HARDWARE RELATED OPTIMIZATIONS IN A JAVA VIRTUAL MACHINE

by

Dayong Gu

School of Computer Science
McGill University, Montréal

December 2007

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
DOCTOR OF PHILOSOPHY

Your file Votre référence
ISBN: 978-0-494-50823-7
Our file Notre référence
ISBN: 978-0-494-50823-7

Abstract

Java Virtual Machines provide a layer of abstraction supporting any services required for the execution of Java programs; from the viewpoint of Java programs, a Java Virtual Machine is a kind of "virtual hardware". However, fundamentally, any job of this virtual hardware is done by the real low level hardware, and behavioural changes in the virtual hardware are eventually reflected by performance variations in the real hardware. Investigating the real hardware performance is thus important for understanding the behaviour of higher levels, including virtual machines themselves and the Java programs they run. Hardware information also has significant potential for optimizing Java Virtual Machines and achieving better runtime performance for Java programs.

In this thesis, we introduce a series of adaptive optimizations in a Java Virtual Machine based on hardware information. We investigate the recurrent behaviour apparent in hardware data and detect the recurrent, periodic phases, *i.e.*, the repetitive behaviour, in high level program execution. These phase detection results can be used for a variety of purposes including optimization and program understanding. For example, phase data can be used to select only the representative portions in program execution for runtime profiling. This selective profiling technique achieves a similar accuracy to that of the continuous profiling with a significant workload reduction. Based on further hardware investigation results we roughly divide the lifetime of a program into different phases and dynamically apply appropriate hot method recompilation strategies which generally improve performance and demonstrate a real world optimization using our technique. Hardware information can also bring benefits to the selection of better garbage collection points. We implement a collector with a garbage collection point analytic model based on our hardware data analyzer and provide a deep study of the relative factors in collection point selection.

Our approach and set of techniques highlight a problem for optimization development and a design that adaptively compensates. As hardware performance becomes an increasingly important factor it becomes a greater consideration in the construction of runtime environments, including Java Virtual Machines. We are able to show in our work that hardware monitoring can be the basis of both high level understanding and many new optimizations.

Résumé

Les machines virtuelles Java fournissent une couche d'abstraction soutenant tous les services exigés pour l'exécution de programmes en Java ; du point de vue des programmes en Java, une machine virtuelle Java est un genre de « matériel informatique virtuel ». Cependant, fondamentalement, n'importe quel travail de ce matériel virtuel est fait par le vrai matériel de bas niveau et des changements comportementaux du matériel virtuel sont par la suite reflétés par les variations d'exécution dans le vrai matériel. L'étude de la vraie performance du matériel informatique est ainsi importante pour la compréhension du comportement des niveaux plus élevés, y compris les machines virtuelles elles-mêmes et les programmes en Java qu'elles exécutent. L'information extraite du matériel informatique a un potentiel signficatif d'être utile pour l'optimisation des machines virtuelles Java et pour réaliser de meilleures performances d'exécution pour les programmes en Java.

Dans cette thèse, nous présentons une série d'optimisations adaptatives dans une machine virtuelle Java qui sont basées sur de l'information provenant du matériel informatique. Nous étudions le comportement récurrent évident dans des données de matériel informatique et détectons les phases récurrentes et périodiques, c.-à-d., le comportement réitéré dans l'exécution de haut niveau du programme. Ces résultats de détection de phase peuvent être employés à une variété de fins, y compris l'optimisation et la compréhension de programmes. Par exemple, des données de phase peuvent être employées pour considérer seulement les parties significatives dans l'exécution de programme pour le profilage d'exécution. Cette technique de profilage sélectif permet d'atteindre une exactitude comparable à celle du profilage continu avec une réduction significative de la charge de travail. En se basant sur d'autres résultats de recherche sur le matériel informatique, nous divisons approximativement la vie d'un programme en différentes phases et nous appliquons dynami-

quement des stratégies de recompilation de méthode appropriées de type « sur le fait » (*hot strategies*) qui, généralement, améliorent la performance et démontrent une optimisation réelle, en utilisant notre technique. L'information du matériel informatique peut également apporter des avantages quant au choix de meilleurs points de récupération de mémoire (*garbage collection*). Nous mettons en application un récupérateur avec un modèle analytique des points de récupération de mémoire basé sur nos données de matériel informatique et fournissons une étude profonde des facteurs relatifs dans le choix des points de récupération.

Notre approche et notre ensemble de techniques mettent en lumière un problème du développement d'optimisation ainsi qu'un concept qui compense de manière adaptative. Comme l'exécution au niveau du matériel informatique devient un facteur de plus en plus important, elle est davantage prise en considération dans la construction d'environnements d'exécution, y compris les machines virtuelles Java. Nous montrons dans ce travail que des mesures sur le matériel informatique peuvent servir à comme fois la base de compréhension à haut niveau ainsi que pour plusieures nouvelles optimisations.

Acknowledgements

Most importantly, I would like to thank my advisors, Professor Clark Verbrugge and Professor Etienne M. Gagnon. It is undoubtedly impossible for me to finish this thesis without your support. Your enthusiastic and patient supervision is indispensable. Thank you, Clark! Thank you, Etienne! Many things I have learnt from you, not only knowledge and research ability. Moreover, I appreciated the guidance from Dr. Karel Driesen. You gave me a hand in the beginning of the journey of a Ph.D. study. This first-stage help is definitely unforgettable.

I also like to thank other professors of SOCS, especially Laurie Hendren and Hans Vangheluwe. Thank you Laurie, you are the best lecturer and presenter I ever met. I enjoyed every talk from you. The impressive "Laurie's laughter" encourages every member in the whole research group. Thank you Hans, for the excellent services you provided as my progress committee member. Every bit of my progress is obtained under your supervision.

The Sable laboratory is a great place, full of support, friendship, and fun. Thank you Chris Pickett, the most "picky" proofreader. No error and flaw can escape your sharp eyes as well as no non-100%-perfect food is tolerable by your picky taste. Gregory Prokopski, you are the best system administrator. The great working environment provided by you is crucial to my research. The discussions about history between us were also very pleasant. We exchanged interesting topics of China, Japan, and Korea with those of Poland, Russia, and Deutschland. I am also particularly grateful to your friend Geneviève for her help in translating the abstract into French. In the past several years, I said "Assalam-O-Alaikum" ("Good morning" in Urdu) to my office-mates Ahmer and Nomair every morning, and they said "Zai Jian" ("Goodbye" in Chinese) to me every night. Most recently, I had two new office-mates, Haiying and Xun. We are busy with thesis submission at the same time. We

enjoy this important period together, encourage and oversee each other. Eric, Lin, Michael, Patrick, Richard and many more colleagues in Sable, I am proud of my experience sharing with you!

This research would not have been completed without financial support. I would like to thank SOCS and FQRNT for providing scholarships during my Ph.D. study course.

Finally, a big thank you to my parents. It has been a long time that I have not been around you, chatting with you, preparing dinner for you, having a trip with you, and many other things that I definitely should share with you. However, every day, every hour, every minute, I can feel your caring, your love, your encouragement and your understanding, across half of the world!

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Java [GJSB00] has become one of the most popular general purpose programming languages in the past decade. Java requires a specific runtime system, the Java Virtual Machine (JVM) [LY99], to support its platform independence and security attributes. Just as indicated by the name, the JVM is an abstract machine or a layer of virtual hardware. The performance of the JVM, however, can be influenced by a number of factors, including the Java program specific behaviour, the implementation details of the virtual hardware (JVM) and the impact of the actual hardware components. In fact, the impact of subtle hardware related issues on JVM performance is much bigger than intuition may lead one to think.

In this thesis, we investigate the impact of hardware on JVM performance. Following a detailed study of hardware and other VM performance concerns, we demonstrate a design for extracting high level behavioural data from low level hardware performance data. Based on this design we are able to implement a variety of novel JVM optimizations and analyses that exploit high level variable length repetitive program phases. This improves both program performance and understanding, and in general shows the strong connection between low level hardware performance and high level VM behaviour.

1.1 Motivation

Java offers a set of benefits in platform independence, runtime flexibility and security over traditional imperative languages such as C/C++. All these benefits are provided by the Java

Virtual Machine, which is an extra layer between application programs and the operating system. The design and implementation of JVMs is currently quite complex, involving many layers of optimization and adaptivity. The JVM performance actually depends on a variety of factors [GVG06]. Some of the factors are surprisingly unintuitive. From our study of JVM performance, we have found that the impact of hardware is often much more significant than may be commonly assumed. Many virtual machine level problems or program inherent behaviours are eventually reflected by performance variations in hardware components. In other words, there exists a tight relation between hardware performance and program execution [LSP+05]. Hardware information can be a good indicator of program runtime behaviour and hence be used to detect program behaviour variations. Fortunately, hardware performance counters are widely available in modern processors. A great deal of microarchitecture level information is thus procurable and can be used for program understanding and adaptive optimization.

Both of these two facts, the close relation between hardware performance and program behaviour and the existence of efficient hardware monitoring, motivate our work using hardware information to improve adaptive optimizations in JVMs. If simple and easily obtained hardware data is indicative of program behaviour then it can also be used for optimization and analysis. We begin our work by detecting program repetitive behaviour, or recurrent phases, based on the analysis of hardware information data. We translate the problem of high level program recurrent behaviour detection into low level repetitive hardware performance detection. Raw hardware data is gathered, coarsen and investigated. Based on these phase analysis results, a set of high level adaptive optimizations can be applied.

We examine that use and value of this hardware-based data on three new high level optimization techniques. These optimizations demonstrate both how this sort of data can be used and the relative impact or value expected.

Runtime profiling is a critical technique for understanding dynamic program behaviour and provides the basis for further adaptive optimizations. It is well known that programs are highly repetitive, and most of the execution time is spent in a small portion of code. It is thus not necessary to keep on profiling across the whole execution to get an accurate or representative profiling result. With recurrent phase information, we can choose the most

valuable portions to do selective profiling.

Hot method recompilation is another essential technique in developing highly efficient JVMs. The recompilation strategy of a JVM is important for the overall performance. Fixed recompilation strategies are straightforward and usually work well. However, dynamic strategies based on the status, or the phase, of a program execution can improve performance and thus are more desirable. Supported by our hardware phase analysis, we develop a dynamic recompilation strategy which better adapts to the runtime program behaviour.

Garbage collection is one of the hot topics in JVM research and development. Data on high level program behaviours may also be useful for optimizing collection performance. For instance, selecting garbage collection points (more) optimally can potentially eliminate a large portion of garbage collection workload, and program patterns of execution relate to use of memory. Program behaviour transition points are thus good for collection; new phases heuristically indicate a change in the liveness of a large number of objects. Our hardware performance detector can thus help the garbage collector to postpone or anticipate a collection until the next large program behaviour variation point.

With hardware performance data, we can get a better understanding of program behaviour. Using hardware data to support optimizations in JVMs is the main theme of our work. In our efforts, we highlight the importance and prove the feasibility of taking the hardware information into consideration in the design and implementation of JVMs. Other than the three concrete applications listed above, many other optimizations can benefit from hardware information. The improvement space is of course still large, and includes many aspects of program understanding as well as performance.

1.2 Contributions

This work contributes to program behaviour analysis and Java Virtual Machines in three tiers as shown in Figure 1.1. Each prior tier motivates and serves as a base of the implementation of the later tier. Each later tier works as an application and also a validation to the previous tier.

We begin this work with hardware performance monitoring and analysis. We find that

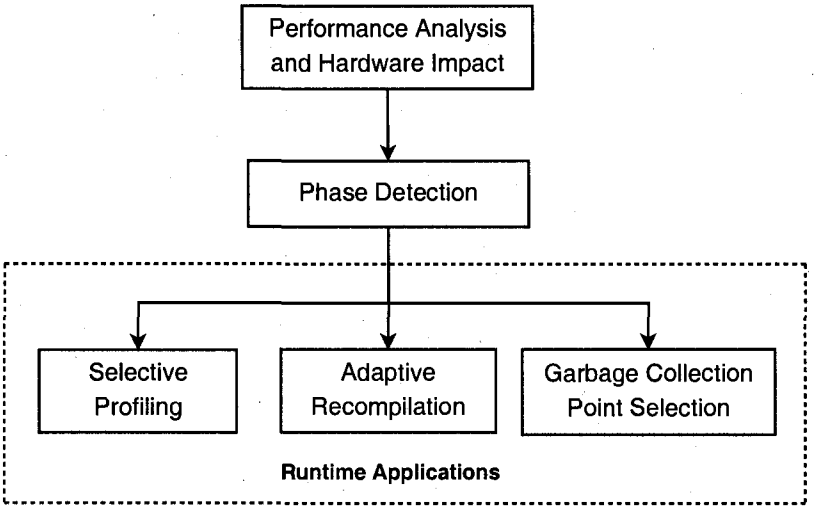

Figure 1.1: *Summary of contributions.*

hardware components can impact program behaviour significantly and can thus be considered as an indicator of variation in program behaviour. We then developed an online phase detection technique that uses our hardware performance data. Three concrete runtime optimizations based on our hardware phase analysis results have also been designed and implemented. The benefits we obtained from these optimizations prove the correctness of our phase detection technique, which in turn confirms that there exists a tight relation between hardware performance and runtime program behaviour.

- **Performance Analysis and Hardware Impact**

  The performance of modern virtual machines can be influenced by a number of factors. We investigate and categorize the relative factors which are essential for objective performance measurement of Java Virtual Machines. We experimentally demonstrate the significant impact of the hardware components on the overall performance, and in particular their surprising and often unexpected magnitude. This investigation also motivates our later work in detecting program phase behaviour based on hardware information analysis.

- **Runtime Phase Detection**

We develop a runtime phase detection algorithm based on hardware performance data analysis. Our technique focuses on the identification of variable length recurrent phases in program execution, a novel and complex form of phase data not previously examined or exploited. To situate our analysis and design, we further explore the area of phase analysis and categorize phase detection problems and the corresponding techniques. We emphasize the importance of detection and provide a new pair of evaluation metrics for recurrent phase detection. We implement three runtime applications based on our hardware event analysis.

- **Selective Profiling**

  We develop a selective runtime profiling mechanism that can reduce the profiling workload to half while preserving the accuracy of profiling results. This technique is a runtime optimization by itself, as well as a concrete proof of the effectiveness of our online phase detection.

- **Adaptive Recompilation**

  Adaptive recompilation is a key factor in the implementation of highly efficient Java Virtual Machines. Employing our hardware phase detection scheme, we have developed a novel hot method recompilation mechanism which exhibits both low overhead and good overall performance and demonstrates a general improvement over other designs. We implement this phase aware recompilation strategy in Jikes RVM; however, the fundamental idea can be helpful for any multiple-level optimization system.

- **Garbage Collection Point Selection**

  The selection of garbage collection points is an interesting and challenging problem. Significant benefit can be achieved by choosing optimal collection points. We develop an automatic collection points selection algorithm for copying collectors. The hardware performance data is used to move the collection point to a better heuristically predicted moment. We study the detailed behaviour of this collector and provide potential improvement directions that advance the research of garbage collection point selection further.

1.3 Thesis Overview

In this thesis we mainly study the challenges and opportunities of using hardware information to explain and improve JVM performance. The structure of the JVM and modern hardware are critical background knowledge to our work. We thus provide background knowledge about hardware components, Java Virtual Machine and our base system, Jikes RVM in Chapter 2. We then study a large number of relative factors for JVM performance analysis in Chapter 3. The relation between hardware performance and program behaviour are investigated and experimentally demonstrated. This result motivates our further work, using hardware information to detect recurrent periodic phases in program execution.

Phase detection is a rather wide area. In Chapter 4, we investigate the phase detection problem and corresponding techniques. We use the entire chapter to give an overview of the phase detection problem. Typical phase detection solutions are described and categorized. We also provide our opinion on this problem and make claims as to the importance of detecting different types of phases, especially recurrent, periodic phases, whose importance is not yet emphasized in current literature. This chapter introduces many related works of Chapter 5 in which we describe our hardware data based online phase detection technique in detail.

In the later half of the thesis, we present three runtime adaptive optimizations based on our hardware phase analysis results. Each of them has an individual, structural complete chapter with introduction to the question, the most important and recent related works, the concrete implementation details, the experimental results, the discussion and the chapter summary. The three runtime applications, selective profiling, adaptive recompilation and garbage collection point selection are introduced in Chapters 6, 7 and 8 respectively.

Finally, we conclude the entire thesis and present directions for future improvements in Chapter 9.

Chapter 2
Background

We apply adaptive optimizations in a Java Virtual Machine based on hardware information. Several hardware components in microprocessors can potentially largely impact program execution. We gather and investigate the performance data of these hardware components. The analysis results are used to apply adaptive optimizations appropriately. We use Jikes RVM as the base Java Virtual Machine to realize our strategy. In this chapter, we will first give an introduction to the most important background knowledge of our work, including:

- A concise introduction to the architecture of modern microprocessors, and details about the memory hierarchy, branch predictors, and hardware performance monitors.
- An overview of the fundamental structure of the Java Virtual Machine.
- A brief introduction to Jikes RVM. Jikes RVM is a complex system with many components and a variety of interesting features. Among the components, the adaptive optimization system, the hardware performance monitoring unit and the memory management toolkit have a tight relation with our work. We thus explain their structure and functionality in detail.

The rest of this chapter is organized as follows: In Section 2.1, we introduce the architecture of hardware components that can have a large impact on program performance. An

overview of the organization of JVMs in general is given in Section 2.2. A specific discussion of Jikes RVM, including the adaptive optimization system, hardware performance monitoring subsystem and the memory management toolkits is in Section 2.3. Finally, we summarize this chapter in Section 2.4.

2.1 Hardware Components

Our work is based on monitoring and investigating low level hardware information. Therefore, we first introduce the structure and functionality of modern microprocessors, especially the components that are able to significantly influence the behaviour of the running programs.

A modern computer is a complex system composed of four main structural components [Sta99]; central processing unit (CPU), main memory, I/O system and system interconnection. The CPU controls all operations and performs data processing functions. As the core of the whole computer system, the speed of the CPU is considered as an important factor to the overall system's performance. However, the CPU needs to read inputs and instructions from memory and to store the results back to memory via system interconnections. Usually, the later can not match the fast pace of the CPU. Therefore, the statement "the gap between the CPU and the memory system" occurs frequently in research papers about the architecture and performance optimization of computer systems. Hardware designers add internal memory, *i.e.*, *caches*, to alleviate the problem. At the same time, the memory requirements of today's programs becomes larger and larger, tending to exceed the capacity of physical memory. Virtual memory is thus widely used. We thus have a multiple-layer memory system, spanning from registers, on-chip caches (L1 caches), external caches ($L2^+$ caches), and main memory, to virtual memory. Hardware components in each of these layers can have an impact on the final performance, specially the caches and *translation lookaside buffer*. These hardware components narrow the speed gap between the CPU and the memory system. However, the words "the gap between the CPU and the memory system" still keeps its high overall relevance since making these components perform well turns out to be a difficult issue. Due to the large impact of memory hierarchy on runtime performance, we give a thorough introduction about it in Section 2.1.1.

Other than on-chip caches, the other major components inside a CPU are the control unit, the arithmetic and logic unit (ALU), registers and CPU interconnection. The execution of one instruction includes several steps: instruction fetch, instruction decode, register fetch, execution/effective address, memory access, and data write back [PH90]. Pipelining is an implementation technique whereby multiple instructions are overlapped in execution. It is one of the key techniques to make fast CPUs. Modern processors often are very deeply pipelined. However, the benefit of pipelining can be seriously damaged by branch instructions. The instruction fetcher may have no idea of which instruction to fetch next until the branch instruction is retired. One solution is to make a guess of the address of the next instruction, or the target of the branch; accurate guessing is the task of branch predictors in deeply pipelined processors. Obviously, the correctness rate of the branch predictors is essential to the performance of processor. Hence, branch predictor is another hardware component which can have a large impact on overall processor performance. A more detailed introduction to branch predictors can be found in Section 2.1.2.

Monitoring the performance of the underlying hardware components can be very helpful for explaining the behaviour of the processor and the running program. The microarchitecture level hardware monitoring system can be a powerful tool to locate performance bottlenecks and discover optimization opportunities. Fortunately, modern processors often provide a specific component namely the *performance monitoring unit* (PMU), or hardware counters. The PMU provides a set of low level hardware information that is worthwhile for investigating program behaviour. We thus introduce the structure and functionality of PMU, the software library for accessing PMU data and the development of PMU in new generation processors in Section 2.1.3.

2.1.1 Memory Hierarchy

Memory access latency is a major performance bottleneck in modern computers. Improvements in memory access speed have not kept pace with the improvements in speed of processors. For this reason, architecture designers put a fast, relatively small memory layer of *cache* between the fast processor and the slow main memory. Caches keep the most useful data for the processor. The system first copies the data required by the CPU from

main memory into the cache(s) and then loads the data into a register in the CPU. The data store action goes the opposite direction. Depending on the cache architecture specific implementation, the data is either immediately copied back to memory (*write-through*), or deferred (*write-back*) [PH90]. To amortize the cost of the memory transfer, more than one element is loaded into the cache each time. The basic unit of transferring is named as *cache line*. Accessing a single data element brings an entire line into the cache.

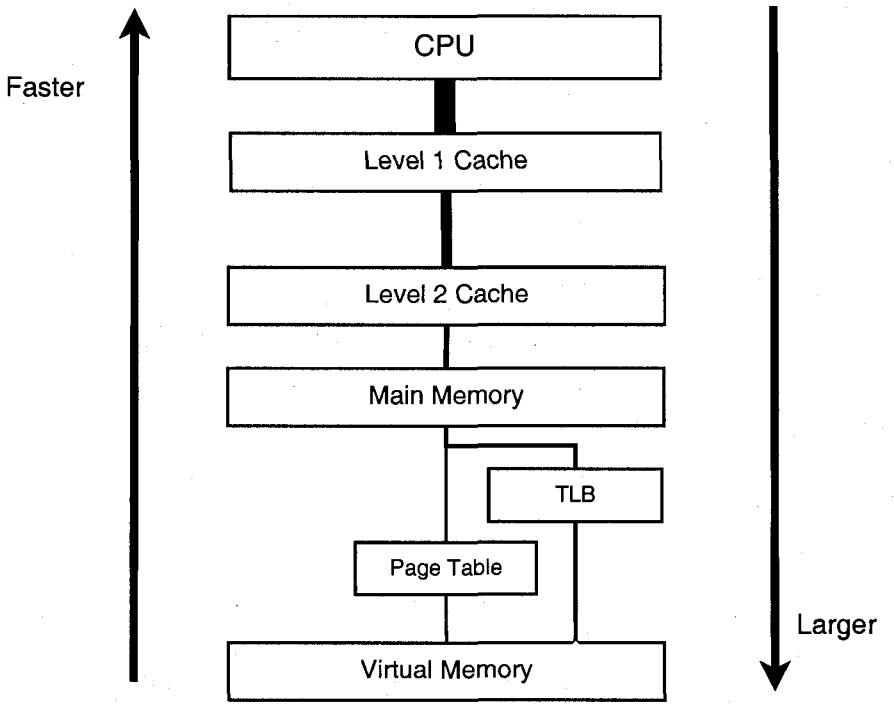

Figure 2.1: *Multiple levels of memory system.*

As shown in Figure 2.1, multiple levels of caches are used in most architectures. The higher the level, the farther away the cache is from the CPU. In most systems, a higher level cache has a larger size and usually slower access speed. Level 1 (L1) cache is on-chip, whereas the higher level(s) is external to the microprocessor.

Caches have a certain organization and a replacement policy. The organization, or mapping scheme of a cache describes in which way the lines are organized within the cache. The replacement policy dictates which line will be evicted from the cache in case

an incoming line must be placed somewhere in the cache.

According to different cache mapping schemes, caches can be categorized into three types:

- **Direct Mapped**

  Direct mapped is a simple and efficient organization. Each line from the main memory has a unique place in the cache where it can reside. Implementing a direct mapped cache is straightforward, and is relatively simple. The placement policy is built-in since the victim line is fully determined by the address of the new line. This organization has the downside of replacing a cache line which will be visited again shortly.

- **Fully Associative**

  The fully associative design solves the potential problem of direct mapped caches. The replacement policy is no longer a function of the new line's address. The new line can take any position in the cache. In a fully associative system, typically the oldest cache line is evicted from the cache which is called *least recently used* (LRU).

  The downside of a fully associative organization is cost. The larger the capacity of the cache, the larger the cost to track the usage of lines. Typically, only on very small caches is a fully associative design is of practical efficiency.

- **Set Associative**

  Set associative design is widely used in popular processors. Set associative caches can be considered as a group of several, typically a small value of power of 2, (*i.e.*, 2, 4, 8) direct mapped caches. A cache controller is responsible to determine which direct mapped cache, or *set*, a new line should go in. Within the set, a direct mapped scheme is used to allocated a slot for the new line.

  Set associative design can significantly reduce the address conflict problem in direct mapped design with a lower cost than fully associative design. Most modern processors use set associative caches, especially for higher level caches.

A cache miss refers to a failed attempt to read or write a piece of data in the cache, which results in a main memory access with much longer latency. Cache misses can be separated into *compulsory misses, capacity misses* and *conflict misses*. Compulsory miss is something unavoidable. However, the density of capacity miss and conflict miss can vary according to different designs and application-specific characteristics. Both data and instruction cache miss density potentially have a significant impact on overall program performance. In [GVG05b], the authors show a situation that large density of data cache miss changes the performance of a garbage collector. Similarly, code layout is another important factor for performance measurement. Code layout changes the program performance via changing the instruction cache miss density. Moreover, the sensitivity of applications to cache performance is program specific. In previous work [GVG06], we study the *cache bias* of a set of regular Java programs and show the different sensitivity to data and instruction cache performance variations.

Compared with the size of cache, the capacity of main memory is large. However, modern programs often require tremendous amount of memory resource. It is usual that main memory does not have the capacity to hold the data associated with a very large program, or a large number of programs coexist in the machine at the same time. In this case, we have to break things down into pieces and move the pieces into and out of main memory. In other words, we need a way to associate the blocks in main memory with location of the same data on outernal storage device, *e.g.*, hard disks. We thus need a *virtual memory* system.

Whereas caches are used to boost performance in a transparent fashion, virtual memory is used mainly for convenience. Virtual memory provides the illusion of memory that is much larger than the available physical memory. Programs using a large virtual memory address space can be executed on systems with varying amount of physical memory. However, the virtual memory address must be *translated* to physical memory before instructions or data are sent to the CPU.

The access unit in a virtual memory system is *page*, which is similar to the concept of *cache line* in cache memory. A two-stage process is often used for memory access: a *page table* is consulted to find out whether a required page is in memory and if so, where it is located then the actual memory access is performed or, in the case of a page fault, access

from disk is initiated. While a program is in execution, the start address of its page table is stored in a special page table register [Par05]. The virtual page number is used as an index into the page table and the corresponding entry is read out.

Page table access essentially increases the memory access delay. This is because accessing a word in memory requires two operations: one to the page table and one to the word itself. In order to reduce this time penalty, the translation lookaside buffer (TLB) is used to keep the record of the most recent address translations. As illustrated in Figure 2.1, when a virtual address is to be translated to a physical address, the TLB is consulted first. TLB can be considered as a special type of cache dedicated to page table entries. Typically, a TLB has tens to thousands of entries, with the smaller size being fully associative and larger ones having lower degrees of associativity.

Both cache and TLB reduce the memory access latency greatly when there is a "hit". However, cache misses or TLB misses can be a big factor for performance reduction. Investigating the data reflecting the performance of these hardware components can help us understand the program performance problems and discover further optimization opportunities.

2.1.2 Branch Prediction

Predicting the targets of branches is essential to the performance of a deeply pipelined processor. Branch prediction enables the processor to begin executing instructions long before the branch outcome is certain. Branch delay is the penalty that is incurred in the absence of a correct prediction. Today, all state-of-art microprocessors have some form of hardware support for dynamic branch prediction. All types of near branches, including conditional, unconditional, calls and returns, and indirect branches, can be predicted by these predictors.

The branch prediction subsystem always contains at least three distinct predictors for three main classes of branches:

- **Conditional Branch Predictor** returns a boolean (taken or not taken) for each conditional branch.
- **Branch Target Buffer (BTB)** predicts indirect branch targets.

- **Return Address Stack (RAS)** predicts return instruction based on prior calls.

Note that it is not necessary to predict the target of unconditional branches since the address of the target is explicitly encoded.

Hardware branch prediction strategies have been studies extensively. Some of the best known techniques are *Gshare prediction* [McF93], *bimodal branch prediction* [oCS95], and *YAGS prediction* [EM98]. Basically, prediction schemes use local or (and) global history information as an index to a prediction table with limited size. At the same time, the impact of branch predictors on performance have been studied. Such as in [GZD02], the impact of branch prediction on dynamic dispatch techniques is investigated. Our phase prediction scheme is also a table-based solution, sharing some similarity in prediction strategy with branch predictors.

2.1.3 Hardware Performance Monitoring Unit

Most modern microprocessors provide a set of special purpose registers that keep track of programmable hardware events at every cycle. This support can be logically viewed as a single hardware component called the *performance monitoring unit* (PMU). The interface of the PMU consists of a set of dedicated registers that can be programmed to count occurrences of certain microarchitecture events, such as the number of elapsed cycles, the number of instructions executed, or the number of cache/TLB/branch prediction misses. Although the implementation varies largely on different concrete processors, we still can roughly divide these registers into *performance monitoring data counters* (PMD), which collect hardware event data, and *performance monitoring configuration* registers (PMC), which configure what is to be monitored.

Extending the PMU

The PMU provides specific data which describes the hardware performance of the underlying platform. It has many uses, including program understanding, system bottleneck detection, runtime optimizations, system reconfiguration and system safety. The PMU has received great attention from both academic researchers and the industry. Since its intro-

duction in the Intel Pentium processor, the functionality of the PMU in modern processors has become more and more complete and complex.

Several Intel IA32 platforms [Int02] provide two 40-bits hardware counters, allowing two events to be monitored simultaneously. An extra 64-bit *time stamp counter* (TSC) is also included to measure the relative time in machine cycles. The AMD Athlon processor provides four 48-bit counters [AMD01]. A large extension is introduced in Pentium 4 processor. The Pentium 4 supports 48 event detectors and eighteen 40-bit event counters [Spr02], enabling the concurrent collection of a larger set of performance event counts. Event detectors control the selection of events and the qualification of event detection by privilege mode (OS and/or USER) and thread ID. The Pentium 4 also provides several instruction-tagging mechanisms that enable counting non-speculative performance events, *e.g.*, events generated by instructions that retire. On the most recent designs of microprocessors, there are further extensions to the PMU. Many interesting features which potentially can bring great benefit has been introduced. For example, on the Intel IA64 Itanium processor, the major additional features include [ME01]:

- **Opcode Matching**: Monitoring can be constrained to certain instructions, based on their encoding or based on the execution unit they use.
- **Address Range Checking**: The PMU can be programmed to record events only when they occur within a certain range of data or code addresses.
- **Event Thresholding**: An event is recorded only when the occurrence number is larger than a certain threshold per cycle.
- **Event Address Registers (EAR)**: The PMU can record cache or TLB events misses by data accessors or instruction fetches. Each sample collects the address where the miss happened.
- **Branch Trace Buffer**: A trace of the executed branch instructions can be recorded. Up to four branches can be recorded in the buffer, and for each, the source and target addresses are provided.

Hardware Performance Monitoring Tools

A number of software libraries and applications can be used to access hardware performance monitors. IBM provides a library PMAPI [IBM] as an extension of the AIX kernel to access counters. Sweeney *et al.* use this library to develop a framework which can be used to explain the behaviour of Java applications from the view of hardware events [SHC+04]. For Intel/AMD processors, PMC [Hel] and PCL [BZM] are libraries supporting hardware event counting. PCL also supports other platforms, including PowerPC, Alpha, R12000 and Ultra SPARC I/II/III. In this work, we employ PAPI [BDG+] which is a specific library providing cross-platform interface to hardware performance counter. The Intel *VTune Performance Analyzer* [Cor] is an application for hardware performance analysis and demonstration with graphic user interface.

2.2 The Java Virtual Machine

Fundamentally, our work is a set of optimizations for Java Virtual Machines. The Java Virtual Machine (JVM) [Ven96, LY99] is an abstract layer over the underlying operating system to support the execution of Java [GJSB00] programs. The JVM specification [LY99] defines a set of features that every JVM must have but leaves concrete implementation choices to the designers. The main job of a JVM is to load class files and execute their bytecodes.

As shown in Figure 2.2, the JVM contains a *class loader*, which loads class files from both Java applications and Java API library. The bytecodes are executed on the *execution engine*. Different implementations can vary largely on the execution engine part, which provides a large space for JVM designers to employ optimizations. A type of execution engine, called *interpreter* just translates the bytecodes into executable code one by one. Interpreters are easy to implement and require less resources, but usually perform slowly. Various techniques can of course be applied to improve the efficiency, such as the *inline-threading dispatcher* used in SableVM [Gag02]. Another type of execution engine, which is faster but requires more resources, is a *just-in-time compiler* (JIT). In a JIT engine, the bytecodes are compiled to native executable code at the first time that they are executed.

Methods are chosen as the basic compilation unit in most cases. Most state-of-art JITs employ multiple levels of compilation, *e.g.*, a method can be (re)compiled to different optimization levels according to its "hotness". Method recompilation is a popular and current topic in JVM research. We will present a hot method recompilation strategy in Chapter 7.

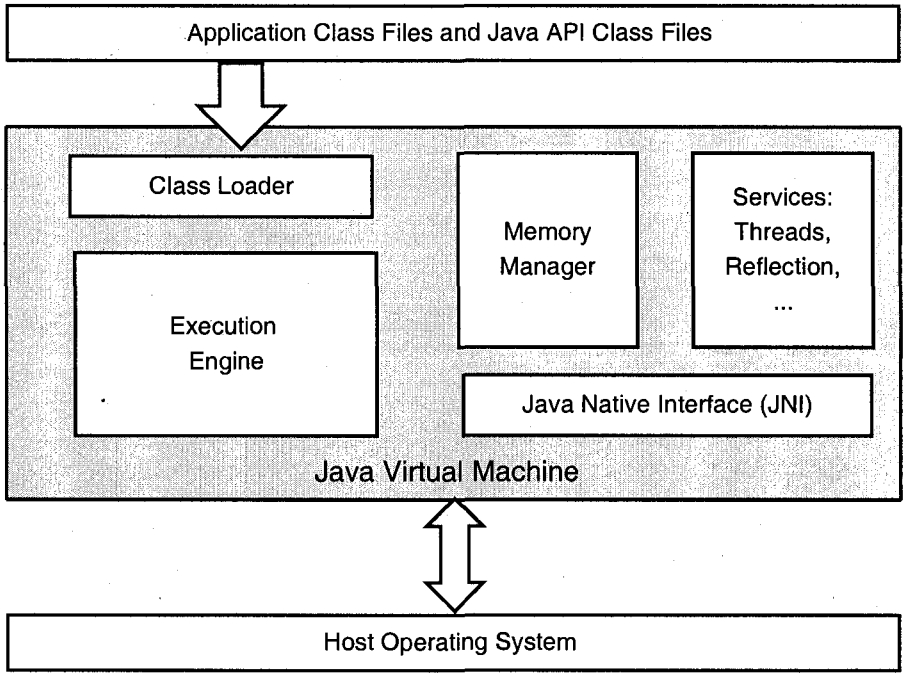

Figure 2.2: *Basic structure of a Java Virtual Machine.*

Other than Java methods which are compiled to bytecodes and stored in class files, there is another type of methods which is essential to execute Java programs, namely *native methods*. Native methods are compiled to native machine code of a particular platform and stored in a dynamic library. Native methods work as the connection between a Java program and the underlying host operating system. A Java program uses native methods to access the resources of the host operating system. As demonstrated in Figure 2.2, JVM contains a *Java Native Interface* (JNI) to load dynamic libraries containing native methods.

