, i.e. phases, in a program's execution to guide optimization and customization of the program and the execution environment, for Java programs. To this end, we isolate two important directions and make significant contributions in each: (1) analysis, characterization, and detection of phases, and (2) their use to guide optimization and further program analysis. Section 8.1 summarizes these contributions.

8.1 Dissertation Summary

Our first contribution, described next, stemmed from our need to extract, and better understand runtime phases in Java programs.

Understanding and Analyzing Phases

We developed an open-source toolkit and JVM extensions to facilitate easy collection, visualization, and analysis of the time-varying behavior of Java programs. This framework incorporates phase analysis techniques used by the binary optimization and architecture communities into JikesRVM, a freely available Research Virtual Machine from the IBM T.J. Watson Research Center, and the toolkit, which is intended for offline use. The data generation framework within the JikesRVM generates a temporal basic block trace for the executing program, which is processed by components of the

toolkit to either classify intervals into phases (Phase Finder), visualize phase behavior (Phase Visualizer), analyze program behavior within phases (Phase Analyzer), or to extract important code sequences from within phases (Code Extractor). The components of the framework are pluggable (different models and metrics can be used to perform phase classification), and parameterizable so that it allows users to experiment with the granularity and similarity parameters, both of which have been shown to impact observed phase behavior.

We used this framework to analyze phase behavior in the SpecJVM benchmark suite, and established that Java programs exhibit phase behavior that can be exploited. Following its initial use, the framework has proved to be equally useful, to us and others in the analysis of new workloads. Having established the existence of phases in Java programs, we next focused on detecting these phases accurately within a Java virtual machine.

Online Phase Detection

Accurate, online phase detection is vital to the efficacy of phase-based, runtime optimization. Almost all extant phase detection approaches either rely on offline techniques, hardware support, or specialized hardware, thus rendering them unsuitable for use within dynamic optimization systems like Java Virtual Machines (JVMs). Our second contribution to enable understanding and exploitation of phase behavior is a JVM

infrastructure for the development, study, and accuracy evaluation of phase detection algorithms. Our goal with this work was to understand the parameter sensitivity of detectors and the accuracy that is possible.

The infrastructure is a novel, parameterizable, framework for online phase detection, multiple instantiations of which produce different online phase detection algorithms. The phase detection framework examines the current execution profile and determines whether the program is in phase or in transition. It consists of a similarity model and a similarity analyzer, both of which can be implemented in many ways. For example, the model can differ in how it consumes, internally represents, and computes the similarity of the profile. Like in our previously described work, this framework can be used to investigate, compare, and evaluate both extant and novel algorithms. To evaluate the accuracy of an online phase detector, we also designed an optimization- and machine independent baseline methodology, and a metric to compare online detectors to this baseline. Using the framework, baseline, and metric, we implemented and evaluated numerous (over 2500 using different models, analyzers and parameters) online phase detection algorithms for several Java programs.

The next set of contributions that we made were two phase-based runtime techniques. The first technique is an accurate, low-overhead profiling scheme for resource-constrained devices that uses phases to drive when to sample the execution of a program. The second technique is a software instruction prefetching mechanism that uses

method-level phase behavior to identify, predict, and prefetch methods that incur a large number of instruction cache misses for emerging Java workloads like database- and application servers. These two techniques span two extremes of execution environments used for Java applications: software for resource-constrained devices at the low end and application servers at the high-end.

Phase-aware Profiling

Phase-aware profiling uses phase behavior to enable efficient collection of accurate profile information. Using program phase behavior, phase-aware profiling summarizes a program as a minimal but diverse set of behaviors that accurately represent overall program behavior. Compared to other commonly used sampling schemes, like periodic or random sampling, phase-aware profiling intelligently chooses fewer samples to generate more accurate profiles.

We employed phase-aware profiling to perform remote profiling of programs on resource-constrained devices, like personal digital assistants and web-enabled mobile phones, which have emerged as new access points to the world's digital infrastructure. The difficulty in testing and optimizing software for all possible hardware-, software configurations, and use scenarios before its release, and the recent explosion in network availability and bandwidth, have made post-deployment monitoring and evolution of software an attractive option. A highly efficient, transparent, and unobtrusive runtime

profiling scheme, like phase-aware profiling, is key to the success of such an approach, given the limited resources available on these devices.

