Profileringstechnieken voor prestatieanalyse
en optimalisatie van Javaprogramma's

Profiling Techniques for Performance Analysis
and Optimization of Java Applications

Dries Buytaert

Promotoren: prof. dr. ir. K. De Bosschere, prof. dr. ir. L. Eeckhout
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. J. Van Campenhout
Faculteit Ingenieurswetenschappen
Academiejaar 2007 - 2008

UNIVERSITEIT
GENT

ISBN 978-90-8578-184-4
NUR 980
Wettelijk depot: D/2008/10.500/3

Nothing in this world can take the place of persistence.
Talent will not; nothing is more common than unsuccessful people with talent.
Genius will not; unrewarded genius is almost a proverb.
Education will not; the world is full of educated derelicts.
Persistence and determination alone are omnipotent.
Calvin Coolidge

Acknowledgements

This thesis would not have been possible without the help of many people. I wish to express my sincere thanks to:

- My two advisors Prof. Koen De Bosschere and Prof. Lieven Eeckhout for offering me the chance to be part of their research group, for being extraordinary mentors, and for encouraging me to pursue my interests.
- The members from my thesis committee: Prof. Jan Van Campenhout, Prof. Koen De Bosschere, Prof. Lieven Eeckhout, Prof. Frans Arickx, Prof. Luc Taerwe, Prof. Filip De Turck, Dr. James Gosling, Dr. Michael Hind and Dr. Johan Vos.
- The people who read early drafts of this dissertation: Prof. Koen De Bosschere, Prof. Lieven Eeckhout, Andy Georges and Peter Bertels.
- My colleagues at Ghent University who provided an enjoyable research environment and who have helped me understand how to do research. In particular, Andy Georges, Dr. Kris Venstermans and Dr. Jonas Maebe who had tremendous impact on this thesis.
- Dr. Michael Hind and Dr. Matthew Arnold from IBM T.J. Watson Research. Without their collaboration this thesis might have looked different as one of our OOPLSA'07 papers might never have been written.
- Benjamin Schrauwen for convincing me to solicit at Ghent University and for entertaining me on our daily commute.
- The Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) for providing me a research grant.
- My wife Karlijn, my son Axl, my parents, my family and my friends for their continuous support. You rock.

Dries Buytaert
Ghent, January 24, 2008

Examencommissie

Prof. Frans Arickx
Vakgroep CoMP, Departement Computerwetenschappen
Universiteit Antwerpen

Prof. Koen De Bosschere, promotor
Vakgroep ELIS, Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. Filip De Turck
Vakgroep INTEC, Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. Lieven Eeckhout, promotor
Vakgroep ELIS, Faculteit Ingenieurswetenschappen
Universiteit Gent

Dr. James Gosling
Vice President
Sun Microsystems

Dr. Michael Hind
Research staff member and senior manager
IBM T.J. Watson Research

Prof. Luc Taerwe, voorzitter
Vakgroep Labo Magnel, Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. Jan Van Campenhout, secretaris
Vakgroep ELIS, Faculteit Ingenieurswetenschappen
Universiteit Gent

Dr. Johan Vos
Co-CEO
Lodgon

Leescommissie

Prof. Frans Arickx
Vakgroep CoMP, Departement Computerwetenschappen
Universiteit Antwerpen

Prof. Filip De Turck
Vakgroep INTEC, Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. Lieven Eeckhout, promotor
Vakgroep ELIS, Faculteit Ingenieurswetenschappen
Universiteit Gent

Dr. James Gosling
Vice President
Sun Microsystems

Dr. Michael Hind
Research staff member and senior manager
IBM T.J. Watson Research

Samenvatting

Beheerde programmeertalen zoals Java hebben de laatste jaren een enorme vlucht genomen. Functionaliteit zoals overdraagbaarheid, dynamische typecontrole, veiligheid, wijzerencapsulatie, het dynamisch laden van klassen en automatisch geheugenbeheer laat software-ontwikkelaars toe om sneller betere code te schrijven.

Om een deel van deze functionaliteit te bewerkstelligen is een gesofisticeerde virtuele machine vereist. Javabroncode wordt vertaald naar Java-bytecode, en die kan dan worden uitgevoerd op elk hardwareplatform waarvoor een Java Virtuele Machine beschikbaar is. In vergelijking met programma's geschreven in traditionele programmeertalen zoals C of C++, beschikken programma's die gebruik maken van een virtuele machine over een extra niveau van abstractie en virtualisatie.

Een gevolg hiervan is dat voor het efficiënt uitvoeren van een Javaprogramma een extra vertaalslag nodig is. Om de uitvoering van Javaprogramma's te versnellen, vertalen de meeste Java Virtuele Machines de Java-bytecode van frequent uitgevoerde methoden naar machinetaal zodat de code rechtstreeks uitgevoerd kan worden op het onderliggende hardwareplatform. Deze vertaling gebeurt meestal tijdens de uitvoering van het programma zelf.

Omdat de uitvoering van toepassingen geschreven in beheerde programmeertalen een geavanceerde virtuele machine vereist, bestaat er een complexe interactie tussen het programma en de onderliggende virtuele machine. Dit maakt het voor software-ontwikkelaars vaak moeilijk om het uitvoeringsgedrag van Javaprogramma's te doorgronden. Bovendien worden Javaprogramma's vaak ingewikkelder; door de opkomst van middelware-lagen breiden Javaprogramma's uit in grootte en complexiteit.

Omwille van deze redenen is het begrijpen, laat staan het verbeteren, van de prestatie van programma's geschreven in een dergelijke programmeertaal verre van triviaal is. Dit is dan ook het kernprobleem dat we behandelen in dit proefschrift.

De oplossing die we aanreiken ligt in *profilering*. Het centrale thema van dit proefschrift is het verzamelen van meer complete profielen ten-

einde de prestatie van Javaprogramma's te begrijpen en te verbeteren. Naarmate programma's complexer worden en virtualisatie aan belang toeneemt, wordt het steeds belangrijker om informatie van verschillende lagen van de uitvoeringsstapel te combineren in één profiel. In dit proefschrift stellen we drie nieuwe profileringstechnieken voor die ons toelaten om meer complete profielen op te meten.

Ten eerste presenteren we *Javana*, een raamwerk dat het mogelijk maakt om op een eenvoudige manier verticale profileringstoepassingen te bouwen. Javana gebruikt dynamische binaire instrumentatie om laag-niveau-informatie te verzamelen en terug te koppelen aan hoogniveauprogrammeerconstructies zoals objecten, methoden en draden. Het gebruik van een dynamisch binair instrumentatieraamwerk is traag, maar functioneel accuraat. We tonen aan hoe Javana kan gebruikt worden om informatie, verzameld op verschillende lagen van de uitvoeringsstapel, te combineren tot één profiel, en hoe software-ontwikkelaars en onderzoekers op die manier meer inzicht kunnen krijgen in de uitvoering van Javaprogramma's.

Vervolgens stellen we *MonitorMethod* voor, een nieuwe profileringstechniek die het fasegedrag van Javaprogramma's gebruikt om informatie verzameld op het niveau van de microprocessor te koppelen met de broncode van die Javaprogramma's. Net zoals Javana maakt MonitorMethod gebruik van instrumentatie, maar dan op basis van hardware-prestatietellers (HPTs) in plaats van dynamische binaire instrumentatie. Het gebruik van HPTs laat ons toe om de profielinformatie te verzamelen met een kleinere overhead dan bij Javana. We tonen aan hoe software-ontwikkelaars deze informatie kunnen gebruiken om prestatieproblemen in hun Javaprogramma's te identificeren, te lokaliseren en te verklaren.

Tot slot stellen we *HPT-bemonstering* voor, een techniek die gebruik maakt van de HPTs om methoden te vinden die in aanmerking komen om door de Java Virtuele Machine geoptimaliseerd te worden. De HPTs worden hier niet gebruikt om inzicht te krijgen in het gedrag van Javaprogramma's, maar om de Java Virtuele Machine te helpen de uitvoering van Javaprogramma's te versnellen. Het optimaliseren van het programma en het verzamelen van de informatie die hiervoor nodig is, moet gebeuren tijdens de uitvoering van het programma. Het is dus belangrijk dat de overhead veroorzaakt door de gebruikte profileringstechniek, tot het minimum herleid wordt. We bereiken dit door gebruik te maken van bemonstering in plaats van instrumentatie en noemen deze techniek dan ook *HPT-bemonstering*. Met HPT-bemonstering illustreren we dat door het propageren van profielinformatie van de hardware naar de Java Virtuele Machine, we sneller en nauwkeuriger belangrijke methoden kunnen identificeren dan in software geïmplementeerde profileringstechnieken.

Het overkoepelende verband tussen deze bijdragen is dat ze informatie verzamelen op verschillende lagen van de uitvoeringstapel en dat deze informatie gecombineerd wordt met als doel meer volledige profielen

te verzamelen. Door gebruik te maken van profielinformatie afkomstig van het microprocessor-niveau kunnen we profielen opstellen die meer accuraat zijn, sneller te bekomen zijn, en die we vroeger niet ter beschikking hadden. Concreet tonen we aan hoe deze verbeteringen in profileringstechnieken ons helpen bij het begrijpen van het prestatiegedrag en het versnellen van Javaprogramma's.

Abstract

Managed languages such as Java become increasingly popular. Features like portability, dynamic type checking, security, pointer encapsulation, dynamic class loading and automatic memory management allow developers to produce higher quality software faster.

To support a number of these features, a sophisticated virtual machine is required. Java source code is translated to Java bytecode, and Java bytecode can then be executed on all host platforms for which a Java Virtual Machine is available. Compared to applications written in traditional programming languages such as C or C++, applications that run on top of a virtual machine benefit from an additional layer of abstraction and virtualization.

A result of the additional abstraction and virtualization is that the execution of Java bytecode is much slower than executing native machine code. To improve performance of Java applications, most Java Virtual Machines compile the Java bytecode of frequently executed methods to machine code so they can run directly on the underlying hardware platform. The compilation typically takes place at runtime while the application is running.

Because the execution of applications written in such languages requires a sophisticated virtual machine, there exists a complex interaction between the application and the underlying virtual machine. As such, for software developers, it is difficult to fully understand the performance of Java applications. In addition to that, as middleware applications are becoming commodity, Java applications are growing in size and complexity.

Understanding and optimizing the performance of applications written in managed programming languages is non-trivial. This is the key challenge that we will address in this dissertation.

The solution lies in *profiling*. The central thesis of this dissertation is that in order to understand and optimize the performance of Java applications, the ability to capture complete profiles is key. As Java applications become more complex and as virtualization becomes more important, we need to investigate profiling techniques that combine information captured at different layers of the execution stack. In this PhD dissertation, we present three novel profiling techniques that capture more complete

profiles.