The JVM's heap stores all objects created by a Java application. This heap is automatically maintained by the *memory manager* of a JVM, *i.e.*, the JVM uses a garbage-collected

heap. A large number of *garbage collection* (GC) algorithms have been developed. We will provide more details on garbage collection techniques in Chapter 8.

The service part consists of a set of sub-components providing the necessary internal support for standard class library features, such as threads and reflection.

2.3 Jikes RVM

Jikes RVM [AAC+99] is an open-source research virtual machine for Java developed at the IBM T.J. Watson Research Center. It is intended to be easily extended, modular, and object oriented. Jikes RVM is implemented mainly in Java. At build time, it is run on a host JVM. A portion of Jikes RVM is a code generator which reads class files and generates the corresponding machine code for the target machine. Running on a host VM, the code generator generates the machine code for the entire optimizing and self-contained VM.

As a research virtual machine, Jikes RVM is composed of a large number of flexible components which bring convenience to researchers that try to innovate on virtual machine theories and techniques. Here we just introduce three components of Jikes RVM which have a tight relation with our work. They are the *adaptive optimization system*, the *hardware performance monitor*, and the *memory management toolkits*.

The Adaptive Optimization System

The Jikes RVM's *adaptive optimization system* (AOS) [AFG+00] contains three components, the *runtime measurement subsystem*, the *controller*, and the *recompilation subsystem*. Figure 2.3 shows the internal structure of AOS and the relation between each subsystem.

The runtime measurement subsystem maintains a set of event listeners to collect different types of information about the executing program. Usually, they perform only extremely limited processing of the gathered raw data. Organizers are a set of threads in the runtime measurement subsystem usually staying in sleep state. When awoken by listeners, organizers analyze raw data and package the data into a suitable form for consumption by the controller. The data can either be stored into an AOS database for further investi-

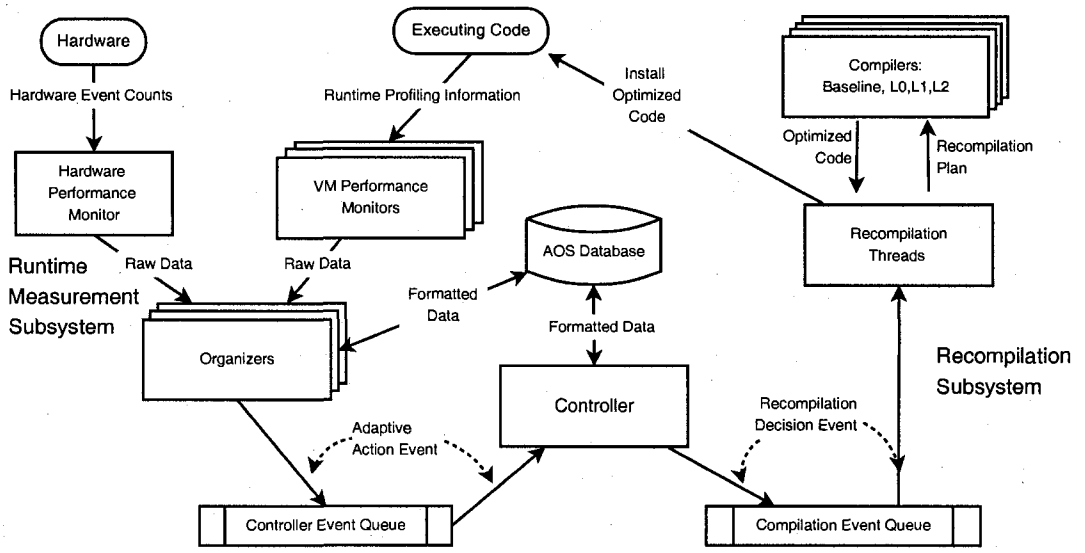

Figure 2.3: *Architecture of the Jikes RVM's Adaptive Optimization System.*

gation, or an event reflecting the information is created and inserted into a priority event queue consumed by the controller. The controller is the core of the whole AOS system. It conducts all the other components. It is the coordinator between the runtime measurement subsystem and the recompilation subsystem. The controller instantiates all runtime measurement subsystem listeners and organizers. Based on the received information and the current data in the AOS database, it makes decisions on the adaptive actions, such as requiring the recompilation subsystem to do recompilations.

Jikes RVM employs a compile-only strategy. It compiles all methods to native code before they execute. There are two types of compilers in Jikes RVM:

- The *baseline* compiler translates bytecodes directly into native code without performing optimizations.
- The *optimizing* compiler translates bytecodes into an intermediate representation, upon which it performs a variety of optimizations. This compiler has three optimization levels:
  - *Level 0* consists mainly of a set of optimizations performed on-the-fly during the

translation, including constant/non-null/copy propagation, constant folding and arithmetic simplification, dead code elimination, and elimination of redundant null-checks, check-casts, and array store checks, etc.

- *Level 1* includes additional local optimizations such as common subexpression elimination, array bound check elimination, and redundant load elimination. Inlining based on a static-size heuristics is employed on this level. Other optimizations on level 1 include copy and constant propagation, scalar replacement and flow-insensitive dead assignment elimination, etc.
- *Level 2* implements SSA-based flow sensitive optimizations.

The recompilation subsystem of AOS consists of compilation threads that invoke optimizing compilers at different levels. These compilation actions follow *compilation plans* that are inserted into the compilation queue by the controller.

The AOS database provides a repository where the adaptive optimization system records decisions, events, and static analysis results. The controller uses the AOS database to record compilation plans and to track the status and history of methods selected for recompilation. The results of runtime profilings, such as the hot method profiling and calling context profiling, are also stored and organized in the AOS database.

The Hardware Performance Monitor

As described in Section 2.1, hardware performance is one of the essential factors for program runtime behaviour. Furthermore, modern processors have provided special hardware counters for monitoring important hardware events. As a JVM with runtime adaptive optimization feature, Jikes RVM provides a component named *hardware performance monitor* (HPM) to access hardware counters.

As a part of the runtime measurement subsystem, the HPM collects hardware information following a listener-organizer cooperation mechanism. A hardware event listener thread is woken up every time a context switch happens. The listener thread reads hardware event counts by invoking native calls to the PAPI [BDG+] library. The raw hardware data is stored in one of two local buffers alternatively. When the current buffer is full, the

listener thread activates the organizer thread and submits its data. At the same time, the second buffer is used to store new hardware event data. Currently, Jikes RVM has a simple hardware data organizer `TraceWriter`. It just writes the hardware data received from the listener into a trace file in a dedicated format. In this work, we extend the HPM of Jikes RVM. An extra organizer of the hardware event listener is added to generate "patterns" to represent the behaviour of hardware. More details about this extension and hardware patterns can be found in Chapter 5.

The Memory Management Toolkit

The *memory management toolkit* (MMTk) [BCM04a] is a toolkit for writing high-performance memory managers. It currently provides the memory management subsystems of the Jikes RVM. MMTk supports a wide range of collectors: copying, mark-sweep, reference counting, copying generational, hybrid generational, *etc.*.

MMTk groups regions of memory into *spaces* and implements garbage collection algorithms with a *policy* that couples a space with an allocation and collection mechanism. Whole heap collectors use one policy for most objects, while generational collectors divide the heap into age cohorts, and use one or more policies. Currently, MMTk implements a *bump pointer allocator*, a free-list allocator and a reference counting scheme. MMTk forms different policies for these spaces:

- **Copy Space** uses bump-pointer allocation.
- **MarkSweep Space** uses free-list allocation and tracing collection by mark-sweep strategy.
- **RefCount Space** uses free-list allocation and a reference counting algorithm to detect the dead objects.
- **Immortal Space** uses bump-pointer allocation and no collection.
- **Large Object Space (LOS)** uses a coarse-grained free-list of pages and *treadmill* collection [JL96].

Based on this infrastructure, a set of different collectors can be implemented with comparatively less effort. Just combining these policies, we can create the following collectors: *SemiSpace, MarkSweep, RefCount,GenCopy* and *GenMS*. Here SemiSpace, MarkSweep, RefCount are classic semi-space copying, mark-sweep and reference counting collections respectively. GenCopy is the classic generational copying collector: it allocates into a nursery copy space, and promotes survivors into a mature space based on a SemiSpace-style copying. GenMS is a hybrid generational collector which is the same as GenCopy except it used a MarkSweep mature space. GenRC is a hybrid generational collector using ulterior reference counting to combine a copying nursery with a RefCount mature space.

The efficiency of different collectors is dependent upon application behaviour and available resources. Soman *et al.* [SKB04] investigate the performance of these above collectors. They test a variety of programs in different resource settings and demonstrate that garbage collection performance is application specific. In most case, GenMS is the one of the best performing or close to the best. However, SemiSpace, which usually works poorly, is the best choice when the heap size is huge relative to the application requirement.

Our GC work uses the MMTk and focuses on improving the performance of GenMS garbage collector which is the best choice in most cases.

2.4 Summary

Its platform independent features make Java one of the most popular object oriented programming languages. The platform independence is provided by the JVM. The design and implementation of components of the JVM have received a large amount of attention. In particular, adaptive optimization in the execution unit is one of the hottest topics of current JVM research.

On the other hand, as the structure of microprocessor becomes more and more complex, the performance of programs can be significantly impacted by the working state of the underlying hardware components. Important hardware information, such as the miss density of I/D caches, the performance of the TLB, and the hit rate of the branch predictors, becomes essential to understanding and improving runtime performance.

In the thesis, we present our solution to apply adaptive optimizations in a JVM based

on hardware information. We presented the most important background of our work in this chapter, including a basic introduction to hardware architectures, especially the memory hierarchy and branch prediction schemes, the performance monitoring unit which can be used to obtain runtime hardware information, and the basic organization of a JVM. Since our work is rooted in Jikes RVM, we also gave a brief introduction to the related parts of Jikes RVM. In the following chapters we build on this to develop and justify high level optimizations from low level hardware data.

Chapter 3
Relative Factors of Java Virtual Machine Performance

Modern, high level languages such as Java provide many benefits, including a significant amount of runtime flexibility in terms of portability, adaptivity, and optimization. Sophisticated runtime environments like the Java Virtual Machine (JVM) are, however rather complex systems, involving multiple layers of optimization and adaptivity. Improvements to JVMs can be influenced by a variety of factors, many of these surprisingly unintuitive. Understanding the source of performance variation is an essential first step in determining if changes in performance are due to external factors or are dependent on a given optimization or design change.

We address the problem of understanding JVM performance measurement in this chapter. We begin with a more detailed problem description of the problem in Section 3.1. In Section 3.2, a GC optimization is given as a case study of performance measurement for JVM techniques. A deep discussion on the relative factors impacting performance in the GC case study can be found in Section 3.3. Finally, in Section 3.4, we summarize the whole chapter and point out how this investigation of performance factors influences our subsequent work.

3.1 Difficulty of JVM Performance Measurement

The actual experienced performance of a Java program depends on multiple factors, with the program itself, the JVM and the underlying hardware all contributing to the final speed. The JVM itself has many tightly-interconnected runtime components, including class loading, GC, JITs, and so forth. Any modification in one component may influence other components, and isolating the effect of a given change or optimization is correspondingly difficult. Moreover, as we mentioned in Section 2.1, the performance of low level, actual hardware components has a significant impact on the running programs. Many performance variations may be caused by the side-effects of hardware designs or optimizations on software level implementations. In addition, programs show different characteristics, responding to optimizations differently in accordance with their individual execution properties. Program-specific reasons are thus important to performance measurement and program behaviour understanding. Software and hardware, both general and program-specific reasons should therefore all be taken into consideration in JVM performance analysis. In most cases, the final performance is a combination of these factors with different weights. Understanding the relative impact of different influences on performance is important to good optimization design and implementation.

In this chapter, we use a GC optimization as an example to investigate the relative factors in performance measurement. The measures of the GC optimization show how modifications on one component (the collector) of JVMs can impact the performance of the other part (the mutator) both significantly and unexpectedly. We then study and discuss both general contributing factors and benchmark-specific factors. We investigate reasons at software and hardware levels, including both data and instruction related issues

Close inspection of relative factors shows that the impact of *code positioning* is surprisingly significant. This is in principle clear and due to large performance variation in cache performance, especially the instruction cache; it is, however an often unintended side-effect of otherwise benign or reasonably well isolated component changes. The study discussed in this chapter has two main contributions to our later work.

- It demonstrates the tight relation between the hardware performance and overall pro-

gram execution. It validates our fundamental idea of deriving high level information by monitoring and investigating low level hardware events.

- The analysis shows that the instruction cache miss density is an outstanding candidate for understanding significant changes in program behaviour. We thus focus on the instruction cache miss density (and variations) in our later work based on hardware performance monitoring.

3.2 GC Case Study

In this section we briefly describe a GC optimization and its implementations in two distinct VM environments, a pure interpreter SableVM [Gag] and the JIT-enabled Jikes RVM [AAC+99]. We will use this example optimization to show the number and subtlety of factors that need to be considered when examining performance results, as well as give concrete evidence of their relative impact.

Our optimization case study is based on a simple and general improvement to *tracing* garbage collectors. Tracing collectors are found in most Java Virtual Machines. Starting from a set of root references (static variables, stack references), a tracing GC visits each *reachable* object seeking references to other reachable objects. Once the live set is determined, the memory storage of non-reachable objects is reclaimed. Gagnon and Hendren proposed a *bi-directional* object layout [GH01] aiming to improve the performance of GC tracing, and here we present a *reference section* tracing strategy that attempts to validate and improve that work.

Below we first describe the basic bi-directional layout design and introduce the *reference section* concept in Section 3.2.1. We talk about our implementations in two JVMs in Section 3.2.2. Experimental results on SPECJVM98 suite are reported in Section 3.2.3. In the same section, we also point out the abnormalities which will be discussed in detail later.

3.2.1 Bi-Directional Layout and Reference Sections

The bi-directional layout is an alternative way of physically representing objects in memory. Traditionally, all the fields of an object are located after the object header. The middle

graph in Figure 3.1 shows the traditional layout of an object of type **C** extending type **B** extending type **A**. Note that in this object layout, the reference fields and non-reference fields are interwoven. The JVM has to store the offsets of the references in some data structure. The tracing operation is thus composed of two steps, visiting the data structure to obtain the reference offset and accessing the reference. This overhead can be easily avoided by using the bi-directional layout.

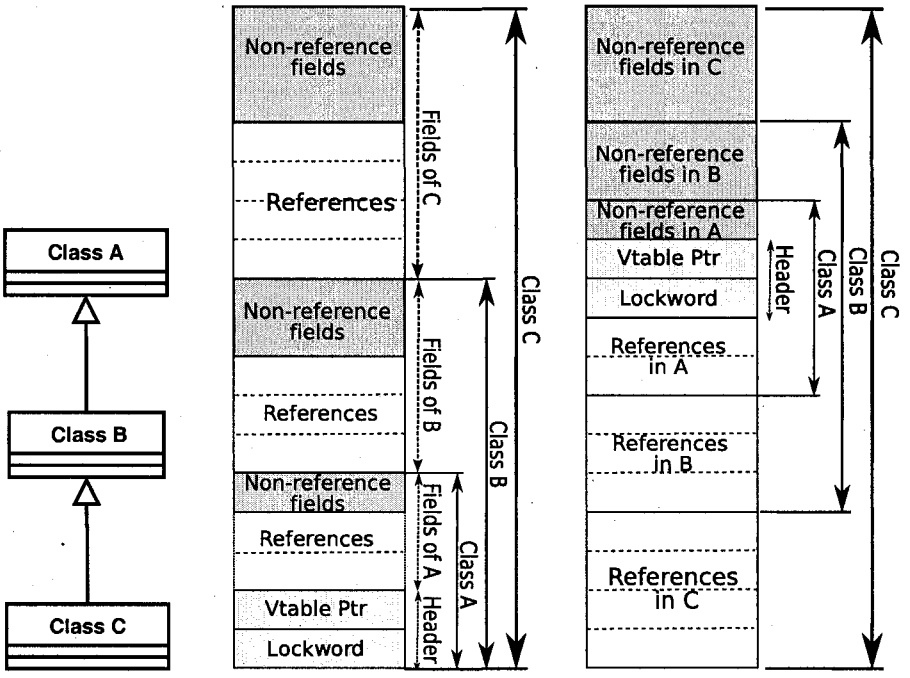

Figure 3.1: *An instance of type **C** extending type **B** extending type **A** in both traditional and bi-directional object layouts.*

The right graph in Figure 3.1 shows the bi-directional layout of the same object. The basic idea of the bi-directional layout is to relocate reference fields before the object header and group them together in a contiguous section; we denote these sections as *reference sections*. The main advantage of the bi-directional layout is the simplicity of locating all references in an object during garbage collection. References are contiguous, and only a single count of reference section size must be stored (usually in the object header). When scanning an object during GC there is no need to access a table of offsets in the object's

type information block to distinguish reference fields from non-reference fields, as must be done with the traditional layout.

Based on the bi-directional layout, we developed a new *reference section* (RS) based scanning strategy to further reduce the required work for tracing from *per object* to *per reference section*: When a new reachable object is found, the location of its reference section (if it does have one) is stored in a work list. The collector then uses this work list, which only contains relevant information, to copy or mark referents.

Compared to normal bi-directional layout tracing, our solution has the following advantages:

- The collector skips tracing of all reachable objects that have no references.
- The compactness of the work list may help improve cache locality while GC is in progress.
- In copying collectors, using a work list allows for depth-first tracing instead of default breadth-first tracing. This usually leads to better cache locality [JL96].

3.2.2 Implementing RS Scanning

RS scanning strategy changes the style of kernel workload of reference tracing. We hope it can bring benefit to all JVMs with a tracing collector. In order to examine the performance of our RS scanning strategy, we implemented it in two distinct JVM environments, a fully-functional Java interpreter, SableVM [Gag] and a JVM using adaptive JIT compilation, Jikes RVM [AAC+99]. Here we give a brief overview of the implementation designs; more details can be found in [GVG05a].

SableVM

SableVM has a semi-space copying collector which uses a two-pointer scanning algorithm [JL96]. The *scan pointer* is used to trace references in copied objects, while the *free pointer* tracks the location of unallocated memory in the target semi-space.

In our RS scanning implementation, the location (start and ending addresses) of reference sections is saved in 512-entry blocks organized in work lists. We use the higher

address end of the *to-space* to store these blocks, and unused blocks are maintained in a free list, ready to be reused. The newly found references are put into a "current" block. When the current block is full, the block is put into a "ready-for-scan" list and a new free block is obtained from the free list to store new references. Reference tracing is accomplished by obtaining a block from the "ready-for-scan" list. When all the references in a block have been accessed, the block is considered as an empty block and is returned to the free list again. Compared to the total size of the heap, the space required for this work list is very small. One 512-entry block is enough for most SPECJVM98 programs. In practice, at most five blocks (in JAVAC), or 20K at the end of the *to-space*, is enough to perform garbage collection on a two 16M semi-spaces heap.

Since our RS scanning strategy can reduce GC workload and improve data cache locality, we expect a significant GC performance improvement in SableVM.

Jikes RVM

Jikes RVM is an open-source research Java Virtual Machine which uses a compile-all strategy, totally different from that of SableVM. We also implemented the bi-directional layout and the RS scanning strategy in Jikes RVM version 2.3.4. We modified the object model component, which controls the layout of the fields, and simplified the routines that compute the offset of fields. Jikes RVM uses type information blocks to maintain the class-specific data, including an array storing the offsets of reference fields. We replaced the array with a single integer storing the number of references for this type (class).

A complication is introduced by Jikes RVM's *hashing* scheme. Jikes RVM uses a lazy hashing style. Initially, there is no field in the object header for the hash code. Once the JVM decides to hash an object, the hash code is inserted at the beginning of the object header. If we follow the same mechanism, the offsets of the references to the header will be changed by a hash code insertion, as we store references before the header. To avoid this situation we thus change the hash storage mechanism append the hash code to the end of an object when the object is hashed.

3.2.3 Experimental Results

We tested the RS scanning strategy on the SPECJVM98 benchmarks [Stac] running with input size "-s 100". We excluded MPEGAUDIO from the suite as it needs no garbage collection in SableVM's default heap settings. Experiments were run under the Debian Linux operating system on an Athlon 1.4GHz workstation with 1GB memory, with some earlier results from a Pentium III 733MHz workstation with 512MB memory. Both environments were isolated and minimized for testing, and we report the average of the medium 3 values in 5 runs. For Jikes RVM, we tested two versions, one using its semi-space copying collection algorithm and one using the GenMS collection algorithm. We chose these two because they are representative GC configurations; the former is a classic tracing GC which can give better performance for some benchmarks when the heap size is large enough [SKB04], while the latter is the best choice for most benchmarks in most heap configurations of Jikes RVM.

The results of performance speedup are shown in Figure 3.2. Here we calculate the speedup as:

$$Speedup = \frac{Original\ Execution\ Time}{RS\ Version\ Execution\ Time}$$

On SableVM, using RS scanning, a significant 1.19 average speedup is obtained, with a maximum of 1.43 speedup on DB. We also measured the impact of RS whole program execution time. Although, the overall performance speedup is still positive in general, we notice an anomalous performance decline in some benchmarks, most obviously RAYTRACE. Equally surprising are the $> 2\%$ performance improvements shown by COMPRESS and DB. GC usually takes less than 1% of execution time in the SableVM interpreter environment for these benchmarks, and so this indicates a significant, unintentional impact on the mutator.

For semi-space copying in Jikes RVM we obtained a stable improvement on the speed of GC for all benchmarks, similar to SableVM. At the same time we also show an overall positive performance for whole program execution time. We note that when using semi-space GC in Jikes RVM, GC takes a large portion of execution time (up to 40%, using semi-space GC). Whole program execution performance is therefore highly dependent on the collector's performance.

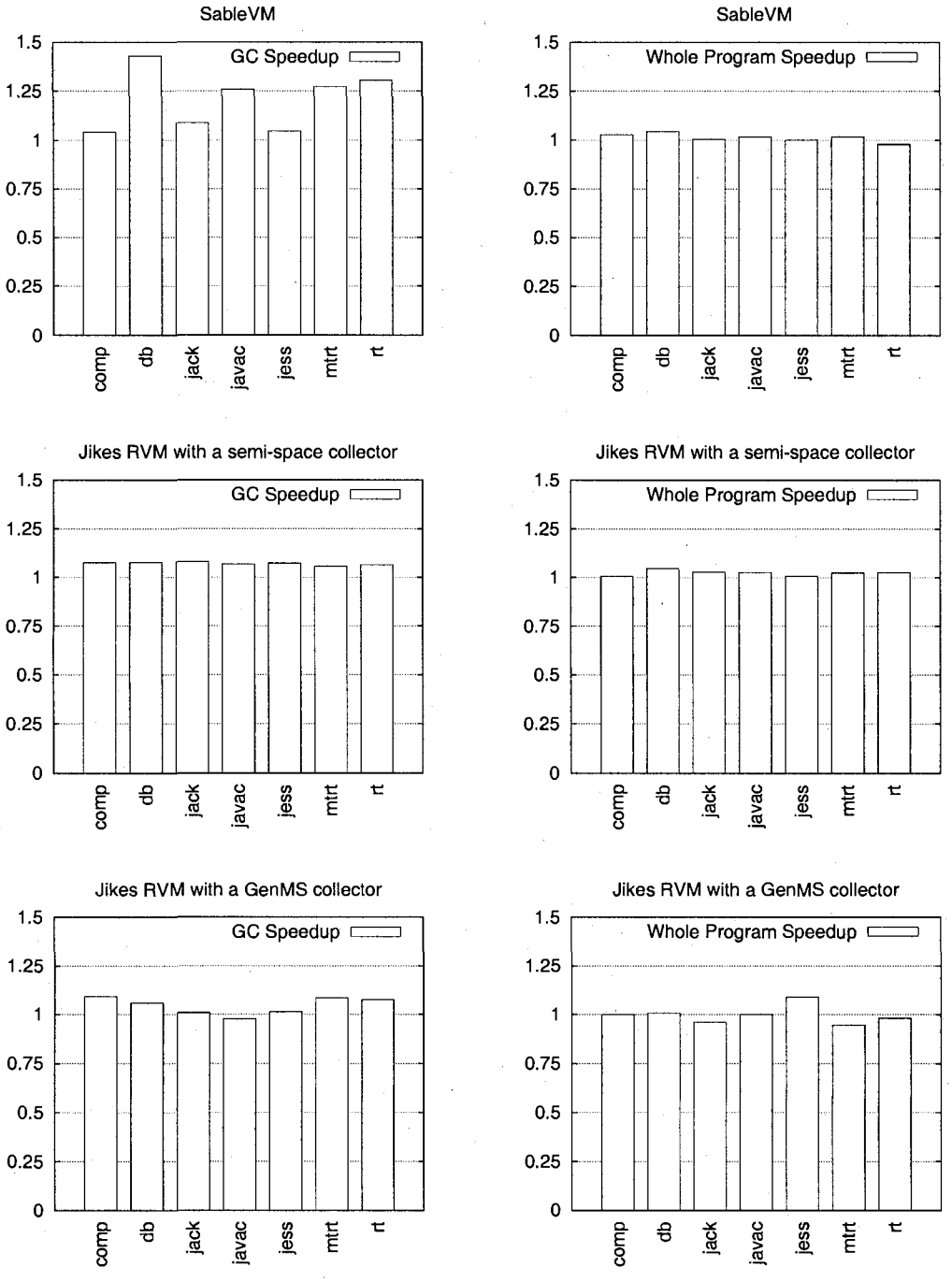

Figure 3.2: *GC and whole program speedup results on SableVM, Jikes RVM with a semi-space collector, and Jikes RVM with a GenMS collector.*

In the case of GenMS, garbage collection performance results for both GC and whole program execution are less consistent. Although the RS strategy still delivers overall GC improvements on most benchmarks, we find a significant negative value for JAVAC. Whole program execution time shows no obvious stable trend, positive or negative, and in particular no obvious correlation with GC performance.

Viewed in isolation our RS scanning improves GC performance in both interpreter and adaptive JIT compiler environments. These results, however, are not well reflected in overall execution time and anomalous measurements suggest a significant variation in the performance of the mutator. A more detailed examination to determine and compare the responsible influences is the subject of the next section.

3.3 Discussion of Relative Factors

To explain program performance on real platforms is tricky. Performance is affected by a variety of factors at different levels, from low level hardware to high level program behaviour. Here we divide potential influences as a rough taxonomy into *general factors* and *benchmark-specific factors.* General factors affect the performance of all Java programs, and can be further subdivided into two concerns *code related*, such as overall instruction workload, hash code location, and code positioning, and *data related*, such as heap organization, data location, and reference scan order, *etc.*.

Benchmark-specific factors can influence performance. These properties include the relative number and distribution of reference fields (relatively unique to our particular optimization), and more generic effects such as variation in GC collection points, GC strategy, and relative cache sensitivity of the benchmarks. We use hardware counters and runtime statistics data to investigate the impact of these factors.

Below we further explain the impact of the main factors involved in the actual performance of our GC study. We conclude with a detailed study of a critical, but not well appreciated general factor, *code-positioning.*

3.3.1 General Factors: Code and Data Management

3.3.1.1 Instruction Workload

As the source code of a virtual machine is compiled, an obvious source of performance difference is in the generated code. Even *improved* source code can generate an increase in hardware workload due to code generation patterns or downstream optimizations.

We used hardware performance counter data to investigate the changes due to our implementation of RS. The final version of RS (used in our measurements) actually reduces the number of instructions executed during GC for most benchmarks on both virtual machines. Furthermore, there is no noticeable difference in the executed instruction count for the mutator (variations were about 0.03% in average); clearly mutator instruction counts are not a significant contributor to the whole program performance differences.

3.3.1.2 Hash Code Location

In support of the `java.lang.Object.hashCode()` method, many virtual machines derive object hash codes from heap addresses, and may also store calculated hashes in the object header. Use of hash codes thus can have an indirect effect on performance if the heap memory is laid out differently. As mentioned on page 30, the position of an object's hash code is another implementation difference between the RS/bi-directional implementation and the original Jikes RVM implementation.

In practice, however, our profiling results indicate that the number of objects that actually use of hash code is quite small for these benchmarks. Most objects in these benchmarks are not hashed. Even in the JAVAC benchmark, which exhibits the largest number of hashed objects, no more than 0.5% of copied objects are hashed. Measuring the precise effect of different hash values is of course quite difficult, but the limited use of hash codes in our benchmarks strongly suggests that any differences have a minimal impact.

3.3.1.3 Code Positioning

Any change to the source code of a program is likely to change the precise location of parts of compiled code, *e.g.*, code position, and *downstream* code with higher memory addresses,

and, depending on code layout heuristics, potentially the entire program. Code positioning changes can happen in any kind of code modifications due to optimizations. Therefore, it can affect the performance of any program, not only Java programs running on a JVM. Our final code-related effect is thus a study of the effect of minor changes in code positioning on performance. In fact, among all the factors, we found that the changes in code position in the executable binary have the largest potential impact on the whole program execution time.

Table 3.2 shows that during GC very few instruction cache misses occur. In fact, in the GC phase the collector mostly works by iterating over a small set of instructions; it is thus unlikely for code position differences to cause any significant impact on GC performance.

On the other hand, Table 3.2 also shows that instruction cache misses are more frequent in the mutator. To gain additional insight on the issue, we performed two experiments.

The second column of Table 3.1 shows the largest performance changes we found in SPECJVM98 benchmark on a series of *code shifted* versions of SableVM. The only difference between these versions is the length of some extra useless space, varying from 0 bytes to double the size of a cache line, reserved in the string table section of the executable binary. This causes later binary executable code to be shifted upward in memory, without actually changing the binary code. Surprisingly, such a trivial modification triggered significant performance differences, up to 6.09%.

As a second experiment, we changed the position of some code in Jikes RVM by hand, and we generated a set of variances. We then compiled two versions of Jikes RVM: one with and one without the *hardware performance monitoring* (HPM) component. In these measurements no HPM code was executed; i.e., we simply added a piece of non-executed code to Jikes RVM. The results are shown in the third column of Table 3.1. Note how the simple addition of some non-executed component to Jikes RVM can affect performance by up to 6.4%! Performance changes due to changing code position clearly have potential to be quite large relative to other "noise" effects.

The results of these two experiments demonstrate the significant impact of the code positioning, which has nothing to do with the actual optimization, on the final performance measures. Fundamentally, this large effect is caused by performance variation in the instruction cache. This fact highlights the tight relation between hardware components,

Benchmark	L.V.F.(%) of Code Shifting	L.V.F.(%) of Extra Component
compress	2.78	1.24
db	6.09	4.80
jack	2.04	5.19
javac	2.00	4.40
jess	2.69	6.39
mtrt	3.69	4.70
raytrace	3.21	6.42

Table 3.1: *Impact of the code shifting in SableVM and adding an extra never executed component in Jikes RVM (L.V.F. for Largest Variation Found in execution time and always positive).*

especially instruction cache, and program execution.

3.3.1.4 Data Location Factors

Our case study optimization changes the position of fields in the object layout, and this has an obvious potential impact on the data cache. For the majority of objects with relatively few fields, however, proximity of data is maintained, and at least within the mutator these changes are expected to be both minor and amortized throughout execution.

A more subtle and important data cache effect can arise from the use of scanning GCs. In a tracing collection based system the order in which references are scanned has a direct impact on the new location of reachable objects in the heap after collection, and thus affects data locality in the mutator and in later collection cycles.

As the bi-directional layout changes the *natural* scan order of references, and thus the natural corresponding layout after collection, we define two scan orders:

- **Original Favourite Order (OFO)**:

  This is the natural reference scan order in the traditional layout, where references of super classes are scanned first. Thus in our case, references defined in a super class of an object are visited and relocated before those defined by subclass.

Benchmark	In Mutator		In GC	
	Instruction	Data	Instruction	Data
compress	239	871	128K	77
db	725	400	341K	152
jack	145	244	38K	123
javac	201	259	264K	138
jess	176	376	80K	146
mtrt	534	312	264K	159
raytrace	475	311	242K	161
Average	356.4	396.1	194K	136.5

Table 3.2: *Benchmark characteristics: average number of cycles between cache misses in SableVM on a Pentium III workstation.*

- **Bi-directional Favourite Order (BFO):**

  This is the natural reference scan order in the bi-directional layout, where references of super classes are scanned last (after those of subclasses). In our case, this means references defined in a subclass will be visited and relocated earlier than parent references.

We modify our implementation and change the scanning order of our collector. Switching the scan order leads to a new heap layout that changes data locality in the mutator. However, there is no obvious winner between the two orders. Most changes in data cache misses are lower than the variance in the execution time. Table 3.2 shows the average number of cycles between two consecutive L1 data or instruction cache misses. Given the low data cache miss density in the mutator part, it is safe to assert that data locality changes due to scan order are not the key issue for the performance anomalies observed in Section 3.2.3.

3.3.2 Benchmark Specific Factors

It is also the case that individual benchmarks may have properties that produce very different reactions to a given optimization. Below we extend our analysis to benchmark-specific factors which can also influence the performance. These properties include the relative number and distribution of reference fields (relatively unique to our particular optimization), and more generic effects such as variation in GC collection points and GC strategy, and relative cache sensitivity of the benchmarks.

3.3.2.1 Reference Field Usage

By its nature, RS scanning will bring larger benefits when accessing long, contiguous reference sections. For objects with a single reference, however, the cost of RS scanning is greater than the cost of normal scanning. The number of reference fields typically found in objects can also thus influence performance, and so we measured the number of reference fields in scanned objects in SPECJVM98 benchmarks.

We found that DB, MTRT and RAYTRACE have more than 40% objects with no reference at all. These objects are skipped by the RS strategy, leading to a significant improvement in GC speed over the original SableVM implementation. A relatively large number of single-reference objects are found in JACK and especially JESS (43.4%), for which our RS strategy brings less improvement. The behaviour of COMPRESS, which has the lightest GC workload of all analyzed SPECJVM98 benchmarks, and of JAVAC, which triggers four *forced* GCs, however, cannot be completely explained from the reference composition data alone. For this we need to also consider more general properties of GC behaviour.

3.3.2.2 GC Frequency and Workload

Our code and data modifications have strong potential to adjust the workload given to GC during program execution. This can have both obvious and subtle consequences. Jikes RVM's garbage collector, for instance, manages both application data and VM-specific data. Thus purely internal VM changes can be reflected in the workload experienced by applications. Our modifications to the Jikes RVM object model in the implementation of

the RS strategy also causes a slight change in GC workload. In particular, the size of surviving objects after a collection for these benchmarks is slightly different (by only a few Kilobytes) between the original and the RS implementations. Given the large heap size, we would not expect any significant impact from this when using a semi-space copying collector. However, in the case of a generational collector, where most of the work is done incrementally, a small size difference can have a much larger impact.

As a further complication, a lower number of GCs does not necessarily mean lower total GC time. The point, or moment, at which a collection is launched can lead to totally different GC performance. In Chapter 8, we will discuss the performance of GC in detail, especially the impact of collection points.