We proposed a hybrid, hardware-software approach to perform phase-aware remote profiling, where phase tracking hardware (developed in prior work by Sherwood et al) efficiently monitors program execution behavior and makes predictions about the next phase, and the software profiler collects and communicates samples only for previously unseen phases, significantly reducing the overhead. We compared phase-aware profiling with random and periodic sampling by evaluating both accuracy and overhead for a variety of profile types. We evaluated accuracy by comparing the profile generated by each of the sampling techniques with an exhaustive profile. Overhead was measured in terms of power, the primary resource for battery-powered devices, required to gather and communicate profile information. Our results indicate that phase-aware profiling can reduce the energy consumption overhead of gathering and communicating profile information by 50-75% over periodic and random sampling.

Call-chain based Instruction Prefetching

With the emergence of new commercial Java workloads, like database-, and application servers, which are characterized by larger instruction working sets and relatively poorer instruction locality, instruction prefetching, as a means to alleviate the performance loss due to instruction cache (icache) misses, has regained importance.

However, historically due to the relatively larger performance cost of data cache misses in most applications, research and development has largely focused on the data cache miss problem. As evidence, only a few architectures (IA-64, PA-RISC, and SPARC v9) include instruction cache prefetch instructions, while many architectures (e.g. IA-32, x86-64, and PowerPC), include no support for instruction prefetching. In contrast, all major architectures include support for software-directed data prefetching.

We performed a detailed characterization of instruction cache performance for IBM's WebSphere Application Server running the industry-standard SPECjAppServer2004 J2EE benchmark to investigate the potential of an all-software, method-level instruction prefetching mechanism that could be eventually implemented within a Virtual Machine [78]. In studying method-level behavior, we found that Java code (JITTED) accounts for the majority (71%) of icache misses, but contrary to the commonly observed 80-20 paradigm, no single method is a significant contributor. In fact, an effective prefetching mechanism would have to target thousands of methods. In this work, we proposed a mechanism that uses repeating patterns, i.e. phases, in the calling context to track, predict and prefetch methods before they are executed.

To evaluate the potential of this scheme for improved instruction cache performance, we performed a trace-driven limit study. We implemented and evaluated a call-chain based prefetching algorithm that individually targets delinquent methods (methods which cause significant icache misses) and chooses suitable predecessor methods in

the call-chain as prefetch points. The method prologue of prefetch points is augmented with instruction cache prefetch instructions for one or more delinquent targets. These prefetch points are selected based on two parameters, miss distance and confidence, that aim to ensure timeliness and usefulness of the issued prefetch. Miss distance is the average number of misses on the path between the prefetch point and the method to be prefetched and confidence is the probability of reaching the method to be prefetched from the prefetch point. Given at least one miss between the prefetch point and the method to be prefetched, prefetch timeliness can be ensured (since the latency due to the miss cannot be avoided). A high confidence value is necessary to avoid cache pollution. We used Mambo, a full system simulator developed at the IBM Austin Research Lab, to gather method entry/exit traces augmented with instruction cache miss information, and to subsequently evaluate our prefetching algorithm for IBM's WebSphere Application Server running the SPECjAppserver2004 J2EE benchmark. We found an 23% reduction in L1 instruction cache misses (for Java code) by targeting only a subset of executed method. Our preliminary results argue in favor of architectural support for software-directed instruction prefetching algorithms.

8.2 Impact and Future Directions

Each of the contributions described in the previous section was an important step towards our end-goal. Moreover, these contributions made significant advances in the area of feedback-directed optimization of Java programs. These advances, their impact, and future directions that have emerged are described below.

Analysis and Visualization of Phase Behavior

When we began this work, detecting and exploiting changes in program behavior at runtime was recognized to be the next frontier in feedback-directed optimization of dynamically compiled programs. However, there was no previous systematic study of phase behavior in Java programs. The framework and toolkit that we developed to facilitate extraction and analysis of phase behavior in Java programs allowed us to study the runtime behavior of Java programs, and to experiment with parameters that affect its characterization. Following their initial use, the analysis and visualization tools have been used by researchers at the San Diego Supercomputing Center in analyzing scientific computing programs as well as at IBM in analyzing industry-standard commercial software like IBM's WebSphere application server.

While we expect our framework and tools to continue to be useful to researchers in analyzing new programs, they could be made even more accessible by integrating them

into development environments to enable automatic visualization and phase analysis of programs under development.