The first contribution is the proposal of *Javana*, a framework for building customized vertical profiling tools. Javana uses dynamic binary instrumentation and an event handling mechanism in the VM to link low-level information to programming language constructs such as objects, methods and threads. By doing so, higher-level profiling tools can be built. Using binary instrumentation is slow but functionally accurate. Application developers and researchers can use Javana to build a wide variety of profiling tools that aggregate information across different layers of the execution stack into one profile.

The second contribution is the proposal of a profiling mechanism that links hardware performance monitors (HPMs) to method-level phases by means of instrumentation. We use this technique to build *MonitorMethod*, a tool that helps software developers identify and explain performance bottlenecks in their Java application. Just like Javana, MonitorMethod uses instrumentation to capture profile information. However, by using hardware performance monitors instead of dynamic binary instrumentation, MonitorMethod has limited overhead compared to Javana.

Finally, the third contribution is the proposal of an online profiling technique that uses hardware performance monitors, not to gain insight in the behavior of applications, but to improve their performance. Here, profile information is used to identify frequently executed methods, and to help the JVM make better optimization decisions at runtime. Because the profiling and compilation of methods happens during the execution of the program, this profiling technique is optimized to collect information with extremely low overhead. To control the overhead, it uses a sampling mechanism rather than instrumentation. Hence, we called this contribution *HPM-sampling*. We illustrate that by propagating profile information captured by the hardware to the Java Virtual Machine, we are able to identify frequently executed methods faster and more accurately compared to profiling techniques implemented in software.

The central attribute of these contributions is that they link information gathered at different levels of the execution stack, and that this information is used to gather more complete profiles. By collecting information at the micro-processor level, we have profile information that is more accurate, faster to obtain, or that was otherwise not available. We show that advances in profiling techniques lead to better understanding of program behavior and faster executing Java applications.

Contents

List of Tables

List of Figures

List of Abbreviations

AOP	Aspect-oriented programming
AOS	Adaptive optimization system
API	Application programming interface
BMP	Branch misprediction
CLI	Common language infrastructure
CopyMS	Copy mark sweep
CPI	Cycles per instruction
GC	Garbage collection
HPM	Hardware performance monitor
IR	Intermediate representation
JIT	Just-in-time
JNI	Java native interface
JVM	Java virtual machine
MSIL	Microsoft intermediate language
OOP	Object-oriented programming
OS	Operating system
OSR	On stack replacement
PAPI	Performance API
RVM	Research virtual machine
VM	Virtual machine

Chapter 1

Introduction

Software development is hard. It is no surprise that high-level managed programming languages such as Java and C# – but also scripting languages like Python, Perl and PHP – have become increasingly popular over the past few years. In fact, a recent Gartner report estimates that 80% of new software development will be in Java or C# [36].

What all of these languages have in common, and what helps drive their adoption, is the fact that they run on a *high-level language virtual machine* (VM). Computers are among the most advanced engineered systems, and the key to managing complexity is abstraction and virtualization. The higher the level of *abstraction*, the more details are hidden, the easier it becomes to program for. The better the *virtualization*, the better the software can be isolated from the underlying host platform, and the less likely developers have to produce separate versions of their application for each target platform of interest.

Compared to applications written in traditional programming languages like C or C++, applications that run on top of a VM benefit from an additional layer of abstraction and virtualization, and this additional layer can provide important features such as portability, dynamic type checking, security, dynamic class loading, automatic memory management, etc. These features, provided by abstraction and virtualization, help developers produce higher quality software faster.

For example, software written in a managed programming language is often compiled to a machine-independent instruction set such as Java bytecode or the Microsoft Intermediate Language (MSIL) in the .NET framework. The VM defines its own instruction set, and the application's source code is compiled to the VM's instruction set much like C or C++ applications are compiled to the processor's instruction set. This is how virtualization makes applications highly portable, as it is not the application, but the VM that needs to be ported to make the application run on a

new host platform.

Another important advantage provided by most VMs is *pointer encapsulation*. VMs often force developers to think in terms of objects rather than in terms of memory locations, and don't allow direct accesses to raw memory addresses. Just like a file system makes it easier to operate a physical hard disk, pointer encapsulation make it easier to use a computer's physical memory. A VM that supports pointer encapsulation prevents inadvertent or malicious memory corruption.

On top of that, most VMs use *automatic memory management* which enables the VM to automatically reclaim objects in memory that are no longer reachable by the application, making it easier to develop applications, and making applications more robust to memory corruption.

Not surprisingly, the abstractions and virtualizations introduced by VMs come at some costs. It is clear that the goals of software engineering and the ability to achieve high performance are often at odds. The execution of applications written in managed programming languages involves a complex interaction between the application and the underlying VM. In addition to that, the applications themselves are growing in size and complexity. As a result, understanding and optimizing the performance of today's applications written in managed programming languages is hard. These are the key challenges that we will address in this PhD dissertation.

In the remainder of this thesis, we will focus on Java Virtual Machines (JVMs). However, the presented techniques and ideas are applicable to many other VM-based platforms as well.

1.1 Challenges in VM performance analysis and optimization

We envision two major challenges in VM performance analysis and optimization: (i) understanding the runtime performance of applications written in managed programming languages, and (ii) automatically improving their runtime performance.

1.1.1 Understanding runtime performance

Application and system software developers need a good understanding of an application's behavior in order to optimize overall system performance. Analyzing the behavior of applications written in languages such as C and C++ is a well understood problem [56, 65, 66, 75], but understanding the behavior of modern software that relies on a VM is much more challenging.

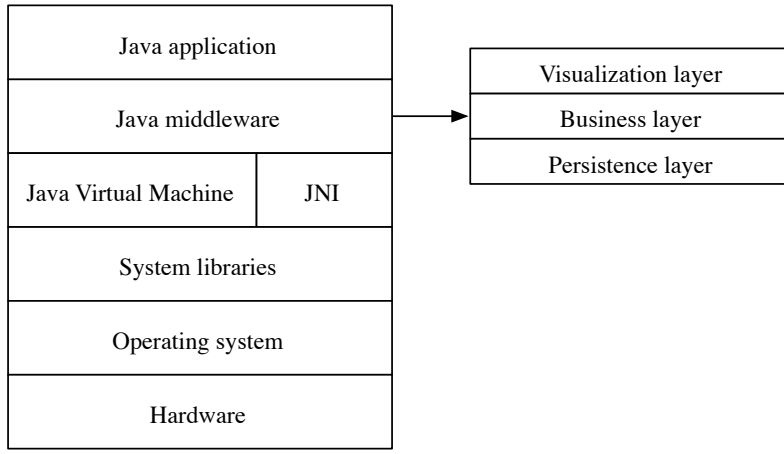


Figure 1.1: An overview of a Java execution stack.

To support the features that modern programming environments like Java offer [42], a sophisticated VM is required [54]. A high-level language VM consists of a number of complex sub-components such as a class loader, an interpreter, a run-time compiler, an optimizer, a garbage collector and much more. The execution of applications involves a complex interaction between the Java code and the different VM components, and between the VM, the operating system and the hardware which makes it challenging to relate hardware performance with source code. The behavior that is observed at the highest level in the execution stack is not just the result of the application, but is a result of the interaction between the application, the VM, the operating system and the hardware.

Furthermore, the applications themselves are growing in size and complexity. Or, as Norman R. Augustine states it: *Software is like entropy. It is difficult to grasp, weighs nothing, and obeys the second law of thermodynamics, i.e., it always increases*. For example, enterprise Java applications often run on top of an application server or framework. A good example of such a framework is a Content Management System (CMS); it provides an additional layer between the VM and the actual applications that are implemented on top of the CMS. Like a CMS, there are many other middelware applications that provide business logic or data access to other software components or applications. Figure 1.1 shows an example Java execution stack.

While it is useful to shield software developers from as many details as possible and to delegate responsibility, it makes it increasingly difficult to conduct a thorough performance analysis. The more layers there are, and the more complex each of these layers becomes, the harder it be-

comes to understand the behavior of the entire application. In general, there is an increasing need to gather more complete profiles that contain information about the behavior of each of the components in the execution stack. Capturing a performance profile that crosscuts multiple layers of the software in a way that is useful for software developers is a key challenge.

1.1.2 Improving runtime performance

The second challenge stems from the fact that VMs need to run an application written in a machine-independent instruction set as efficiently as possible. Early VM implementations interpreted programs by emulating the machine-independent instructions. Unfortunately, an interpreted execution is much slower than source code compiled to native machine code. In particular, Java source code is compiled to a machine-independent instruction set, called *Java bytecode*, and early versions of the JVM were not known for their blistering performance at interpreting and executing Java bytecode.

To overcome this problem, Sun Microsystems added a *runtime compiler* to their Java Virtual Machine (JVM). The run-time compiler translates the machine-independent instructions to native machine code during the execution of the program. This technique is often referred to as *dynamic compilation* and was first pioneered by the Smalltalk and Lisp community in the 1980s. In the early 1990s, the Self group at Sun Microsystems worked on dynamic compilation techniques to make Self perform well [85]. Later, in 1996, when Java entered the scene, many of the research results and innovations of the Self group became part of the Java Virtual Machine. In 1997, Sun Microsystem purchased Longview Technologies, the company that originally developed the *HotSpot Virtual Machine*, widely respected for its runtime compiler. In 1999, HotSpot became the default Sun JVM in Java 1.3 [67].

Run-time compilers typically compile the machine-independent instructions on a per-method basis just before the code is about to be executed; hence also the name *just-in-time compiler*, or *JIT compiler* for short [7, 15, 67, 80]. Whereas *static compilers* are often limited by the information available at compile time, JIT compilers can take advantage of the profile information collected at runtime to dynamically adapt to changes in program execution behavior and can therefore concentrate their efforts on methods where they can be most effective [2, 8, 10, 69].

Not surprisingly, a JIT compiler's biggest challenge lies in balancing the overhead of profiling and JIT compilation versus code quality. Compiling a method from Java bytecode to native machine code can be expensive. Compiling and optimizing a method that will only be executed once is likely to result in a performance loss. It is important that the per-

formance gain obtained from compiling and optimizing a method amortizes the cost of profiling and run-time compilation. Unlike static compilers that have the luxury of spending a lot of time analyzing the source code, run-time compilers are restricted in the scope of the analysis and optimizations they can perform. As a consequence, dynamic compilation systems need to balance code quality with the time spent compiling the method.

To balance these costs and to achieve high-performance, production Java virtual machines contain at least two modes of execution: (i) *unoptimized* execution, using interpretation [61, 67, 80] or a simple dynamic compiler [7, 15, 24, 47] that produces code quickly, and (ii) *optimized* execution using an optimizing dynamic compiler. Methods are first executed using the unoptimized execution strategy. An online profiling mechanism is used to find a subset of methods to optimize during the same execution. Many systems enhance this scheme to provide multiple levels of optimized execution [7, 61, 80], with increasing compilation cost and benefits at each level. This approach is commonly referred to as *adaptive* or *selective compilation*.