3.3.2.3 Hardware Related Benchmark Characteristics

Not all benchmark characteristics of interest are most easily seen as high level considerations, and so we also use an instrumented Jikes RVM to study benchmark behaviour through a variety of hardware events. Here we briefly discuss results on L1 instruction and data cache misses for some sample benchmarks, COMPRESS, DB and JACK. The corresponding cache miss data is shown in Figures 3.3, 3.4 and 3.5 respectively, and represent data gathered at each thread context switch. In these three figures, "GCs" stands for "garbage collections", "L1DCM/Cyc" stands for "Level 1 data cache misses per cycle", and "L1ICM/Cyc" stands for "Level 1 instruction cache misses per cycle".

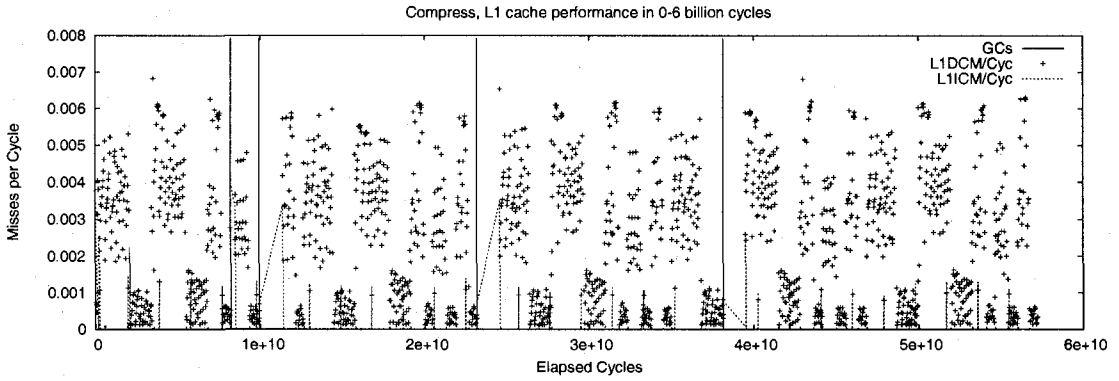

Figure 3.3: COMPRESS *hardware event trace.*

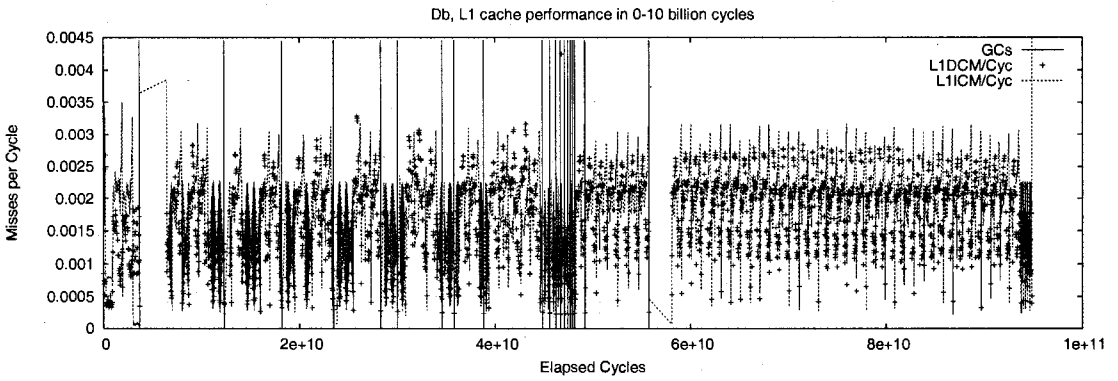

Figure 3.4: DB *hardware event trace.*

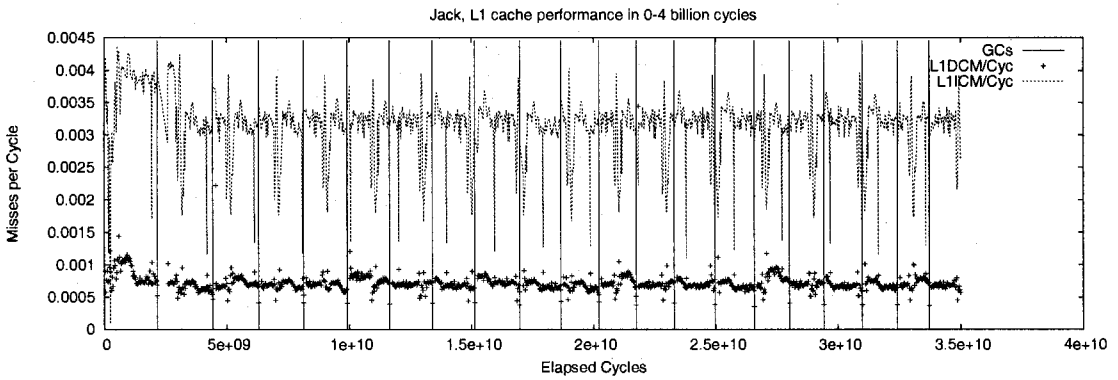

Figure 3.5: JACK *hardware event trace.*

All these benchmarks show recurrent patterns, particularly in the instruction cache miss rate. This corresponds to the various execution phases of these benchmarks. More interesting is the proportion of cache misses attributed to instruction or to data. In COMPRESS data cache misses dominate, whereas in JACK instruction cache misses dominate; DB lies between, with both kinds of misses equally important. Relative dominance of cache misses should correlate with the general sensitivity of benchmarks to instruction versus data cache effects; *e.g.*, a benchmark with a dominant and tightly recurrent pattern of instruction cache misses likely contains a small but very "hot" section of code, and could be strongly affected by small changes in code positioning.

Figure 3.6 extends the idea of a cache sensitivity "bias" (I-Cache versus D-Cache) to all our benchmarks. In this graph a benchmark's position is determined by the I-Cache (x-axis) and D-Cache (y-axis) miss density. Benchmark COMPRESS, for instance, is quite biased toward the data cache, while many benchmarks, such as JACK and JAVAC, are highly biased toward the instruction cache. The performance of DB and MTRT have similar relative dependencies on these two caches.

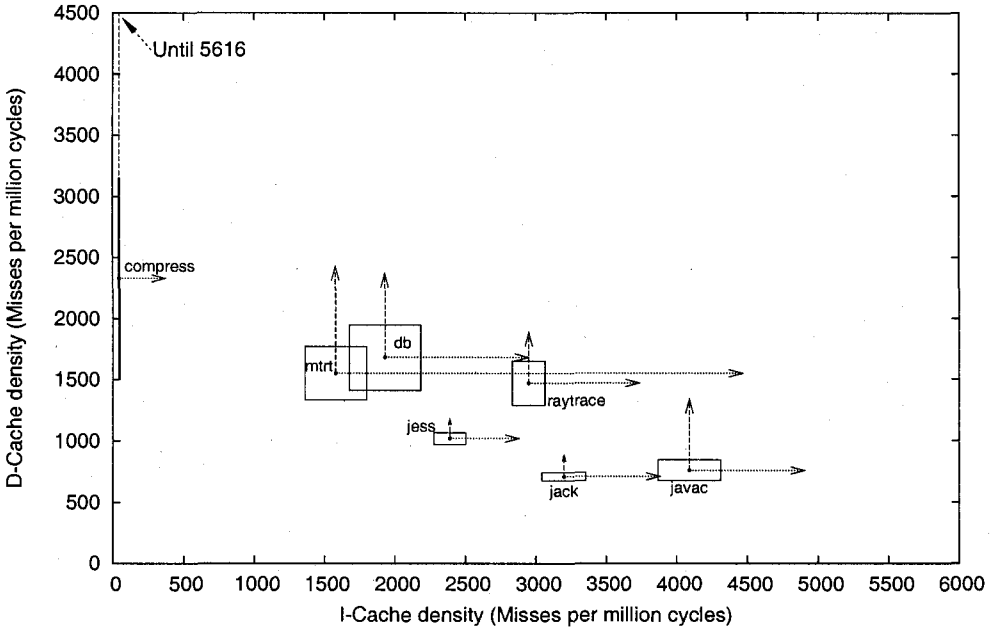

Figure 3.6: *Benchmark cache bias.*

The rectangular area for each benchmark data point functions as error-bars, showing the size of one standard deviation in the variations between consecutive measurements. A box elongated in one direction represents a benchmark that has a larger variation in the corresponding hardware event, and thus a larger potential for variation due to optimizations; e.g., in COMPRESS data cache performance varies much more than I-Cache. The arrows associated with each point show the average of the top 10% largest cache miss variations between two consecutive sample points. A very long arrow thus means that the largest performance variation is very different from the more typical case, whereas a small arrow

indicates a more uniform and stable result. The length of this arrow is thus a rough indicator of the validity of the measurement for detecting program phase transition points: a measurement that varies little will not be a good indicator of program behaviour changes.

The results in these figures are heuristic indicators only, but show that individual benchmarks may have very different properties with respect to how they respond to a particular optimization, even at a very low level: these effects are not obviously and trivially amortized away by a long or complex execution. An optimization may be viewed quite differently given a set of benchmarks that are primarily I-Cache (resp. D-Cache) driven, and this can easily result in a spurious overall evaluation of the optimization effect.

3.4 Summary and Future Work

Optimizations in a modern virtual machine environment clearly have the potential for complex interactions with various system aspects, high and low level.

The above investigations and coarse taxonomy provides a number of insights into the sources of different influences on program and optimization performance. We have attempted to be exhaustive with respect to influences related to our specific optimization case study, while demonstrating both general principles and a typical, relative weighting of factors. From the analysis in this section we can summarize that:

- The performance of JVMs can be significantly affected by unintended code motion side-effects. Instruction cache effects are not typically deeply considered in modern, high level optimization studies, but even in cases where an optimization does not intentionally alter I-Cache behaviour, minor code position changes can induce a misleading understanding of the optimization effect.

- The relation between kinds of benchmarks and design choices can be a complex source of variance, and cannot always be ignored as an amortized, unimportant cost. The reference composition of the objects, for instance, is an important factor in determining the suitability of our RS scanning strategy.

- Major VM components optimized for general cases do not give a consistent improve-

ment across all benchmarks. The garbage collector, for instance, behaves differently depending on the specific benchmark and workload size. This situation exhibits some potential for adaptively setting the nursery size to improve performance.

- Benchmarks show a wide variation in sensitivity to code versus data cache effects. Which factor dominates for a given benchmark depends strongly on the benchmark itself. This highlights the impact of low level system design on high level performance, as well as the need to apply quantitative methods for ensuring benchmark representability.

Of course a potential variance is also a potential source of optimization. At a fine grain the cache behaviour shows strong repetitive sequences. Future work on adaptive optimizations that branch on early detection of these qualities may be very applicable. Monitoring the hardware performance, detecting repetitive phase behaviours from the hardware data, and employing adaptive optimizations according to the hardware information is thus the main theme of our work presented in later chapters.

Chapter 4
Phase Detection Theory and Techniques

Detecting phases in program execution has been receiving more and more attention lately. A significant part of our work is also a type of phase detection based on hardware information. In this chapter, we introduce the fundamental idea, the basic categorization, as well as techniques and application areas of the phase detection problem. This chapter can be considered as the background and related work introduction to Chapter 5.

We begin with a brief introduction to the fundamental idea and main application areas of phase detection techniques in Section 4.1. We then explore a systematic classification of phase detection theory and prediction techniques. Many existing phase techniques are based on data collected during fixed length intervals. We will introduce a set of representative techniques in Section 4.2. Recent research has identified potential benefit in investigating variable length phase detection. Several variable length phase techniques are introduced in Section 4.3. Our phase detection technique is also an instance of this type of phase detection. Program phase detection is a broad area; in Section 4.4, we classify a large number of techniques and distinguish our technique from the others. Finally, we summarize the whole chapter in Section 4.5.

4.1 Phase Detection and Applications

It is well known that the behaviour of a program is not random. A typical program performs similar work, loads similar resources, and shows stable performance over significant

periods of time. Most programs are also quite repetitive, with similar behaviour occurring cyclically throughout the whole execution.

We use the term *phase* to represent a set of intervals or portions within a program's execution that have similar behaviour, regardless of temporal adjacency. Detecting these intervals/portions is the process of *phase detection*. Phase detection techniques can be used to capture the beginning of relatively stable executions, and also to identify repetitive cycles during the program execution. Both of these properties are valuable for improving program understanding, reducing profiling and simulation overhead, applying system reconfigurations, and employing adaptive optimizations.

- **Program Understanding and Debugging**

  Phase detection techniques can determine the boundaries of each sub-portion of program execution. Such results can be used to analyze the workload of a program at different stages, locating bottlenecks and detecting program defects at a finer granularity than the whole program scope. A. Georges *et al.* [GBEB04] associate the major workload of a program with representative methods. By measuring hardware events only for these selected methods, hardware related performance bottlenecks can be located with much less effort. Compile-time data reordering frameworks can also benefit from phase information mapped to static program structures, by focusing optimizations within the actual critical areas [SCF03].

- **Reducing Simulation and Profiling Workload**

  Program simulation, especially on accurate, cycle-level hardware simulators, can be time-consuming. It is very worthwhile to select the crucial simulation periods to model, and thus save a large portion of the total simulation time. Phase detection techniques can be used to help simulators find the interesting points to simulate. Sherwood *et al.*, for example, use phase detection techniques to determine the simulation portion of execution and to guide computer architecture research [SPHC02].

  Similarly, workloads for both offline and online profiling can be reduced by only sampling representative parts selected by phase detectors. This also benefits trace size; many profilers can generate huge traces, and phase detection can also function

as a lossy compression solution to the trace files that attempts to preserve the most meaningful information. W. Liu *et al.* demonstrate the use of phases for reducing profile cost by giving a phase-driven simulation mechanism that can obtain acceptable accuracy while only simulating a small portion of the code [LH04]. Nagpurkar *et al.* present a flexible scheme to reduce network-based remote profiling overhead based on repetitive phase information gathered from remote programs [NKS05]. They implement their phase identification mechanism on `SimpleScalar` [BA97] which is a cycle-level hardware simulator. In the case of online profiling, reductions in sample content and frequency have been recognized as important; various authors mention that optimizations based on runtime profiling need to be applied judiciously, or the cost will outweigh the benefit in many situations [ABD+97, KF03, AHR02].

- **System Reconfiguration**

  Embedded or mobile systems often have demanding resource requirements. It is valuable to reconfigure the system dynamically to minimize resource consumption. Dhodapkar and Smith, for example, introduce tuning points based on phases; these are selected to save power and improve overall performance by enabling or disabling resources adaptively [DS02b]. Similarly, the phase detection technique introduced by Shen *et al.* has been shown to be effective in adaptive cache resizing and memory re-mapping [SZD04]. Trade-off between speed and energy use of a system based on phase information have also been explored [BABD00, DS02a, HRT03].

- **Adaptive Optimizations**

  Runtime, adaptive optimization is an exciting application of phase detection, and many adaptive systems are built on determining and exploiting phases. The *software code trace* in Dynamo, for example, is refreshed based on monitoring the generation rate of new *code traces* in recent intervals [KS03]. In fact, this is a type of phase detection, and most systems that attempt to locate "hot" code based on runtime data can be seen as phase detectors. M. Arnold *et al.* [AFG+05] give a survey of adaptive optimization techniques, especially in a virtual machine environment. Many techniques introduced in that work may benefit from phase information.

Of course successful application requires a good understanding of the form of phase detection being offered; a number of phase detection approaches exist, based on a variety of different phase properties. Scientific and computationally-intensive applications may benefit more from stable phase prediction techniques than irregular applications based on dynamic data structures. We designed and developed a phase detection techniques for the JVM. It is the basis for several further runtime adaptive applications discussed in later chapters.

4.2 Fixed Length Interval Based Phase Detection

A large number of phase detection techniques are based on data gathered from fixed length intervals. They share a common style:

- The program execution is divided into fixed length intervals by some means.
- Specific profiling data is collected in each intervals.
- If the difference of the profiling data between two consecutive intervals is larger than a predefined *threshold*, a phase transition point is detected.

4.2.1 Definition

M. Hind *et al.* give a basic classification [HRS03] of phase detection. They formally give an abstract definition of a phase detection problem that takes a profile string $\mathscr{P}$ as input.

Definition 1 *Let* PD$[\tau, \sigma](\mathscr{P})$ *represent the phase detection problem based on input* $\mathscr{P}$ *and parameters:*

- **Granularity**(τ) *specifies how a profile is partitioned into fixed-length, atomic units of comparison, denoted* chunks. *Granularity size is also the minimum size of a detectable phase.*
- **Similarity**(σ) *is a boolean function that, give two strings, determines if the two strings are similar. That is* $\sigma_1(S_1, S_2)$ *returns* true *if* S_1 *is similar to* S_2*, and* false

> *otherwise. Using continuous output (*e.g.*, the interval* $[0.0, 1.0]$*) from such a function can provide detail on relative similarity, although a binary decision must be made at some level.*

Using this model, Hind *et al.* take two input strings (traces), convert each string into an abstract representation, and compute the similarity between the abstract representations. They then give a generic algorithm based on this model and demonstrate it on a simple alphabet string example.

The above approach, and its specific instantiations, are in fact based on recognizing *stable phases*. A stable phase can be defined as above, or more abstractly as: *a maximal length sequence of consecutive intervals containing no large performance change.* Such definitions are very appropriate for identifying phases in programs in which long sequences of unchanging behaviour occur frequently. Scientific benchmarks, for instance, tend to exhibit such activity, and studies of the SPECCPU95 [Stab] and SPECCPU2000 [Staa] suites show the utility of this kind of phase detection [SPC01, DS02b, SSC03, LH04, LSC05].

The fundamental mechanisms of these works are more or less similar. The differences among them are in what type of profiling data is selected, how the data is organized, how the threshold is set, and what type of comparison algorithm is used. Below we address fixed length interval approaches by dividing them into two major approaches, pure *detection* works, and techniques that aim at phase *prediction*.

4.2.2 Detection

Phase detection efforts are built on a variety of basic structures and data sources. High and low level events of different forms have been considered, and both online and offline techniques have been developed.

Sherwood *et al.* make use of moderately high level program structure *basic block vectors* (BBVs) to detect phase changes [SPC01]. A BBV is an array with an entry for each static basic block in the program. BBVs are used to track the execution frequency of individual basic blocks; the value of an array entry is simply the number of times that a given basic block has been executed during a given interval. Phase changes are detected when

the *Manhattan distance*

$$\Delta_{i,i-1} = \sum_{j=0}^{\infty} |BBV_i[j] - BBV_{i-1}[j]|$$

between consecutive intervals i and $i-1$ exceeds a predefined threshold Δ_{th}. In [SPC01], this technique is applied to select crucial simulation points.

Using a lower level perspective, A. Dhodapkar and J. Smith use the *instruction working set* to detect phase transitions [DS02b]. This allows the computation of a *relative working set distance*

$$\delta = \frac{|W(t_i,\tau) \bigcup W(t_j,\tau)| - |W(t_i,\tau) \bigcap W(t_j,\tau)|}{|W(t_i,\tau) \bigcup W(t_j,\tau)|}$$

where a *working set* $W(t_i,\tau)$ for i=1,2,...ω, is a set of distinct *segments* $s_1, s_2, ..., s_\omega$ touched over the i^{th} *window* of size τ. "Segments" here are memory regions of fixed size (*e.g.*, pages). The instruction working set is hashed into a n-bit vector, the *working set signature*. Combined with a suitable threshold, the distance between working set signatures over time is then the basis for a fixed interval phase analysis.

Another low level data choice is provided by Balasubramonian *et al.* who use *conditional branch counts* as the monitoring data [BABD00]. They use a counter to measure the number of dynamic conditional branches executed over a fixed execution interval. In their scheme, no fixed threshold is set; instead the detection algorithm dynamically varies the threshold throughout the execution of the program. This work is based on `SimpleScalar` [BA97], interacting with the phase detection scheme. Phase analysis is used to determine whether the current state is *stable* or *unstable*. In the latter case, hardware reconfiguration mechanism is launched to adjust to the new state. Dhodapkar *et al.* make a comparison between these detection techniques based on *basic block vectors, instruction working sets* and *conditional branch counts* [DS03], respectively. They evaluate the techniques mainly from their *sensitivity* and *stability*.

The techniques described above are neither aiming at Java programs nor implemented in Java Virtual Machines. Recently, Nagpurkar *et al.* present an online phase detection model [NHK+06] in Jikes RVM. Their phase detector calculates a similarity value between the profile elements in *current window* (CW) and *trailing window* (TW). They provide an abstract representation of inputs. Thus, their model allows a wide variety of input types,

such as basic blocks, methods, addresses loaded, *etc.*.

4.2.3 Prediction

Detection techniques work in a *reactive* manner; program behaviour changes are observed only after the occurrence of the change. This delay is minimally one interval long, often much more in order to achieve good confidence of stable behaviour. However, if the behaviour changes can be *predicted*, the delay between observation and reaction can be reduced. Prediction techniques can be roughly divided into two types:

Statistical Predictors

Simple statistical predictors can be used to estimate future behaviour based on historical behaviour [DCD03]. Many statistical predictors have been developed, including (among many others):

- *Last value* predictors assume the next value of a memory location or computation is the same as the last. This approach works well within a stable phase, but not in phase transitions or more complex phase behaviour.
- *Average(N)* predictors use the average of the last N intervals as the predicated value for the next interval.
- *Freq(N)* predictors choose the most frequent value in the last N intervals as the prediction for the next interval.
- Exponentially weighted moving average *(EWMA(N))* predictors place more emphasis on the most recent history, weighting a historical value's contribution to a predicted value by an exponential function of age.

Statistical prediction strategies have been widely used in optimizations based on *(return) value prediction* [PV04, Bur02, GVdB01, OHL99]. Hu *et al.*, for instance, present a parameterized *stride predictor* and give return value prediction data for SPECJVM98 [Stac] benchmarks on simulated hardware [HBJ03]. In general a variety of strategies can be applied to estimate single value from related historical data; most are based on exploiting

stable phases, but stride, context and a few other predictors can provide small scale "phase" detection for individual variables.

Table-Based Predictors

Different from the statistical predictors, *table-based predictors* predict values using information other than just the most recent history. This approach has been applied to create a memoization predictor for return value prediction [PV04], but can also be applied to predict phases. In general, table-based predictors encode a current state as well as history as the index into a prediction table. The prediction of the future is stored in the table and can be updated when large behaviour changes are identified. The differences between individual implementation can be:

- What type of data is used to build the prediction
- What is the detailed construction and organization strategy of the historical data
- What algorithm is used to create the index into the prediction table
- What kind of a mechanism is used to update the predicted value in accordance with the most recent measurement

E. Duesterwald *et al.* give a general study on predicting program behaviour [DCD03]. A set of predictor models of both statistical and table-based types on fixed size intervals are introduced and compared. Their experimental results show that table-based predictors can cope with program behaviour variability better than statistical predictors. This work uses hardware data from Power3 and Power4 architecture on SPECCPU2000 [Staa] benchmarks.

T. Sherwood and S. Sair [SSC03] present a *run length encoding phase* (RLEP) predictor using low level branch data. First, a phase ID is built for each interval based on its *footprint* for the executed branches. As shown in Figure 4.1, the PC of a branch is hashed into an index of the accumulator table, and the number of instructions executed are added into the corresponding entry. After the execution of an interval the most significant parts of the accumulator entries are combined to construct the *footprint* of this interval. If the *footprint*

is "unique" enough according to their definition a new phase ID is assigned to this interval. They evaluate their work on SPECCPU2000 benchmarks using `SimpleScalar`.

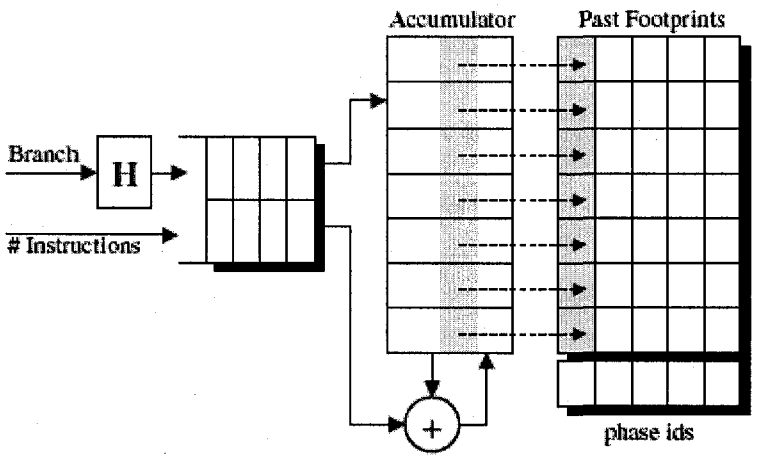

Figure 4.1: *RLEP: Building the phase ID from the branch footprint in [SSC03].*

In a subsequent step the phase ID of the current interval and the number of consecutive repetitions of the phase are hashed into the prediction table to find the phase for the next interval. This process is shown in figure 4.2. Similar general strategies have been followed

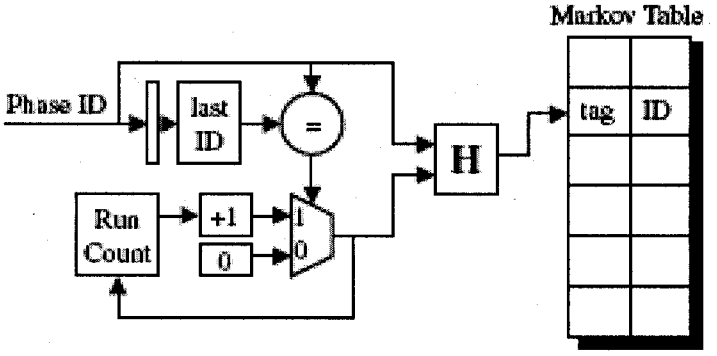

Figure 4.2: *RLEP: Using phase ID and the number of repetitions to predict the next phase in [SSC03].*

in other work [LSC05].

4.3 Variable Length Periodic Phase Detection

In the previous section we introduced a collection of representative, state-of-the-art phase detection techniques. These approaches share a number of properties:

- Split the program execution into *fixed length* intervals.
- Use *predefined* metrics to measure the differences between intervals.
- Detect behavioural differences by observing noticeable variations between *consecutive* intervals.

All those techniques are able to perform well in certain situations. However, in some situations we cannot obtain satisfying results from the data of fixed length intervals, mainly due to two reasons: the *out-of-sync problem* and *inappropriate granularity*.

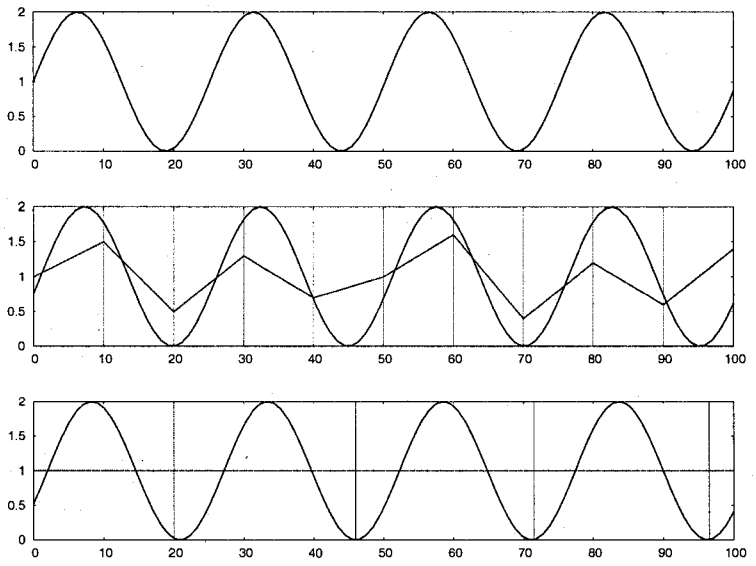

Figure 4.3: *The synchronization problem for fixed length intervals techniques [LPH+05].*

Lau *et al.* point out that fixed lengths can become "out-of-sync" with the intrinsic period of the program [LPH+05]. This problem can make a program's periodic phase behaviour

difficult to detect using fixed length interval solutions, and they graphically show that variable length intervals are necessary in some situations. Figure 4.3 from [LPH+05] show a simple example of how the fixed length interval solutions can fail in capturing the actual phase because of asychronization. A sinusoid signal is shown in the top figure. Two unsuitable fixed interval division are provided in the lower two figures. The average value of the intervals are shown by the solid lines. It is clear that no obvious repetitive features of the input sinusoidal curve are preserved in the lower two figures.

Lau *et al.* also graphically demonstrate that there are multiple levels of phases in programs that current fixed length interval techniques cannot handle at all. This motivates an initial study on variable interval phase detection using the `SimPoint` simulator [SE02]. Programs are instrumented with `ATOM` [SE94] to generate traces of each loop branch, method call and method return. Based on these traces, they construct a hierarchy of variable length phases using *SEQUITUR*, a linear-time, context free grammar algorithm that infers a hierarchical structure from a sequence of discrete symbols [NMW97]. *SEQUITUR* recursively replaces repetitive sequences with a grammatical rule that can generate the sequence. This result is a hierarchical representation of the original sequence that can offers insights into its lexical structure. An example is shown in Figure 4.4.

- **S := BBA**c
- **B := A**b
- **A :=** aa

Figure 4.4: *Grammar generated for the input "aabaabaac" by* SEQUITUR *from [LPH+05].*

Although still at an early stage, the main contribution of this work is important. They show that programs have a hierarchy of phase behaviour at many different levels of granularity, and point out limitations of fixed length interval solutions.

4.3.1 Definition: Periodic Phase

In fact, the *out-of-sync* problem is also caused by using an inappropriate granularity. In the cases where the techniques based on fixed length interval data fail to identify correct phases, we can always make it work by cutting the program execution into finer enough intervals. Of course, if we increase the length of intervals or phase transition threshold greatly, we can also obtain some kind of extreme stable phase detection results, *e.g.*, the whole program is in one stable phase.

Unfortunately, it is not free to choose the granularity of data collection, especially for techniques aiming at runtime/online applications. The smaller the granularity we choose, the larger the overhead it has, and thresholds are practically necessary.

For scientific computation programs (used by many works in Section 4.2.2 as benchmarks), there do exist long term, quite flat, stable execution portions. It is not necessary to set the granularity to a small value. On the contrary, many real world applications, including many object-oriented programs, do not share this property of scientific programs. There can be no stable portions unless data is collected on intervals with very fine granularity. In the case of Java programs, due to the extra layer of the JVM, there exists more non-determinism when measuring most kind of runtime data. At the same time, in a runtime environment, too fine-granularity in measurement is not acceptable due to the corresponding heavy overhead. For example, the context switch point is a good chance to apply adaptive optimizations in a JVM. Figure 4.5 shows L1 instruction cache miss (an important runtime measurement data) gathered at each context switch point (a coarse granularity for doing adaptive optimization on a JVM) of benchmark JACK in SPECJVM98 suite. There is almost no stable phase which can be found even by close human inspection. However, Figure 4.5 also demonstrate that there are obvious repetitive behaviours.

Repetitive information is very useful for understanding program behaviour and making adaptive optimizations. However, stable phase detection techniques and even the definition of stable phase given in Section 4.2.1 are insufficient to describe this situation. It is necessary to give a new definition for such kind of *periodic phases* as:

Definition 2 *A* variable length periodic phase *is a tuple* $\mathscr{P} < \alpha, \theta, \delta >$ *where,*

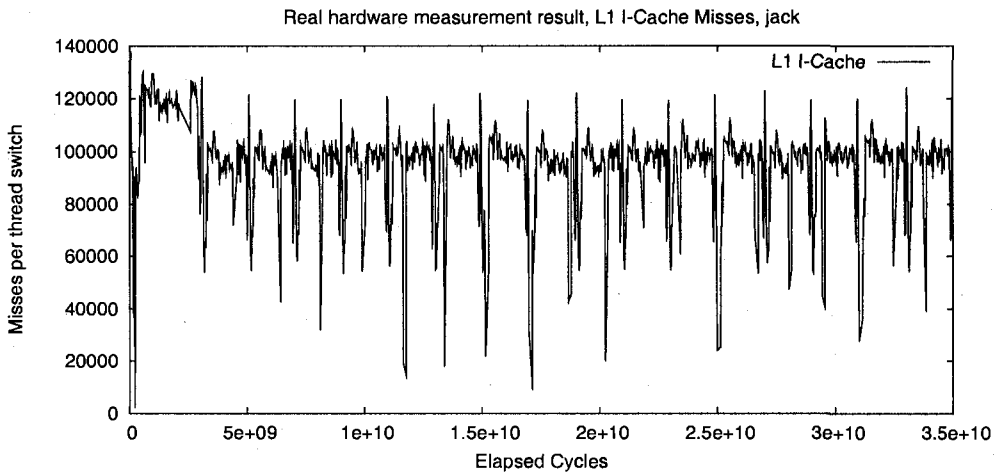

Figure 4.5: *The obvious repetitive behaviour of* JACK *at a coarse granularity, L1 instruction cache miss counts are gathered every thread context switch.*

- α *is a set of segments* $S_1, ..., S_n$ *appear in program execution and* $n > 1$.
- θ *is a function computes the correlation between each two items in* α.
- δ *is a threshold. For two arbitrary items* $S_x, S_y \in \alpha$ *where* $1 \leq x, y \leq n$, *the following inequality must be hold:* $\theta(S_x, S_y) > \delta$

More simply, periodic phases are repetitive patterns in program execution.

4.3.2 Periodic Phase Detection Techniques

Compared with fixed interval based phase detection works, investigating variable length periodic phases is new. Researchers present techniques based on different types of data and employ a variety of approaches.

A. Georges *et al.*, for example, have developed an analysis for detecting "method level phase behaviour in Java" [GBEB04]. The authors develop an *offline* analysis technique for Java workload. After the execution time is measured for each method invocation, they use an offline tool to analyze the dynamic call graph and then identify phases corresponding to method executions. Methods that take a large portion of the whole execution time but

which have a less frequent invocation count are then candidates for major method level phases.

Shen *et al.* [SZD04] detect long range variable phases using a quite different technique. Their offline/online mixed phase detection solution does an offline computation on a trace of *reuse distance* data of programs. Reuse distance is defined as the number of distinct data elements accessed between two consecutive references to the same element. Apparently, reuse distance can cover a large portion of the program execution and is not fixed length interval. They use a *discrete wavelet transform* [Dau92] as a filter to remove the least significant changes and locate the most important ones. They use `ATOM` [SE94] to insert *phase markers* into program to label the significant phase points. Shen *et al.* apply their phase analysis to "cache resizing," and test their work on the `Cheetah` [SA93] cache simulator. Simulation data suggests this phase analysis can help considerably, reducing cache size up to 40% without significantly increasing the number of cache misses.

4.4 Problem Classification

We have introduced a large number of different phase detection techniques. All these techniques cover a broad area. Each solution is distinguished from, as well as shares some common features with others. In order to highlight the differences and relations among them, we categorize them in three manners. That is, we treat the whole solution space of phase detection as a three-dimension space. Each technique can have its specific position in this space. The three axes are:

1. **Phase Type**

   As we said in previous sections, all phase detection works are either aiming at stable phases or periodic phases. We actually follow this axis to introduce all the phase detection techniques.

2. **Data Source**

   Phase detection can work on different types of data.

   - Static software data

Some works use measurement for particular static program units, such as method [GBEB04], basic block [SPC01] and loop or branch [LPH+05].

- Dynamic software data

  Some phase detection works are based on dynamic counting for a special software concern, such as the "instruction working set" used in [DS02b] and the "data reuse distance" used in [SZD04].

- Simulated hardware data

  Hardware data has received a large attention. However, for practical reasons, many researchers use hardware simulators to investigate their techniques. From this precise data is possible and a set of phase detection works are developed, including the RLEP phase predictor [SSC03] and Balasubramonian's work based on "conditional branch counts" [BABD00].