Online, Software-level Phase Detection

Most prior work on phase detection was based on low-level program behavior, and proposed the use of special-purpose hardware. This work was the first online, all-software approach to detecting phases, and therefore an important step towards phase-based adaptive optimization in virtual execution environments, in the absence of special-purpose hardware. The client- (or optimization-) independent and parameterizable design of this framework makes it useful in designing online phase detection algorithms for many potential phase-based optimizations.

In the work described in this dissertation, we focused on the accuracy of detecting phase-shifts, i.e. changes in behavior, leaving efficiency, and the ability to recognize repeating behaviors to future work, provided we could justify it by designing and evaluating techniques that could take advantage of this information. We now believe that with additional research, online phase-shift detection could be a useful standard feature in virtual execution environments, playing the role of behavior sensors that trigger automatic, proactive responses. The next important step to make this possible would be to focus on the efficiency of online phase detection, and ways of leveraging existing runtime profiling and analysis mechanisms employed by current virtual execution en-

vironments. Phase-shift detection coupled with the ability to detect repeating patterns would enable even more sophisticated optimizations.

Phase-aware Profiling

Our first application of phase behavior, to drive efficient profiling, addressed an important problem: post deployment evolution of software, using profile information gathered from billions of users in the ever-expanding realm of hand-held devices. Much like several commercial applications that use error reporting to gather information about problematic execution of software after deployment, we used our technique to remotely perform continuous, low-overhead profiling of deployed programs to enable performance-oriented optimization in subsequent versions. This extremely low-overhead, but accurate, profiling technique opened up opportunities for feedback-directed optimization on resource-constrained devices. Continuing in this direction, distributing the effort of profile collection across multiple users, could be used not only to further reduce the overhead, but also to enable us to isolate distinct behaviors in complex programs.

We chose continuous profiling on resource-constrained devices, since it allowed us to evaluate phase-aware profiling under extreme circumstances, in addition to addressing an important problem. Phase-aware profiling, however, is a general-purpose technique that could be applied to any scenario that requires efficient and accurate pro-

file collection. In fact, this novel application of phase behavior, falls into the class of proactive runtime techniques that we mentioned in conjunction with efficient phase-shift detection, earlier in this section.

Phase-based Instruction Prefetching

In our second attempt at uncovering the potential of phase-based optimization, we used repeating patterns in method invocations in designing an adaptive instruction prefetching mechanism. This is one of very few extant, software-level instruction prefetching mechanisms, and has helped in identifying necessary architectural support, which if provided in future processor architectures, would enable dynamic compilers to drive adaptive memory system optimizations. Our work on instruction prefetching described in this dissertation brought out the potential in phase-based prefetching (especially for new, commercial Java workloads like application servers), and the need for architectural support; one of our future goals is to integrate this mechanism into a Java virtual machine as a new adaptive optimization to improve the performance of commercial Java programs. In a broader context, we also think that with the increasing complexity of both software and processor architectures, and the inability of current software to utilize hardware resources efficiently, such software-level optimizations will become increasingly important.

Bibliography

[1] Java technology: Brief history. This is an electronic document. Date of publication: unknown. Date retrieved: May 25, 2007.http://www.java.com/en/about.

[2] T. M. Aamodt, P. Marcuello, P. Chow, A. Gonzalez, P. Hammarlund, H. Wang, and J. Shen. A framework for modeling and optimization of prescient instruction prefetch. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2003.

[3] A. Adl-Tabatabai, M. Cierniak, G. Lueh, V. Parikh, and J. Stichnoth. Fast,Effective Code Generation in a Just-In-Time Java Compiler. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 280–290, May 1998.

[4] A. Ailamaki, D. Dewitt, M. Hill, and D. Wood. DBMSs on a modern processor: where does time go? In *The VLDB Journal*, pages 266–277, 1999.

[5] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–221, 2000.

[6] J. Anderson, W. Weihl, L. Berc, J. Dean, S. Ghemawat, M. Henziger, S. Leung, R. Sites, M. Vandevoorde, and C. Waldspurger. Continuous Profiling: Where Have All the Cycles Gone? *ACM Transactions on Computer Systems (TOCS)*, 15(4):357–390, 1997.

[7] M. Annavaram, J. Patel, and E. Davidson. Call graph prefetching for database applications. In *International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2000.

[8] M. Annavaram, J. Patel, and E. Davidson. Call graph prefetching for database applications. *ACM Transactions on Computer Systems*, 21(4):412–444, Nov. 2003.