A crucial component to improve runtime performance is the ability to capture the necessary profile information in a low-overhead and accurate manner. The quantity and the quality of the information gathered directly impacts the performance of runtime environments. As advances in online profiling techniques will lead to advances in performance, improving existing online profiling schemes is our second key challenge.

1.2 Profiling

As hinted in the previous section, the key solution to both challenges – understanding the behavior of layered applications and optimizing the performance of virtualization software – lies in *profiling*. We define the term *profiling* to mean, in a broad sense, the ability to capture information on the execution of the program.

We now provide background information with respect to the state-of-the-art in offline and online profiling techniques prior to the work presented in this dissertation.

1.2.1 Profiling for program understanding

The most obvious way to profile the performance of an application is by manually instrumenting the Java source code of the application. Manual instrumentation is tedious, hence there exists a large body of work about offline profiling tools [9, 14, 22, 25, 75]. *Offline* meaning that the pro-

files are collected in a separate preparatory run of the program and that the information is then used to understand and (potentially) optimize the program's performance.

The Java Virtual Machine Profiler Interface (JVMPI) defines a general purpose mechanism to obtain profile data from the JVM [81]. It is intended for tool vendors to develop profilers that work in conjunction with Sun's JVM implementation. Profiling tools like IBM Jinsight, JProbe, HProf, OptimizeIt, etc. obtain their profile information from the virtual machine using the JVMPI [43]. The JVMPI is a two-way API between the JVM and a profiler agent. The profiling agent specifies to the JVM what kind of events it is interested in, and the JVM will tell the agent when an event occurred. The JVMPI supports CPU-time profiling for threads and methods, heap profiling, and monitor contention profiling. The JVMPI does not allow events to be matched to host-specific events, or events that occurred at lower levels of the execution stack. As such, the JVMPI does not allow vertical profiles to be collected.

Building customized profiling tools

In 1994, Amitabh Srivastava and Alan Eustace of Digital Equipment Corporation published a paper describing ATOM [75]. ATOM is a *static binary instrumentation tool*; it provides a mechanism that allows elements of the program executable, such as instructions, basic blocks, and procedures to be queried and manipulated. In particular, ATOM allows one to add custom code to an executable. This custom code runs in the same address space, making it possible to collect information about registers used, branch conditions, data-flow analysis, etc. This technique, modifying a program to profile itself, is known as *instrumentation*.

The approach taken by ATOM proved to be one of the most general and efficient techniques for the transformation and instrumentation of programs. As a consequence, a lot of systems have been developed that provide ATOM-like functionality on various platforms. For example, it leads to the development of *dynamic binary instrumentation tools* like PIN [56], Valgrind [65, 66] or DIOTA [60]. Given an instrumentation specification, these tools transform the binary application at runtime to intercept memory operations, code execution, signals, function calls, system calls, etc.

In the context of Java, the first academic works on Java bytecode instrumentation and transformation were BIT (Bytecode Instrumentation Toolkit) [52] and BCEL (Bytecode Engineering Library) [6]. Both tools can be used to alter the behavior of existing Java applications. Program transformation tools developed with BIT or BCEL typically take Java class files as input, and output modified version of these class files. This can be used to eliminate dead code or unused variables, to add instrumentation code

that locates where a program spends its time, or to collect information that can be used as input for profile-based optimization systems.

Unfortunately, these tools are inadequate to understand the complex interactions that exist in virtual execution environments. As programming environments become more complex, and as middleware applications are becoming commodity, there is a desire for more complete profiles that crosscut multiple layers in the execution stack. Unfortunately there is no ATOM-like tool that allows one to create a wide variety of profiling tools for Java applications.

Using hardware performance counters

Modern processors are often equipped with a set of performance counter registers. These registers are designed to count microprocessor events that occur during the execution of a program. They allow to keep track of the number of retired instructions, elapsed clock cycles, cache misses, branch mispredictions, etc.

Sweeney et al. [82] present a system to profile microprocessor-level behavior of Java workloads. At every thread switch, one or more hardware performance monitors are read. Doing so, they generate traces of hardware performance monitor values while executing Java applications. They also present a tool for graphically exploring the performance counter traces.

In later work [44, 45], Hauswirth et al. found that using hardware performance monitors was not enough for a complete understanding of certain performance phenomena, and concluded that they also needed information from higher layers in the execution stack. They extended their previous work [82] to coin a new profiling technique called *vertical profiling*. They used the term to describe a profiling setup in which a stream of hardware performance monitors were recorded and later on synchronized with events occurring at the VM level. This information can be used to explain unusual, unexpected or undesirable aspects of the application's performance. For example, they used it to show that a dip in the number of cycles per instruction (CPI) corresponds to garbage collector activity, and that an increase in CPI was due to a method being recompiled at a higher optimization level.

While the recent work illustrates the usefulness of using hardware performance monitors to better understand program behavior, the results obtained with their techniques are coarse-grained as they only read the hardware performance monitors at every thread switch.

1.2.2 Profiling for runtime optimization

The goal of *online* profiling techniques is to collect profile information during the execution of the application and to consume that information within the same run [2, 8, 10, 69].

Applying offline profiling techniques online

There has been significant interest in making offline profiling techniques online. Online profiling tools are fundamentally different from offline profiling tools. An online system needs to profile the application during program execution and when profiling is expensive, it is difficult for optimizations to amortize this cost. As a consequence, offline profiling tools tend to favor precision, whereas online profiling tools tend to focus on achieving low-overhead.

A number of solutions have been proposed to limit the overhead of offline profiling techniques to enable them being used online. On-stack replacement (OSR) [37] is a technique used to enable and disable program specialization. Using OSR, an executing method can be recompiled or transformed, and its stack frame can be dynamically replaced with that of the new version. This allows methods with instrumentation code to be replaced with methods that do not execute instrumentation code. Arnold and Ryder [11] proposed a general framework for performing *instrumentation sampling*, allowing previously expensive instrumentation to be performed with low overhead. Their framework performs code duplication and uses compiler-inserted counter-based sampling to switch between instrumented and non-instrumented code. The reduction in overhead provided by such systems allows profiling to be performed for a longer time and makes it easier to control the overhead of online profiling.

Identifying methods to optimize

As explained, to achieve high performance, production Java virtual machines combine an unoptimized and optimized execution mode. Methods are initially interpreted or compiled with a simple baseline compiler. However, an online profiling mechanism is used to find a subset of methods to optimize during the same execution.

Two approaches that are commonly used to find optimization candidates are method *invocation counters* [24, 61, 67, 80] and *timer-based sampling* [7, 15, 61, 80, 86]. Although invocations counters can be used for profiling unoptimized code, their overhead makes them a poor choice for use in optimized code.

Most VMs rely on an operating system timer interrupt to perform sampling, but this approach has a number of drawbacks. First, the min-

imum timer interrupt varies depending on the version of the operating system, and in many cases can result in too few samples being taken. Second, the sample-taking mechanism is untimely and inaccurate because there is a variable delay between the timer going off and the sample being taken. Third, the minimum sample rate does not change when moving to newer, faster hardware; thus, the effective sample rate (relative to the program execution) continues to decrease as hardware performance improves.

1.3 The focus and contributions of this thesis

The central thesis of this PhD dissertation is that in order to understand and optimize the performance of Java applications, the ability to capture complete profiles is of key importance. As we go forward, we need to investigate profiling techniques that combine information captured at different layers of the execution stack.

This PhD dissertation presents three novel profiling techniques representing different trade-offs in accuracy versus overhead. The central attribute of these contributions is that they link information gathered at different levels of the execution stack, and that this information is used to gather more complete profiles. By collecting information at the microprocessor level, we have profile information that is more accurate, faster to obtain, or that was otherwise not available.

The first contribution is the proposal of *Javana*, a framework for building customized vertical profiling tools. Javana uses dynamic binary instrumentation and an event handling mechanism in the VM to link low-level information to programming language constructs such as objects, methods and threads. By doing so, higher-level profiling tools can be built. Using binary instrumentation is slow but functionally accurate. Application developers and researchers can use Javana to build a wide variety of vertical profiling tools. Example applications of Javana include object lifetime analysis, memory address tracing, etc.

The second contribution is the proposal of a profiling mechanism that links hardware performance monitors to method-level phases by means of instrumentation. Using hardware performance monitors instead of dynamic binary instrumentation makes that this technique can be used with limited overhead. We used this technique to build *MonitorMethod*, a tool that helps developers identify and explain performance bottlenecks in their Java application.

Finally, the third contribution is the proposal of an online profiling technique that uses hardware performance monitors, not to gain insight in the behavior of applications, but to improve their performance. Here, profile information is used to help the JVM make better optimization de-

cisions at runtime. Because it is an online profiling tool, it is optimized to collect information with extremely low overhead. To control the overhead, we use a sampling mechanism rather than instrumentation. Hence, we called this contribution *HPM-sampling*. Table 1.1 compares the different properties of these contributions.

Contribution 1: Javana	Contribution 2: MonitorMethod	Contribution 3: HPM-sampling
- Offline	- Offline	- Online
- Uses binary instrumentation	- Uses hardware performance monitors	- Uses hardware performance monitors
- Uses instrumentation	- Uses instrumentation	- Uses sampling
- For application developers and researchers	- For application developers and VM developers	- For the Java Virtual Machine
- Very detailed profile information	- Moderately detailed profile information	- Coarse-grained profile information
- Slow	- Fast	- Fastest

Table 1.1: A comparison of the contributions in this PhD dissertation.

We now discuss each of these contributions in more detail.

1.3.1 Contribution 1: Offline profiling using binary instrumentation

The first technique uses *dynamic binary instrumentation* to create customized Java program analysis tools. A dynamic binary instrumentation tool runs underneath the JVM. The JVM informs the instrumentation layer about a number of events, for example when an object is created, moved or collected, or when a method gets compiled or re-compiled, etc. The dynamic binary instrumentation tool then catches these events and links instruction pointers and memory addresses to high-level language concepts such as objects and methods.

We use this technique to build Javana, a proof-of-concept tool that provides developers an easy-to-use instrumentation framework to develop profiling tools that crosscut the Java application, the JVM and the native execution layers. To do so, the Javana instrumentation framework provides the end user with both high-level and low-level information. The high-level information relates to the Java application and the VM, such as thread IDs, method IDs, source code line numbers, object IDs, object types, etc. The low-level information consists of instruction pointers, and memory addresses. The end result is that Javana knows for all native

instructions from what method and thread the instruction comes and to what line of source code the instruction corresponds; and for all accessed memory locations, Javana knows what objects are being accessed. Running the Java application of interest within the Javana system along with user-specified instrumentation routines then collects the desired profiles of the Java application. This allows for building a wide variety of profiling tools, such as memory address tracing, vertical cache simulation, object lifetime computation, etc.