- Hardware data

  Use of real hardware data is desirable. E. Duesterwald *et al.* use hardware data from Power3 and Power4 architecture to detect phases in C language benchmarks. Shen *et al.* present an extension of their *Wavelet* based analysis work to hardware trace data, such as IPC (Instruction-per-cycle) and cache hit rates [SDDS05]. Our approach is also based on real life hardware event data for a JVM.

3. **Application Time**

- Offline analysis

  Many phase detection techniques are actually offline data analyses [SZD04, GBEB04, SDDS05, LPH+05]. Sophisticated analysis is applied on program trace data. In general, offline analysis can work on trace data for as many passes as it requires and provides comparatively accurate phase detection results, or phase analysis results. They are very useful for program understanding, but will not be suitable for online optimizations due to the (usually) large computation overhead. However, the results of offline analysis can be somehow used by runtime systems, such as the "phase markers" used in [SZD04].

- Simulated online

  Some phase detection algorithms designed for online applications are firstly presented in a simulated manner. The "simulated" manner here either means implemented on hardware simulators as works in [BABD00, SSC03] or presented as a prototype for online solutions working on a trace file in only one pass. Our offline pre-study is a simulated online algorithm, as well as the phase detection model given in [HRS03].

- Online implementation

  Online phase detection technique on real world system is challenging. Nagpurkar *et al.* designed an online phase detection model [NHK+06] which aims at detecting stable phase. Their framework can accept different type of data as input. We also implement phase detection in a purely online manner, on real world hardware measurement data, but aim at detecting periodic phases.

4.4.1 Online Hardware Based Phase Detection

We will present a phase detection approach for detecting variable length periodic phases in Java programs. The data source we used is obtained from realistic hardware components. The application time of the final approach is purely online, and we do an offline pre-study working in a simulated online manner. The results of our approach can bring benefit for better program understanding and provide valuable information for runtime adaptive optimizations.

Using the hardware event data commonly available in modern processors, we detect and predict the recurrent behaviour in programs. The hardware event data is gathered at every context switch, *e.g.*, we choose a coarse granularity which is practical to online implementation in realistic systems.

Before designing and implementing the online algorithm, we first built an offline prototype to validate the feasibility. The prototype is actually a simulation on hardware trace data generated using Jikes RVM.

Our algorithm captures and associates the beginning *pattern* (binary representation of hardware event data) of each periodic phase with the phase itself. We thus use initial

patterns to predict upcoming periodic phases. We analyze the similarity between patterns and replay the other part of a period of execution when we identify that there is a recurrence of a pattern. The algorithm just reads the trace file in one pass. Therefore, it is a simulation of an online application, and can be transplanted into an online implementation with less effort.

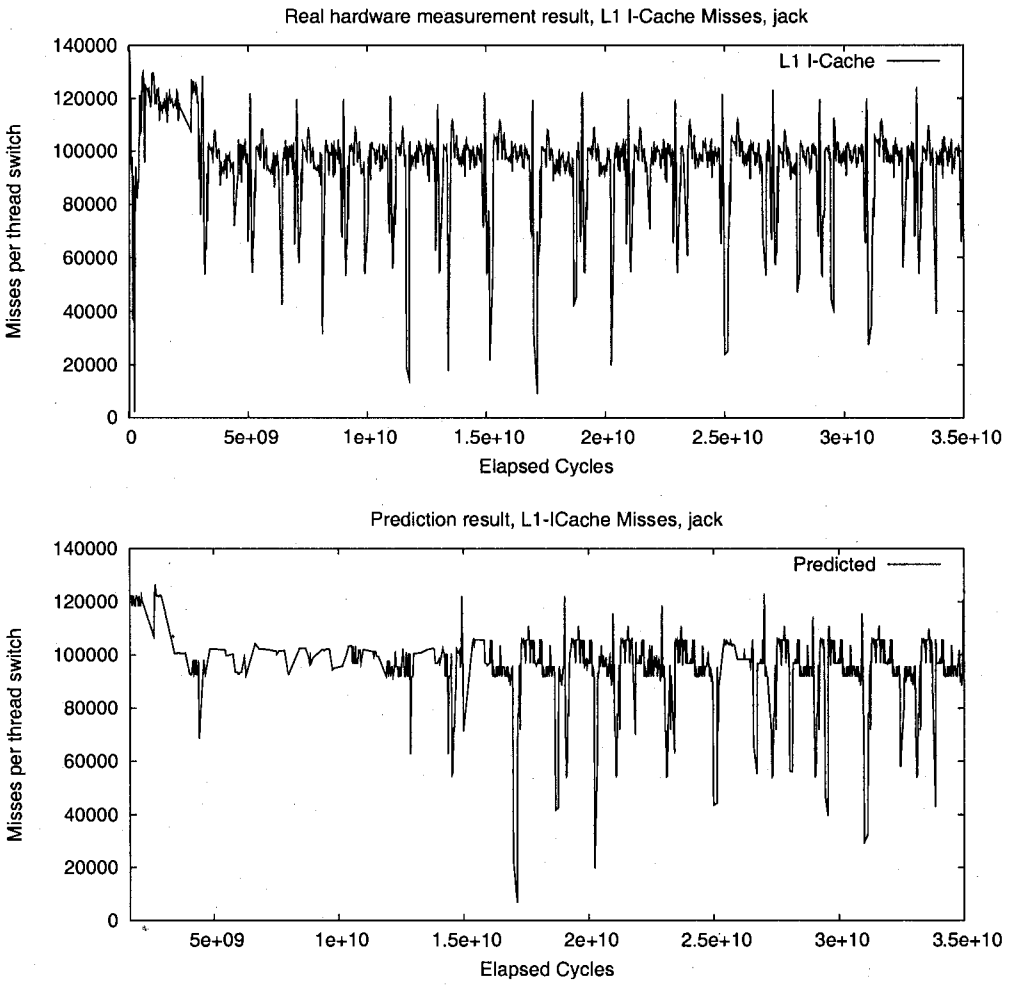

Figure 4.6: *The comparison between the real measurement result (top) and the phase prediction (offline pre-study) result (bottom) on* JACK. *The hardware event used here is Level 1 instruction cache.*

The bottom graph in Figure 4.6 shows a sample result from our offline pre-study work on JACK in predicting the L1 instruction cache miss data. The actual program execution is shown in the top graph. These results demonstrate that our solution performs well after the initial learning period, with most major features quite accurately predicted in the latter half of the program. This result is typical of the Java benchmarks we have investigated.

4.4.2 Distinguishing Characteristics of Our Approach

Our phase detection approach is a variable length periodic phase detection technique which is based on real world hardware event data, in coarse granularity (context switch) and is implemented in a pure online manner for non-scientific, general Java programs.

Most pre-works in phase detection are detecting stable phases. These approaches work well for flat, stable programs or on fine-granularity. Due to the irregular bahaviour of general Java programs and the coarse granularity that an online implementation can afford, we developed a variable length periodic phase approach which is different from many works.

The other variable phase detections presented in [GBEB04], [LPH+05] and [SZD04] require comparatively heavy offline data analysis. Furthermore, our approach is based on real world hardware events, implemented in an existing architecture, which is distinguishing from the works [BABD00, SSC03] that are also based on hardware data but implemented on hardware simulators.

4.5 Summary

In this chapter we gave a general introduction to the program phase detection area. In fact, the concept "program phase" is not well defined yet, by just using the definition for stable phases. We gave a definition to periodic phase for irregular, real life, object oriented programs investigated from the perspective coarse granularity.

We investigated a large set of different phase detection techniques and classified them into different types according to three concerns: *phase type, data source type* and *application time*. We used these three concerns to build a solution space for phase detection. Each technique can find its position in such a space. Our approach holds its own position,

different from all the pre-existing techniques.

Finally, we built an offline prototype to prove the feasibility of our online approach. The offline pre-study results illustrate that it is practical to make online phase detection on real world hardware for general Java programs. This study builds a solid base for our further work on phase detection techniques which will be introduced in later chapters.

Chapter 5
Hardware Based Online Phase Detection

In this chapter we introduce the core part of later work, detecting program phases from hardware information analysis. The output of our phase detector is used to support further optimizations introduced in later chapters. We begin with an overview of our phase detection technique in Section 5.1. In Section 5.2 we describe the details of our phase analysis design. A set of phase detection metrics presented by other researchers are introduced in Section 5.3. We also point out the limitations of these metrics if applied to variable length, recurrent, periodic phases and explain our evaluation metrics in the same section. Section 5.4 gives the experimental data and finally we summarize and discuss future work in Section 5.5.

5.1 Overview

Most programs are highly repetitive; a large portion of execution time is typically spent in just one or more small code segments. Detecting, or even predicting repetitive, "phase-like" behaviour can be important for many reasons, including program understanding, identification of execution "hot spots," runtime adaptation, and so forth. In Chapter 4.1, we have given an overview for program phase detection and the state-of-art techniques. Phase detection is a rather wide topic. We should be aware of the fact that phases can have different types and hold different properties. Both the application areas and mechanisms to detect different type of phases are varying. Similarly, we need different type of evaluation metrics

suitable to different type of phase problems. In this chapter, we will introduce an online algorithm for phase detection in Java programs based on real world hardware performance data. Moreover, we propose a pair of evaluation metrics for variable length recurrent periodic phase detection results.

Phases can have different properties; many phase analysis techniques concentrate on finding short-term, fixed-length phases representing periods of stable program execution. This is appropriate and reasonable for many programs, especially "regular" and scientific computations, but not necessarily appropriate for programs with more variable behaviour and/or more long-term phase structure.

Understanding performance, including the nature of program phases, requires understanding the underlying execution system as well as the program code. Modern processors are complex, with many internal components and designs; pipelines, multiple-level caches, TLBs, branch predictors, multiple cores, etc. These features are very effective, but introduce a significant amount of complexity when trying to determine why a program behaves the way it does. In Chapter 3 and in previous work [LSP+05, GVG06], we had shown that there exists a tight, and often unintuitive relation between the hardware performance and program behaviour. Hardware performance data is thus critical for developing a good understanding of program performance.

Recently, and following the general maturation of hardware performance monitoring techniques in commercial machine designs, hardware event data has begun to receive more and more attention as a basis for understanding program behaviour [SHC+04], detecting program phases [DCD03, BABD00, GBEB04], and for employing adaptive optimizations [DS02a, RSEW04, Jim05].

In this chapter we present an online technique to detect repetitive behaviour in Java program execution using hardware data. Our work considers the important problem of finding variable length periodic phases, something we show is usefully present in many programs. Our design is based on creating *patterns* representing the variation in hardware event data collected from low level hardware profilers. These patterns can then be used to detect higher-level phase changes, and incorporated into sophisticated table-based techniques to help predict program behaviour and guide runtime adaptation.

Formal evaluation of phase detection and prediction is of course critical to demonstrate

the quality of phase analysis. In previous works, a set of evaluation metrics for phase detection have been presented, including *sensitivity , stability, transition correlation, etc.*. However, all these existing metrics are either only suitable to measure stable phase detection results or are fairly naïve and cannot cover some essential aspects of variable length, recurrent phases. We thus propose *Confidence* and *Possible Miss Rate (PMR)* measures to quantitatively evaluate the quality of variable length, recurrent phase detection results. These calculations give a good understanding of the quality of phase data, and are the first such measures to be formally described. This pair of metrics is very helpful in selecting pattern creation algorithms that most effectively represent the similarity and regularity of the recurrent portions in the program execution. The final algorithm we choose results in high quality repetitive-phase detection.

5.2 Design

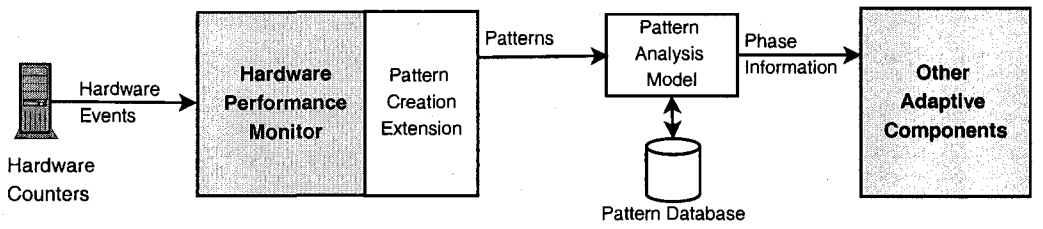

Figure 5.1: *System structure for recurrent phase detection.*

Our work is an extension to Jikes RVM [AAC+99], and Figure 5.1 shows an overview of the design. Raw hardware event data is read from hardware counters through the *hardware performance monitor* (HPM), a pre-existing component in Jikes RVM. We augment the HPM with a *pattern creation extension* that generates *patterns* representing the hardware performance. This analyzes the hardware data between two consecutive sample points, summarizing the "shape" or pattern of variation in low level performance. If we observe that the same sequence of variation in events has been encountered before, a (new) repetitive sequence will be considered.

Created patterns are transferred to a *pattern analysis model* for deeper analysis. The

pattern analysis model maintains a *pattern database* to store the received patterns. The pattern analysis model makes the ultimate decision on the identification of and response to phase changes. Below we describe the two main mechanisms in more detail: the *pattern construction mechanism,* and the *pattern analysis and prediction.*

5.2.1 Pattern Construction

A wide variety of properties of hardware events can be used to detect repetitive behaviour: increasing or decreasing trends, range of variation, and distance and similarity measures of various forms. Obviously there are trade-offs in terms of complexity and data size (cost) and improvements to phase detection and prediction. In order to select appropriate properties and pattern building strategies, we implemented a variety of heuristics and evaluated them quantitatively using the metrics developed in Section 5.3.2. Here we present our most successful and general approach in detail. As shown in Figure 5.2 this design is mainly based on three attributes of the hardware event curve: the *level* of variation (as shown in the top graph), the *shape* or the direction of variations (second from the top), and the *length* of the more significant varying part of each repetitive period of the curve (third graph from the top). Finally, we cut the whole curve into recurrent phases based on the similarity between the beginning parts of each period according to these three attributes (bottom graph).

Our implementation summarizes this low level behaviour using (short) bit-vectors that encode the overall pattern of variation. Translating hardware event data to bit-vector patterns involves first coarsening the (variation in) data into discrete *levels,* and then building a corresponding bit-vector *shape* representation.

- "Levels": A basic discretization is applied to (variations in) event density data to coarsen the data and help identify changes that indicate significant shifts in behaviour. We compute the density of events over time for each sample. By comparing the density of the current sample with that of the previous sample, we obtain a variation V. The variation V is discretized to to a corresponding level, P_V. For the number of levels, we test a series number of power of 2, *e.g.*, 2, 4, and 8. We decide to categorize the variation into four levels for an optimal tradeoff between the ability of distinguishment, the noise tolerance, and the overhead of encoding.

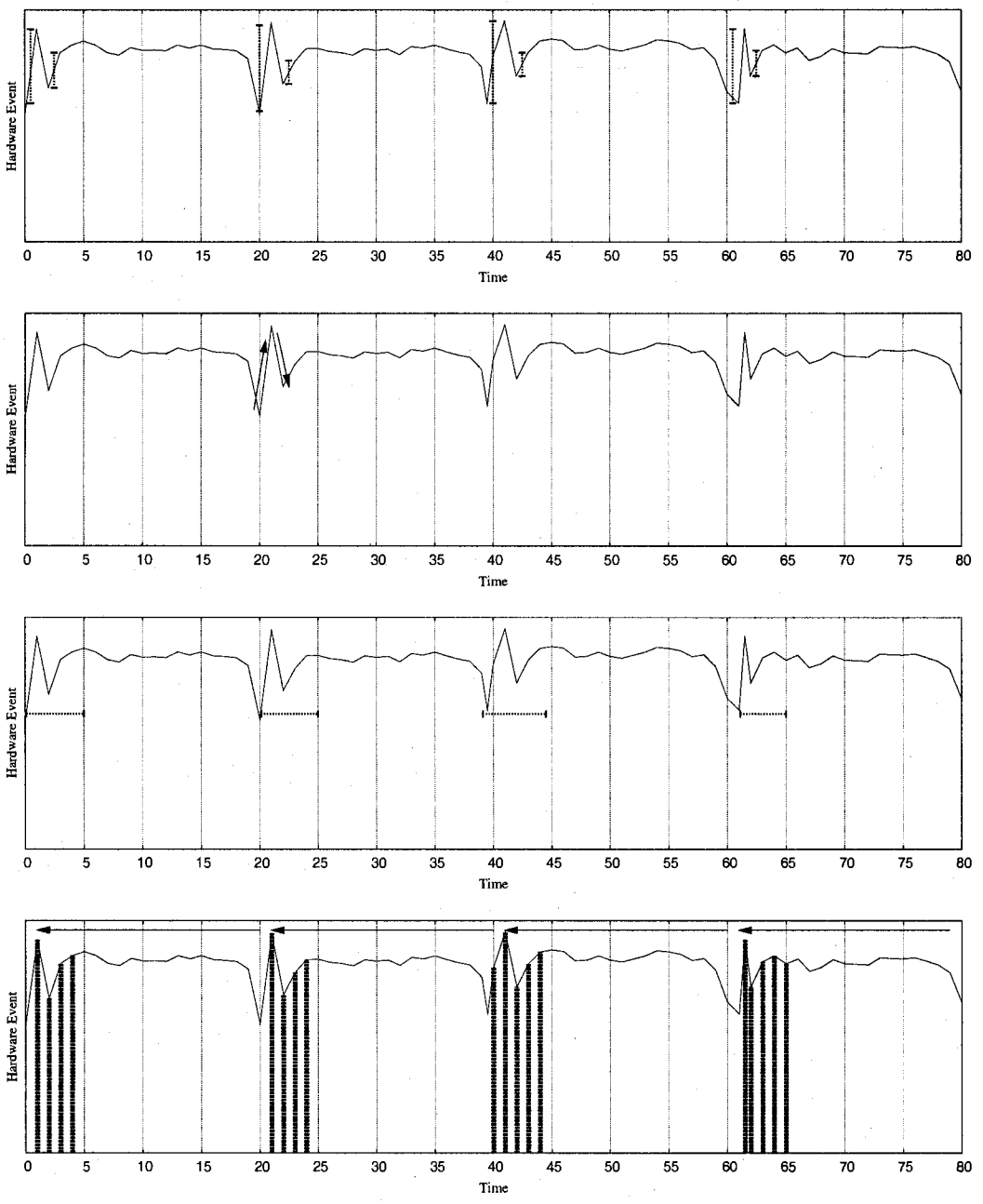

Figure 5.2: *Main attributes used to build patterns. The top three graphs show the three attributes of the hardware event curve: the variation level, the variation shape, and the length of the significantly varying part. The bottom graph shows the result of recurrent phase identification based on similarity of the beginning part of each phase.*

- Pattern "shapes" are then determined by observing the direction of changes, positive or negative, between consecutive samples. Complexity in shape construction is mainly driven by determining when a pattern begins or ends.

  Each shape construction is represented by a pair $(P_V, \vec{v})$, where P_V is a level associated with the beginning of the shape, and $\vec{v}$ is a bit-vector encoding the sign (positive, negative) of successive changes in event density. Given data with level P_V, if there is no shape under construction a new construction begins with an empty vector: $(P_V, [\,])$. Otherwise, there must be a shape under construction $(Q_W, \vec{v})$. If $Q_W = P_V$, or we have seen $Q_W > P_V$ less than n times in a row, then shape creation continues based on the current shape construction $(Q_W, \vec{v})$: a bit indicating whether $V > 0$ or not is added to the end of $\vec{v}$.

  The following conditions terminate a shape construction:

  1. If we find $Q_W < P_V$ we consider the current shape building complete and begin construction of $(P_V, [\,])$. Increases in variation of event density are indicative of a significant change in program behaviour, and so motivate the decision to begin a new phase.
  2. If we find $Q_W > P_V$, n times in a row the current shape has "died out." In this case we also consider the current shape building complete. In our experiments we use $n = 2$, which is long enough. In our observation, it is extremely rare that a major variation will happen after two very flat intervals coming in a row.
  3. If in $(Q_W, \vec{v})$ we find $|\vec{v}|$ has reached a predefined maximum length we also report the current construction as complete. In our experiments we use a maximum of 10 bits. We thus can store the patterns in direct-mapping table structure of less than 1K entries.

A rough overview of the pattern creation algorithm is shown in Figure 5.3. After obtaining hardware data D, we compute the variation V between D and the same data (D_{last}) for the previous interval. V is then mapped from a real value to an integer value $P_V \in \{0, \ldots, n\}$, representing the "level" of V. As shown in the formal description of this algorithm, we use Q_W to represent the level of the pattern currently under construction. Initially the value of

Q_W is set to -1 to indicate no pattern is under construction. If $P_V > Q_W$ then we are facing a larger, and hence more important variation than the one that began the current pattern construction. The current pattern is thus terminated and and a new pattern construction associated with level P_V is begun. The value of P_V is assigned to Q_W and the shape code vector (denoted as *ShapeCode* in Figure 5.3) is blanked. Otherwise ($P_V \leq Q_W$) and the current pattern building continues.

The actual pattern encoding is based on the relation between P_V, Q_W and the sign of V. Two bits will be appended to the current *ShapeCode* each time a pattern grows: "01" means a positive variation at level Q_W, "10" represents a negative variation at level Q_W, and 00 means either a positive or negative variation at a level below Q_W. Binary 1s in our scheme thus indicate points of significant change. Construction continues until one of the pattern termination conditions is met, at which point we report the pattern to the pattern analysis model. A concrete example of the creation of a pattern is shown in Figure 5.4.

Of course choice of primary data is also important; the above strategy can be applied to many different hardware events. In our actual system we make use of the instruction cache miss density as a good indicator of code activity. We have considered other hardware events and combination of events (see page 123), but a thorough study is left for future work.

5.2.2 Pattern Analysis and Prediction

Pattern analysis and prediction consumes patterns generated by the pattern creation module. Here we further examine the patterns to discover recurrent phases and generate predictions of future program behaviour. All created patterns are stored into a *pattern database.* The recurrent pattern detection and prediction are based on the information in the pattern database and the incoming pattern.

The recurrent detection is straightforward: if we find a newly created pattern that shares the same pattern code as a pattern stored in the pattern database we declare it to have recurred. An actual recurrent phase, however, is not declared unless the current pattern also matches the prediction results.

The prediction strategy we use is a variant of fixed-length, local/global mixed history,

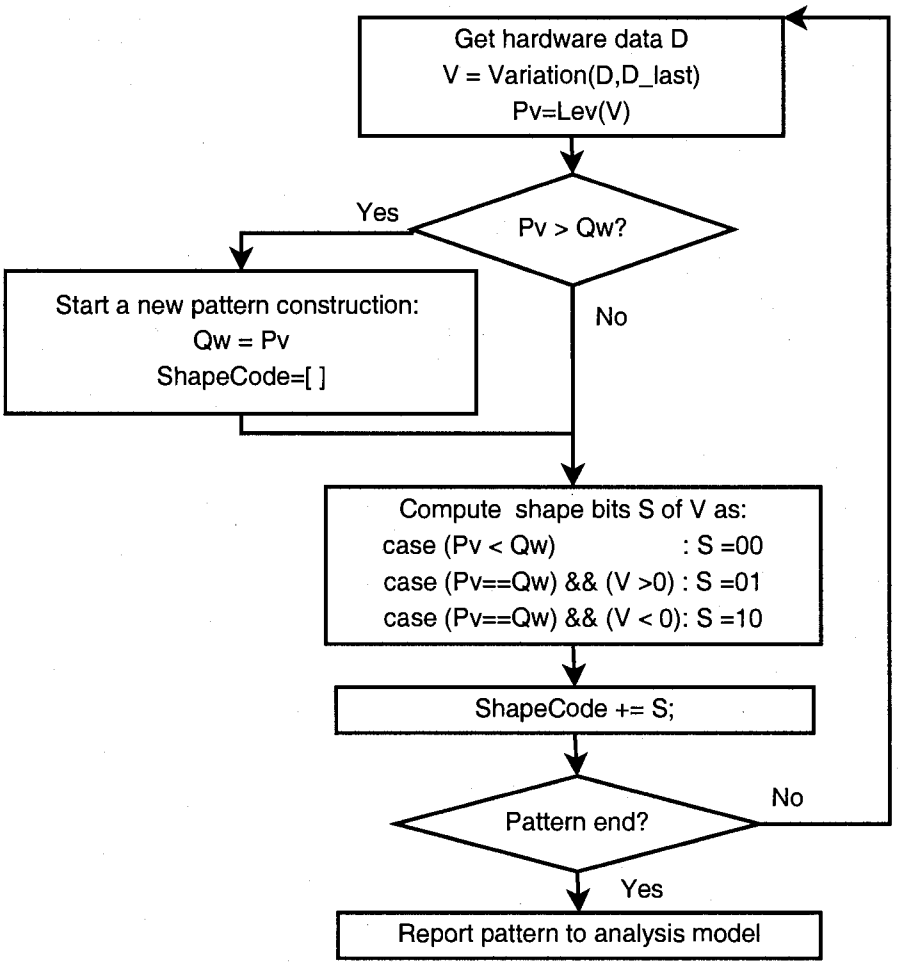

Figure 5.3: *A flow chart for pattern creation.*

table-based prediction. Unlike more direct table-based methods, our predictions include an attached "confidence" value; this allows us to track multiple prediction candidates and select the most likely.

Figure 5.5 gives an overview of our prediction scheme. For each pattern, we keep the three most popular repetition "distances" from a former occurrence to a later one—the use of three candidates is based on experimentally balancing predictor performance and accuracy. In our initial experiments, we notice a long pattern recurrent period can be interrupted by a shorter period in the middle. By tracking three distances longer periods

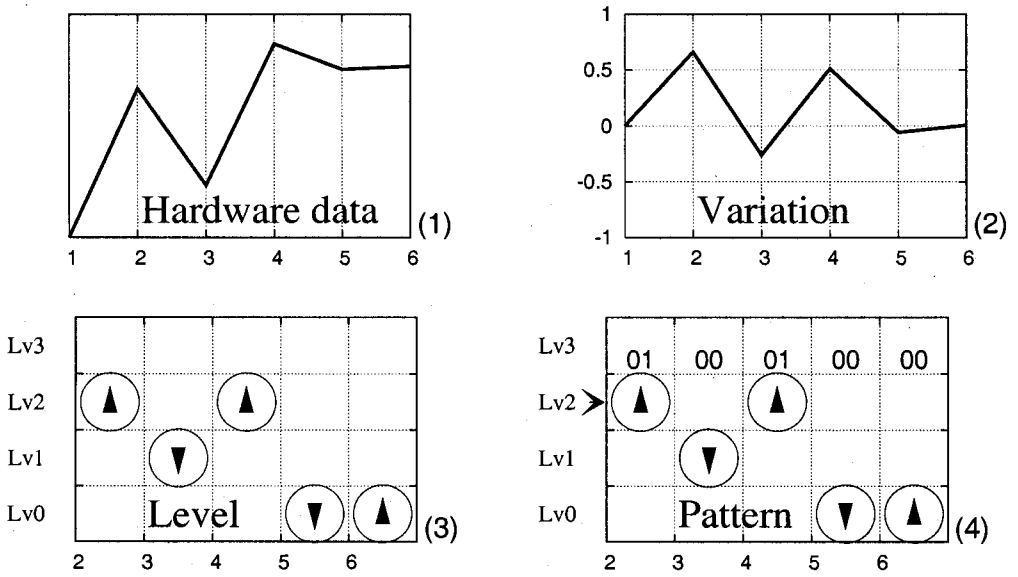

Figure 5.4: *Pattern construction example. (1) Acquire the raw hardware data. (2) Calculate the variation between consecutive points. (3) Coarsen the variation into different levels; the triangles inside each circle show the direction (negative/positive) of variation. (4) The final pattern creation results; the arrow on the y-axis indicates that we obtain a level 2 pattern; the number above each circle shows the 2-bit code for each variation. The four trailing zeros are omitted (the pattern has died out), and the final pattern code is 010001.*

are better able to survive. Prediction updates are performed by heuristically evaluating these distances for a given incoming pattern to find the most likely, variable-length pattern repetition. Our *tri-distance selection algorithm* updates the likely choices for an incoming pattern p by tracking three repetitions D_i, $i \in \{0,1,2\}$:

- For each D_i we keep a repetition length L_i, measured by subtracting time stamps of occurrences, and a "hotness" value H_i.
- The difference T_i between the current pattern occurrence p and the ending point of each of D_i is calculated.
- If the different rate $DR_i = \frac{|T_i - L_i|}{L_i} \times 100\%$ between T_i and L_i is smaller than a threshold T, the hotness H_i is doubled. The hotness of the best fit distance gets a further

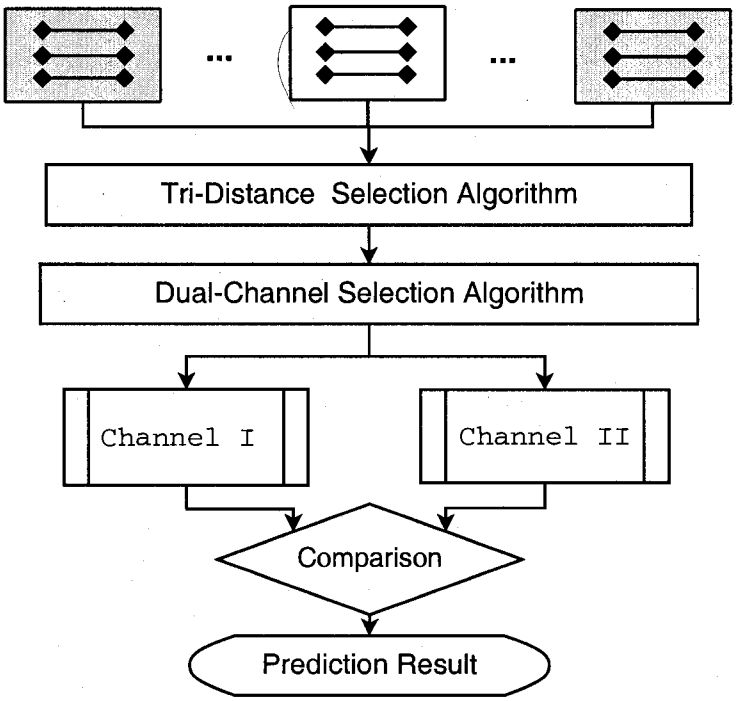

Figure 5.5: *Overview of the prediction mechanism.*

doubling. We then right shift of the hotness values of all the three distances. Consequently, if the different rate of a distance is larger than T, the hotness of it is decreased to a half. This adaptive approach ensures new, hot patterns can be quickly recognized and less useful aging patterns to be discarded.

- If all the different rates of the three distances are larger than T, we replace the D_j with the lowest hotness with a new D_j. The length, L_j is based on the distance to the closest of the current set of D_i, and hotness, H_j, is initialized to a constant value representing a low but positive hotness in order to give the new pattern a chance to become established. The value of H_j is chosen as five arbitrarily, which will not become zero in two consequent right shift operations.

- We use the D_i with the greatest hotness as the prediction result; H_i further functions as a confidence value for this prediction.

Steps	Distances			Events and Actions
1	D_1[100,200]	$L_1 = 100$	$H_1 = 10$	Get pattern at time 297
	D_2[70,180]	$L_2 = 110$	$H_2 = 7$	
	D_3[150,200]	$L_3 = 50$	$H_3 = 5$	
2	D_1[100,200]	$L_1 = 100$	$H_1 = 10$	Compute $DR_1 = 3.0\%$
	D_2[70,180]	$L_2 = 110$	$H_2 = 7$	Compute $DR_2 = 6.4\%$
	D_3[150,200]	$L_3 = 50$	$H_3 = 5$	Compute $DR_3 = 194.0\%$
3	D_1[100,200]	$L_1 = 100$	$H_1 = 40$	Update the hotness
	D_2[70,180]	$L_2 = 110$	$H_2 = 14$	values, according to
	D_3[150,200]	$L_3 = 50$	$H_3 = 5$	the different rates
4	D_1[100,200]	$L_1 = 100$	$H_1 = 20$	Shrink the hotness
	D_2[70,180]	$L_2 = 110$	$H_2 = 7$	values to half
	D_3[150,200]	$L_3 = 50$	$H_3 = 2$	
5	D_1[200,297]	$L_1 = 97$	$H_1 = 20$	Use D_1 as the
	D_2[70,180]	$L_2 = 110$	$H_2 = 7$	prediction and
	D_3[150,200]	$L_3 = 50$	$H_3 = 2$	update its content

Table 5.1: *A concrete example of the tri-distance algorithm. The difference threshold T is set to 10%.*

A concrete example is shown in Table 5.1. Here we set the different threshold T as 10%. In the first step, a pattern comes at time slot 297 and the state of the distances are shown in the second column. We then compute the different rate for each distance in step 2. In step 3, the hotness values are updated according to the different rates. D_1 fits the best and thus the hotness of D_1 is increased four times to 40. In step 4, we shrink all hotness values to half. The hotness of D_3, whose different rate is larger than T, is thus shrunk to a half of the original value in step 1. Finally, in step 5, D_1 is chosen for prediction and the beginning point, the ending point, and the length are updated reflecting the latest pattern.

With the current prediction updated, we then make a final prediction from the global set of pattern updates. It is frequently true that the current prediction, even if with a high confidence value, does not belong to the most important recurrence of a program. We

thus use two global prediction "channels" to avoid losing the more important prediction in the history. We found that using two channels is sufficient to identify the most important active period while keeping the overhead low. This setting also aims to limit the cost of choosing among all possible patterns. Our *dual-channel selection algorithm* is similar to the tri-distance selection algorithm:

- We have two prediction channels; each stores a prediction from a pattern, and each channel holds a "hotness".
- If the current prediction from the tri-distance predictor matches one of the prediction channels, the channel's hotness is increased by the prediction confidence
- In the case that the current prediction matches neither of the prediction stored in the channels, the coldest channel is replaced by the current prediction.
- The channel with a higher hotness determines the global prediction result.
- After each global prediction, the hotness of both channels is shrunk to half.

5.3 Evaluation Metrics

Any specific technique provides a specific solution to a specific problem. To evaluate the result of a technique, we specify metrics that cover the most important characteristics of the specific problem on which the technique is applied.

A number of phase detection evaluation metrics have been provided. These metrics are mainly rooted in their experience in different type of phase detection problems and cover the most important aspects of particular type of phase detection problems.

In this section, we first give an introduction to the existing evaluation metrics. Most of them are designed for evaluating stable phase detection results, and are thus not suitable for our case. We then present a pair of novel metrics to measure the result of variable length recurrent phase detection problems. This evaluation pair covers two important aspects of recurrent phase detection: *similarity* and *regularity*.

5.3.1 Existing Metrics

Most stable phase detectors are based on measures of fixed length intervals. Basically, a program is split into a set of flat, stable portions, called phases. The special portions between phases are named phase transitions. Most of the existing metrics are based on measures for these two different states of programs.

5.3.1.1 Stability and Average Phase Length

An outstanding stable phase detector should logically detect more stable phases than other detectors when applied to the same program. Dhopapkar *et al.* [DS03] employ *stability* and *average phase length* to compare phase detection results from different algorithms.

Stability is defined as the fraction of intervals that belong to a detected stable phase; a higher stability means a more complete coverage of the program. Similarly, *average phase length* is defined as the number of intervals that are part of stable phases, divided by the total number of stable phases.