[9] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Oct. 2000.

[10] M. Arnold and D. Grove. Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines. In *International Symposium on Code Generation and Optimization*, March 2005.

[11] M. Arnold, M. Hind, and B. Ryder. Online feedback-directed optimization of Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Nov. 2002.

[12] M. Arnold and B. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, Jun 2001.

[13] M. Arnold and P. F. Sweeney. Approximating the calling context tree via sampling. IBM Research Report, July 2000.

[14] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[15] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proc. of the 32nd International Symposium on Microarchitecture*, pages 245–257, Dec. 2000.

[16] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W. mei W. Hwu. Vacuum packing: Extracting hardware-detected program phases for post-link optimization. In *International Symposium on Microarchitecture*, pages 233–244, Nov. 2002.

[17] L. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *Proc. of the 26th Intl. Symp. on Computer Architecture*, May 1999.

[18] I. T. P. Benchmark. Trade6. ”https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?s

[19] E. Berk. Jlex: A lexical analyzer generator for Java. www.cs.princeton.edu/ appel/modern/java/JLex.

[20] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. Hensbergen, and L. Zhang. Mambo: A full system simulator for the powerpc architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):8–12, March 2004.

[21] J. Bowring, A. Orso, and M. Harrold. Monitoring Deployed Software Using Software Tomography. In *Proceedings of ACM SIGPLAN-SIGSOFT Worshop on Program Analysis for Software Tools and Engineering*, pages 2–9, 2002.

[22] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of SuperComputing 2000 (SC'00)*, Nov. 2000.

[23] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[24] H. W. Cain, R. Rajwar, M. Marden, and M. H. Lipasti. An architectural evaluation of Java TPC-W. In *Proc. of the Seventh Intl. Symp. on High-Performance Computer Architecture*, pages 229–240, Monterrey, Mexico, January 2001.

[25] Q. Cao, P. Trancoso, J. Larriba, J. Torrellas, B. Knighten, and Y. Won. Detailed characterization of a quad pentium pro server running TPC-D. In *Proc. of the IEEE International Conference on Computer Design*, 1999.

[26] C. Chambers and D. Ungar. Making Pure Object-Oriented Languages Practical. In *Proceeding of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA*, volume 26, pages 1–15. ACM Press, 1991.

[27] Y. Chen, R. Dios, A. Mili, L. Wu, and K. Wang. An Empirical Study of Programming Language Trends. *IEEE Software*, 22(3):72–78, 2005.

[28] M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–26, June 2000.

[29] S. Clarke, E. Feigin, W. Yuan, and M. Smith. Phased behavior and its impact on program optimization.

[30] M. Corliss, E. Lewis, and A. Roth. DISE: Dynamic Instruction Stream Editing. Technical Report MS-CIS-02-24, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, Jul 2002.

[31] M. Corliss, E. Lewis, and A. Roth. A DISE Implementation of Dynamic Code Decompression. In *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 232–243, Jun 2003.

[32] M. Corliss, E. Lewis, and A. Roth. DISE: A Programmable Macro Engine for Customizing Applications. In *Annual International Symposium on Computer Architecture*, pages 362–373, Jun 2003.

[33] M. Corliss, E. Lewis, and A. Roth. Low-Overhead Interactive Debugging via Dynamic Instrumentation with DISE. In *International Symposium on High Performance Computer Architecture*, pages 303–314, Feb 2005.

[34] S. P. E. Corporation. Specjappserver2004 benchmark. http://www.spec.org/jAppServer2004/, 2004.

[35] A. Das, J. Lu, and W.-C. Hsu. Region monitoring for local phase detection in dynamic optimization systems. In *ACM Conference on Code Generation and Optimization*, Mar. 2006.

[36] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos. Profileme : Hardware support for instruction-level profiling on out-of-order proce ssors. In *International Symposium on Microarchitecture*, pages 292–302, 1997.

[37] P. Denning. Working sets past and present. In *IEEE Transactions on Software Engineering*, 1980.

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

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

[40] E. Duesterwald and V. Bala. Software Profiling for Hot Path Prediction: Less is More. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2000.

[41] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *International Conference on Parallel Architecture and Compilation Techniques*, Sept. 2003.

[42] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *International Conference on Parallel Achitecture and Compilation Techniques*, Sept. 2003.

[43] K. Ebcioglu, E. R. Altman, M. Gschwind, and S. W. Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 50(6):529–548, 2001.