To illustrate the usefulness of Javana consider Table 1.2 which is the output of a simple profiling tool built with Javana that computes the cache behavior per line in the Java source code. As we will show in this dissertation, the `shell_sort` method of the SPECjvm98 `db` benchmark is an important performance bottleneck – we will describe the experimental setup in Chapter 2. Table 1.2 shows the `shell_sort` method annotated with cache miss information, i.e., L1 and L2 data cache miss rates are annotated to each line of source code. Line 13 seems to be the primary source for the high cache miss rate in the `shell_sort` method. The reason is that the `j+gap` index results in a fairly random access pattern into a 61 KB `index` array.

Instrumentation tasks like the example above look conceptually simple, however, in practice they are challenging to implement without Javana. To accurately capture cache miss information, the JVM and its compilers need to be adjusted in numerous ways, we have to deal with native code that is called through the Java Native Interface (JNI), the standard class libraries need to be instrumented, etc. Javana is useful because it enables one to easily build accurate profiling tools without having to manually instrument the application, the JVM, the standard class libraries or the native code. The code required to build the profiling tool to compute Table 1.2 was roughly 200 lines and will be presented in Chapter 2.

Another key advantage of Javana is that it is functionally accurate (by construction) because the dynamic binary instrumentation layer can track every executed machine code instruction. The pay-off however is that tracking individual instructions results in a slowdown that varies between a factor 125× and 850×. A large part of that overhead can be attributed to the dynamic binary instrumentation layer that is required to build a profiling framework like Javana.

This work is described in[1]:

- *Javana: A system for building customized Java program analysis tools*, Jonas Maebe, Dries Buytaert, Lieven Eeckhout and Koen De Boss-

[1]This was joint work with Jonas Maebe. The focus of Jonas' contributions was on the aspect-oriented instrumentation specification and the dynamic binary instrumentation. My contribution was the overall system design, the vertical profiling infrastructure and demonstrating the use of Javana for building customized profiling tools. The work presented in this dissertation represents my contributions.

Source code		DL1 accesses	DL1 misses	DL2 accesses	DL2 misses
1	`void shell_sort(int fn) {`				
2	`  int i, j, n, gap;`				
3	`  String s1, s2;`				
4	`  Entry e;`				
5					
6	`  if (index == null) set_index();`	67	0 (0%)	0	0 (0%)
7	`  n = index.length;`	134	1 (0%)	1	0 (0%)
8					
9	`  for (gap = n/2; gap > 0; gap/=2)`	938	0 (0%)	0	(0%)
10	`   for (i = gap; i < n; i++)`	12,276,499	910 (0%)	1,083	3 (0%)
11	`    for (j = i-gap; j >=0; j-=gap) {`	23,064,743	8,179 (0%)	9,615	33 (0%)
12	`     s1 = (String)index[j].items.elementAt(fn);`	157,553,557	29,772,665 (19%)	36,551,726	6,095,594 (17%)
13	`     s2 = (String)index[j+gap].items.elementAt(fn);`	157,553,557	24,036,992 (15%)	29,456,752	15,581,062 (53%)
14					
15	`     if (s1.compareTo(s2) <= 0) break;`	45,015,302	128 (0%)	153	1 (0%)
16					
17	`     e = index[j];`	32,322,537	219 (0%)	228	0 (0%)
18	`     index[j] = index[j+gap];`	75,419,253	2,654 (0%)	3,228	811 (25%)
19	`     index[j+gap] = e;`	43,096,716	0 (0%)	0	0 (0%)
20	`    }`				
21	`  fnum = fn;`	67	61 (91%)	73	61 (84%)
22	`}`				

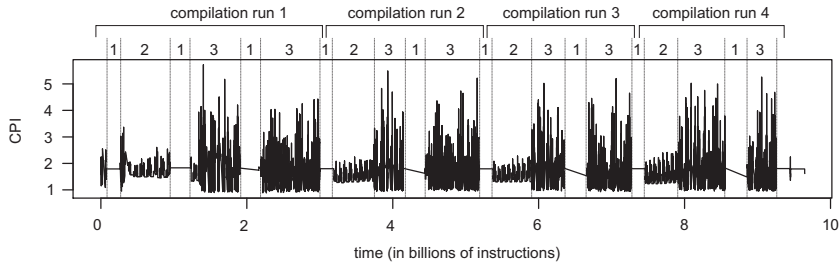
Table 1.2: The `shell_sort` method from db annotated with cache miss information computed using Javana.

chere. In Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '06), Portland, USA, October 2006.

1.3.2 Contribution 2: Offline profiling using hardware performance monitors

The second contribution is an offline profiling technique that uses hardware performance monitors and phase behavior of Java applications to gain more insight in the execution behavior of a Java application on real hardware. The goal is to identify method-level phase behavior and to use that information to correlate hardware performance characteristics directly to the source code of the application and the virtual machine.

The technique reads the hardware performance monitors (i.e., cycles per instruction, cache miss rates and branch misprediction rates) at the beginning and end of each execution phase, and links the microprocessor-level information to the source code of the Java application. The use of phase behavior in combination with hardware performance monitors allows us to collect profile information without the overhead that is inherent to a similar profiling tool built with a system like Javana.



Method / phase	Region	Time (%)	CPI	L1-D	L2-D	BMP
SourceClass.check	3	7.06	2.25	8.06	1.74	23.20
SourceClass.compileClass	3	26.03	1.74	5.14	0.98	16.26
Garbage collector	1	28.99	1.80	4.48	2.55	4.76
Parser.parseClass	2	22.68	1.48	2.93	0.54	18.26
Benchmark average	n/a	n/a	1.67	4.28	1.32	13.26

Figure 1.2: The graph and the table present example phases in a javac -s100 run. The time is given as a percentage of the total execution time. The L1 data cache miss rates, the L2 data cache miss rates and the branch misprediction rates are given as the number of events per 1,000 instructions.

We used this technique to build a prototype tool called MonitorMethod. The output of MonitorMethod helps answering three fundamental questions programmers might ask when optimizing their application: (i) what are the application's performance bottlenecks, (ii) why do these performance bottlenecks occur and (iii) when do the performance bottlenecks occur? To illustrate this point, consider the execution of the SPECjvm98 javac benchmark application. Figure 1.2 shows a graph produced by MonitorMethod that plots javac's cycles per instruction (CPI) over time when measured on an AMD Athlon XP system. The vertical separators group phases in regions with similar performance characteristics. Note that javac with the `-s100` input set compiles the same Java classes four times. Profile information captured at each context switch is used to aggregate all profiling data into a single graph. To answer the first question (what is the performance bottleneck?), we ordered the phases by their CPI values as shown in the table of Figure 1.2. Note that the table depicts only a small subset of all the phases in javac. Methods whose CPI is worse than the average CPI, are potential performance bottlenecks. To answer the second question (why does the performance bottleneck occur?), one can investigate the corresponding metrics such as cache miss rates and the branch misprediction rate, see the table in Figure 1.2. Finally, to answer the last question (when does the performance bottleneck occur?), one can use region information to relate phases to the time behavior of an application, see also Figure 1.2. This example illustrates that to understand how, why and when bottlenecks occur, it is valuable to link hardware performance monitors, captured at the lowest level of the execution stack, with the source code of a Java application.

By exploiting phase behavior we can capture the hardware performance monitors at an acceptable performance cost; the overhead of profiling javac was only 2.11%, compared to a slowdown of 125× and more that we observe for similar profiling tool built with Javana.

This work is described in[2]:

- *Method-level phase behavior in Java workloads*, Andy Georges, Dries Buytaert, Lieven Eeckhout and Koen De Bosschere. In Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04), Vancouver, British Columbia, Canada, October 2004.

[2]This was joint work with Andy Georges. While the main contribution of Andy was the statistical data analysis, my primary contribution was on the vertical instrumentation aspects of the overall system design. The work presented in this dissertation represents my contributions.

1.3.3 Contribution 3: Online profiling using hardware performance monitors

While the first and second contribution where offline techniques, the third contribution is an online profiling technique. Unlike the previous techniques that helped developers identify performance bottlenecks in their Java applications, this technique looks for methods that can be optimized by the VM during the execution of the program itself. Methods are first executed unoptimized and then an online profiling mechanism is used to find a subset of methods that should be optimized during that same execution.

The technique uses hardware performance monitors to identify methods that might be good candidates for optimization. Besides reading the counting hardware performance monitors at instrumentation points, as done by MonitorMethod, the hardware performance monitors can also be configured to generate an interrupt when a counter overflows. This interrupt can be converted to a signal that is delivered to the process using the hardware performance monitors, which can track which methods are being executed. In other words, the approach that we propose uses event-based sampling, not instrumentation, to identify the methods that can be optimized by the VM. This allows us to collect the information necessary to drive optimization decisions with extremely low overhead; typically the profiling overhead is no more than 0.2%.

We empirically evaluate the design space of several profilers for dynamic compilation and show that existing online profiling schemes suffer from several limitations. They provide an insufficient number of samples, are untimely, and have limited accuracy. Next, we describe and comprehensively evaluate HPM-sampling, a simple but effective profiling scheme for finding optimization candidates using hardware performance monitors that addresses the aforementioned limitations. We show that HPM-sampling is more accurate; has low overhead; and improves performance of existing Java applications by 5.7% on average and up to 18.3% when compared to results obtained with existing profilers.

This work is based on[3]:

- *Using HPM-Sampling to Drive Dynamic Compilation*, Dries Buytaert, Andy Georges, Michael Hind, Matthew Arnold, Lieven Eeckhout, Koen De Bosschere. In Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '07), Montreal, Canada, October 2007.

[3]The principal contributors to this work are Andy Georges and Michael Hind. Andy's main contribution is with helping to determine the accuracy and stability of the HPM-sampling approach. Michael Hind provided feedback on the technical aspects of this work and took the lead on writing the OOPSLA'07 paper.

1.4 Thesis outline

In Chapter 2, we present the first contribution, an offline profiling technique that uses binary instrumentation to build a profiling framework for creating customized Java program analysis tools. We show how this technique can be used to build a wide variety of profiling tools that accurately relate microprocessor information to Java source code.

In Chapter 3, we describe the second contribution, an offline profiling technique that uses hardware performance monitors instead of binary instrumentation to relate hardware performance monitors to the Java application's source code. We present a detailed case study showing that linking microprocessor-level performance characteristics to the source code is helpful for identifying performance bottlenecks.

In Chapter 4, we present the third contribution of this thesis, and study how the VM identifies methods that can be optimized at runtime. We evaluate the design space of existing online profiling schemes, and propose an online profiling mechanism that uses hardware performance monitors to identify methods that are optimization candidates.

In the last chapter, Chapter 5, a conclusion and future research directions are given.

Chapter 2

Offline profiling using binary instrumentation

This chapter describes Javana, a system for building customized Java program analysis tools. The Javana instrumentation framework provides the end user with both high-level and low-level information. The high-level information relates to the Java application and the VM, such as thread IDs, method IDs, source code line numbers, object IDs, object types, etc. The low-level information consists of instruction pointers and memory addresses. Running the Java application of interest within the Javana system along with user-specified instrumentation routines then collects the desired profiles of the Java application.