This pair of metrics are based on an assumption that phase detection is applied to a program with a large proportion of stable/flat portions. The metrics pair considers the phase detection output with a larger number of intervals identified as stable and with longer continuous flat intervals as a better result. These metrics are thus particularly designed for stable phase detection and are not suitable for examining periodic phase detection techniques working on a data set with large variations.

5.3.1.2 Sensitivity and False Positives Rate

T. Sherwood *et al.* [SSC03] present a pair of measures to evaluate how often a phase detection algorithm identifies phases correctly.

Sensitivity measures the ability of a phase detection mechanism to identify a phase change after there is a "significant" performance change. It is defined as the fraction of intervals showing significant performance changes with respect to the preceding interval over all intervals.

The *false positive rate* is the fraction of intervals where the performance shows no

"significant" change, but is nevertheless claimed as a phase transition by the detector.

Both of these measurements are dependent on the definition of "significant" changes. They are not suitable for the case where there is no long term stable phases with respect to a single level of granularity. For the situation shown in Figure 4.5 on page 57, if the value of "significant" is set to a too small value, we will determine that each interval is different from the preceding interval. Otherwise, if the significance is set too high, we will end-up with an equally meaningless result, saying all intervals are in the same phase. Even if we are lucky enough to select the optimal significance, this pair of metrics still fails to examine one important aspect of the periodic phases: the regularity of the recurrences of phases.

Different from other metrics, the false positives rate measures the result from the down-side employing the concept that a better solution should also make fewer wrong decisions. We also use the same concept when developing our evaluation metrics specific to recurrent periodic phase detection.

5.3.1.3 Transition Correlation and Accuracy Score.

Nagpurkar *et al.* [NHK+06] propose an evaluation strategy based on a theoretical perfect phase detector. The perfect detector provides a "correct" phase boundary solution for a particular program's execution. By comparing the results of the perfect detector and a given, real detector they define the *transition correlation* as

$$TrCorrelation = \frac{BothInPhase + BothInTransition}{TotalEvents}$$

BothInPhase is the total number of profile elements for which both detectors agree it is in a stable phase. Similarly, *BothInTransition* is the total number of profile elements for which both detectors agree it is in a period of phase transition.

In combination with *Sensitivity* and *False Positive*, they further introduce a novel accuracy scoring metric, defined as

$$Score = \frac{TrCorrelation}{2} + \frac{Sensitivity}{4} + \frac{False\ Positive}{4}$$

The *Score* weights correlation equally with the sum of sensitivity and false positive. Although reasonable, the authors did not provide data or arguments to support the values of weights they used, and other weightings and combinations might be possible as well.

Again, however, this pair of metrics is also designed for stable phases. More importantly perhaps, it requires a "perfect detector" which is not available in many cases. Their accuracy scoring metric examines the results from multiple directions. This idea is helpful for us to design our evaluation metrics that will be introduced in Section 5.3.2.

5.3.1.4 Performance Variance and Coefficient of Variation

In the case where program presents relatively small performance variations, a small *performance variance* in a stable phase is a sign that the phase detector has identified the phase boundary correctly [SSC03]. A poor phase detection result will show a comparatively large performance variance within a phase due to the inclusion of more intervals than is strictly necessary. Of course the concrete definition of this metric must be considered in the context of the whole program variation, and thus is highly application-specific.

Coefficient of variation (CoV) is a statistical measure of standard deviation as a percentage of the average:

$$CoV = \frac{stddev}{mean}$$

Here, *stddev* stands for standard deviation; *mean* is the average of all measures of the intervals in the same phase.

For stable phase detection, a lower CoV is desired; in an extreme case, all the intervals in a detected phase would have exactly the same value in the measurement data, resulting a CoV of zero, or perfect phase identification.

This metrics pair still assumes that intervals identified in the same phase performs similarly. Thus, these metrics still only work for stable phase detection results. Different from other existing metrics, this metrics pair makes use of statistical computations other than just fractions between the counts of intervals.

We aim at detecting periodic phases in Java programs that show larger performance variance. More sophisticated statistical measures are required to give an evaluation on the phase detection algorithms. The existing metrics are not enough for this situation, but some ideas behinds these metrics are also helpful for us to design new evaluation metrics suitable to recurrent periodic phase detection.

5.3.2 Periodic Phase Evaluation

We have introduced a number of evaluation metrics for stable phase analysis results in Section 5.3.1. Meanwhile, we also mentioned the reason why these metrics are not suitable for measuring the variable-length program periodicity we investigate here. Nevertheless, some idea behinds these metrics are valuable for developing new metrics, such as:

- To evaluate the result of a phase detection technique, it is not enough to only consider one simple measure; a combination of multiple measures must be considered.
- To evaluate the results of a phase detection technique, it is not enough to only consider positive cases, but also the negative cases as well.
- To evaluate the results of a phase detection technique, it is not enough to only consider simple rates between different types of counts. Novel statistical computations are required.

We define two metrics, *Confidence* and *Possible Miss Rate (PMR)*. *Confidence* gives a measure of the similarity between repetitive periods identified by our algorithm, while *PMR* measures the amount of execution which could have been identified as repetitive but which was not done so by the phase detection algorithm. These metrics are well-suited long term variable length phase, and are practical to compute as well.

Both *Confidence* and *PMR* are based on the same pair of fundamental metrics measuring the *similarity* and *regularity* between execution segments that may be allocated to the same repetitive group, *i.e.*, the instances (occurrences) of the same pattern.

Suppose we have a pattern P which has N instances. All the instances P_i compose a group, which can be represented by an ordered set $G(P) = \{P_i | i = 1, 2, ..., N\}$. Each instance P_i is actually a segment in a program execution, and can be formally represented as $P_i = [b_i, e_i]$. The b_i and e_i are the start and end time, measured as the time stamp number of the data collection points of the segment P_i respectively. We also have $b_{i+1} > e_i$, which means the beginning point of P_{i+1} is later than the ending point of P_i. We use two basic metrics to quantify the similarity and regularity of a set $G(P)$:

- **Similarity:**

We calculate the *Pearson correlation* between each pair, P_x and P_y, in $G(P)$ as in in formula 5.1,

$$C(P_x,P_y)=\frac{\Sigma P_xP_y-\frac{\Sigma P_x\Sigma P_y}{N}}{\sqrt{(\Sigma(P_x)^2-\frac{(\Sigma P_x)^2}{N})(\Sigma(P_y)^2-\frac{(\Sigma P_y)^2}{N})}} \tag{5.1}$$

Given that $G(P)$ has N items, we can obtain totally $\frac{N(N-1)}{2}$ pearson correlation results between each two segments in $G(P)$. We thus use a mathematical average to represent the similarity of a group. We denote this value as $C_{G(P)}$:

$$C_{G(P)}=\frac{\Sigma C(P_x,P_y)}{\frac{N(N-1)}{2}}=\frac{2*\Sigma C(P_x,P_y)}{N(N-1)},(x=1,2,...,N-1;y=x+1,x+2,...,N)$$

- **Regularity:**

  The difference between start times for each pair of adjacent P_i provides a basic "distance" measure between pattern instances, *i.e.*, the distance between $P_x=[b_x,e_x]$ and $P_y=[b_y,e_y]$ is b_y-b_x. Without losing generality, here we assume $b_y>b_x$. The extent to which pattern instances are well clustered shows regularity; we measure it using a *k-means* clustering algorithm [McQ67] applied to the set of all distance pairs. For each cluster, we obtain the absolute value of the difference between each pair of item and the centroid of the cluster. The sum of all these differences becomes a measure of the regularity of the pattern group $G(P)$, and we denote this value as $D_{G(P)}$.

Combining the above calculations, we provide an overall evaluation of $G(P)$ as:

$$E_{G(P)}=C_{G(P)}*D^{-1}_{G(P)}$$

Given different repetition detections for the same pattern P a higher $E_{G(P)}$ heuristically indicates better results.

Our actual metrics can now be defined in terms of the above calculations.

- **Confidence:**

  For each $G(P)$, we generate a set $\hat{G}(P)^j$ by removing the j^{th} pattern instance of $G(P)$. If $\hat{G}(P)^j$ has a better quality (a higher value of E) than $G(P)$, then we have less

confidence on the j^{th} pattern instance being a member of the group, and thus reduced confidence in the grouping itself. Otherwise, the j^{th} instance makes the whole group better and improves confidence.

We thus give a confidence score *Conf(P_j)* of j^{th} item of $G(P)$ as:

$$\text{Conf}(P_j) = \begin{cases} 1.0 & E_{G(P)} > E_{\hat{G}(P)^j} \\ \frac{E_{G(P)}}{E_{\hat{G}(P)^j}} & \text{Otherwise} \end{cases}$$

Confidence in the detection results of pattern P, denoted as *Conf(P)*, is then the sum of *Conf(P_j)* for all j.

Our final *Confidence* in a complete detection result on all m patterns $P^1, P^2, ..., P^m$ appearing in the result is the sum of confidence in each pattern weighted by the number of the instances of the pattern.

Confidence basically indicates the degree to which the pattern detection results represent at least a local maximum. High confidence indicates patterns are well-categorized, while low confidence suggests some execution segments may be misclassified.

- **Possible Miss Rate (PMR):**

The *PMR* evaluates how much of the execution was potentially mis-identified as non-repetitive. We define it as follows:

$$PMR = \frac{Number\ of\ PMPI}{Number\ of\ PMPI + Number\ of\ DPI} \tag{5.2}$$

In formula 5.2 above, *PMPI* stands for "Possible Missed Pattern Instances" and *DPI* represents "Detected Pattern Instances". Somewhat dual to *Confidence,* the fundamental idea of *PMR* is to add an execution segment as an instance of a pattern and check whether this new grouping is better or worse.

Given a pattern detection result $G(P)$, we treat all the execution segments that are not covered by $G(P)$ as potential elements of *PMPI.* We then insert each such execution segment s into $G(P)$ and build a new group $\check{G}(P)^s$. Segment s is then included as a member of *PMPI* if $E_{G(P)} < E_{\check{G}(P)^s}$.

Similar with that of the false positive rate described in page 77, the purpose of *PMR* is also to measure the quality of the "noise resistance" property of a detection algorithm. The difference here is that we use more novel statistics suitable for our case, detecting recurrent period phases with less or no flat stable intervals.

5.4 Experimental Results

In this section, we make use of the metrics developed in the previous section to experimentally evaluate our technique.

5.4.1 Setting and Benchmarks

Our implementation is based on Jikes RVM 2.3.6; results were measured on an Athlon 1.4GHz workstation with 1GB memory (Debian Linux, 2.6.9 kernel). We report phase detection results derived from L1 instruction cache miss events. Benchmarks include the industry standard SPECJVM98 suite [Stac], and two larger examples, SOOT and PSEUDOJBB. SOOT is a Java optimization framework which takes Java class files as input and applies optimizations to the bytecode; in our experiments, we run SOOT on the class files for JAVAC in SPECJVM98 with options "–app -O". The benchmark PSEUDOJBB is a variant of SPECJBB2000 [Sta00] which executes a fixed number of transactions in multiple warehouses. Our experiments run one to eight warehouses, 100 000 transactions in each warehouse. For SPECJVM98 we use the recommended (large) input size "-s 100". For quality analysis we built a canonical sample profile from 15 typical runs, while the phase driven profiling results are the average of 5 runs. The threshold T for tri-distance selection is set to 10%. Note that all the experimental results reported in this thesis use the same system setting and benchmark suite introduced here. We will not repeat this basic system and benchmark parameter settings in later chapters.

5.4.2 Results

We implement our online phase detection algorithm introduced in Section 5.2 in Jikes RVM. Trace files recording pattern creation are generated and an offline analysis is ap-

plied to evaluate the quality according the metrics described in Section 5.3.2.

The results are given in Table 5.2. The five data columns (columns 2 through 6) are the number of different patterns, the number of occurrences of all patterns, *Confidence* results, *PMR* results and *PMR* results on the most important (major level) patterns.

Benchmark	Number of Patterns	Number of Occurrence	Confidence	PMR (%)	PMR Major(%)
compress	32	158	0.94	60.78	2.78
db	29	451	0.95	35.94	1.25
jack	29	352	0.94	22.65	0.05
javac	23	214	0.93	32.42	6.58
jess	25	182	0.88	48.71	5.88
mpegaudio	28	111	0.91	68.71	13.49
mtrt	27	78	0.85	27.58	0.10
raytrace	18	69	0.85	16.17	4.44
soot	49	11106	0.99	28.45	0.03
PseudoJbb	35	7093	0.98	37.80	0.01
Average	—	—	0.92	37.98	3.46

Table 5.2: *Pattern detection evaluation results. Hardware patterns are built based on performance data of L1 instruction cache.*

On average we have a 92% *Confidence* that the segments identified by our algorithm are actual repetitive portions. Unfortunately we also have a comparatively high average *PMR*, 38%. This means we potentially miss over a third of repetitive segments in the execution. Deeper investigation shows that most of the missed segments are likely instances of patterns at the lowest levels (0 and 1). As described in Section 5.2, pattern constructions at lower levels can be interrupted when a higher level variation is encountered. It is therefore not surprising that many possible repetitions of lower level patterns are ignored by our algorithm; larger, more significant changes are expected to be more important for capturing the important repetitive behaviour of a program, and our algorithm weights such patterns higher. In Table 5.2, the "PMR Major" column gives the PMR value for only the upper range of variance (levels 2 and 3). For these signals the data shows that we only miss on

average about 3.5% of possible repetitive periods.

We had tried other solutions for pattern creation and recurrence detection. Absolute values vary a lot from program to program and are not appropriate to build a universal discretization scheme for all programs. Variants such as considering only upwards or downwards variations, encoding the distance between upwards and downwards performance directions, using finer or coarser level settings (8 levels or 2 levels) were also investigated. Our design was represented we feel a local optimum with respect to obvious variation in design parameters, and reasonable different settings showed no further general improvement over our basic design for variable length phase detection.

5.5 Summary

In this chapter, we presented our approach for online phase detection for general Java programs. Our technique is based on real world hardware information. There exist a number of evaluation metrics for phase detection. However, most of them are designed only for stable phase based on fixed interval measures. Some of them use counting on intervals directly; some others make an assumption that the phases are flat and stable portions in program executions. Since the existing metrics are not suitable for evaluating long term, highly varying, periodic phases in general Java programs, we defined a set of novel metrics to evaluate our results. Our experimental data demonstrates that our phase detection and prediction mechanism can provide accurate results. On average, we have a high confidence in the phase detection results, and our algorithm only misses a small number of possible repetitions in program execution at major variation levels.

In upcoming chapters, we will show a series of runtime adaptive applications based on our phase detection mechanism. As consumers of our phase information, we are able to use them to further confirm the correctness and accuracy of our phase analysis results. They are adaptive optimizations by themselves at the same time.

Chapter 6
Phase Based Selective Profiling

Profiling is essential to some runtime and offline optimizations. In this chapter, we present a selective runtime profiling technique which uses our hardware phase detection mechanism. In Section 6.1, we first categorize the profiling technique and mention the contributions of our work. The most important related works are given in Section 6.2. The implementation details and the evaluation metrics of our profiling technique are discussed in Section 6.3. Experimental results are presented in Section 6.4 and we summarize this chapter in Section 6.5.

6.1 Profiling Categorization

Program profiling is an important technique for understanding the dynamic behaviour of programs. To application developers, profiles provide insight into a program's resource utilization and help to identify performance bottlenecks. For compiler constructors, profiling data can be used to guide static code optimization. For the designers of JVMs or other runtime environments, both online profiling results or offline analysis on profiling data can be used to improve runtime adaptive strategies.

Profiling data can be produced in some different ways.

- **Program Instrumentation**

By inserting intrusive instrumentation in a running program, a wide variety of profile data can be collected completely at a fine granularity. However, complete profiling at a fine granularity can bring intolerable overhead for runtime system or huge trace files for offline applications.

- **Sampling**

  A sampling mechanism allows the system to collect a subset of the profiling events. Most sampling systems are timer based; they examine events and program states only once per timer interval or timeout. Sampling techniques can greatly reduce the cost of profiling over more exhaustive techniques, albeit with a lower accuracy.

- **Selective Profiling**

  General instrumentation or sampling techniques are actually applied throughout the whole life of a program, *i.e.*, profiles are continuously taken. However, a program usually does most of its work in a comparatively small portion of its code. Hence, it is not necessary to take profiles continuously across the whole program execution. Compared with continuous sampling profiling, selective profiling can reduce the profiling overhead while keeping profiling accuracy at the same time. The key point to benefit from selective profiling is to choose critical profiling points that reflect the most important periods, or phases, of program execution.

Profiling techniques that provide detailed/accurate information with low overhead are especially important for runtime environments. Even for offline work, selective profiling can reduce the size of trace files largely without losing of important information. In this chapter, we introduce a selective runtime profiling technique based on our hardware phase detection results. We use low overhead hardware monitoring to reduce about half of the profiling workload with almost no degradation in profiling accuracy.

6.2 Related Work

One of the crucial technical challenges for adaptive optimizations is to gather accurate profiling data with as low an overhead as possible. Profiles can be obtained from program

instrumentation or from a sampling scheme. By adding instrumentation into a program, we can gather accurate profiles at a fine granularity, and instrumentation techniques are widely used in doing adaptive optimization. Dynamo [BDB00], for example, uses instrumentation to guide code transformations. Instrumentation techniques are also very useful in program understanding; Daikon [EPG+06] is a system for dynamic detection of likely invariants in a program through instrumentation. Even commercial JVMs provide a basic instrumentation interface through Sun's JVMTI specification [Sunb]. Unfortunately, instrumented profilers can also be fairly heavyweight, producing potentially large runtime overheads [CKJA98, CFE99]. This inspires work on reducing instrumentation overhead reduction, such as that by Kumar *et al.* in their "INS-op" system that optimizes (reduces) instrumentation points [KCS05].

Alternatively, runtime profiles can be gathered by sampling. In a sampling approach, only a subset of the execution events are considered, and this can greatly reduce costs. Many systems, such as the Jikes RVM [AFG+00], use a timer-based approach to determine sampling points. On some other systems, such as IBM's Tokyo JIT compiler [SYK+01] and Intel's ORP JVM [CEG+05], a count-down scheme is used. An optimization candidate method is chosen when an associated "counter" reaches a pre-defined value. Arnold and Grove [AHR02] present an approach that combines the timer-based and count-down schemes; based on the original timer-based scheme in Jikes RVM, a stride counter is set to control a series of non-contiguous burst count-down sampling actions.

A sampling-based strategy allows the the system to reduce the profiling overhead with the profiling accuracy as a tradeoff. Many techniques have been developed to reduce profiling overhead while maintaining profiling accuracy at a reasonable level. Zhuang *et al.* [ZSCC06], for instance, develop an adaptive "bursting" approach to reduce the overhead while preserving accuracy. The key idea of this work is to do detailed, heavy profiling only at selective points.

Our work uses program *phase* information to reduce the profiling workload. Phase information can be very useful in locating stable or repetitive periods of execution at runtime, and has been used in various adaptive optimizations [CH02, SZD04, NKS05] and designs for dynamic techniques. Nagpurkar *et al.* present a flexible scheme to reduce network-based profiling overhead based on repetitive phase information gathered from re-

mote programs [NKS05]. Their *phase tracker* is implemented using the `SimpleScalar` hardware simulator [BA97]. As described in Chapter 5, our implementation is done on real world hardware and addresses the problem for general Java programs.

6.3 Methodology and Evaluation Metrics

6.3.1 Profiling Control Mechanism

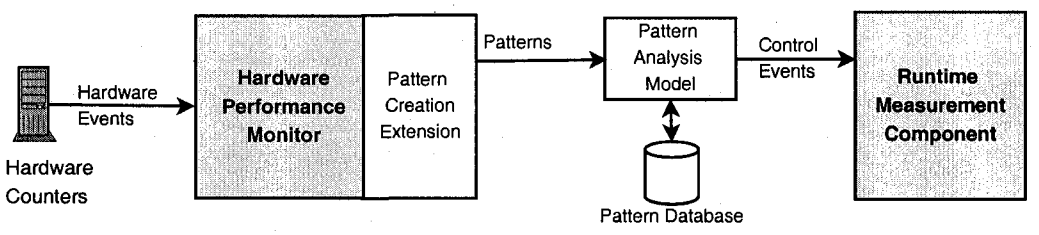

Figure 6.1: *Use recurrent phase detection to control profiling. This figure is the same as Figure 5.5 except that we replace the rightmost block "Other Adaptive Component" with a concrete adaptive component addressed here, the "Runtime Measurement Component" of Jikes RVM.*

As shown in Figure 6.1, we use the repetitive phase detection and prediction results to control the normal runtime profiling mechanism of Jikes RVM. The profiling result is used to guide adaptive optimizations. When there is no recurrent pattern, the runtime measurement component takes profiles as usual. When a recurrent pattern is detected, we compare it with the previous prediction. If it changes the prediction result, we still keep collecting profiles, but also save the profiles into an extra, variable-length local buffer. If the predicted pattern is the same as the last prediction we stop the profiling and instead "replay" the samples in the local buffer. Real program behaviour can of course drift from predicted behaviour over time, and so to ensure profiling accuracy, we have a count-down, rechecking scheme to re-enable the profiling periodically irrespective of prediction.

Different from a normal communication between organizers and the controller, here we build a shortcut control channel between pattern analysis model and runtime measurement component due to two considerations:

- The interface to the runtime measurement component must be as simple as possible in order to keep the perturbation to runtime measurement at a low level.
- Hardware performance may vary quickly. We want the decision made on current hardware event data to be applied as soon as possible before it is out-of-date.

Here we simplify the control mechanism to a simple *set/unset* action on a profiling flag. The runtime measurement component only works when the flag is set as TRUE. The relation between pattern analysis and profiling actions is summarized in Table 6.3.1.

Pattern Analysis Result	**Profiling Flag**	**Action**
No pattern	True	Profiling
New pattern *P*	True	Profiling
Recurrent pattern *P* Prediction changed	True	Profiling Remember local results
Recurrent pattern *P* Prediction not changed	False	Stop profiling Reuse local results

Table 6.1: *The relation between phase detection/prediction, profiling flag and actions of the runtime measurement component in the phase driven adaptive profiling.*

6.3.2 Profiling Metrics

Our application is an improvement to the runtime profiling component in Jikes RVM used to support its adaptive compiler [AHR02]. This profiler samples execution periodically in order to identify "hot methods" and make (re)compilation decisions; sampling rates heuristically trade off accuracy for profiling cost. We provide two metrics for evaluating the impact of phase prediction on profiling:

- **Profiling Rate (Pr):**

  Profiling rate Pr is defined as:

$$Pr = \frac{Number\ of\ Actual\ Profiling\ Points}{Number\ of\ All\ Possible\ Profiling\ Points} * 100\%$$

An unmodified version of the runtime profiling mechanism has a Pr of 100%. Based on phase predictions, we disable some profiling points; a lower value of Pr indicates a reduction in the profiling workload.

- **Coverage Score (Cov):**

  The Jikes RVM profiler makes use of the *relative* number of probe results in each method. Our predicted results should thus produce the same intended effect.

  A method profiling result R on methods $M_i, i \in \{1,\ldots,m\}$ can be represented as:

  $$R = \{< M_i, Per_i^R >\}$$

  where Per_i^N is the percentage ratio of samples in method M_i to the total number of program samples. Given a canonical sample result $N = \{< M_i, Per_i^N >\}$. The *Cov* of R is calculated as:

  $$Cov(R) = \sum_{i=1}^{m} Min(Per_i^R, Per_i^N)$$

  To compare the accuracy of phase based profiling to the original profiling results, we obtain a canonical N by averaging multiple standard executions of the original profiling mechanism. In practice N is reasonably stable. The *Cov* for a phase based profiling run compared with the average *Cov* of each of our standard runs provides an *accuracy score* that indicates how much a given phase based profile varies from typical runs.

6.4 Experimental Results

The profiling workload reduction and accuracy results are shown in Table 6.2. On average we reduce the profiling workload by about a half, although results vary significantly by benchmark. Profiling accuracy, however, is uniformly very high; on average we achieve a 94.3% accuracy, profiling at 51% of possible profiling points. For comparison purposes we show the accuracy score for a simple profiling reduction strategy, denoted as "Simple 50%", that just omits every other probe, also a factor of 2 workload reduction. On benchmarks with small hot method sets, such as COMPRESS and DB, profiling results are not sensitive

Benchmark	Profiling Rate	Accuracy Score (%)	
		Phase Driven	Simple 50%
compress	52.2	91.72	91.71
db	37.5	85.61	89.54
jack	46.0	95.55	68.56
javac	54.8	99.32	76.87
jess	47.3	91.92	79.12
mpegaudio	49.7	92.47	83.76
mtrt	77.7	97.15	83.00
raytrace	97.9	99.97	83.82
soot	27.2	93.83	61.43
PseudoJbb	30.0	94.71	64.84
Average	51.02	94.31	78.26

Table 6.2: *Phase driven profiling workload reduction and accuracy.*

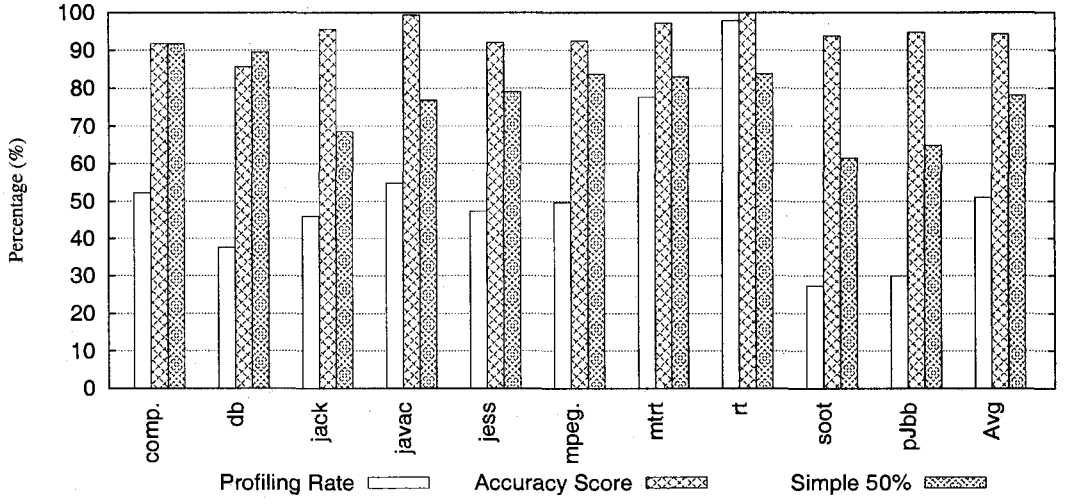

Figure 6.2: *Profiling workload reduction and accuracy results.*

to profiling rate. On more complicated benchmarks, such as JACK, SOOT and PSEUDOJBB, our technique is significantly more accurate, usually with less than a 50% profiling rate.

These results are also illustrated in Figure 6.2, in which "comp." stands for COMPRESS, "mpeg." stands for MPEGAUDIO and "rt" stands for RAYTRACE. The obvious difference between the "Profiling Rate" bar and the "Accuracy Score" bar for each benchmark demonstrates the effect of our profiling workload reduction mechanism. Of course, we understand that the relation between the profiling rate and the accuracy of profiling result is not linear. Usually, a N% reduction in profiling workload will not led to a N% in the accuracy. Just as shown in the case for the straightforward "Simple 50%" solution, the accuracy results is much better than the profiling rate which is 50% for all benchmarks. However, there is a big gap between the accuracy bars for our solution and the "Simple 50%". Given the fact that our solution takes similar or even fewer profiles than that of the simple solution, a on average 16% better in the accuracy indicates that we select a more representative subset to take profiles than the simple solution. This confirms that our phase detector discovers the repetitive period in program execution well. We notice that on MTRT and RAYTRACE, our solution cannot reduce the profiling rate as greatly as the others. The small number of patterns for them shown in Table 5.2 on page 84 shows the reason. These benchmarks only have a relatively small instruction working set. Thus there are only very slight changes in the instruction cache performance, and our hardware pattern constructor cannot generate enough patterns to feed latter processing and analysis.

6.5 Summary

In this chapter, we presented an optimized, phase-driven runtime profiling mechanism which uses the phase detection and prediction technique described in Chapter 5. Our profiling mechanism achieves a significant reduction in profiling workload over the original sampling mechanism in Jikes RVM and still ensures high accuracy. As a sample application, the profiling results confirm that our phase detection and prediction based on hardware information is able to provide useful information to locate the most important, repetitive behaviours in Java programs.

Chapter 7
Phase Based Adaptive Recompilation

Adaptive recompilation is crucial to high efficient JVMs. Better recompilation strategies can bring large benefits to the final performance. In this chapter, we discuss how hardware phase detection results can be used to improve adaptive recompilation decisions. We provide both an offline limitation study and an online implementation. The motivation and a list of contributions are given in Section 7.1. In Section 7.2, we discuss related work on hot method set identification and adaptive optimizations in virtual execution environments, *e.g.*, JVM. Our offline and online implementation details are described in Section 7.3. Performance results and analytical measurements are reported in Section 7.4, and Section 7.5 provides detailed data analysis and discussion. Finally, we conclude and provide directions for future work in Section 7.6.

7.1 Motivation

Many of today's Java Virtual Machines (JVMs) [LY99] employ *dynamic recompilation* techniques as a means of improving performance in Java programs. At runtime, the dynamic Just-in-Time (JIT) compiler locates a "hot set" of important code regions and applies different optimizations, balancing the overhead costs of optimized (re)compilation with expected gains in runtime performance. In this chapter, we introduce a mechanism to select better (re)compilation points and optimization levels. Again, this adaptive recompilation mechanism is based on our hardware performance monitoring and recurrent pattern

construction results.

Building a high-performance, adaptive recompilation strategy in a JVM requires making resource-constrained choices as to which methods to optimize, what set or level of optimization to apply, and when the optimized compilation should be done. Heuristically, the earlier the method is compiled to its "optimal" optimization level, the better. Naively assuming that optimal means more optimizations, the potential for such improvements is illustrated schematically in Figure 7.1. In each image the x-axis is samples (normalized time), and the y-axis is optimization level. More time at higher optimization heuristically means better performance, and so the area under each curve roughly represents how well a method is optimized. The upper left image shows a typical method history, compiled initially at a low level, and progressively recompiled to higher optimization levels. Better prediction of future behaviour allows a method to move more quickly between these steps (upper right), or to skip intermediate steps (lower left). The lower right image demonstrates the case of making an initial "ideal" choice, skipping all intermediate recompilation. Note that even in the latter case at least one sample is required to identify the hot method. The area under the curve (rectangle) summarizes the "amount" of optimized method execution. On the bottom right a method is compiled to its highest optimization level immediately; this roughly represents an upper limit for the potential performance gains, at least assuming simple models of method execution and optimization impact.

One of the key factors involved in finding ideal recompilation choices for a given method is method *lifetime*. Method lifetime is an estimate of how much future execution will be spent in a given method based on current and past behaviour; techniques for estimating method lifetime are critical in making online recompilation decisions. A straightforward solution used in the Jikes RVM [AAC+99, AFG+00, AAB+05] adaptive recompilation component is to assume that the relative proportion of total execution time that will be spent in a given method is the same as its existing proportion: the ratio of future lifetime to past lifetime for every method is assumed to be 1.0. This is a generally effective heuristic, but as an extremely simple predictor of future method execution time it is not necessarily the best general choice for all programs or at all points in a program's execution.

Our work aims at investigating and improving the prediction of future method execution

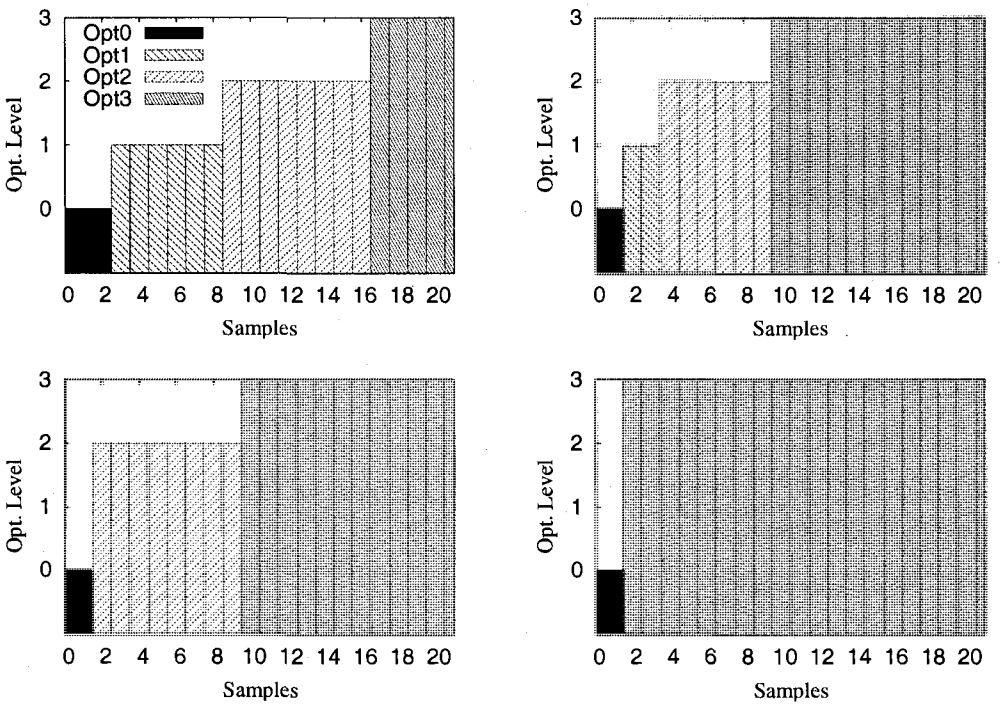

Figure 7.1: *Sources of optimization due to improved recompilation decisions for a given method.*

times in order to improve adaptive optimization decisions.

To achieve better predictions we divide Java program execution into coarse phases; different phases imply different recompilation strategies, and by detecting or predicting phase changes we can appropriately alter recompilation behaviour. We perform an *offline* analysis of the practical "head space" (maximum potential improvement) available to such an optimization that depends on a *post mortem* analysis of program traces, allowing the method recompilation system to perform as in the bottom right of Figure 7.1. We also develop an *online* analysis that is more practical and dynamically gathers and analyzes phase information. To keep our online system lightweight, we base our phase analysis on hardware counter information, recovering high level phase data from low level event data. Based on our implementations in Jikes RVM, we observe an average of 8.5% and up to 21% speed improvement in our benchmark suite using the offline approach, and an average of 4.5% and up to 18% speedup in our benchmarks using our online system, including all

runtime overhead.

Although these results demonstrate significant potential, changes to the dynamic recompilation system introduce feedback in the sense that different compilation times and choices perturb future recompilation decisions. There are also many potential parameters of our design, and different kinds of benchmarks can respond quite differently to adaptive recompilation—programs with small, core method execution sets and long execution times can be well-optimized without an adaptive recompilation strategy, while programs with larger working sets and more variable behaviour should perform better with adaptive recompilation. We consider a number of confounding factors and include a detailed investigation of the source and extent of improvement in our benchmarks, including potential variability due to choice of recompilation algorithm. Our results show that our phase based optimization provides greater benefits in terms of performance, stability, and consistency than current designs or simpler optimizations.

Contributions of this work include:

- We give the results of an offline study of the head space for optimization in the selection of hot-method recompilation points based on our phase information. In the case of repeated or allowed "warm up" executions our study represents an effective optimization by itself.

- We present a new dynamic, phase based hot-method recompilation strategy. Our implementation incorporates online data gathering and phase analysis to dynamically and adaptively improve recompilation choices and thus overall program performance.