[44] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate CPI components. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[45] S. Fink and F. Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *International Symposium on Code Generation and Optimization (CGO)*, Mar. 2003.

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

[47] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VEE)*, 2004.

[48] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 156–164, Oct 2004.

[49] D. Hilbert and D. Redmiles. Extracting usability information from user interface events. *ACM Computing Surveys*, 32(4):384–421, 2000. "http://www.ics.uci.edu/ dhilbert/papers/".

[50] M. Hind, V. Rajan, and P. Sweeney. Phased Behavior and Its Impact on Program Optimization.

[51] M. Hind, V. Rajan, and P. F. Sweeney. Phase detection: A problem classification. Technical Report 22887, IBM Research, Aug. 2003.

[52] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *Fourth ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

[53] The Java HotSpot Virtual Machine, Technical White Paper. `http://java.sun.com/`

products/hotspot/docs/whitepaper/
Java_HotSpot_WP_Final_4_30_01.ps.

[54] H.Saputra, M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. Hu, H.Hsu, and U.Kremer. Energy conscious compilation based on voltage scaling. In *LCTES02-SCOPES02*, June 2002.

[55] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2006.

[56] C. Isci and M. Martonosi. Identifying program power phase behavior using power vectors. In *WWC '03: Proceedings of the Sixth International Workshop on Workload Characterization*, 2003.

[57] C. Isci and M. Martonosi. Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques. In *HPCA '06: Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture*, 2006.

[58] Jikes Research Virtual Machine (RVM). http://jikesrvm.sourceforge.net.

[59] K. Keeton, D. Paterson, Y. He, R. C. Raphael, and W. Baker. Performance characterization of a quad pentium pro smp using oltp workloads. In *Proc. of the 26th Intl. Symp. on Computer Architecture*, pages 25–26, May 1998.

[60] T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Transactions on Programmins Languages and Systems*, 25(4):500–548, 2003.

[61] T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems*, 25(4):500–548, July 2003.

[62] T. P. Kistler. *Continuous Program Optimization*. PhD thesis, University of California, Irvine, 1999.

[63] U. Kremer, J. Hicks, and J. Rehg. A compilation framework for power and energy management on mobile computers. In 14th *International Workshop on Parallel Computing (LCPC'01)*, August 2001.

[64] C. Krintz. Coupling On-Line and Off-Line Profile Information to Improve Program Performance. In *International Symposium on Code Generation and Optimization (CGO)*, Mar. 2003.

[65] C. Krintz, Y. Wen, and R. Wolski. Application-level Prediction of Battery Dissipation. In *International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2004.

[66] J. Lau, E. Perelman, and B. Calder. Selecting software phase markers with code structure analysis. In *ACM Conference on Code Generation and Optimization*, Mar. 2006.

[67] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals and hierarchical phase behavior. In *IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2005.

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

[69] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture (MICRO)*, pages 330–335, 1997.

[70] B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug Isolation via Remote Program Sampling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 141–154, June 2003.

[71] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, Apr. 1999.

[72] J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu. Design and implementation of a lightweighted dynamic optimization system. *Journal of Instruction-Level Parallelism*, 6, 2004.

[73] C.-K. Luk and T. C. Mowry. Architectural and compiler support for effective instruction prefetching: A cooperative approach. *ACM Transactions on Computer Systems*, 19(1):71–109, Feb. 2001.

[74] A. Madison and A. P. Batson. Characteristics of program localities. *Commun. ACM*, 19(5):285–294, May 1976.

[75] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proc. of the Sixth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, 1994.

[76] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. mei W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *International Symposium on Computer Architecture*, June 1999.

[77] H. Mousa and C. Krintz. HPS: Hybrid Profiling Support. In *International Conference on Parallel Achitecture and Compilation Techniques*, pages 38–50, Sep 2005.

[78] P. Nagpurkar, H. W. Cain, M. Serrano, J.-D. Choi, and C. Krintz. A Study of Instruction Cache Performance and the Potential for Instruction Prefetching in J2EE Server Applications. In *Tenth Workshop on Computer Architecture Evaluation Using Commercial Workloads (CAECW-10)*, Feb. 2007.

[79] P. Nagpurkar, H. W. Cain, M. Serrano, J.-D. Choi, and C. Krintz. Call-chain Software Instruction Prefetching in J2EE Server Applications. In *International Conference on Parallel Achitecture and Compilation Techniques*, Sept. 2007.