The Javana system consists of a VM along with a dynamic binary instrumentation tool that runs underneath the VM. The virtual machine communicates with the dynamic binary instrumentation tool through an *event handling* mechanism. The virtual machine informs the instrumentation layer about a number of events, for example when an object is created, moved or collected, or when a method gets compiled or recompiled, etc. The dynamic binary instrumentation tool then catches these events and subsequently builds a *vertical map* that links instruction pointers and memory addresses to high-level language concepts.

The dynamic binary instrumentation tool captures all natively executed machine instructions during a profiling run within Javana; this includes instructions executed in native functions called through the Java Native Interface (JNI). Instrumenting all natively executed machine instructions causes a substantial slowdown, however, it enables Javana to know for all native instructions from what method and thread the instruction comes and to what line of source code the instruction corresponds. Also, for all accessed memory locations, Javana knows what objects are

accessed.

As such, Javana enables the building of vertical profiling tools, i.e., profiling tools that crosscut the Java application, the VM and the native execution layers. Vertical profiling tools are invaluable for gaining insight into the overall performance and behavior of a Java application.

When looking at the lowest level of the execution stack, i.e., when looking at the individual instructions executed on the host machine, it is hard to understand the application's behavior because of the fact that the virtualization software gets intermixed with application code. However, when the goal is deep understanding of the application's behavior, the lowest level of the execution stack really is the level to look at. Vertical profiling enables gaining such insights and Javana makes vertical profiling easy to do. Building equally powerful profiling tools without Javana is both tedious and error-prone. Dynamic binary instrumentation underneath the virtual machine as done in Javana alleviates this issue.

In this chapter we demonstrate the power and accuracy of Javana through three applications.

Our first application is memory address tracing. A recent study published by Shuf et al. [74] analyzed the memory behavior of Java applications based on memory address traces. They instrumented the virtual machine to trace all heap accesses, but did not trace stack accesses. As we will show in this chapter, we found that on average 58% of all memory accesses in a Java application are non-heap accesses. Hence, not including non-heap accesses in a memory behavior analysis study may significantly skew the overall results. The Javana system captures all memory accesses and consequently is more accurate.

In our second application we built a vertical profiling tool for analyzing the cache behavior of Java applications. This cache performance profiling tool tracks cache miss rates per object type and per method and thus allows for quick computation of the top-most cache miss causing lines of code, the top-most cache miss causing object types, etc. This is invaluable information for an application developer who wants to optimize the memory performance of his software.

Our third application shows how easy it is to build an object lifetime analysis tool in Javana. Previous work [70] has shown that object lifetime is an important characteristic which can be used for analyzing and optimizing the memory behavior of Java applications. Computing an object's lifetime, although conceptually simple, is challenging in practice without Javana because the virtual machine needs to be adjusted in numerous ways in order to track all possible accesses to all objects, including accesses that occur through the Java Native Interface (JNI). This requires an in-depth understanding of the virtual machine. Computing object lifetime distributions with Javana on the other hand, is easy to set up and in

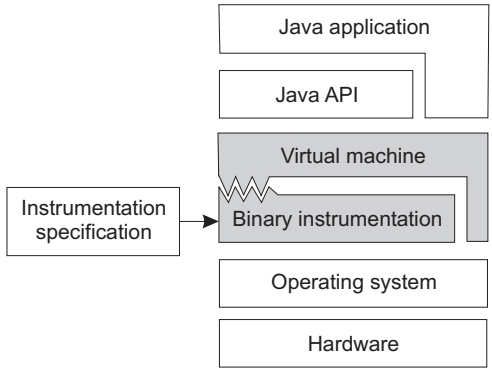


Figure 2.1: The Javana system for profiling Java applications.

addition, is guaranteed to deliver accurate object lifetimes.

2.1 The Javana system

Figure 2.1 illustrates the basic concept of the Javana system. The top of the execution stack shows a Java application that is to be profiled. The Java application together with a number of Java libraries runs on top of a virtual machine. The virtual machine translates Java bytecode instructions into native instructions. A dynamic binary instrumentation tool resides beneath the virtual machine and tracks all native instructions executed by the virtual machine.

The key point of the Javana system is that the virtual machine informs the dynamic binary instrumentation tool through an event handling mechanism whenever an object is created, moved, or deleted; or a method is compiled, or re-compiled; or a thread is created, switched or terminated, etc. The dynamic binary instrumentation tool then uses these events to build *vertical maps* that associate native instruction pointers and memory addresses with objects, methods, threads, etc. The dynamic binary instrumentation tool also intercepts all memory accesses during the execution of the Java application on the virtual machine. This includes instructions executed in native JNI functions, but excludes kernel-level system calls as will be discussed later. Using the vertical maps, the binary instrumentation tool associates native machine addresses to high-level concepts such as objects, methods, etc. This high-level information along with the low-level information is then made available to the end user through the Javana instrumentation framework.

The remainder of this section discusses the Javana system in more de-

tail. We discuss the events that are triggered by the virtual machine, the dynamic binary instrumentation layer in the Javana system, the event handling mechanism, the vertical instrumentation, the perturbation of the Javana system and finally our Javana proof-of-concept implementation. All of the subsections below give a general description of what the issues are for building a vertical profiling system; the final subsection then discusses our own proof-of-concept implementation.

2.1.1 Events triggered by the virtual machine

The Javana system requires that the virtual machine is instrumented to trigger events. These events communicate information between the virtual machine and the dynamic binary instrumentation tool. The Javana system supports the following events:

- Class loading: When a new class is loaded and a new object type becomes available, the new class name is communicated to the binary instrumentation tool.
- Object allocation: When a new object is allocated, the object's type and memory location (object starting address and its size) are communicated.
- Object relocation: When an object is moved by the garbage collector, the object's new location is communicated to the instrumentation tool.
- Method compilation: When a method is compiled, its name, memory location and a 'code to line number' map are communicated to the instrumentation tool.
- Method recompilation: When a method is recompiled, the method's location and 'code to line number' map are updated in the binary instrumentation tool.
- Method relocation: When code is moved by the garbage collector, the code's new location in memory is communicated.
- Memory freed during garbage collection: When memory is freed, the address range of the freed memory space is communicated to the binary instrumentation tool.
- Java thread creation: When a new Java thread is created, the thread's ID and name are communicated.
- Java thread switch: When a Java thread switch occurs, the newly scheduled Java thread's ID is communicated.

- Java thread termination: When a Java thread has ended execution, this is communicated to the dynamic binary instrumentation tool.
- Java thread stack switch: When a Java thread stack is relocated, the thread ID, the old stack location and the new stack location are communicated.

Note that this event list is just an example event list that can be tracked within a Javana system. Additional events can be defined and added to this list if desired. As we will show in Section 2.1.3, implementing events in a virtual machine is easy to do. However, we found that this list of events is sufficient for our purpose of building powerful Java program analysis tools, as will be shown in the remainder of this chapter.

2.1.2 Dynamic binary instrumentation

A dynamic binary instrumentation tool takes as input a binary and an instrumentation specification [56, 59, 60, 65, 66]. The binary is the program of interest; this is the Java application running in a virtual machine in our case. The instrumentation specification indicates what needs to be instrumented in the binary; it drives the customized profiling. The dynamic binary instrumentation tool then instruments the program of interest at run time. Upon the first execution of a given code fragment, the instrumentation tool reads the original code, modifies it according to the given instrumentation specification and stores the result as part of the instrumented binary. The instrumented version of the code is then executed and the desired profiling information is collected while executing the instrumented binary.

The data memory addresses referenced by the loads and stores in the instrumented binary are identical to those of the uninstrumented binary. By keeping the original binary in memory at its original address while generating the instrumented binary elsewhere, the instrumented binary obtains correct data values from the original uninstrumented binary in case data-in-code is read. The instrumentation tool also keeps track of correspondences between instruction pointers in the original binary versus the instrumented binary. By doing so, the instrumentation routines see instruction pointers and memory addresses as if they were generated during the execution of the original binary.

Running a dynamic binary instrumentation tool underneath a virtual machine requires that the instrumentation tool can deal with self-modifying code. The reason is that most virtual machines implement a dynamic optimizer that detects and (re-)optimizes frequently executed code fragments. A similar issue occurs when garbage is collected; copying collectors may copy code from one memory location to another. This

requires that the dynamic instrumentation tool invalidates the old code fragment and replaces it with an instrumented version of the newly generated code fragment.

Note that the dynamic binary instrumentation tool does not track kernel-level system calls. This limits the use of Javana to user-space instrumentation.

2.1.3 Event handling method

The virtual machine triggers events by calling empty functions; these empty functions are native C functions. The dynamic binary instrumentation tool intercepts such function calls and in response calls the appropriate event handlers. Event handlers can accept arguments because the arguments placed on the stack by the virtual machine are available to the binary instrumentation tool as well. For example, when allocating an object, the virtual machine calls the `AllocateObject` procedure with a number of arguments, namely the object type `t`, its address `m` and its size `s`. The dynamic binary instrumentation tool intercepts such events by inspecting the target addresses of the function calls. If the target address corresponds to the `AllocateObject` function in the above example — the dynamic binary instrumentation tool knows this function by name from the symbol information of the virtual machine — the dynamic binary instrumentation tool transfers control to the appropriate event handler which in turn reads the arguments from the stack and adds this information to its internal data structures. When the event handler has finished execution, control is transferred to the return address of the event's function call, i.e., the instrumented binary gets control again.

Event handling enables the dynamic instrumentation tool to build the *vertical map*. In the above example with the `AllocateObject` event, the event handler adds the following information to the vertical map: an object of type `t` is allocated in the memory address range `m` to `m+s`. Similar event handlers exist for all the events mentioned in Section 2.1.1.

The dynamic instrumentation tool captures *all* native instructions and memory accesses from both the application and the virtual machine during the execution of a Java application within Javana. The vertical map then enables the dynamic binary instrumentation tool to know for each memory access what object is being accessed and what the object's type is; and for every instruction pointer, the dynamic binary instrumentation tool knows to what method, to what line of source code and to what thread the instruction corresponds. As a result, Javana allows for easy tracking of *all* Java object accesses, which is much harder to do without a vertical map and dynamic binary instrumentation support.

2.1.4 Perturbation issues

An important property of any instrumentation framework is that the results that are obtained during profiling may not suffer from perturbation. The end user wants the instrumentation framework to be completely transparent, i.e., the instrumentation framework must not impact the results from profiling.