- We provide non-trivial experimental data, comparative results, and detailed analysis to show that our design achieves a significant and general improvement. Potential variations, identification of influences, and consideration of the precise source of improvements and degradations are important for optimizations in complex runtime services of modern virtual machines.

7.2 Related Work

Virtual machines provide several advantages over traditional, statically compiled binaries, including platform independent representation, some safety guarantees, automatic memory management and dynamic program composition and optimization. However, in many cases, these dynamic features introduce new challenges for achieving high runtime performance. In response to this situation, many adaptive techniques have been investigated to improve performance by monitoring a program's behaviour. The most important characteristic of the runtime techniques used in modern virtual machines is to do optimization selectively.

Modern interpreter-based JVMs have employed a variety of techniques to improve performance over the naïve *switch-based* implementations. Using *threaded code* [Bel73] is probably the most important improvement for Java interpreters. Note that the word "thread" here has nothing to do with the *thread* in concurrent programming. With a basic direct-threaded technique, the interpreter jumps with indirect branches from the implementation of one bytecode to the next, eliminating the central dispatch. Recent work has improved on basic threading by using runtime translation. For example, Gagnon and Hendren implements an inline-threaded interpretation [GH03] scheme in SableVM [GH01].

In a compiler based JVM (JIT), bytecode is compiled into native code immediately before it executes, which is much faster than interpretation. However, the JIT strategy introduces compilation overhead before any code can execute. This can impose a heavy burden if complex optimization actions are employed during the compilation course. Therefore, compiling and optimizing all the code of a program can easily introduce far too much overhead, both in time and resources. JIT compilers thus typically attempt to identify a smaller hot set on which to concentrate optimization efforts. This kind of adaptive optimization allows sophisticated optimizations to be applied selectively, and has been widely explored in the community [KS03, PVC01, AFG+00]. Most of this work focuses on methods as a basic compilation unit, but other choices are possible; For instance, Hansen's AF [Han74] recompiled basic blocks and single-entry regions with loops selectively. Whaley presents an approach to determining important intra-method code regions from dynamic profile data [Wha01]. On the other hand, Chambers and Ungar [CU91] apply optimizations

across method boundaries via inlining.

Modern virtual execution environments often have a compiler with more than one optimization level. In general, code compiled at a higher optimization level provides faster speed as a trade-off for heavier compilation overhead. In a system with multiple optimization levels, only recompiling the most important (hot) code to a higher level is a common sense, *i.e.*, making selective optimization. In a system such as in SELF-93 [HU96], all methods are firstly compiled into a non-optimizing level and the the optimizing compiler is invoked only for frequently executed methods. SELF-93 uses method invocation counts to figure out hot method, the counts decaying over time. Detlef and Agesen [DA99] use a fast JIT compiler and a slow "traditional" compiler adaptively. They found that a combination of the fast JIT and judicious use of the slow JIT on the longest running methods shows the best results on their benchmark suite. The Sun's HotSpot Server JVM [PVC01] resembles the technique used in SELF-93. Similarly, the IBM Mixed-mode interpreter system [SYK+01] also relies on invocation counts to determinate recompilation decisions. In addition to a counter-based selective optimization heuristic, the Intel's ORP JVM [CEG+05] also use a count-down scheme to identify hot methods.

All these counter-based policies rely on a myriad of heuristic tuning values. Recently, more theoretically involved policies have received more and more attention. Kistler *et al.* [KF03] consider a sophisticated online decision for driving compilation in the Oberon virtual machine. Each compiler phase estimates its own speedup based on a rich set of profile data. Jikes RVM [AFG+00] uses call stack sampling to support a model-driven optimization policy, relying on a cost-benefit model. Krintz [Kri03] provides a dynamic compilation system based on Jikes RVM. Offline profiling results for the top hottest methods are annotated and works as a suggestion for a compilation task to the adaptive engine. Our offline mechanism follows a similar style, but stores all recompilation history from multiple runs and makes a summary trace from the traces of these multiple executions.

In all these efforts, recompilation overhead is reduced by avoiding compiling and optimizing rarely used code, based on either the assumption that "future = past," or by using simple counter-based schemes to determine relative execution frequency. Our work here augments these approaches by concentrating on the specific problem of providing additional predictive information to the adaptive engine of a JVM in order to improve optimiza-

tion decisions, rather than providing the concrete adaptive optimization framework itself.

7.3 Methodology

We have introduced our hardware event based phase detection and prediction model in chapter 5. Employing phase information, the adaptive recompilation engine of Jikes RVM can potentially improve performance by executing highly optimized code more often and decreasing the overhead of successive recompilations. We investigate the improvement from two perspectives. The first is an offline technique based on trace data; this mainly serves to give a sense of the maximal benefit that could be achieved given optimal information. The second is a purely online implementation, that uses our low level profiling and online phase detection systems to improve predictions of future life for methods.

7.3.1 Adaptive Recompilation System in Jikes RVM

Before describing both of the offline and online recompilation mechanisms, we first make the current adaptive recompilation strategy used in Jikes RVM clear to our readers.

The adaptive recompilation system [AFG+00] of Jikes RVM involves three main subsystems. A *runtime measurement component* is responsible for gathering method samples. An *analytic model* reads this data and makes the decisions on whether to recompile a method and the appropriate optimization level. These recompilation decisions are fed to the *recompilation subsystem* which carries out the actual recompilation.

The crucial point is the decision-making strategy of the analytic model. This selects between different optimization levels, based on an estimate of the potential benefit of each level. For each optimization level i $(0 \leq i \leq N)$, Jikes RVM gives an estimate of the execution speed Sp_i of a method m. The value of N can be different for different platforms; in our system, $N = 3$. A recompilation decision is then made based on the following computations:

- T_p: The time of the program already spent in m. It is computed as

$$T_p = SampleNumber * TPS$$

TPS stands for "time per sample," a constant value in Jikes RVM.

- T_i: The expected time of m at level i, if it is not recompiled. In the original implementation, the system assumes:

$$T_i = T_p \tag{7.1}$$

- C_j: The cost of recompiling method m at level j, for $i \leq j \leq N$.

- T_j: The expected time the program will spend in m in the future, if it is recompiled at level j:

$$T_j = T_i * \frac{Sp_i}{Sp_j}$$

The analytic model chooses the level j that minimizes the value of $C_j + T_j$, the compile time overhead and expected future time in m. If $C_j + T_j < T_i$, then m will be recompiled to level j.

7.3.2 Offline Trace-Driven Mechanism

Recompiling a hot method to an ideal optimization level at the earliest point will in general maximize the benefit of executing optimized code, as well as eliminate further potential compilation overhead from the method. For a recompilation mechanism based on runtime sampling data, knowledge of the final optimization level of a method at the time when the first sample of it is taken represents ideal results with minimal profiling overhead. Optimality is bounded by the accuracy of the estimation, including heuristic choices that balance optimization costs and benefits. Here we implement an offline trace-driven optimization technique to discover the approximate improvement head space if optimal choices are made in the sense of attempting to maximize the heuristic benefit.

Implementation of the offline mechanism (*Offline*) is straightforward. A set of traces from training runs is gathered, analyzed, averaged, and used in a subsequent replays of the program to select an appropriate optimization level for each recompiled method. Use of multiple runs accommodates minor variations in performance; sources of noise in recompilation data is discussed more fully in Section 7.5.

Implementation details include that:

- First, training data is gathered; a Java program is executed N times to produce trace files $T_i (1 \leq i \leq N)$.
- Each trace T_i is composed of a set of pairs $< M, L_i >$. M is a particular method, and L_i is the last and highest optimization level of M in T_i.
- A summary trace T_s is constructed, composed of pairs $< M, L_s >$. For a given M, $L_s = Max(L_1, L_2, ..., L_N)$.
- In the tested runs, T_s is loaded at the beginning of execution. Each time a method sample M is taken, if we can find a record $< M, L_s >$ for it in T_s, we recompile M to level L_s directly, and mark the recompilation as a final decision. No further compilation will be applied to M.
- It is possible that speed gains due to better adaptive recompilation allows a method not recompiled in any training run to be added to the hot set in an actual run. If we cannot find a record for M in T_s, M is treated per Jikes RVM's original recompilation strategy. Note that in our experiments such cases are rare and involve infrequently executed methods; the impact of this divergence in hot set identification is reasonably expected to be small.

Performance results from the offline strategy are given in Section 7.4.1. On some benchmarks the benefit obtained is quite significant, confirming both the potential available to a more flexible online optimization, and the value of our offline design as an optimization unto itself.

7.3.3 Online Mechanism

The success of an online recompilation system depends on the accuracy of method *life-times*, or the future time spent in a method, as well as other recompilation cost and benefit estimates. Underestimating future method execution time results in missed optimization opportunities, while overestimating runs the risk of being overly aggressive in compilation, wasting time on unnecessary recompilations and/or high optimization levels. This is particularly true early and late in program executions, where code execution variability is

high and the expectation of continued behaviour is lower. This can also occur when programs make major phase changes, shifting into markedly different modes of execution. The kernel of our online mechanism is thus a system that detects coarse grained and variable length program phases and uses this information to control the relative aggressiveness of the recompilation subsystem in Jikes RVM. The resulting improved recompilation choices improve overall program performance.

The existence of basic startup, core execution, and shutdown phases are well known. Our phase identification is based on identifying *age*, but further allows programs to *rejuvenate*, as a means of allowing for the identification of multiple major execution phases. These phases imply distinct patterns of control for recompilation, and are classified as follows:

- **Newborn:** At startup a Java program tends to spend time on a set of methods that perform initialization actions, and these are often not executed after basic setup is complete. When considering whether past behaviour is a good predictor of future behaviour we can heuristically expect that the future execution time of a given method will be less than the past: $Future < Past$.

- **Young:** After a period of time, the program comes into the main application or kernel code and will spend a comparatively long time on the same set of methods. Methods executed at this stage are likely to be executed even more in the future: $Future > Past$.

- **Mature:** After the program works within its kernel code for a while, we consider the program to be *mature*. In this case, we assume the runtime profiling subsystem has gathered enough samples to support the recompilation engine in determining suitable optimization levels. Here the original estimate that future and past performance will be similar is most valid: $Future \approx Past$.

- **Rejuvenated:** Experience with coarse grained phase analysis of Java programs shows some programs will have distinct, kernel-based phases, and at runtime will have more than one hot method set. When a program enters a new hot set it thus transitions to the young phase again. Once so *rejuvenated* as such, however, we have a slightly more cautious estimate as to the future behaviour of the new hot set: $Future > Past$.

Phase	Hardware Event Behaviour	Recompilation
Newborn	No recurrence of patterns	Less aggressive
Young	Recurrence of patterns	More aggressive
Mature	Less new patterns More old patterns	Moderately aggressive
Rejuvenated	More new patterns Invalidation of old patterns	More aggressive

Table 7.1: *Program phase, hardware patterns, and recompilation aggressiveness.*

The second column of Table 7.1 describes how program phases are heuristically determined from the underlying hardware event data. Changes in how low level patterns are identified in the data suggest corresponding changes in the program code, and thus phase or age. At program startup, a wide variety of "execute-once" startup code is executed, and few recurring low level patterns are found. A young program will start to show significant recurrences of new patterns as it begins to execute its kernel code. The mature phase is detected by noticing the balance tipping from discovery of new patterns to recurrence of old patterns, and the rejuvenated phase by a subsequent loss of old patterns and introduction of new ones.

Understanding program phase allows for heuristic control of the relative aggressiveness of the recompilation engine. In cases where the future performance is not equal to the past the expected execution time should be appropriately scaled. The third column in Table 7.1 gives a summary of how age affects the behaviour of the recompilation engine. A newborn program is less likely to repeat its behaviour, and recompilation should be more conservative. A young program enters into its kernel; the new code is likely to be executed much more than it has been in the past, and recompilation becomes aggressive. As the execution enters a mature phase aggressiveness is decreased; in such a relatively stable environment the recompilation engine is expected to have sufficient past data for making good decisions. A program that enters a new significant kernel of execution requires again ramping up the aggressiveness of recompilation.

The *aggressiveness* of the adaptive recompilation engine is controlled by using a scaling

parameter in the estimation of future execution times. We introduce a variable *futureEstimator* and change the definition of T_i in Formula 7.1 to:

$$T_i = T_p * futureEstimator \tag{7.2}$$

Figure 7.2 shows a high level overview of the complete online algorithm. Each hardware pattern *PAT* has a field *occNum* which remembers the number of occurrences. If the adaptive recompilation model finds a recurring *PAT*, such that, *PAT.occNum* is more than one, the estimated "age" of a program (*Prog.age*) is increased. When *Prog.age* is larger than a threshold *youngThresh*, the program has left the newborn phase and become young. From then on, each time there is a *fresh* pattern *PAT* such that the occurrence number is less than a threshold *matureThresh*, the value of *futureEstimator* is increased; otherwise its value is decreased. A larger value of *futureEstimator* drives the adaptive recompilation model to make more aggressive recompilation decisions, assuming methods will run for longer than currently estimated. Fixed upper and lower bounds protect the *futureEstimator* value from diverging in cases of extended bursts of fresh or mature patterns. Based on initial experiments we limit *futureEstimator* to the range $[0.9, 5.0]$.

7.4 Experimental Results

The experimental platform and benchmark suite are the same as we introduced in Section 5.4.1. For performance evaluation we measured our benchmarks quantitatively using a baseline (original), and using our offline and online strategies. Overall execution time for the online approach includes all overhead for phase analysis and low level profiling. In the case of the offline approach the overall execution time includes the overhead of processing the recompilation trace. Full results for our benchmarks in absolute and relative terms are shown in Table 7.2. The "Original" column represents the data collected from the version of Jikes RVM where we began our work. This "Original" version already includes the whole adaptive engine.

To gain greater insight into the source of improvement, and inspired by our intuition as to potential performance gains in introductory Figure 7.1, we also developed more abstract,

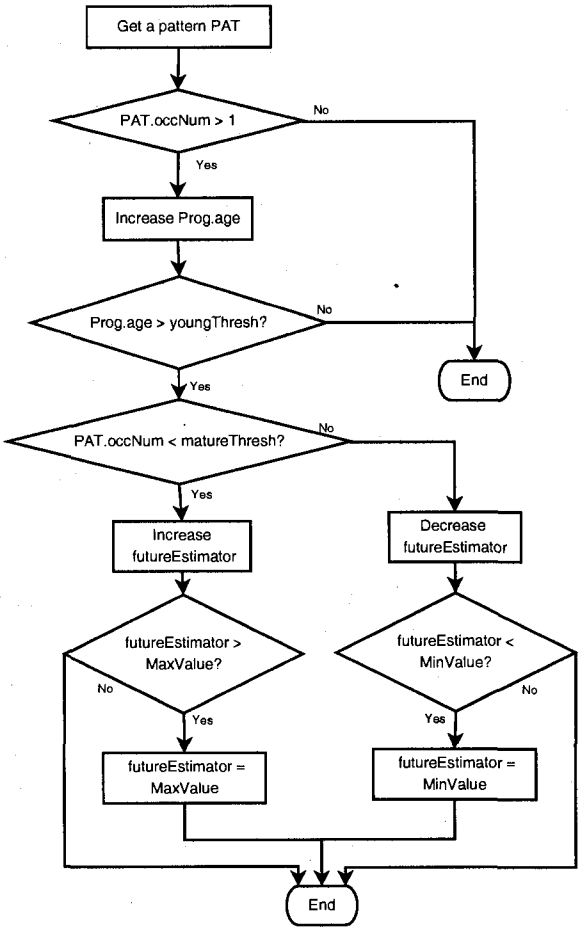

Figure 7.2: *An overview of the algorithm used in the computation of the* futureEstimator.

analytical measures that summarize the *amount* of optimized code executed. Our abstract measures of optimization quality are shown in Figure 7.3 and Figure 7.4.

To measure the relative proportion of code executed at different optimization levels we developed a *method-level speed* (MLS) metric that can be applied to individual methods in individual program executions. MLS is computed as the sum of the time, measured in samples, spent at different optimization levels, weighted by the proportion of time at each optimization level. Each partial sum for an optimization level in this calculation is scaled by an estimate of optimization quality, namely the *speed* of the code under the given optimization level; Jikes RVM provides fixed estimates for these latter values. Figure 7.3

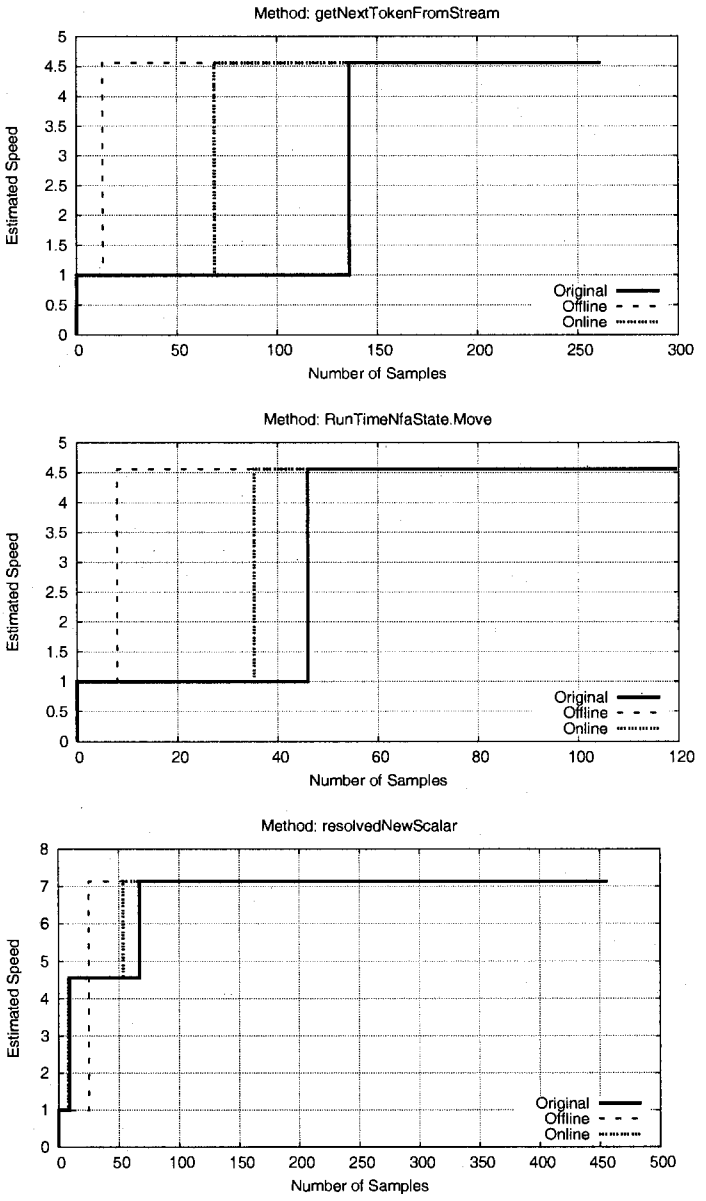

Figure 7.3: *Dynamic* Method Level Speed *measurements over time for each of our baseline, offline and online recompilation approaches. Each graph is a distinct method from* JACK.

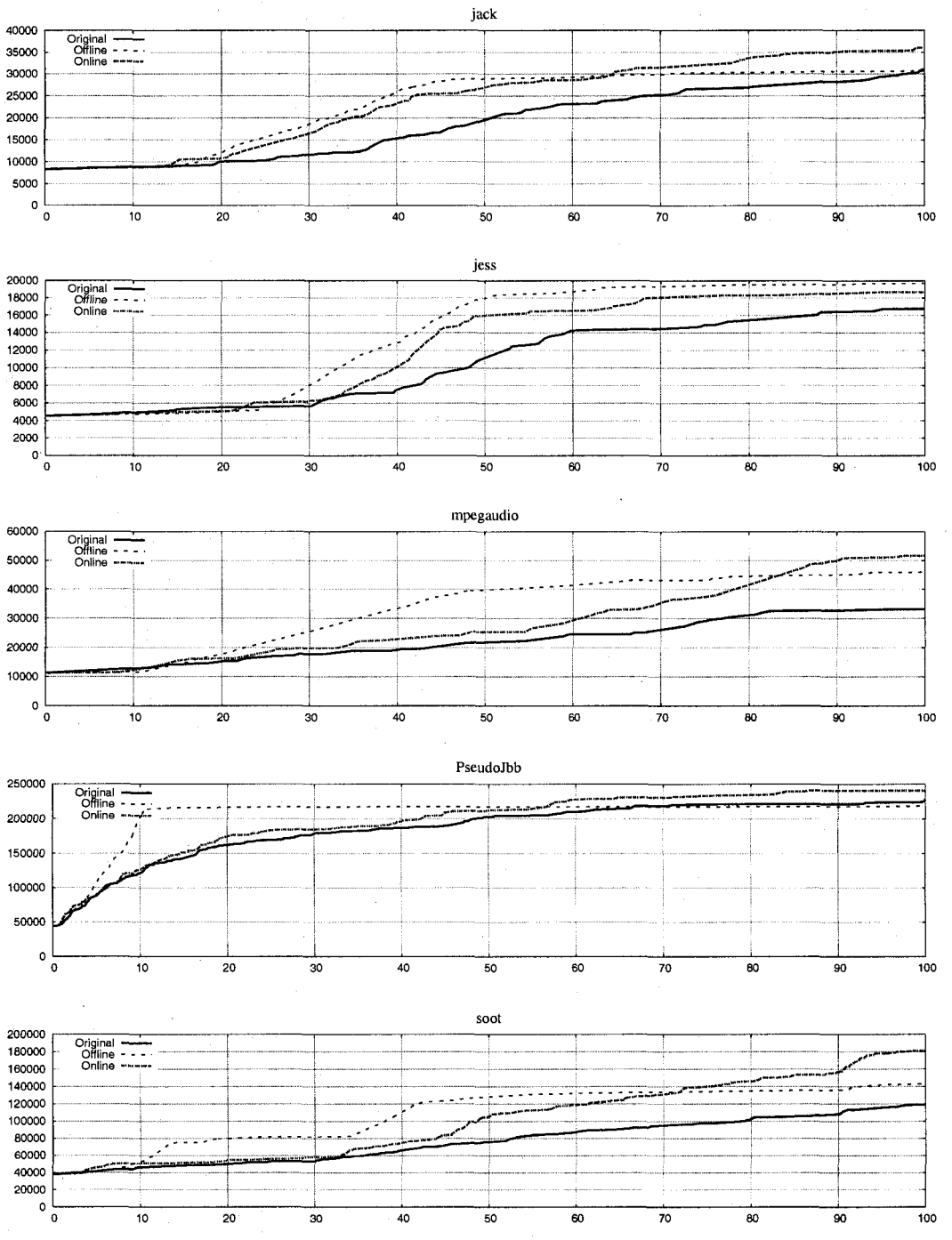

Figure 7.4: *Weighted optimized methods:* JACK, JESS, MPEGAUDIO, PSEUDOJBB *and* SOOT.

Benchmark	Original	Offline		Online		Benchmark Characteristics	
	Time (s)	Time (s)	Impr. (%)	Time(s)	Impr. (%)	Patterns	Optimized methods
compress	15.75	15.55	1.3	15.73	0.1	157.9	17.6
db	37.97	37.22	2.0	37.72	0.6	450.5	25.3
jack	22.59	20.08	11.2	19.78	12.5	343.5	90.0
javac	11.78	10.72	9.4	11.10	5.7	193.9	36.9
jess	18.11	14.25	21.3	14.87	17.9	204.5	50.0
mpegaudio	20.24	17.81	12.1	19.79	2.3	103.6	58.9
mtrt	15.14	14.29	6.4	15.42	-1.8	58.8	36.4
raytrace	14.35	13.30	7.3	14.21	0.8	63.9	35.3
soot	303.12	278.45	8.1	291.28	3.9	2542.3	408.2
PseudoJbb	753.95	705.90	6.4	735.62	2.5	7832.8	331.8
Average	-	-	8.5	-	4.5	-	-

Table 7.2: *Execution results, number of patterns created in the online version, and number of methods optimized for our benchmark suite. Values are the arithmetic average of the middle 11 out of 15 runs. "Impr." stands for the improvement over the original version.*

shows the results for a measurement of MLS for the three methods with the largest MLS values in JACK, ordered from top to bottom. The x-axis in these graphs is time, measured in samples, while the y-axis is the estimated speed for different optimization levels in Jikes RVM. An upward step in the graph corresponds to a recompilation at a higher optimization level. The size of the area under each curve gives an estimate of how MLS changes under different recompilation strategies—greater area means greater use of optimized code, and hence heuristically improved performance.

In Figure 7.4 we show a summary of the same basic property, but summarized over the entire execution and all methods. To simplify calculations, method contributions are weighted here not by actual number of runtime samples, but by static method size. Note that we are aware that a dynamic version of executed code size is potentially more accurate, but we have found that the static method size is sufficient to discover the main trend of the execution. Runtime code to measure dynamic execution sizes also brings extra runtime overhead, which if not carefully optimized may pollute the experimental results. This therefore provides a more approximate picture of behaviour, akin to a static versus dynamic analysis, but also demonstrates the effect is robust in the face of different and less

precise forms of evaluation. In these figures the x-axis is normalized execution time, and the y-axis is "weighted optimized methods", a sum of weighted method size of all sampled methods, where each weighted sum is again scaled by the appropriate optimization *speed* factor provided by Jikes RVM. The interpretation of these graphs is similar to that used for Figure 7.3; a higher curve means there are more methods optimized to a higher level and the execution speed should be faster, with the area underneath approximating relative amount and quality of optimized code executed.

7.4.1 Offline

The results of our offline mechanism in absolute terms as well as relative improvement over the original version are given in the third and fourth columns of Table 7.2. The offline version does achieve significant improvements on some benchmarks. On JESS, it improves execution time by 21.3%. On JACK, JAVAC and MPEGAUDIO, the improvements are also quite large. On average, the offline version saves 8.5% of the execution time, although the effect is not uniform; for some benchmarks, such as COMPRESS and DB, there is little to no improvement at all. We will discuss these benchmark-specific behaviours in more detail in Section 7.5.

In the weighted optimized methods graphs, the curves for our offline implementation are shown as dashed lines. Corresponding with the faster execution speeds, these curves are also the highest ones in these graphs. Interestingly, in most of the benchmarks, there is only one major upwards trend. In the graph for SOOT, however, there are two such increasing phases. This shows the existence of programs with multiple major phases that can require large and relatively abrupt changes in identified hot method sets.

7.4.2 Online

The execution time results for the online mechanism are shown in the fifth and sixth columns of Table 7.2. For benchmarks where the offline version shows a large improvement, the online version also performs well. We obtain up to nearly 18% improvement for JESS, quite close to the 21% improvement found for JESS offline. On average the online version achieves a 4.5% improvement, about 53% of the possible performance improvement

demonstrated in the offline version. For the 4 benchmarks that responded most positively to the offline version, the improvement online is on average 9.6%, or 71% of the offline result.

In the weighted optimized methods graphs, the curves for the online version are shown as dotted lines, and typically lie between the curves for the offline and original implementations. In the graph for SOOT (the bottom graph in Figure 7.4), the online curve reflects the multiple phases that are more clearly seen in the offline curve; our online system correctly identifies the rejuvenated phase, as we discuss in more detail in Section 7.5.1.

Further details on performance can be seen in the behaviour of specific methods, as shown for JACK in Figure 7.3. As with the weighted optimized method results, the offline version has the greatest area and provides higher optimization earlier, with the online implementation lying between the offline and original versions. Note the bottom graph shows the offline implementation optimizing the method later than both the original and online versions. This is a result of resource management in the recompilation system, prioritizing requests for relatively fast lower levels of optimization over more expensive requests for longer, highly optimized compilations.

7.4.3 Variance and Overhead

Figure 7.5 shows 99% confidence intervals for our original, offline, and online data measurements. Our evaluation is experimentally quite stable and deterministic, with confidence ranges for the three variations generally showing good separation. Note that the intervals for JACK are among the largest and have clear overlap; the $\approx 1\%$ performance gain for JACK online as opposed to offline could be attributed to data variance and/or the intrinsic imprecision of simple optimization benefit/cost estimates. We discuss accuracy and noise concerns in depth in the following section.

Overhead in profiling systems is always a major design concern. In our case we make use of hardware counters that are sampled at every process context switch; at a few tens of machine cycles per read and only on the order of thousands of context switches over a program's lifetime this technique is extremely cheap. Pattern construction and phase analysis provide the bulk of our actual overhead, and to measure total overhead costs we

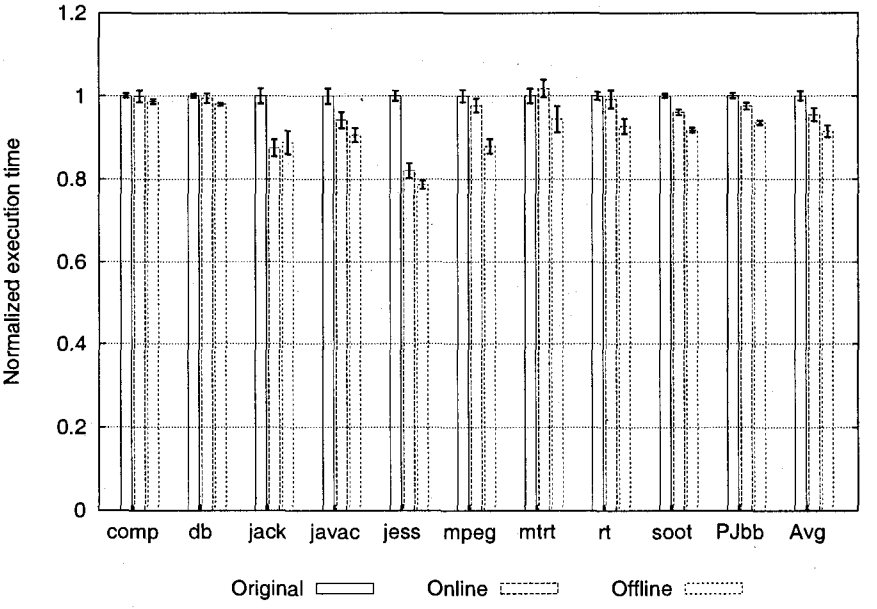

Figure 7.5: *Normalized execution time of* SPECJVM98, SOOT *and* PSEUDOJBB *with 99% confidence interval errorbars for each of our three test scenarios: original, online and offline.*

compared the original, baseline Jikes RVM with an implementation of our online technique that computes phases as normal but does not actually change the adaptive recompilation settings (*futureEstimator*). Figure 7.6 shows the computed relative overhead. Overhead comes from sources such as hardware monitoring, pattern construction, phase prediction, and building control events for the recompilation component. On average there is a 1.33% slowdown across these benchmarks due to our data gathering and phase analysis system. There is always room for improvement, but this relatively small cost is in most cases greatly exceeded by the benefit, and demonstrates the practical low overhead of our technique; again, speedup and other experimental data includes all overhead.

7.5 Discussion

Initial recompilation choices affect later recompilation choices, and there are many potential parameters and choices in our, or any, recompilation design. A good understanding of

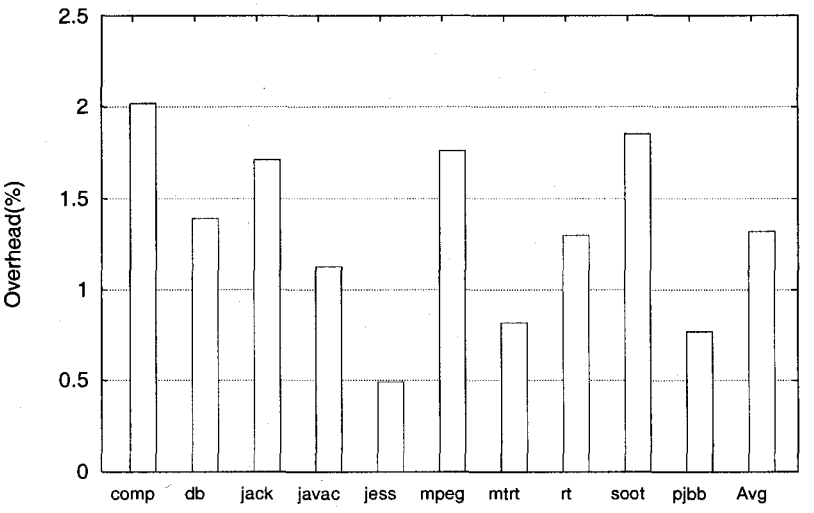

Figure 7.6: *Relative overhead in the online system compared with the original.*

potential variation and relative performance gain is therefore important to making good, general selections of recompilation strategies.

We have chosen algorithmic parameters to include resource requirements and performed extensive initial experimentation and numerical validation of the parameter space to justify our main approach; this numerical evaluation is described in [GV07]. Here we discuss various factors that can influence our performance, and present data validating the general stability and effectiveness of our design. We first consider different benchmark characteristics that are important in our approach. This is followed by a detailed comparison of our design with other simple optimizations to the recompilation system, again showing the practicality of our work and the generally good quality of the result.

7.5.1 Benchmark Characteristics

Benchmarks in our study demonstrate a wide range of responses to our optimization. Several benchmark-specific factors can be seen to influence whether and where performance will be realized using our techniques. Benchmark length, the stability of the hot set, as well

as more general sensitivity of the program to our profiling and optimization systems can all affect the relative success.

Benchmark Execution Time

In our benchmark suite, the SPECJVM98 benchmarks finish in a comparatively short time while SOOT and PSEUDOJBB execute for an order of magnitude or so longer, and also recompile many more methods than other benchmarks, as seen in the last column of Table 7.2. Longer running programs have an advantage in that recompilation has more data to work with as there are more sample points. Furthermore, any reduction in speed due to less optimal recompilation choices can be amortized over a longer period and often a larger hot set. For shorter programs our mechanism helps the VM locate the hot set more quickly; the reduction in overhead obtained by promoting methods more quickly to their final optimization level is also a greater benefit. This factor can be seen in the results for the longer and shorter running programs. SOOT and PSEUDOJBB show an average improvement of 7.3% and 3.2% using offline and online analyzes respectively, while the other, shorter benchmarks improve on average of 8.9% and 4.8%.

Hot Set Stability

We observe that many programs contain a single hot set of methods that is more-or-less stable over the course of execution. Some benchmarks, however, do have large, distinct execution phases, and show a clear hot set variation. SOOT in our benchmarks demonstrates this quite clearly; in Figure 7.4, the SOOT curve of the offline version obviously has multiple stages. Each large incline corresponds to a major change in the hot set.

Using our offline implementation with perfect knowledge of the future, we can detect the hot set variation or *rejuvenated* phase correctly and quickly, resulting in relatively steep slopes upward as the new hot set is optimized. The original implementation, on the other hand, has no apparent sensitivity to this change in program behaviour and shows a gradually increasing curve with no obvious bursts of optimization. Our online implementation achieves an intermediate level between these two. It has a moderate sensitivity to the hot set variation and goes through a couple of smaller steps at approximately the same points

in time, rising more quickly to the level of the offline analysis.

An unfortunate side effect of our optimization for detecting rejuvenation, or variations in the hot set is a certain overzealousness of optimization toward the end of execution. The online curves of JACK, MPEGAUDIO and SOOT in Figure 7.4 tend to rise above even that of the offline curve by the end of execution, indicating that optimized recompilation may be being overused, recompiling and optimizing methods that will only be used in the final fraction of program execution. We experimented with identifying a termination phase, but termination tends to look like any other phase change (rejuvenation) with our current pattern analysis and data. Solutions based on incorporating extra, high level information such as knowledge of termination-specific methods may be more profitable. In practice, these sub-optimal online decisions at termination time do not have an overly large impact, and so we leave reducing this "tail" problem to future work.