[80] P. Nagpurkar, M. Hind, C. Krintz, P. Sweeney, and V. Rajan. Online Phase Detection Algorithms. In *International Symposium on Code Generation and Optimization (CGO)*, March 2006.

[81] P. Nagpurkar and C. Krintz. Visualization and analysis of phased behavior in Java programs. In *ACM Principles and Practices of Programming in Java*, June 2004.

[82] P. Nagpurkar and C. Krintz. Phase-based visualization and analysis of java programs. *Elsevier Science of Computer Programming – Special Issue on Priciples Practices and Programming in Java*, 59(1–2):64–81, January 2006.

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

[84] P. Nagpurkar, H. Mousa, C. Krintz, and T. Sherwood. Efficient remote profiling for resource-constrained devices. *ACM Transactions on Architecture and Code Optimization*, 3(1):1–32, March 2006.

[85] T. Nguyen. PANIC Laboratory at Rutgers University. `http://www.panic-lab.rutgers.edu/`.

[86] A. Orso, D. Liang, M. Harrold, and R. Lipton. Gamma System: Continous Evolution for Software after Deployment. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, pages 65–69, 2002.

[87] C. Pavlopoulou and M. Young. Residual Test Coverage Monitoring. In *Proceedings of International Conference on Software Engineering*, pages 277–284, 1999.

[88] C. Pereira, J. Lau, B. Calder, and R. Gupta. Dynamic phase analysis for cycle-close trace generation. In *International Conference on Hardware/Software Codesign and System Synthesis*, Sept. 2005.

[89] K. Pettis and R. Hansen. Profile guided code positioning. In *International Conference on Programming Language Design and Implementation (PLDI)*, June 1990.

[90] The phoenix framework from microsoft research. http://research.microsoft.com/Phoenix/technical.aspx.

[91] S. Sastry, R. Bodík, and J. Smith. Rapid Profiling via Stratified Sampling. In *Annual International Symposium on Computer Architecture*, pages 278–289, July 2001.

[92] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2004.

[93] T. Sherwood and B. Calder. Time Varying Behavior of Programs. Technical Report Technical Report, UC San Diego, Aug. 1999.

[94] 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.

[95] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming Languages*, Oct. 2002.

[96] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[97] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. Eickemeyer, and J. Joyner. Power5 system microarchitecture. *IBM Journal of Research and Development*, 49((4-5)):505–522.

[98] SpecJVM'98 Benchmarks. `http://www.spec.org/osg/jvm98`.

[99] J. M. Spivey. Fast, accurate call graph profiling. *Software–Practice and Experience*, 34(3):249–264, 2004.

[100] L. Spracklen, Y. Chou, and S. Abraham. Efective instruction prefetching in chip multiprocessors for modern commercial applications. In *Proc. of the Eleven Intl. Symp. on High-Performance Computer Architecture*, pages 225–236, San Francisco, CA, January 2005.

[101] R. Srinivas. The 10-yr Story: Java and the networked world. *The Financial Express*, 2005. This is an electronic document. Date of publication: May 2005. Date retrieved: May 25, 2007.

[102] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[103] The Standard Performance Evaluation Corporation. SPEC JBB 2000. http://www.spec.org/osg/jbb2000, 2000.

[104] The Standard Performance Evaluation Corporation. SPEC JVM 1998. http://www.spec.org/osg/jvm98, 2000.

[105] J. Whaley. A Portable Sampling-based Profiler for Java Virtual Machines. In *Proceedings of ACM JavaGrande Conference*, pages 78–87, 2000.

[106] J. Whaley. Partial Method Compilation using Dynamic Profile Information. In *Proceeding of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA*, pages 166–179. ACM Press, Oct. 2001.

[107] F. Xie, M. Martonosi, and S. Malik. Compile-time dynamic voltage scaling scheduling using mixed-integer linear programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*, june 2003.

[108] L. Zhang. Personal communication, 2007.

[109] L. Zhang and C. Krintz. Code unloading. Technical Report 2003-14, University of California, Santa Barbara, 2003.

[110] X. Zhuang, M. Serrano, H. Cain, and J. Choi. Accurate, efficient, and adaptive calling context profiling. In *International Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[111] C. Zilles and G. Sohi. A programmable co-processor for profiling. In *HPCA*, pages 241–, 2001.