More in particular, in our Javana system, care needs to be taken that the profiling results are not perturbed by the event handling mechanism. Recall that the virtual machine triggers events by calling an empty method with a number of arguments. Computing the arguments, pushing them onto the stack, and finally calling the empty method introduces some overhead. Since the dynamic binary instrumentation tool instruments all natively executed instructions, the instructions executed for triggering an event in the virtual machine get instrumented as well. In order to avoid this issue, and to remove any perturbation due to the event handling mechanism, we communicate the address ranges of the virtual machine code for event triggering. The dynamic binary instrumentation tool knows that the code executed in these address ranges needs to be disregarded.

Another issue is that many virtual machines use timer-based sampling, i.e. to detect frequently executed methods that needs to be scheduled for optimization. This can be done by sampling the call stack at regular timer intervals; when the number of samples of a given method gets above a given threshold, the method is considered for optimization. We will discuss how methods are selected for optimization in more detail in Chapter 4. The Java thread scheduling also relies on the notion of real time. Java threads get time quanta for execution and when a time quantum has finished, another Java thread can be scheduled. Running a virtual machine within a Javana system causes the virtual machine to run slower, and by consequence, this affects timer-based virtual machine events such as code optimization and Java thread scheduling. While we haven't solved this problem, we believe it could be solved by using deterministic replay techniques [23].

2.2 The Javana language

A system for building customized Java program analysis tools also requires an easy-to-use instrumentation framework. The instrumentation framework is the environment in which the end user will build his profiling tools. For Javana we designed the Javana instrumentation language for building Java program profiling tools. The Javana instrumentation language is inspired by the Aspect-Oriented Programming (AOP)

paradigm because AOP matches the needs in instrumentation very well. A formal description of the Javana language can be found in our OOPSLA'06 paper [58]. The remainder of this chapter will provide several practical examples of the Javana language.

2.3 A proof-of-concept Javana system

The Javana system is a general framework for building customized Java program analysis tools. Any virtual machine could be employed in this framework and any dynamic binary instrumentation tool could be used as well. In our experimental framework, we use the Jikes RVM as our virtual machine and we use DIOTA as our dynamic binary instrumentation tool.

2.3.1 Jikes RVM

In May 1995, Sun Microsystems introduced the Java programming language. The technology quickly gained popularity and a few years later Java was already in wide use by many big players, including IBM.

In November 1997, a research group was founded at IBM Thomas J. Watson, with the goal to develop an internal research platform for VM technologies. The project, formally known as Jalapeño, strived to be flexible and extensible, so it could serve as a tool to research, prototype and evaluate VM implementation techniques. During that time, various aspects of Jalapeño were documented in research papers [2, 7, 20].

In October 2001, IBM decides to release the Jalapeño project as Open Source software, and to rename the software to *Jikes Research Virtual Machine*, or *Jikes RVM* for short. Ever since, there has been a growing pool of professors, students and researchers that use Jikes RVM for their work, and Jikes RVM established itself as one of the *de facto* research platforms in the Java world [3].

One of the remarkable properties about Jikes RVM is that it uses a 'compile only' strategy. Methods are never interpreted, but are always compiled to machine code just before they are about to be executed the first time (*lazy compilation* or *deferred compilation*). Compilation is done by a simple and fast baseline compiler that translates the Java bytecode to native machine code that emulates the Java operand stack. That is to say, the machine code implements a stack machine rather than taking full advantage of the underlying processor's register architecture.

Jikes RVM is shipped with an optimizing compiler. Just like the baseline compiler, the optimizing compiler gets Java bytecode as input, and generates native machine code as output. However, instead of fast and

straightforward compilation, the optimizing compiler translates the Java bytecode to an intermediate representation (IR). The various optimization phases in the compiler operate at the IR-level to perform optimizations. After the optimizations, the actual machine code is produced. (For completeness, we note that Jikes RVM actually uses three levels of IRs, and that each IR has its own format and set of available optimizations.)

The optimization phases implemented by Jikes RVM include, but are not limited to method inlining, eliminating null- and array checks, branch optimization, loop unrolling, SSA-based optimizations, linear register allocation, code placement, and much more. All of these optimization phases can be arranged in three groups and will also be executed group by group. These groups are: level 0, level 1 and level 2 [7, 8]. The higher the level, the more optimizations that will be performed or the more aggressive certain optimizations become.

Compiling and optimizing a method can be expensive, and because these optimizations have to be performed during the execution of the program, not all methods can be optimized. Optimizing all methods might make the execution of the program slower. Selecting what methods to optimize and what methods not to optimize is the task of the *adaptive optimization system* (AOS). The AOS consists of 3 components: (i) an online profiler that collects information during the execution of the program, (ii) a component that uses this profile information to decide what methods are good candidates for optimization, and (iii) the compiler.

Because Jikes RVM itself is used in many research projects around the world, it includes many of the latest developments in terms of runtime compilation, dynamic optimization, garbage collection, and so on. For example, Jikes RVM ships with at least five state-of-the-art garbage collectors and memory allocators [16]. This makes it easy to compare one strategy to another.

Furthermore, Jikes RVM is written almost entirely in the Java programming language; unlike most other JVMs that are written in C or C++. This has two key advantages: (i) it provides a high-level and strongly typed programming environment which makes it easier to prototype new ideas, and (ii) it enables the JVM to optimize itself, including its garbage collector and its optimizing compiler.

The combination of all the aspects above, makes Jikes RVM a great research platform. For those reasons, we have chosen to use Jikes RVM for our research. Throughout this work, we'll highlight more aspects of the Jikes RVM where appropriate.

Relevant to the work in this dissertation is also Jikes RVM's threading mechanism. The threading system multiplexes n Java threads (application and VM) onto m native (kernel) threads that are scheduled by the operating system. A command line option specifies the number of ker-

nel threads that are created by the Jikes RVM. Usually, there is one kernel thread used for each physical processor, also referred to as a virtual processor because multiple Java threads can be scheduled by the VM within the single kernel thread. In our setup, we have used a single virtual processor.

For our experiments with Javana we use the *FastAdaptive* configuration: all methods are initially compiled using a baseline compiler, and hot methods are recompiled using an optimizing compiler.

Making the Jikes RVM Javana-enabled was easy. We only had to insert around two hundred lines of code (including comments) into the virtual machine in order to trigger the events intercepted by the dynamic binary instrumentation tool. More specifically, we added an event to the class loader, to the object allocator, to all garbage collectors when an object or code is being moved or deleted, to all compilers and optimizers when a method is being compiled or optimized, and to the thread management system when a thread is created, switched or terminated.

Because the Jikes RVM itself is written in Java, there are also some peculiarities. One such peculiarity is with instrumenting this VM as done with Javana. Instrumentation cannot be activated until the virtual machine is properly booted. This means that there are some virtual machine methods and objects that cannot be communicated to the binary instrumentation tool during virtual machine startup. This can be solved by communicating these virtual machine methods and objects as soon as the virtual machine is properly booted. From then on, the instrumentation tool intercepts all method calls and object accesses during the program execution.

2.3.2 DIOTA

The dynamic binary instrumentation tool that we use in our proof-of-concept Javana system is DIOTA [60]. DIOTA stands for Dynamic Instrumentation, Optimization and Transformation of Applications and is a dynamic binary instrumentation framework for use on the Linux operating system running on x86-compatible processors. Its functionality includes intercepting memory operations, code execution, signals, system calls and functions based on their name or address, as well as the ability to instrument self-modifying code [59].

DIOTA is implemented as a dynamic shared library that can be hooked up to any program. The main library of DIOTA contains a generic dynamic binary instrumentation infrastructure. This generic instrumentation framework can be used by so-called backends that specify the particular instrumentation of interest that needs to be done. The backend that we use for Javana is a memory operation tracing backend, i.e., this

backend instruments all memory operations.

The general operation of DIOTA is very similar to that of other dynamic binary instrumentation frameworks such as PIN [56] and Valgrind [65, 66]. All of these operate in a similar way as described in Section 2.1.2.

2.3.3 Java applications

We use the SPECjvm98, SPECjbb2000 and DaCapo benchmark suites in this dissertation. An overview is given in Table 2.1.

Suite	Benchmark	Description
SPECjvm98	compress	Compresses a number of input files using an LZW method.
	jess	Solves a number of puzzles with varying degrees of complexity.
	db	Performs a set of database requests on a memory resident database.
	javac	Compiles a number of Java files.
	mpegaudio	Decompresses MPEG Layer-3 audio files.
	mtrt	Renders a scene using ray tracing.
	jack	Parses grammar files and generates a parser for each.
DaCapo	antlr	Parses grammar files and generates a parser and lexical analyzer for each.
	fop	Takes an XSL-FO file, parses it and formats it, generating a PDF file.
	hsqldb	Executes a number of transactions against a memory resident database.
	jython	Interprets a series of Python programs.
	ps	Reads and interprets a PostScript file.
	xalan	Transforms XML documents into HTML.
SPECjbb2000	PseudoJBB	Performs transactions on a three-tier transaction system.

Table 2.1: The benchmarks used in this PhD dissertation.

SPECjvm98 [77] is a client-side Java benchmark suite consisting of

seven benchmarks. For each of these, SPECjvm98 provides three inputs sets: s1, s10 and s100. Contradictory to what their names suggest, the size of the input set does not increase exponentially. For some benchmarks, a larger input indeed increases the problem size. For other benchmarks, a larger input executes a smaller input multiple times. SPECjvm98 was designed to evaluate combined hardware (CPU, caches, memory, etc.) and software aspects (virtual machine, kernel activity, etc.) of a Java environment. However, they do not include graphics, networking or AWT (window management).

The DaCapo [17, 84] suite is a relatively new benchmark suite. The DaCapo benchmark suite was designed for research on memory management studies. In the complete suite there are 10 benchmarks, of which we use six. The others did not run reliably under the Jikes RVM versions that we used.

SPECjbb2000 (Java Business Benchmark) [76] is a server-side benchmark suite focusing on the middle-tier, the business logic, of a three-tier system. We have used a modified version of this benchmark, known as PseudoJBB, which executes a fixed number of transactions, instead of running for a predetermined period of time. The benchmark was run with 8 warehouses.

These benchmarks are the *de facto* benchmarks in today's Java research. No distributed benchmark applications were used; all benchmarks were run on a single uniprocessor machine.

For our work on Javana, we run all SPECjvm98 benchmarks with the largest input set (`-s100`). All of the SPECjvm98 benchmarks are run on the Jikes RVM using a 64 MB heap and the generational mark-sweep (GenMS) garbage collector; the DaCapo and PseudoJBB benchmarks are run with a 500 MB heap. For the DaCapo benchmark suite we used release version beta050224.

2.3.4 Hardware platform

Our measurements are done on a 2.8 Ghz Intel Pentium 4 system with a 512 KB L2 cache and 1 GB main memory. The operating system on which we run our experiments is Gentoo Linux with a 2.6.10 kernel.