Appropriateness of Data Source

It is interesting that low level events can expose high level behaviour, even for complex, object-oriented programs with non-trivial control flow. We have successfully used the I-cache miss rate as a base event, but this does impact not only what can be measured but also how it can be measured, and of course other choices and event combinations are possible.

Although a good choice in general, for some benchmarks I-cache miss rate provides somewhat reduced information. RAYTRACE and MPEGAUDIO, for instance, have a relatively small instruction working set. Thus we observe only slight changes in I-cache performance, and as can be seen from the 2nd-last column in Table 7.2 our pattern creator finds significantly fewer patterns in these cases. This provides less information to the recompilation engine, and thus recompilation choices are not much better than in the original version: RAYTRACE and MPEGAUDIO show marginally positive improvements, while MTRT shows a 2% reduction. The fact that performance even in this situation is close to the original and not significantly degraded is evidence of the low overhead of our implementation design in general, and sample-based hardware monitoring specifically.

Other benchmarks have instruction working sets large enough to produce significant misses as different code paths are exercised, allowing our online solution to identify pat-

terns easily. The performance difference resulting from the improved information is evident in benchmarks such as JACK, JESS, and JAVAC. Some benchmarks, however, exhibit cache performance changes, but the actual hot method set remains quite small. If a small set of methods are called frequently, as for COMPRESS and DB, the original adaptive recompilation engine has the chance to gather enough samples to recompile a method relatively quickly. In these cases, the potential improvement available by reducing the delay of recompilation is small. The marginal benefit achieved by our offline solution can be mainly attributed to reductions in optimization overhead due to skipping redundant intermediate recompilations for some methods.

Programs can also exhibit *bias* with respect to different hardware events. We previously showed, for instance, that some programs like JESS and JACK are highly "instruction cache sensitive", meaning that from a processor-level point of view the instruction cache performance has a large impact on the execution time of the program [GVG06]. On the other hand, DB and especially COMPRESS are highly data cache biased. There is limited room to improve performance from the code side if data usage has a dominating impact. In these cases even the offline version only obtains a small improvement. We expect that programs with large memory requirements and hence garbage collection overhead, heavy I/O, and so forth will also respond less well to our design, as in general programs that are dominated by other costs than code execution speed will receive reduced benefits from adaptive code optimization techniques.

The above discussion suggests that monitoring different or multiple hardware events may be a route to further optimization. We have explored a few hybrid forms of pattern-building based on combinations of I-cache miss rate, D-cache miss rate, branch instruction counts, and brand prediction miss rates. So far, these designs have not shown useful improvement above that of one based on a simple I-cache miss rate; further exploring this space is, however, potentially fruitful future work.

7.5.2 Stability and Comparison with Simple Approaches

Understanding which benchmarks can work well is important, but differentiating them online may be non-trivial, and a good recompilation system should perform reasonably well

over a range of benchmarks. For our adaptive system to be useful, it is also important to know that the adaptivity is effective. Both our online and offline strategies generally increase the aggressiveness of recompilation choices, and we must consider that similar effects could be achieved by simply making the the Jikes RVM estimator more aggressive without adaptation.

Testing the effects of trivial, constant increases in recompilation aggressiveness provides a baseline that shows both the variability of performance of different recompilation strategies and in comparison with our online approach, the actual impact of adapting to program phases. We evaluate several versions of Jikes RVM with no hardware monitoring or phase analysis, but incorporating our scaled time estimate formula in Formula 7.2 with *futureEstimator* set to different fixed, constant factors to increase recompilation aggressiveness. Table 7.3 shows the normalized overall execution time for our benchmarks when the future time estimate of methods is increased by values between 1.5× and 3.0×; this represents the range of average increase in aggressiveness used by our online system for benchmarks in our suite (Table 7.3, last row).

futureEstimator	**compress**	**db**	**jack**	**javac**	**jess**	**mpegaudio**	**mtrt**	**raytrace**	**soot**	**PseudoJbb**
1.5×	0.997	0.991	0.987	0.970	0.924	0.960	1.017	0.983	0.966	0.991
2.0×	0.970	1.008	1.041	0.955	0.879	0.924	1.039	1.010	0.950	0.978
2.5×	1.018	1.022	1.063	0.975	0.856	0.925	1.127	1.057	0.945	0.969
3.0×	1.018	1.025	1.080	0.991	0.852	0.948	1.151	1.053	0.969	0.975
online	0.999	0.993	0.876	0.942	0.821	0.978	1.018	0.990	0.961	0.976
online average	3.06	1.98	2.16	2.40	2.34	2.44	2.22	1.99	1.35	1.09

Table 7.3: *Fixed setting of* futureEstimator *versus the online version. The "online average" row shows the average* futureEstimator *value used in the online version, weighted proportionally over program execution.*

The data in Table 7.3 shows that there is certainly no one fixed setting that is optimal for all benchmarks; benchmarks respond differently, and simply increasing aggressiveness overall is not a generally effective strategy. This is more apparent graphically, as seen in Figure 7.7. Some benchmarks have a large variance in performance as *futureEstimator* changes, and some are relatively unaffected. For all benchmarks except MPEGAUDIO and

COMPRESS, our online version is optimal or within variance of optimal. In comparison with simple approaches, our online design provides stable and good results overall, significantly more so than the base version or any of the constant aggressiveness settings.

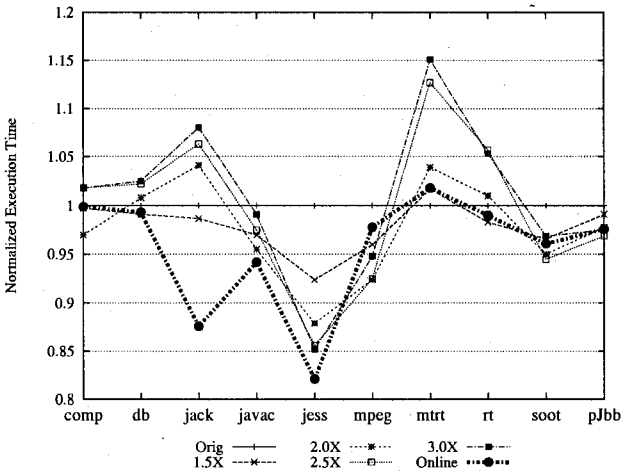

Figure 7.7: *Normalized execution time for benchmarks using different recompilation optimization strategies.*

Recompilation Algorithm Sensitivity

We can separate benchmarks into those that exhibit a low sensitivity to recompilation decisions (less than $\approx 5\%$ variance between approaches), and those that show relatively high variance due to such choices. The former are shown in Figure 7.8 and the latter in Figure 7.9.

The less sensitive benchmarks in Figure 7.8 correspond reasonably well with our discussion of benchmark-specific behaviours that impair the effectiveness of our technique. SOOT and PSEUDOJBB are long-running with large hot sets, while COMPRESS and DB contain hot sets that are easily identified under all scenarios. JAVAC is a marginal inclusion; like RAYTRACE it has a small working set, but falls within the threshold of insensitive benchmarks in our simple binary division.

More sensitive benchmarks where recompilation decisions can have a relatively large

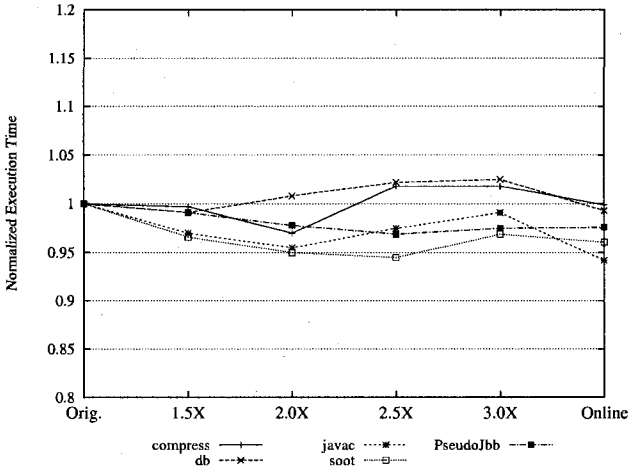

Figure 7.8: *Normalized execution time for benchmarks using different recompilation optimization strategies. These benchmarks are insensitive to strategy.*

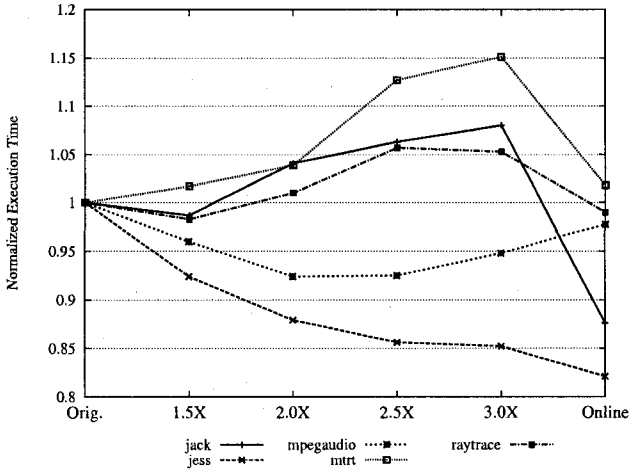

Figure 7.9: *Normalized execution time for benchmarks using different recompilation optimization strategies. These benchmarks are quite sensitive to strategy.*

performance impact are shown separately in Figure 7.9. Adaptivity accommodates benchmarks where greater aggressiveness usually improves performance such as JESS, and benchmarks where greater aggressiveness decreases performance, such as JACK and MTRT. A more detailed view of typical benchmark behaviour found in our experimental data is shown in Figure 7.10, with the upper row showing the normalized performance of benchmarks that improve or degrade performance as an almost linear function of recompilation aggressiveness. More aggressive recompilation is in general good for benchmarks like JESS (upper left), bad for others like MTRT (upper right), while some such as SOOT and MPEGAUDIO have an intermediate sweet spot in terms of overall recompilation aggressiveness. In the first three cases the online system adapts well; for MPEGAUDIO the online performance is improved over the baseline but does not achieve optimal performance. For benchmarks such as SOOT and MPEGAUDIO, however, a "sweet spot" exists in terms of overall aggressiveness, in both cases here around 2.0–2.5. Adaptation is not as successful overall for MPEGAUDIO while for SOOT adaptation finds a good performance level, albeit in a context where the total performance variation is small. Universally good performance under these conditions is hard to achieve; however, the online system, generally does quite well in adapting to different benchmark conditions and is clearly an overall better choice than either the current or other fixed aggressiveness schemes.

7.6 Summary

For many programs, sub-optimal choices in recompilation can result in reduced performance. We have shown how improvements to recompilation strategy can result in better performance, and provided a design using coarse grained, variable length phase prediction to adaptively improve recompilation choices. Using offline trace data for prediction provides an experimental high performance watermark for such a technique, and functions as a useful optimization when program executions are repeated exactly. Our fully online implementation makes choices based on dynamically acquired data, and exhibits both low overhead and good overall performance.

Multiple factors influence performance in a recompilation system, and to show meaningful improvement a close evaluation of performance under different scenarios and with

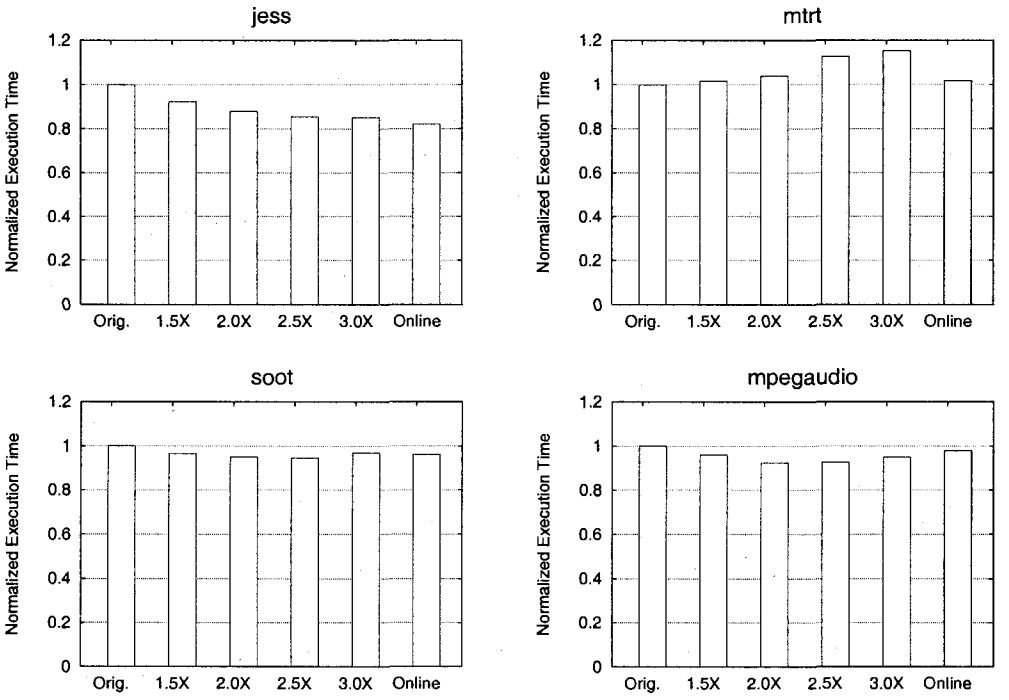

Figure 7.10: *Typical behaviour of benchmarks in response to different recompilation strategies.*

different levels of detail is important. We have explored our optimization in terms of execution time, and further validated our results with analytical measurements. Detailed examination of benchmark behaviour reveals that benchmarks respond in different ways to the relative aggressiveness of a recompilation engine, and we considered a wide variety of benchmark-specific factors, including high level considerations such as overall runtime and low level influences such as the density of hardware event data. Under these highly variable and "noisy" conditions our adaptive online system achieves a significantly improved performance.

There exist a number of possible extensions to this work. The success of our approach, like most adaptive online systems, depends on the extent of variability in runtime execution data. We have expended a great deal of effort to understand and experimentally validate potentially critical factors, ensuring our approach is a generally robust optimization. Further understanding and detection of benchmark characteristics may improve our design, and

could also be used to help select benchmark-specific responses by the adaptive optimization system. *Profile repositories,* aggregating profile data from multiple executions may be a useful way of moving online performance closer to that of offline performance [AWR05]. Mixing profile data from multiple runs or using offline/online hybrid data might also help with the "tail problem" of predicting the termination phase of a program.

We intentionally exploit coarse grained phase information to allow complex optimizations time to act and improve performance. Startup phases are well-known, but the use of high level and variable length phase information, when cheaply gathered, is also obviously of value. Predicting major phase changes may be useful for scheduling garbage collection, heap data reorganization or any other design for larger scale adaptive execution. Additional or different hardware event data may be useful for more "data-centric" applications, and part of our current investigations include the use of multiple and hybrid hardware event sources.

Chapter 8
Garbage Collection Point Selection

Programmers are increasingly turning to object-oriented languages with automatic memory management (*garbage collection*). Java provides a garbage-collected heap which improves productivity of programmers by reducing errors that result from explicit memory deallocation. The implementation, optimization, and performance analysis of garbage collection (GC) algorithms have been a hot topic for a while. A variety of factors can impact the overall garbage collection performance, including the *collection point selection* (CPS) addressed here. In this chapter, we present our exploration in improving GC performance by selecting garbage collection point.

In Section 8.1, we show our motivation on studying the CPS problem, the reason why it matters, and the fundamental idea of our solution. We discuss the algorithm, optimization and performance factors of garbage collection in Section 8.2. The details of our design are introduced in Section 8.3 followed by experimental results and a deep discussion in Section 8.4. Finally, we summarize our study on CPS, both achievements and limitations, and then provide a list of potential improvements as future work in Section 8.5.

8.1 Motivation

The implementation of garbage collection is JVM-specific. Most JVMs employ a *tracing* garbage collector. When the heap runs out of space, a tracing collector begins its work gathering objects that are directly reachable from root set. The root set consists of global

and stack local references (or pointers) and other references in JVM internal data structures. The collector then traverses all the references in the currently identified reachable objects to find other reachable objects. This tasks is done recursively. Finally, all the memory occupied by non-reachable objects, *i.e.*, garbage, is collected and the space is made available for the future objects. One interesting observation of tracing GC is that the GC work will be accomplished faster when there are fewer reachable objects and it will claim more free space. In other words, the collector produces more product (the free space) when it does less work. When the workload is heavy, the collector produces less product. This "less work, more achievement" phenomenon contradicts usual intuition. Picking a suitable point to do collection can thus potentially improve the performance significantly, if collection points correspond to small reachable sets.

A good garbage collection point is a moment at which the program just leaves an old phase and reaches a new phase in its execution. In such a moment, a large amount of objects may reach the end of their life range and turn into garbage. If a collection happens at this moment, it can be accomplished with a light workload and release a large amount of memory. On the contrary, when a collections happen at an inappropriate point, it may endure a very heavy workload, and only release a small amount of memory. An impressive example is the JAVAC benchmark in SPECJVM98 suite. As we know, JAVAC invokes four passes during a Java program compilation. After each pass, it forces the JVM to make a collection. Apparently, at the forced GC points lying between two passes of compilation, a large set of objects just become garbage. Hence, these forced GC points are the optimal GC points for this program. On the other hand, if a collection happens shortly before these forced GC points, it is very likely to be less efficient.

Table 8.1 shows the GC statistics on two settings for JAVAC. Here we use a GenMS collector pre-existing in Jikes RVM. In the "Inappropriate Setting", there is a normal GC right before each forced GC. In the "Appropriate Setting", we slightly increase the size of nursery space and remove all the normal collections before the forced collections except the first one.

The total number of pages released across the running are similar in both cases. However, the appropriate setting accomplishes the same task with fewer collections. All collections, except the first one, finish in a much shorter time. The overall *Throughput*, *i.e.*,

GC #	Inappropriate Setting			Appropriate Setting		
	GC Type	Time(ms)	Released Pages	GC Type	Time(ms)	Released Pages
1	Normal	2208.24	25152	Normal	1975.25	26496
2	Forced	75.38	1372	Forced	255.46	4880
3	Normal	1414.05	21184	Forced	40.01	21896
4	Forced	196.49	1620	Forced	34.57	16344
5	Normal	1269.71	15468	Forced	48.18	17752
6	Forced	425.15	4128	—	—	—
7	Normal	938.81	11216	—	—	—
8	Forced	663.59	3564	—	—	—
Sum	—	7191.42	83704	—	2353.47	87368
Thr.	11.63 pages/ms			37.12 pages/ms		

Table 8.1: *The impact of selecting optimal GC points, using* JAVAC *as an example.* Thr. *stands for* Throughput.

the measures of released pages per millisecond, of the appropriate setting is 3.2× of that of the inappropriate setting. If we do not count the first normal GC which is unavoidable and performs similarly for both cases, the difference of throughput is an astonishingly 13.4×!

Table 8.1 demonstrates the large impact of the GC points selection. Making a collection at an appropriate point can possibly reduce collection time greatly. In order to choose these appropriate GC points, here we make a hypothesis that:

Major program behaviour transition points are also appropriate GC points; a large hardware performance variation reflects a major program variation.

Our hardware phase detector detects large performance variations by generating hardware patterns at a high variation level (details of hardware patterns are described in Section 5.2). By this hypothesis, we translate the the appropriate CPS problem into postponing a collection until the next occurrence of a high level hardware pattern, which is much more implementation-friendly.

Is it possible to postpone a collection when there is a collection request? There is a positive answer for copying style collectors, by reducing the *copy reserve*. The copy reserve

is a space in the heap held by the collector for copying reachable object from the "from-space" or nursery space. In practice, the survival rate of objects is usually very low. This is especially true for objects in the nursery space maintained by a generational collector. In Jikes RVM using a GenMS collector, we observe the survival ratio is frequently below 20%. As a result, most of the space allocated as a copy reserve for the nursery is wasted. Therefore, we reduce the copy reserve to delay the collections until the next appropriate point. Our solution can thus obtain benefit from two directions, selecting a potentially more productive collection point and reducing the copy reserve overhead. In most time, it is safe to do so. However, a garbage collection algorithm must account for the worst case. In our situation, the worst case is when the copy reserve overflows because of too aggressive reduction. We solve this problem by launching a full heap GC. However, full heap GC is very expensive, and thus should be avoided as much as possible. We thus use several mechanisms to shield from extreme situations.

- We maintain an upper bound on the copy reserve reduction rate based on the survival ratio of the last nursery GC.
- We start a collection after receiving a hardware pattern for major program variation. It usually happens before the copy reserve reduction reaches the upper bound, and thus keeps the system safe.
- Based on the history of received patterns, we make a prediction of future pattern occurrence time and associate it with the memory allocation amount. We only postpone a collection when the predictor indicates that it is valuable to postpone GC until the upper bound. This prediction can further reduce the probability of the worse case.

However, selecting GC points is something very tricky. A large number of factors or noise can influence the overall performance of garbage collection. Our solution does obtain an improvement over the original collector used when we began this work, but it is already not a perfect solution for this problem. We investigate and discuss the current results and possible actions that can be taken for further improvement. Our work is a worthwhile initial study of this interesting problem, selecting garbage collection points.

8.2 Related Work

A detailed and rather complete introduction to GC algorithms and GC related problems can be found in Jones and Lin's [JL96] GC book. The three classical methods of garbage collection are *reference counting, mark-sweep* and *copying*. Reference counting is a direct method, based on counting the number of references to each memory cell from others. The strength of the reference counting method is that memory management overheads are distributed throughout the computation. The major drawback of primitive reference counting is the inability to reclaim cyclic structures. Both mark-sweep and copying algorithms are tracing collections, which can handle cyclic data naturally. Usually, a mark-sweep GC offers a better performance but tends to fragment memory, scattering cells across the heap and reducing data locality. Copying GC involves moving a large number of reachable objects and therefore has more overhead, however it compacts reachable objects together and eliminates the fragmentation problem. Furthermore, copying GC usually uses a bump-pointer allocator. Hence, the cost of allocation is low.

GC has been a target of optimization for decades. Many improvements have been made to these classic algorithms in a variety of directions, concerning different factors that affect GC performance. Ungar's *generational scavenging* [Ung84] technique and more recent works on *Age-based* GC [SMM99] *Older-first* GC [SHB+02] and *Beltway GC* [BJMM02], for instance, all aim to improve performance by adjusting collection time according to object lifetimes.

Other approaches for improving GC are available. Reachable objects can be aggregated into regions in the heap based on a selection of object attributes. This either aims to improve data locality in the program [HBM+04, GM04], or to reduce the memory access overhead of the collector [QH02]. Optimizations on data prefetching and lazy sweeping [CHV04, Boe00] aim to improve data cache performance. Gagnon *et al.* use a bi-directional layout in SableVM [GH01]. By grouping all the reference fields together, the copying GC algorithm can be greatly simplified. Further improved reference scanning strategies based bi-directional layout are described in [GVG05b, GVG06]. Both the presented RS and the TBP reference tracing strategies can largely reduce the tracing workload.

Some other works specifically study GC performance. S. Blackburn *et al.* [BCM04b]

discuss performance *myths* of canonical GC algorithms on widely used Java benchmarks. They compare the performance of classic GC and memory allocation algorithms in different configurations and environments. The impact of special implementation factors, such as "write barriers" and the size of nursery space of generational collectors, on mutator and GC performance are carefully studied. The impact of the heap size on garbage collection is further studied in [SKB04]. A set of garbage collection algorithms in Jikes RVM are investigated. GenMS is usually the one performing the best. However, SemiSpace performs well in some special heap setting.

Usually, the garbage collection points are only determined by the heap size and program memory requirement. A collection is triggered when the heap is full. However, adaptively adjusting the garbage collection points is possible. Chen *et al.* [kCBC+06] present a *proactive* garbage collection. Collection is triggered before the moment it should be in order to reorganize the heap and improve data locality. Our garbage collection solution also changes the collection points. However, we delay the collection point for a better collection point. Delaying the collection point is practical for any GC algorithm with copy reserve space. The copy reserve can be reduced to support the delay. Reduced copy reserve is used in Sun's Hotspot JVM [Suna]. However, Hotspot uses a fixed size nursery, rather than an Appel-style variable-sized nursery addressed by our solution. Variable-sized nursery reduces the space wasted in nursery with a low survival rate [App89]. McGachey *et al.* presents [MH06] an improvement on the GenCopy GC of Jikes RVM by reducing the copy reserve which makes use of the same concern of our work. Their work set the rate of reserve to a fixed value before execution. They investigate the impact of a large set of different fixed settings for the benchmarks. Our solution is based on GenMS GC of Jikes RVM, which is a best choice in general, and we adjust the reserve rate dynamically, according to the runtime behaviour of programs.

8.3 Design

In Chapter 5, we described our extension to the HPM component of Jikes RVM. With this extension, we can generate patterns to represent performance variations in hardware and use these hardware patterns to discover important program behaviours. Still using

the hardware patterns, here we aim to obtain appropriate GC points. As mentioned in Section 8.1, we assume that large variations in hardware performance reflect behaviour transitions in programs. These behaviour change points are the appropriate GC points as well. Launching a GC just after such a moment has a higher potential to achieve a better throughput, *e.g.*, releasing more pages in shorter collection time. The crucial point here is to defer a collection until the next detected major phase transition. This collection delay is possible for any collector with a copy reserve. Usually, the size of the copy reserve is the same as that of the *from-space* (in a semi-space collector) or the nursery space (in a generational copying collector). We can postpone GC by reducing the copy reserve.

Copy reserve reduction is safe most of the time. In the case when the survival ratio is very high, a rescue space is allocated to store the objects until the accomplishment of the current nursery collection. We use the *emergency allocation* mechanism pre-existing in the *heap growth manager* of Jikes RVM to allocate this rescue space. A full heap collection is then launched immediately afterwards. Of course, several heuristics have been used to avoid this situation as much as possible.

Our GC mechanism is rooted in the pre-existing GenMS collector of Jikes RVM. GenMS is a generational copying and mark-sweep hybrid collector. It uses a variable size nursery space and reserves the same amount of memory in the mature space. Each time a nursery collection happens, the surviving objects are promoted into mature space. The size of the nursery is thus shrunk after each nursery collection accordingly. When the nursery runs out of memory, a full heap collection is scheduled.

We implemented a *Garbage Collection (GC) point analyzer* that uses information from the hardware performance detector, memory allocation requirement and the heap. The co-operation of GC point analyzer, HPM and memory allocator is shown in Figure 8.1. The layout of the heap is shown in the bottom of the figure. From left to right, we have the nursery space, the copy reserve for nursery in mature space, the mature space occupied by surviving objects from former GC(s) and the other special spaces, *e.g.*,immortal space, large object space (LOS), *etc.*. Two thresholds *MinGCThresh* and *MaxGCThresh* label the lower and upper bound of an "GC enabled" area. Within this area, the GC point analyzer decides whether or not to launch a GC based on the state of two internal flags. The value of the internal flags are set depending on the hardware patterns transfered from our ex-

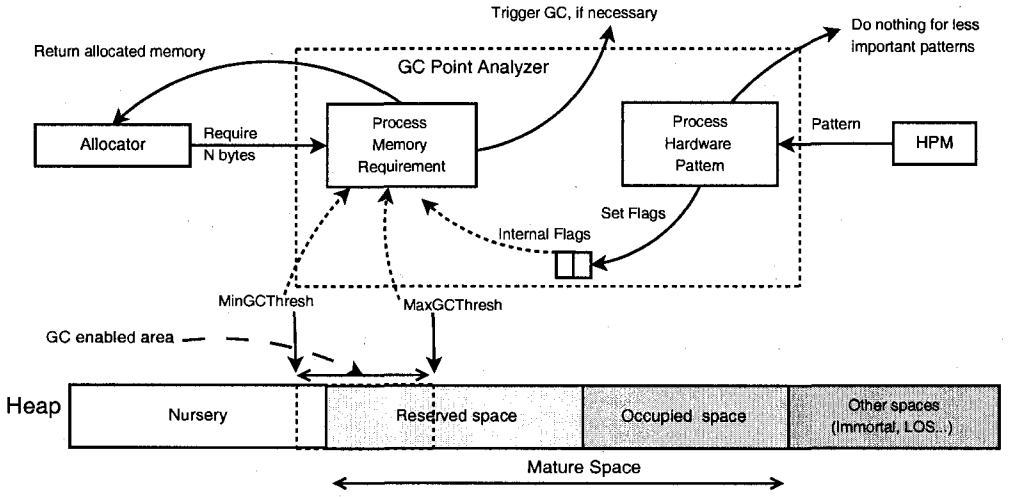

Figure 8.1: *Overview of GC point selection.*

tended version of HPM. Recall that the generated patterns represent hardware performance variations.

The GC point analyzer is actually composed of two parts: the *hardware information processing* and the *memory allocation processing*. As shown in Figure 8.2, the hardware pattern generator feeds the GC point analyzer with hardware patterns. In this special case, the concrete pattern code is not important. We are only concerned with the variation level of the pattern. If the pattern is in a *major level* (see page 84) which means there is a large performance variation, we set a boolean flag *HW_Flag* to true. We use the value of HW_Flag to remember whether we have met a major behaviour change point, *e.g.*, a potentially appropriate GC point. The next step is to make a prediction of the next possible major hardware variation point based on the history of hardware patterns. We associate the amount of memory allocated to the patterns received by the GC point analyzer and reuse the the tri-distance prediction mechanism introduced in Chapter 5 to make a prediction. The prediction result indicates the distance to the next major performance variation point measured by the memory allocation in bytes. We denote this value by *NextHW*. Then, we compare the value of *NextHW* with a threshold *MaxGCThresh*. This comparison tells whether the predicted next behaviour change point is within the current setting of the largest nursery extension. If not, this prediction is invalidated and the process finishes. Otherwise,

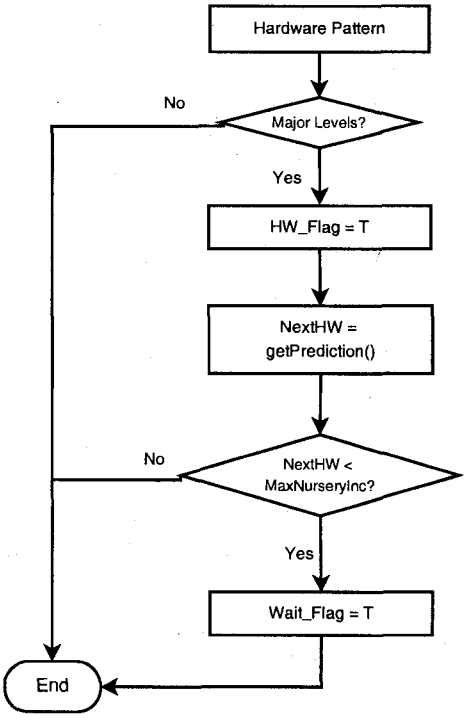

Figure 8.2: *Process the hardware information.*

it is worthwhile to wait for the next major behaviour change point. In the latter case, a boolean flag *Wait_Flag* is set to true to tell the collector to postpone a GC if there is a collection request.

The main flow of how the GC point analyzer responses to a memory allocation request is illustrated in Figure 8.3. After receiving a N bytes memory requirement from the allocator, the GC point analyzer adds N to the current allocated memory in the nursery space. If the sum is lower than a threshold *MinGCThresh*, there is still plenty of memory in nursery space. It is not necessary to make a GC even if the *HW_Flag* is set. In this case, no collection will be scheduled and the flag *HW_Flag* is set to false. If the sum is larger than the prediction value *NextHW* which sets the *Wait_Flag*, this prediction becomes invalided and *Wait_Flag* is set to false. If the sum is larger than the *MaxGCThresh* threshold, we have increased the nursery to a large amount. In order to avoid an out-of-memory problem, the GC point analyzer decides to launch a GC anyway, and sets the boolean flags to false. If

the sum lies between the upper and lower threshold, whether there is a GC is depend on the value of the boolean flags. A collection is scheduled if the *HW_Flag* is true or the both *HW_Flag* and *Wait_Flag* are false, *i.e.*, there is neither a prediction or an actual occurrence of a major performance variation in hardware.

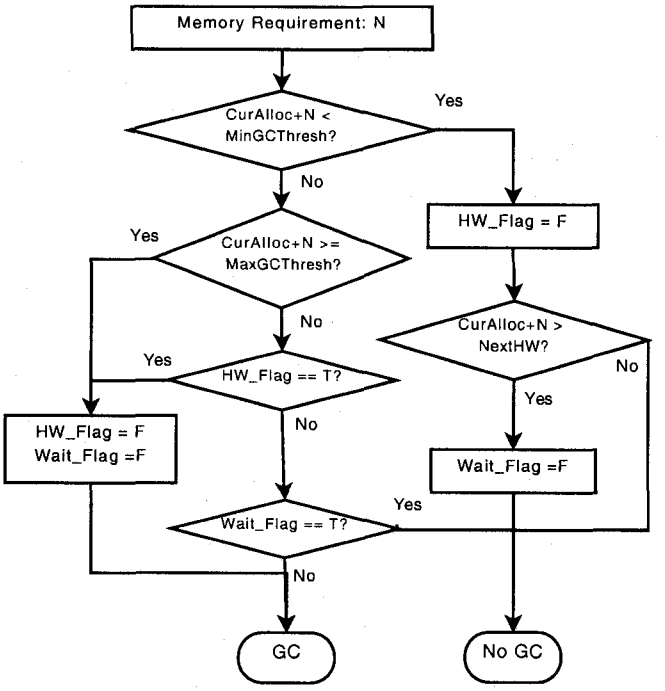

Figure 8.3: *Process the memory allocation requirement.*

The value of the upper and lower bounding thresholds are set as follows:

- Initially, the *MaxGCThresh* is set to 1.33 $\times$ of the nursery size, which is the half of expected free memory reserved for the nursery. The *MinGCThresh* is initialized with the nursery size.

- After each GC, given a nursery survival ratio R and a current expected nursery size S, the value of these two thresholds are reset as follows,

$$MaxGCThresh = S \times \frac{2}{1+R}$$

$$MinGCThresh = S \times \frac{1}{1+R}$$

- When the size of the nursery is extremely small, say less than 4KB, we suspend the copy reserve reduction by setting both *MaxGCThresh* and *MinGCThresh* to the size of the nursery space.

These overflow avoiding mechanisms work well. In practice, no nursery reserve overflow is encountered.

8.4 Experimental Results and Discussion

The basic experimental setting is the same as in Chapter 5. We choose the same benchmark suites and testing platform. The reported results are collected from the median 27 of 33 runs. Other than our automatic garbage collection selection version (CPS) and the original implementation, we also test a set of fixed nursery copy reserve versions. Here we test to fixed increases of 1.2, 1.4, 1.6 and 1.8 times to the default nursery size. We refer them as F1.2, F1.4, F1.6, and F1.8 respectively. For SPECJVM98 benchmarks, we test seven different heap settings from 40MB to 100MB, increasing 10MB in each step. SOOT and PSEUDOJBB apparently require larger heap. PSEUDOJBB fails with an out-of-memory error on a 128MB heap. The same benchmark requires very frequent collections when the heap is set to 192MB, which indicates that even a 192MB heap is still not enough for this program. We thus test seven heap size settings from 256MB to 640MB, increasing 64MB in each step.

We first report the nursery increment results of the CPS version. We record the increase rate of the nursery at each collection time and calculate the geometric mean (Geomean). CPS obtains a significant speedup over the original implementation. We also make the comparison between the CPS version and the fixed nursery increment rate versions and evaluate the CPS solution and potential further improvement for it.