2.3.5 Javana overhead analysis

Running a Java application within Javana obviously introduces overhead. There are a number of contributors to the overall overhead:

- First, the dynamic binary instrumentation tool that runs underneath the virtual machine causes overhead independent of the instrumen-

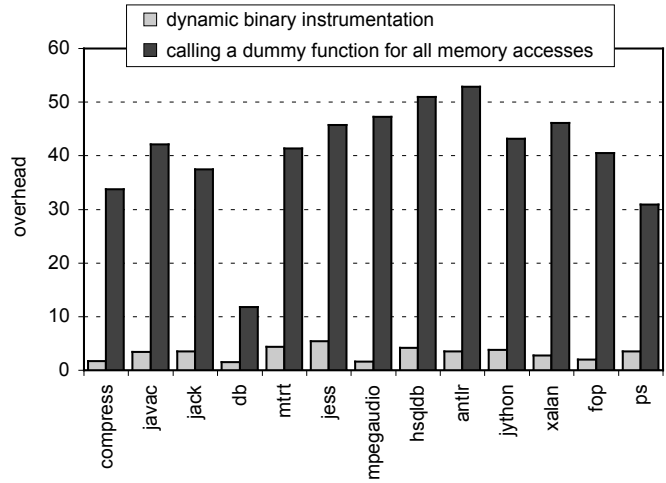


Figure 2.2: The slowdown due to dynamic binary instrumentation.

tation specification provided by the end user of the Javana system.

- Second, the event handling mechanism that communicates high-level language concepts from the virtual machine to the dynamic binary instrumentation tool also introduces overhead. In addition, the event handler needs to process this information for updating the vertical map in the dynamic binary instrumentation tool.
- Third, executing instrumented code requires that the binary instrumentation tool searches the vertical map for every memory location accessed.
- And finally, executing the instrumentation code itself as implemented by the end user of the Javana system also causes additional overhead.

We will now quantify the overhead caused by each of these four overhead contributors.

Dynamic binary instrumentation overhead

We first quantify the overhead of the binary instrumentation (bullet one from above). There are two contributors to this overhead. First, whenever control is transferred, the binary instrumentation engine needs to perform an address look-up. The overhead that we observe for DIOTA in our Javana system ranges from 1.5× to 5.5×, see Figure 2.2.

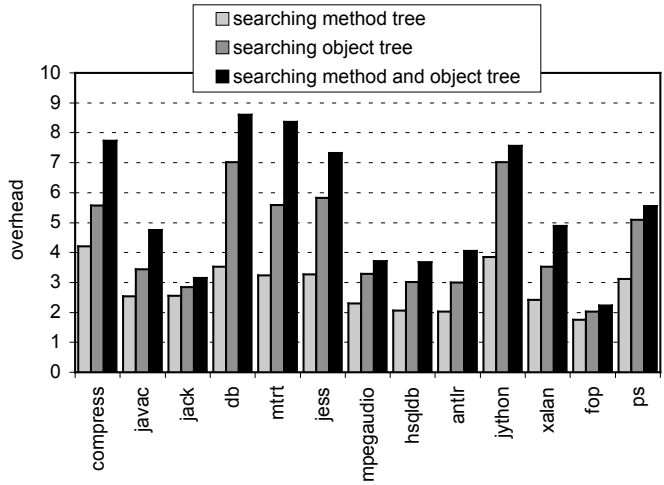


Figure 2.3: The slowdown due to vertical instrumentation.

The second contributor is due to calling an instrumentation routine for all natively executed memory operations. We quantify this overhead by calling a dummy (empty) function for each memory operation. The overhead varies between a factor 12× and 53× depending on the benchmark, see Figure 2.2. This overhead is inherent to dynamic binary instrumentation. Most of this overhead comes from saving the caller-saved registers and preparing the parameters for the dummy function call. One of the parameters is a data structure that contains the thread ID, the instruction pointer, the type of memory access and the number of bytes accessed. This data structure needs to be constructed for every memory access.

Vertical instrumentation overhead

We now quantify the overhead caused by the event handling mechanism and by searching the vertical map for every memory location accessed (bullets two and three from above). We collectively refer to this overhead as vertical instrumentation overhead, i.e., this is the overhead that enables cross-layer instrumentation. In our experiments we observed that the event handling mechanism is only a very small part of the total vertical instrumentation overhead.

Figure 2.3 quantifies the overhead from vertical instrumentation.

- The first bar for each benchmark shows the overhead for the Javana system during a vertical profiling run that only considers method-

related information.

- The second bar depicts the overhead when enabling vertical profiling for measuring object-related information.
- The third bar shows the overhead when both the method- and object-related information is captured.

The average vertical instrumentation overhead varies between a factor 2.8× and 5.5× depending on what information is to be kept track of.

Overall overhead

From the above enumeration, it follows that the total slowdown of a Java program analysis tool built within Javana equals the product of the dynamic binary instrumentation slowdown, the vertical instrumentation slowdown and the slowdown due to the user-defined instrumentation routines, i.e., the advice code included in the instrumentation specification.

The total slowdown for the dynamic binary instrumentation and the vertical instrumentation varies between a factor 90× and 345×. This is the overhead caused by using Javana. The additional overhead due to the instrumentation routines, increases the overall overhead to the 125×-850× range; this is for the vertical cache simulation which is the most demanding vertical profiling tool that we built with Javana.

According to our experience, this is an acceptable slowdown. Compared to simulation, Javana is fast; simulation typically causes a slowdown by at least a factor 10,000× [19]. In cases where a 90× to 345× slowdown is undesirable, sampling can be employed to reduce this slowdown. However, this comes at the price of accuracy; our measurements were done without applying any sampling.

2.4 Applications of Javana

We now discuss three example applications of the Javana system: memory address trace generation, vertical cache simulation and object lifetime computation. These applications demonstrate the real power of Javana: Javana provides an easy-to-use instrumentation environment that allows for quickly building customized (vertical) Java program analysis tools. The key benefit is that easy-to-build program analysis tools increase a software designer's productivity. In addition, the results show that we need to look at profiling techniques that combine information captured at different layers of the execution stack.

```
1:  before any:access
      (location_t const *loc, type_t const *type,
        void **userdata) {
2:    printf ("access by insn @ %p to memory location
        %p of size %d\n", loc->ma->ip, loc->ma->addr,
        loc->ma->size);
3:  }
```

Figure 2.4: The code for the memory address tracing tool in Javana.

2.4.1 Memory address trace generation

Our first application is memory access tracing; the instrumentation specification for building this profiling tool is shown in Figure 2.4. This profiling tool captures all memory accesses during program execution and writes each access' instruction pointer, memory address and size to a file. The procedure shown in Figure 2.4 is called before each memory access. The first parameter is a data structure that collects information concerning the 'location' of the object or memory event. The `ma` field in this structure is a pointer to a `mem_access_t` structure that contains (i) the instruction pointer of the native instruction performing the memory access, (ii) the object's memory location or in case of a memory operation, the memory location being accessed, (iii) the size of the object or in case of a memory operation, the number of bytes accessed in memory, (iv) whether this memory access is a load or store operation, and (v) the thread ID of the thread performing the object or memory operation. The second parameter in the parameter list is a pointer to a data structure that specifies information concerning the 'type' of the object or memory operation. The third parameter in the parameter list (`void **`*userdata*) allows the end user to maintain object-specific information. As can be seen, the Javana instrumentation language only requires three lines of code for building this profiling tool. In other words, the expressiveness of the Javana language is high while the code itself is very intuitive.

Recent work done by Shuf et al. [74] analyzed the memory behavior of Java applications. For doing so, Shuf et al. modified the virtual machine to trace all accesses to the heap, however, they did not trace accesses to the stack — presumably because it is very difficult to track all memory accesses including stack accesses by instrumenting the virtual machine. Using Javana we built a profiling tool that traces all heap memory accesses and all stack memory accesses. We found that on average only 42% of all memory accesses are heap accesses, see Figure 2.5 which shows the fraction heap accesses compared to the total number of data memory accesses. In other words, Shuf et al. captured only 42% of the

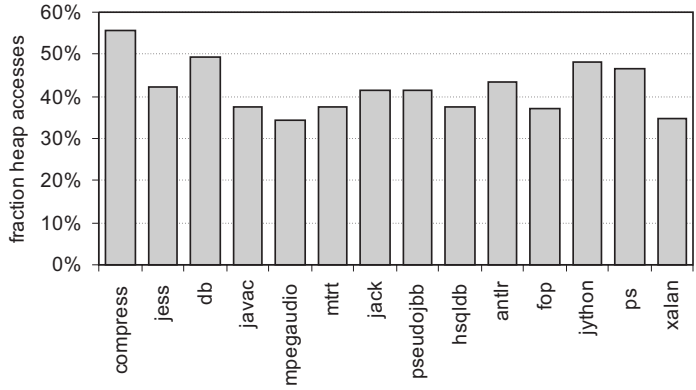


Figure 2.5: The fraction of all memory accesses that are heap accesses.

total number of memory accesses on average. Consequently, not capturing the large fraction of non-heap accesses has a significant impact on the observed memory system behavior.

Figure 2.6 shows the fraction L1 and L2 misses as a ratio to the total number of memory accesses. These results are for a simulated 4-way set-associative 32 KB 32-byte line L1 cache and an 8-way set-associative 1 MB 128-byte line L2 cache. Both are write-back, write-allocate caches. The cache simulation routines were taken from the SimpleScalar Tool Set [19]. The perturbation introduced by Javana should not significantly affect the results; all measurements – including the measurements to reproduce Shuf's results – were done using Javana. We observe that only considering heap accesses results in a severe overestimation of the actual cache miss rates. The difference in miss rates varies by a factor 1.8 and 2.9 between tracking heap accesses versus tracking all memory accesses. Therefore, we conclude that a methodology that analyzes heap accesses in a memory performance study, is questionable.