Benchmark	Geomean of Nursery Increment Rate						
	40M	**50M**	**60M**	**70M**	**80M**	**90M**	**100M**
compress	1.55	1.56	1.37	1.45	1.45	1.42	1.46
db	1.25	1.38	1.40	1.39	1.30	1.30	1.22
jack	1.47	1.46	1.52	1.43	1.57	1.55	1.47
javac	1.14	1.10	1.23	1.08	1.12	1.29	1.13
jess	1.48	1.47	1.47	1.48	1.47	1.48	1.44
mpegaudio	1.32	1.04	1.04	1.03	1.00	1.00	1.00
mtrt	1.44	1.41	1.40	1.36	1.35	1.35	1.35
raytrace	1.43	1.38	1.36	1.33	1.30	1.28	1.29
	256M	**320M**	**384M**	**448M**	**512M**	**576M**	**640M**
soot	1.40	1.35	1.31	1.23	1.22	1.44	1.46
PseudoJbb	1.44	1.47	1.55	1.48	1.52	1.55	1.57

Table 8.2: *Geometric mean of nursery increment rate (increased nursery size normalized to default nursery size).*

8.4.1 Nursery Increase

The geometric mean of the nursery space increase results are shown in Table 8.2. The nursery increment rate in different situations varies a lot, from 1.0 to 1.57. Comparatively, the increment rate for one individual benchmark with different heap sizes varies less, but is not stable. JACK, JESS and PSEUDOJBB have a comparatively stable increment rate. The rate of MPEGAUDIO becomes 1.0 when the heap gets larger than 80MB due to the fact there is no collection needed.

8.4.2 Performance Comparison

The speedup of CPS over the original version is demonstrated in Table 8.3. In general, CPS performs much better than the original. COMPRESS is the only benchmark both versions have a similar behaviour. For the other programs, we obtain speedup ranging from 1.05 to 1.31, measured by the geometric mean across all the heap settings used. There are even large speedups at particular heap settings, *e.g.*, the 1.43 speedup obtained on JACK with a

100MB heap and 1.52 speedup obtained on PSEUDOJBB with a 512MB heap. Note that we do not count the extremely large speedup in the 70MB heap data of JAVAC. JAVAC together with MPEGAUDIO are the pair of benchmarks that show special behaviours. The large performance speedup on JAVAC is mainly due to the number of unnecessary collections eliminated before the forced collection, as mentioned in Section 8.1. For MPEGAUDIO, no collection is needed when the heap is larger than 80MB.

Benchmark	**Geomean of Speedup**							
	40M	**50M**	**60M**	**70M**	**80M**	**90M**	**100M**	**Average**
compress	0.99	1.05	1.00	0.95	0.98	1.01	1.00	1.00
db	1.02	1.05	1.07	1.16	1.01	0.91	1.38	1.08
jack	1.21	1.33	1.24	1.31	1.39	1.27	1.43	1.31
javac	0.93	1.03	1.24	2.33	1.01	1.02	1.03	1.16
jess	1.17	1.17	1.17	1.14	1.13	1.10	1.15	1.15
mpegaudio	1.02	0.98	1.06	1.02	—	—	—	—
mtrt	1.06	1.07	1.06	1.09	1.02	1.04	1.07	1.06
raytrace	1.15	1.13	1.11	1.12	1.11	1.09	1.00	1.10
	256M	**320M**	**384M**	**448M**	**512M**	**576M**	**640M**	**Average**
soot	1.03	1.06	1.05	1.02	1.01	1.01	1.13	1.05
PseudoJbb	1.16	1.20	1.20	1.19	1.52	1.18	1.56	1.28

Table 8.3: *The speedup of the CPS version over the original collector.*

The significant performance improvement of CPS comes from two sources: the increase of the nursery size and selecting the collection points after hardware performance variations. We further investigate the impact of these two sources. We build and test a series of fixed nursery increment versions F1.2, F1.4, F1.6 and F1.8 as described above. The performance of all these fixed nursery increment versions together with CPS solution (CPS) and original implementation (Orig(1.0)) are illustrated in Figure 8.4 and Figure 8.5.

Not surprisingly, a version with a larger nursery space increment rate usually performs better. However, this is not necessarily true for all data points. In some specific benchmarks and particular heap settings, a larger increment in the nursery space can end up with a slowdown, *e.g.*, on DB with a 80MB heap. Most benchmarks are sensitive to nursery

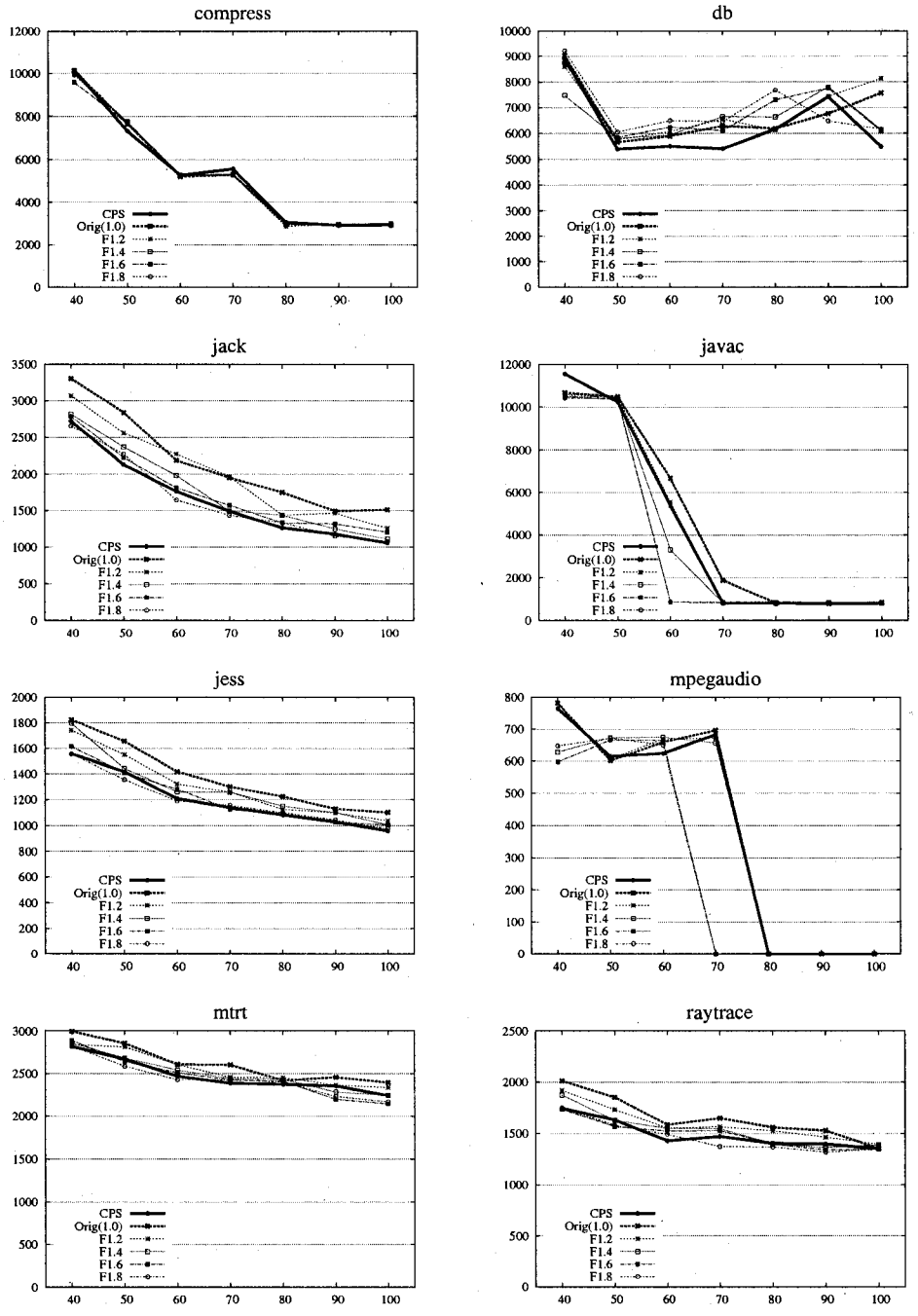

Figure 8.4: *The garbage collection results of* SPECJVM98 *benchmarks. X-axis is the heap size in MB; Y-axis is the collection time in milliseconds.*

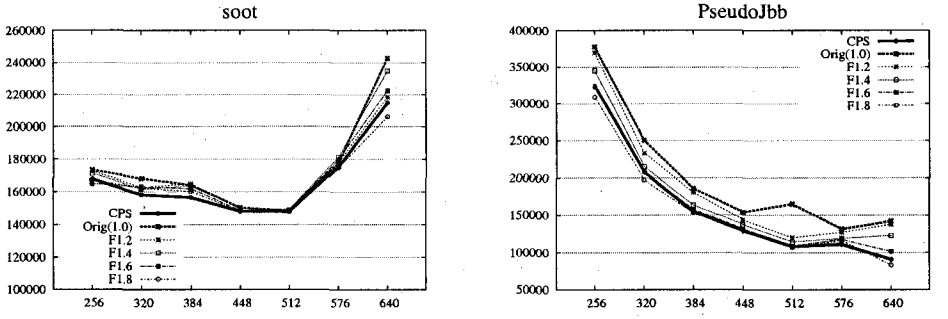

Figure 8.5: *The garbage collection results of* SOOT *and* PSEUDOJBB. *X-axis is the heap size in MB; Y-axis is the collection time in milliseconds.*

changes. However, COMPRESS shows a unique behaviour. The nursery increase does not have observable impact on collection time.

Our CPS implementation is among/close to the best solution in most test points. Especially on DB, CPS performs better than all of the other versions. Comparatively, the performance variation on JAVAC is large until the heap is larger than 80MB. When the heap is smaller than 80MB, the performance of different collection algorithms is mainly determined by how many collections before the four forced collections are eliminated. As soon as the heap is equal to or larger than 80MB, all versions eliminate all the unnecessary collections and perform similarly. MPEGAUDIO requires less memory resource than others and thus requires no collection when the heap reaches 80MB (or larger). F1.8 increases the nursery size more than others and requires no collection even on a 70MB heap setting. For the same reason, F1.8 gives the best overall performance.

Our experimental results show that the nursery increment is a major factor for performance improvement. To investigate the the impact of hardware information, we calculate an expected performance for fixed nursery increment at the same increment rate as that of the CPS solution.

We use the the four fixed increment rates to split the range $[1.0, 2.0]$ into five sections. Within each section, we assume the performance changes linearly. Formally, suppose the nursery increment rate of CPS is X, $X \in [L, H]$, the collection times of fixed solutions at rate L and H are F_L and F_H, then the expected collection time of F_X is calculated as,

$$F_X = F_L + \frac{X-L}{H-L} \times (F_H - F_L)$$

Benchmark	Geomean of Speedup							
	40M	**50M**	**60M**	**70M**	**80M**	**90M**	**100M**	**Average**
compress	0.99	1.05	1.00	0.96	0.97	1.01	1.00	1.00
db	0.94	1.08	1.08	1.23	1.03	1.03	1.44	1.14
jack	1.03	1.09	1.07	1.01	1.07	1.10	1.08	1.07
javac	0.92	1.02	0.97	1.82	1.02	1.03	1.04	1.11
jess	1.11	1.01	1.05	1.06	1.04	1.04	1.05	1.05
mpegaudio	0.90	0.99	1.06	1.01	—	—	—	—
mtrt	1.02	1.01	1.03	1.03	1.03	0.98	1.01	1.01
raytrace	1.06	1.00	1.09	1.06	1.04	1.02	1.03	1.04
	256M	**320M**	**384M**	**448M**	**512M**	**576M**	**640M**	**Average**
soot	1.02	1.03	1.05	1.01	1.01	1.03	1.08	1.03
PseudoJbb	1.05	1.03	1.05	1.05	1.03	1.06	1.15	1.06

Table 8.4: *The speedup of the CPS version over the expected collection time of fixed nursery increasing solution with the same increment rate.*

The comparison between CPS and the expected fixed result with the same rate is shown in Table 8.4. In support of using hardware information, the CPS version performs better. We obtain a speedup of more than 1.05 in half of the benchmarks. The best case is DB where the speedup is 1.14. Moreover, CPS never performs worse than the expected value. Unfortunately, CPS is still slightly weaker than that of F1.8 solution which is usually the best one across all the tested implementations. The speedup of CPS compared with F1.8 is shown in Table 8.5. In all the 63 validating data points, CPS is better in 29 of them and F1.8 wins 32 points. If we only count the points where the difference is larger than 0.05, *i.e.*, 5% performance variation, the rate is 11 versus 12. Therefore, in general, CPS gives a comparable performance with the fixed version with a much larger nursery increment rate.

In summary, the CPS solution performs significantly better than the original version. The nursery increase takes an important role in the performance gain. CPS is slightly

Benchmark	Geomean of Speedup							
	40M	**50M**	**60M**	**70M**	**80M**	**90M**	**100M**	**Average**
compress	0.98	1.05	0.99	0.95	0.98	1.00	0.98	0.99
db	1.04	1.12	1.18	1.19	1.24	0.88	1.12	1.11
jack	0.97	1.06	0.94	0.96	1.06	0.97	1.02	1.00
javac	0.90	1.02	0.16	1.05	0.95	1.06	1.04	0.77
jess	0.99	0.95	0.99	1.02	1.02	1.02	1.02	1.00
mpegaudio	0.85	1.09	1.04	—	—	—	—	—
mtrt	1.02	0.97	0.98	1.02	1.01	0.95	0.97	0.99
raytrace	1.00	0.97	1.04	0.94	0.98	0.94	0.99	0.98
	256M	**320M**	**384M**	**448M**	**512M**	**576M**	**640M**	**Average**
soot	1.00	1.03	1.02	1.00	1.01	1.00	0.96	1.00
PseudoJbb	0.95	0.95	0.99	1.02	0.99	1.05	0.92	0.96

Table 8.5: *The speedup of the CPS version over the collection time of fixed nursery increasing solution with an increment rate 1.8 (F1.8).*

weaker than F1.8 since the latter increases the nursery more aggressively. Another important observation is that CPS also performs better than the estimation of a fixed solution with the same increment rate. This fact supports that choosing collection points according to runtime analysis results, such as in the hardware variation we used here, we are able to obtain benefits for the collector. However, we do not claim the current CPS is a perfect implementation due to the fact that it does not win over a more aggressive fixed setting F1.8. We also notice that there are a number of ways to improve it further and better address the problem of garbage collection point selection. We will discuss these potential improvement as future work in the following section.

8.5 Summary and Future Work

The performance of garbage collection can be impacted by a large number of factors, including the algorithm, program characteristics, the heap size, and garbage collection points.

In this chapter, we introduced an exploration for improving garbage collection perfor-

mance by selecting appropriate garbage collection points. We began with a special example to show the significant potential effect of choosing collection points on the final performance. We then presented a solution to do selective GC according to hardware pattern detection results. The fundamental idea is to postpone collection points later until the next major performance variation point.

We achieved a large improvement over the original implementation on which we began this work. We then made a deep investigation on the possible factors affecting our experimental results. We studied the impacts of increasing of the heap size and hardware information on the collector. Our solution, in general, works better than straightforward solution with the same nursery increment rate. The improvement we obtained is thus not purely from the use of a larger nursery space.

Adjusting collection points is a complex problem. Although, the current CPS is not a perfect solution for this problem, we consider that our work is a valuable exploration of this hard problem. We addressed a challenging problem and obtained a large improvement over the original algorithm. We also investigated several internal issues of this problem. A number of possible improvements can be employed in the future. A simple improvement could be developing better heuristics for determining the lower bound of the GC enabled area.

There are other major potential improvements:

- First, associating hardware variation with software structures, such as methods, or loops, can potentially improve the accuracy of selecting the optimal collection points and reduce the overhead.

- Second, runtime data shape profiling [PV06] traces could be used to figure out potential good collection points. We can apply an offline analysis of the runtime data shape analysis to locate large variation points of data shapes and mark them as suggested collection points.

- Moreover, a novel static analysis could be developed to locate possible optimal GC points by discovering the points at which a large amount of the objects just leave their reachable region.

- A combination of offline static analysis and online hardware variation detection mechanism is another very hopeful direction for addressing the problem discussed here.

Chapter 9
Conclusions and Future Work

Modern hardware architectures are getting increasingly complex. The impact of hardware performance on software execution thus becomes more and more significant. Therefore, hardware performance has become a critical concern of Java Virtual Machine design and implementation. We have presented our exploration on developing virtual machine techniques based on hardware information. In this last chapter, we summarize the entire thesis. We give conclusions in Section 9.1 and discuss several future research directions in Section 9.2.

9.1 Conclusions

Modern virtual machines are complex runtime environments. Any optimization in a modern virtual machine has the potential for complex interactions with various factors, high and low level. We have investigated and provided a coarse taxonomy for the relative factors. Our efforts provide a number of insights into the sources of different influences on program performance. Using our investigation results, we have discovered that the performance of Java Virtual Machines can be significantly affected by hardware related issues, such as unintended code motion side-effects. These hardware related factors make the performance measurement of JVMs become more challenging, but opportunities can co-exist with challenges. The correlation between changes in program behaviour and hardware performance variations suggests that there is a chance to improve JVM performance through hardware

information analysis. In this thesis, we used hardware data to detect recurrent, periodic phases in program execution as an example of the latter.

Program phase detection has been a hot topic for a while. However, most existing phase detection techniques focus on the identification of stable phases. We thus gave a definition to and clarified the importance of recurrent periodic phase for irregular, real life, object oriented programs investigated from the perspective of a coarse granularity. We have presented our approach to online phase detection for general Java programs based on real world hardware information. Most pre-existing phase evaluation metrics are specifically designed for stable phase detection results. We thus have defined a set of novel metrics which are suitable for the recurrent periodic phase detection problem, and demonstrated a practical implementation with potential many applications.

Our hardware information analysis results can bring benefits to many adaptive optimizations in JVMs and other runtime environments. Our selective profiling mechanism reduces the profiling workload significantly over the original sampling mechanism and still ensures high accuracy. Other than a concrete runtime technique itself, the selective profiling mechanism also serves as a sample application for our recurrent behaviour detection. The profiling accurate results confirm that our hardware data based phase detection scheme provides useful information to locate the most important, repetitive portions in the execution of Java programs.

Adaptive recompilation is an essential factor for highly efficient JIT. We have shown improvements to adaptive recompilation by employing a dynamic strategy based on program phases. Our online adaptive recompilation engine makes recompilation choices based on dynamically acquired hardware phase data, and exhibits good overall performance. We have also evaluated this optimization in terms of several analytical measurements.

We have made an exploration of improving garbage collection using hardware variation information. A solution have been provided to select good garbage collection points according to hardware patterns. Hardware patterns reflect the hardware performance variations which, in turn, represent large behaviour changes in the running program. Our solution performs much better than the original implementation. We also have made a deep discussion on the garbage collection point problem. Our work touched on a variety of important aspects of this problem and is helpful for further investigating this topic.

9.2 Future Work

Several improvements can be applied to our hardware phase detection mechanism. The most direct optimization over the current implementation is to use a combination of several hardware events as the performance indicator. Of course, the concrete composition and the weights of events in this combination require further study. We have discovered that programs have different sensitivities to different hardware events. To further optimize the hardware phase detection, one potential solution is to use benchmark-specific performance indicators. The event or event combination that are appropriate for a particular program can be found from offline analysis. In fact, an offline/online hybrid implementation can bring benefit in many aspects of the current implementation in general, *e.g.*, supporting the online engine to choose benchmark-specific values for the settings, such as the bounds of variation levels in pattern construction. Moreover, better hardware pattern construction algorithms might also be further investigated.

Concrete adaptive optimizations can get benefit from wider directions other than using offline-online mixed mechanisms and considering benchmark specific issues. One potential improvement is to employ a hardware-software hybrid strategy. We can associate hardware detection results with concrete software structures of the programs. This can be used to calculate better garbage collection points or fix the "tail" problem (discussed in page 116) in the adaptive recompilation strategy. Several techniques presented by other researchers can also be integrated with our work, such as the *profile repositories* introduced in [AWR05] and the *fall-back compaction* [MH06] technique presented by McGachey *et al.*. Furthermore, static analysis of program structures can still bring benefit to the optimizations addressed here, *e.g.*, locating better garbage collection point by analyzing the life range of objects, or the dominating area for important allocation sites.

The functionality of hardware performance monitoring, and in particular the performance monitoring unit (PMU) has become stronger and more and more complete. Many impressive new features of PMU (details of these features are described in page 15) had been introduced in recent processors. These new features provide more chances for hardware related optimizations. For example, with the event address register, we can easily locate the address of regions with serious data cache conflicts. We can thus design accu-

rately targeted solutions for dynamic data relocation. The branch trace buffer can also be helpful to identify hot code traces of a program and also changes in hot code regions. We could employ dynamic code reordering and deliver better runtime code layout.

Bibliography

[AAB+05] Bowen Alpern, Steven Augart, Stephen M. Blackburn, Maria Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark Mergen, J. Eliot B. Moss, Ton Ngo, Vivek Sarkar, and Martin Trapp. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal*, 44(2):399–417, April 2005.

[AAC+99] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 314–324, Oct. 1999.

[ABD+97] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Computer Systems*, 15(4):357–390, Nov. 1997.

[AFG+00] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, 2000.

[AFG+05] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[AHR02] Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online feedback-directed optimization of Java. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 111–129, New York, NY, USA, 2002. ACM Press.

[AMD01] AMD. Amd athlon processor x86 code optimization guide. 2001. `http://www.amd.com/`.

[App89] A. W. Appel. Simple generational garbage collection and fast allocation. *Softw. Pract. Exper.*, 19(2):171–183, 1989.

[AWR05] Matthew Arnold, Adam Welc, and V. T. Rajan. Improving virtual machine performance using a cross-run profile repository. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 297–311, New York, NY, USA, 2005. ACM Press.

[BA97] Douglas C. Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-1997-1342, 1997.

[BABD00] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general purpose architectures. In *MICRO 33:the 33rd Annual Intl. Sym. on Microarchitecture*, pages 245–257, Dec. 2000.

[BCM04a] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 137–146. IEEE Computer Society, May 2004.

[BCM04b] Steve M. Blackburn, P. Cheng, and Kathryn S. McKinley. Myths and realities: The performance impact of garbage collection. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 25–36, June 2004.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2000. ACM Press.

[BDG+] S. Brown, J. Dongarra, N. Garner, K. London, and P. Mucci. PAPI. `http://icl.cs.utk.edu/papi`.

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[BJMM02] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J Eliot B Moss. Beltway: getting around garbage collection gridlock. *SIGPLAN Not.*, 37(5):153–164, June 2002.

[Boe00] Hans-J. Boehm. Reducing garbage collector cache misses. In *ISMM '00: Proceedings of the 2nd international symposium on Memory management*, pages 59–64, Oct. 2000.

[Bur02] Martin Burtscher. An improved index function for (D)FCM predictors. *Computer Architecture News*, 30(3):19–24, June 2002.

[BZM] Rudolf Berrendorf, Heinz Ziegler, and Bernd Mohr. PCL-the performance counter library. `http://www.fz-juelich.de/zam/PCL/`.

[CEG+05] Michal Cierniak, Marsha Eng, Neal Glew, Brian Lewis, and James Stichnoth. The open runtime platform: a flexible high-performance managed runtime environment: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(5-6):617–637, 2005.

[CFE99] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization, 1999.

[CH02] Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 199–209, New York, NY, USA, 2002. ACM Press.

[CHV04] Chen-Yong Cher, Antony L. Hosking, and T. N. Vijaykumar. Software prefetching for mark-sweep garbage collection: hardware analysis and software redesign. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 199–210, Oct. 2004.

[CKJA98] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. *SIGPLAN Not.*, 33(11):139–149, 1998.

[Cor] Intel Corporation. VTune performance analyzer. `http://www.intel.com/software/products/vtune/`.

[CU91] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 1–15, New York, NY, USA, 1991. ACM Press.

[DA99] David Detlefs and Ole Agesen. The case for multiple compilers. In *OOPSLA'99 Workshop on Peformance Portability, and Simplicity in Virtual Machine Design*, pages 180–194, 1999.

[Dau92] Ingrid Daubechies. *Ten lectures on wavelets*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[DCD03] Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. Characterizing and predicting program behavior and its variability. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 220. IEEE Computer Society, Sep. 2003.

[DS02a] A. Dhodapkar and J. Smith. Dynamic microarchitecture adaptation via co-designed virtual machines, 2002.

[DS02b] Ashutosh S. Dhodapkar and James E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 233–244. IEEE Computer Society, 2002.

[DS03] Ashutosh S. Dhodapkar and James E. Smith. Comparing program phase detection techniques. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 217. IEEE Computer Society, 2003.

[EM98] A. N. Eden and T. Mudge. The yags branch prediction scheme. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 69–77, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[EPG+06] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006.

[Gag] Etienne M. Gagnon. SableVM. http://www.sablevm.org/.

[Gag02] Etienne M. Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, McGill University, 2002.

[GBEB04] Andy Georges, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere. Method-level phase behavior in Java workloads. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 270–287, Oct. 2004.

[GH01] Etienne M. Gagnon and Laurie J. Hendren. SableVM:A Research Framework for the Efficient Execution of Java Bytecode. In *Proceedings of the Java*

Virtual Machine Research and Technology Symposium (JVM '01), pages 27–40. USENIX Association, April 2001.

[GH03] E. Gagnon and L. Hendren. Effective inline-threaded interpretation of Java bytecode using preparation sequences. 2622:170–184, 2003.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[GM04] Samuel Z. Guyer and Kathryn S. McKinley. Finding your cronies: static analysis for dynamic object colocation. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 237–250, Oct. 2004.

[GV07] Dayong Gu and Clark Verbrugge. Using hardware data to detect repetitive program behavior. Technical Report SABLE-TR-2007-2, Sable Research Group, School of Computer Science, McGill University, Montréal, Québec, Canada, March 2007.

[GVdB01] Bart Goeman, Hans Vandierendonck, and Koen de Bosschere. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 207–216. IEEE Computer Society, January 2001.

[GVG05a] Dayong Gu, Clark Verbrugge, and Etienne Gagnon. Assessing the impact of optimization in Java virtual machines. Technical Report SABLE-TR-2005-4, Sable Research Group, McGill University, October 2005.

[GVG05b] Dayong Gu, Clark Verbrugge, and Etienne Gagnon. Code layout as a source of noise in JVM performance. *Studia Informatica Universalis*, 4(1):83–99, March 2005.

[GVG06] Dayong Gu, Clark Verbrugge, and Etienne Gagnon. Relative factors in performance analysis of Java virtual machines. In *VEE '06: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, New York, NY, USA, June 2006. ACM Press.

[GZD02] Dayong Gu, Olivier Zendra, and Karel Driesen. The impact of branch prediction on control structures for dynamic dispatch in java. Technical Report RR-4547, Publication INRIA, 2002.

[Han74] Gilbert Joseph Hansen. *Adaptive Systems for the Dynamic Run-time Optimization of Programs*. PhD thesis, Carnegie-Mellon University, 1974.

[HBJ03] Shiwen Hu, Ravi Bhargava, and Lizy Kurian John. The role of return value prediction in exploiting speculative method-level parallelism. *JILP*, 5:1–21, November 2003.

[HBM+04] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: improving program locality. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 69–80, Oct. 2004.

[Hel] Don Heller. Performance monitoring counter. `http://www.scl.ameslab.gov/Projects/Rabbit/`.

[HRS03] Michael J. Hind, Vadakkedathu T. Rajan, and Peter F. Sweeney. Phase shift detection: A problem classification. Technical Report IBM Research Report RC-22887, IBM T. J. Watson, August 2003.

[HRT03] Michael C. Huang, Jose Renau, and Josep Torrellas. Positional adaptation of processors: application to energy reduction. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 157–168, New York, NY, USA, 2003. ACM Press.

[HU96] Urs Hölzle and David Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, 1996.

[IBM] IBM. Pmapi. `http://www.alphaworks.ibm.com/tech/pmapi`.

[Int02] Intel. Intel architecture software developer's manual. 2002. `http://developer.intel.com/`.

[Jim05] Daniel A. Jiménez. Code placement for improving dynamic branch prediction accuracy. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 107–116, New York, NY, USA, 2005. ACM Press.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996.

[kCBC+06] Wen ke Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao, and Weihaw Chuang. Profile-guided proactive garbage collection for locality optimization. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 332–340, New York, NY, USA, 2006. ACM Press.

[KCS05] Naveen Kumar, Bruce R. Childers, and Mary Lou Soffa. Low overhead program monitoring and profiling. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 28–34, New York, NY, USA, 2005. ACM Press.

[KF03] Thomas Kistler and Michael Franz. Continuous program optimization: A case study. *ACM Trans. Program. Lang. Syst.*, 25(4):500–548, 2003.

[Kri03] Chandra Krintz. Coupling on-line and off-line profile information to improve program performance. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 69–78, Washington, DC, USA, 2003. IEEE Computer Society.

[KS03] Ho-Seop Kim and James E. Smith. Dynamic software trace caching. In *the 30th International Symposium on Computer Architecture (ISCA 2003)*, 2003.

[LH04] Wei Liu and Michael C. Huang. Expert: expedited simulation exploiting program behavior repetition. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 126–135, New York, NY, USA, 2004. ACM Press.

[LPH+05] Jeremy Lau, Erez Perelman, Greg Hamerly, Timothy Sherwood, and Brad Calder. Motivation for variable length intervals to find hierarchical phase behavior. In *2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*, March 2005.

[LSC05] Jeremy Lau, Stefan Schoenmackers, and Brad Calder. Transition phase classification and prediction. In *HPCA*, pages 278–289, 2005.

[LSP+05] Jeremy Lau, Jack Sampson, Erez Perelman, Greg Hamerly, and Brad Calder. The strong correlation between code signatures and performance. In *ISPASS '05: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, page 220. IEEE Computer Society, March 2005.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[McF93] Scott McFarling. Combining Branch Predictors. Technical Report TN-36, Digital Western Research Lab, June 1993.

[McQ67] J. McQueen. Some methods for classification and analysis of multivariate observations. In L. M. LeCam and N. Neyman, editors, *the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297, 1967.

[ME01] David Mosberger and Stephane Eranian. *IA-64 Linux Kernel: Design and Implementation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[MH06] Phil McGachey and Antony L. Hosking. Reducing generational copy reserve overhead with fallback compaction. In *ISMM '06: Proceedings of the 2006 international symposium on Memory management*, pages 17–28, New York, NY, USA, 2006. ACM Press.

[NHK+06] Priya Nagpurkar, Michael Hind, Chandra Krintz, Peter Sweeney, and V.T. Rajan. Online phase detection algorithms. In *CGO '06: Proceedings of the international symposium on Code generation and optimization*, Washington, DC, USA, March 2006. IEEE Computer Society.

[NKS05] Priya Nagpurkar, Chandra Krintz, and Timothy Sherwood. Phase-aware remote profiling. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 191–202, Washington, DC, USA, 2005. IEEE Computer Society.

[NMW97] C. Nevill-Manning and I. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm, 1997.

[oCS95] IEEE Transactions on Computers Staff. Optimal 2-bit branch predictors. *IEEE Trans. Comput.*, 44(5):698–702, 1995.

[OHL99] Jeffrey T. Oplinger, David L. Heine, and Monica S. Lam. In search of speculative thread-level parallelism. In *PACT '99*, pages 303–313. IEEE, 1999.

[Par05] Behrooz Parhami. *Computer Architecture: From Microprocessors to Supercomputers (Oxford Series in Electrical and Computer Engineering)*. Oxford University Press, Inc., New York, NY, USA, 2005.

[PH90] David A. Patterson and John L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[PV04] Christopher J. F. Pickett and Clark Verbrugge. Return value prediction in a Java virtual machine. In *Proceedings of the 2nd Value-Prediction and Value-Based Optimization Workshop (VPW2)*, pages 40–47, October 2004.

[PV06] Sokhom Pheng and Clark Verbrugge. Dynamic data structure analysis for Java programs. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 191–201. IEEE Computer Society, 2006.

[PVC01] Michael Paleczny, Christopher A. Vick, and Cliff Click. The Java HotSpot server compiler. In *Java Virtual Machine Research and Technology Symposium*, pages 1–12, 2001.

[QH02] Feng Qian and Laurie Hendren. An adaptive, region-based allocator for Java. In *ISMM '02: Proceedings of the 3rd international symposium on Memory management*, pages 127–138, June 2002.

[RSEW04] Rodric M. Rabbah, Hariharan Sandanagobalane, Mongkol Ekpanyapong, and Weng-Fai Wong. Compiler orchestrated prefetching via speculation and predication. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 189–198, Oct. 2004.

[SA93] Rabin A. Sugumar and Santosh G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Measurement and Modeling of Computer Systems*, pages 24–35, 1993.

[SCF03] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. Compile-time composition of run-time data and iteration reorderings. *SIGPLAN Not.*, 38(5):91–102, 2003.

[SDDS05] Xipeng Shen, Chen Ding, Sandhya Dwarkadas, and Michael Scott. Characterizing phases in service-oriented applications. http://www.cs.rochester.edu/u/xshen/Abstracts/tr848.html, 2005.

[SE94] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[SE02] A. Srivastava and A. Eustace. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[SHB+02] Darko Stefanović, Matthew Hertz, Stephen M. Blackburn, Kathryn S. McKinley, and J. Eliot B. Moss. Older-first garbage collection in practice: evaluation in a Java virtual machine. In *MSP '02: Proceedings of the 2002 workshop on Memory system performance*, pages 25–36, June 2002.

[SHC+04] Peter F. Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. Using hardware performance monitors to understand the behavior of Java applications. In *VM'04:Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, May 2004.

[SKB04] Sunil Soman, Chandra Krintz, and David F. Bacon. Dynamic selection of application-specific garbage collectors. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 49–60, Oct 2004.

[SMM99] Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 370–381, Oct. 1999.

[SPC01] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[SPHC02] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior, 2002. In Tenth International Conference on Architectural Support for Programming Languages and Operating Systems, October 2002. http://www.cs.ucsd.edu/users/calder/simpoint/.

[Spr02] Brinkley Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, 2002.

[SSC03] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 336–349, 2003.

[Staa] Standard Performance Evaluation Corporation. SPECCPU2000 benchmarks. `http://www.spec.org/cpu2000/`.

[Stab] Standard Performance Evaluation Corporation. SPECCPU95 benchmarks. `http://www.spec.org/cpu95/`.

[Stac] Standard Performance Evaluation Corporation. SPECjvm98 benchmarks. `http://www.spec.org/osg/jvm98`.

[Sta99] William Stallings. *Computer Organization and Architecture: Designing for Performance*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[Sta00] Standard Performance Evaluation Corporation. SPECjbb2000. `http://www.spec.org/osg/jbb2000`, 2000.

[Suna] Sun Microsystems, Inc. The Java Hotspot Virtual Machine V1.4.1. `http://java.sun.com/products/hotspot/index.html`.

[Sunb] Sun Microsystems, Inc. The Java Virtual Machine Tools Interface. `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/`.

[SYK+01] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 180–195, New York, NY, USA, 2001. ACM Press.

[SZD04] Xipeng Shen, Yutao Zhong, and Chen Ding. Locality phase prediction. *SIGPLAN Not.*, 39(11):165–176, 2004.

[Ung84] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 157–167, Apr. 1984.

[Ven96] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, Inc., New York, NY, USA, 1996.

[Wha01] John Whaley. Partial method compilation using dynamic profile information. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 166–179, New York, NY, USA, 2001. ACM Press.

[ZSCC06] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 263–271, New York, NY, USA, 2006. ACM Press.