2.4.2 Vertical cache simulation

The second application relates cache miss rates to high-level concepts such as methods, source code lines, objects and object types. This is invaluable information for software developers that are in the process of optimizing their code for memory performance. As is well known, the memory-processor speed gap is an important issue in current computer systems. Poor memory behavior can severely affect overall performance. As such, it is very important to optimize memory performance as much as

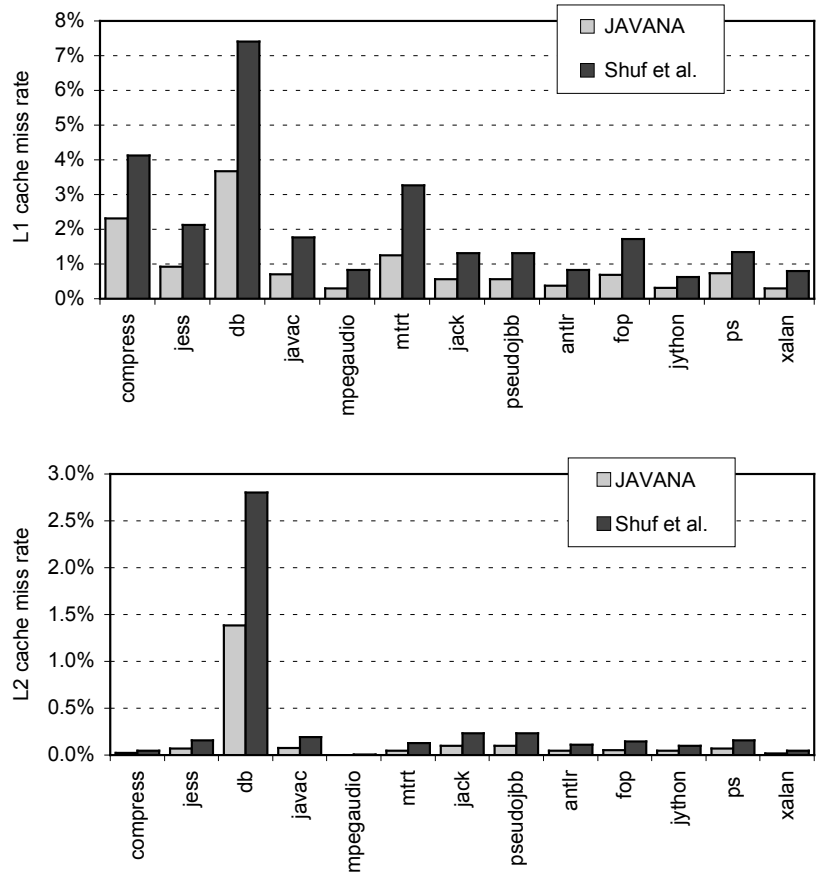


Figure 2.6: The cache miss rates using Javana versus Shuf et al.'s methodology: L1 data cache miss rates (number of L1 data cache misses divided by the number of L1 data accesses) at the top and global L2 data cache miss rates (number of L2 data misses divided by the number of L1 data accesses) at the bottom.

```
0:  #pragma requires object_info
1:  #pragma requires method_info

2:  before object:access (location_t const *loc, type_t const *type, void **userdata) {
      /* compute whether this object reference is a cache miss or not */
3:    hit = simulate_memory_access (loc->ma->addr, type->type_ID);
      /* update the per-type hit/miss information */
4:    update_per_type_miss_rate (type->type_ID, hit);
5:    update_per_method_miss_rate (loc->method_name, loc->line_number);
6:  }

7:  before nonobject:access (location_t const *loc, type_t const *type, void **userdata) {
      /* update the simulated cache content */
8:    simulate_memory_access (loc->ma->addr, -1);
9:    update_per_method_miss_rate (loc->method_name, loc->line_number);
10:  }
```

Figure 2.7: The code for the vertical cache simulation tool in Javana.

```
0:  #pragma requires object_info

1:  typedef {
2:    unsigned long long creation_time;
3:    unsigned long long last_access;
4:  } object_info_t;

5:  static unsigned long long timestamp = 0;

6:  after object:create (location_t const *loc, type_t const *type, void  **userdata) {
7:    object_info_t ** const objectinfo = (object_info_t**)userdata;

8:    (*objectinfo) = diota_malloc(sizeof(object_info_t));
9:    (*objectinfo)->creation_time = timestamp;
10:   (*objectinfo)->last_access = 0;
11: }
10cm
12: before object:access (location_t const *loc, type_t const *type, void  **userdata) {
13:   object_info_t ** const objectinfo = (object_info_t**)userdata;

14:   timestamp++;
15:   (*objectinfo)->last_access = timestamp;
16: }

17: before nonobject:access (location_t const *loc, type_t const *type,  void **userdata) {
18:   timestamp++;
19: }
```

Figure 2.8: The code for object lifetime computation tool in Javana.

Method	Accesses (%)	DL1 misses	DL2 misses
compress			
`Compressor.compress()V`	42.7	130,906,173 (7.8%)	533,099 (0.4%)
`Decompressor.decompress()V`	42.7	21,390,490 (1.3%)	485,995 (2%)
`Input_Buffer.readbytes([BI)I`	1.8	247,545 (0.3%)	55,151 (18.4%)
`Compressor.output(I)V`	4.5	207,706 (0.1%)	35,700 (14%)
`Output_Buffer.putbyte(B)V`	0.9	125,551 (0.3%)	25,169 (17.2%)
db			
`Database.shell_sort(I)V`	59.8	132,442,434 (10.5%)	50,720,398 (31.6%)
`Entry.equals(Ljava/lang/Object;)Z`	3.5	3,413,385 (4.6%)	1,720,280 (36%)
`Database.set_index()V`	6	3,924,453 (3.1%)	1,345,881 (28.7%)
`Database.read_db(Ljava/lang/String;)V`	0.8	36,682 (0.2%)	9,152 (13%)
`spec.io.FileInputStream.read()I`	0	5,078 (0.7%)	3,927 (60%)
mtrt			
`OctNode.FindTreeNode(LPoint;)LOctNode;`	11.8	27,446,406 (12.1%)	463,179 (1.5%)
`PolyTypeObj.Intersect(LRay;LIntersectPt;)Z`	3.9	1,718,595 (2.3%)	184,134 (9.9%)
`Vector.<init>(FFF)V`	0.3	715,338 (13.7%)	177,453 (22%)
`OctNode.Intersect(LRay;LPoint;F)LOctNode;`	16.4	146,3021 (0.5%)	145,254 (8.9%)
`Face.GetVert(I)LPoint;`	14.3	9,920,823 (3.6%)	113,315 (1%)

Table 2.2: The top 5 methods for some of the benchmarks sorted by the number of L2 cache misses.

Type	Accesses (%)	DL1 misses	DL2 misses
compress			
[B	20.6	13,001,417 (1.6%)	1,038,857 (7.1%)
[I	9.2	100,254,792 (27.8%)	56,465 (0%)
[S	4.7	41,699,357 (22.6%)	54,699 (0.1%)
[Ljava/lang/Object;	0.4	4,010 (0%)	406 (8.8%)
[[I	0.1	1,632 (0%)	368 (19.6%)
db			
Ljava/util/Vector;	15.3	36,375,367 (11.2%)	17,288,803 (38.8%)
[Ljava/lang/Object;	7.2	24,118,101 (15.7%)	11,838,596 (41.3%)
[C	11.7	22,697,725 (9.1%)	11,598,229 (42.4%)
LEntry;	4.1	2,8511,936 (32.6%)	7,941,652 (22.7%)
Ljava/lang/String;	13	22,717,143 (8.3%)	4,348,649 (15.6%)
mtrt			
LVector;	5.5	3,968,361 (3.7%)	554,763 (12.7%)
LPoint;	10.6	15,020,935 (7.4%)	358,453 (2.1%)
[LPoint;	3.5	10,720,458 (16%)	114,210 (1%)
[I	4.7	1,055,491 (1.2%)	97,335 (8.4%)
LFace;	3.3	6,796,479 (10.7%)	82,116 (1.1%)

Table 2.3: The top 5 objects types for some of the benchmarks sorted by the number of L2 cache misses.

Source code		DL1 accesses	DL1 misses	DL2 accesses	DL2 misses
1	`void shell_sort(int fn) {`				
2	`  int i, j, n, gap;`				
3	`  String s1, s2;`				
4	`  Entry e;`				
5					
6	`  if (index == null) set_index();`	67	0 (0%)	0	0 (0%)
7	`  n = index.length;`	134	1 (0%)	1	0 (0%)
8					
9	`  for (gap = n/2; gap > 0; gap/=2)`	938	0 (0%)	0	(0%)
10	`   for (i = gap; i < n; i++)`	12,276,499	910 (0%)	1,083	3 (0%)
11	`    for (j = i-gap; j >=0; j-=gap) {`	23,064,743	8,179 (0%)	9,615	33 (0%)
12	`     s1 = (String)index[j].items.elementAt(fn);`	157,553,557	29,772,665 (19%)	36,551,726	6,095,594 (17%)
13	`     s2 = (String)index[j+gap].items.elementAt(fn);`	157,553,557	24,036,992 (15%)	29,456,752	15,581,062 (53%)
14					
15	`     if (s1.compareTo(s2) <= 0) break;`	45,015,302	128 (0%)	153	1 (0%)
16					
17	`     e = index[j];`	32,322,537	219 (0%)	228	0 (0%)
18	`     index[j] = index[j+gap];`	75,419,253	2,654 (0%)	3,228	811 (25%)
19	`     index[j+gap] = e;`	43,096,716	0 (0%)	0	0 (0%)
20	`    }`				
21	`  fnum = fn;`	67	61 (91%)	73	61 (84%)
22	`}`				

Table 2.4: The `shell_sort` method from db annotated with cache miss information.

possible. Vertical profiling is a very valuable tool for hinting the software developer what to focus on when optimizing the application's memory behavior.

Vertical cache simulation requires that an instrumentation specification be written as shown in Figure 2.7. Lines 0 and 1 specify that the instrumentation needs to keep track of both per-object and per-method information. Upon a memory access to an object (lines 2-6), the memory address is used by the cache simulator to update the cache's state. The type-specific and method-specific data structures maintained by the instrumentation tool are updated to keep track of the per-type and per-method miss rates. Other memory accesses, i.e., to non-objects (lines 7-10), update the cache's state and update the per-method miss rate information. The per-type miss rate information is not updated because these memory references do not originate from object accesses.

The instrumentation specification for this profiling tool was no more than 200 lines of code, including comments. The output of the profiling run is a table describing cache miss rates per method, per line of code, per object and per object type.

Selecting the per-method and per-object type cache miss rates and sorting them by decreasing number of L2 misses results in Tables 2.2 and 2.3. In both tables we limit the number of methods and object types to the top five per benchmark in order not to overload the tables. The first column in each table mentions the method or object type, respectively. The second column shows the percentage memory references of the given method or object type as a percentage of the total number of memory references. The two rightmost columns show the number of L1 and L2 misses, respectively, along with the percentage local miss rates, i.e., the number of misses divided by the number of accesses to the given cache level. Results for benchmarks not listed in Tables 2.2 and 2.3 are available in [58].

Software developers can use these tables to better understand the memory behavior of their software for guiding memory optimizations at the source code level. For example, from Table 2.2 it is apparent that the `shell_sort` method in db is a method that suffers heavily from poor cache performance. About 60% of the memory references in db occur within the `shell_sort` method. Of these memory references, 10.5% result in an L1 cache miss, and 31.7% of the L2 cache accesses are cache misses. As such, this method is definitely a method of concern to a software developer when optimizing the memory performance of db.

Table 2.3 shows per-object type miss rates for the various benchmarks. The poor cache behavior for db seems to be apparent across a number of object types. For example, this table shows that the cache behavior for the `Vector` class is relatively poor with an L1 cache miss rate of 11.4% and an L2 miss rate of 38.6%. Note that our framework also allows for going

yet one step further, namely to tracking down miss rates to individual objects. This allows the software developer to isolate the source of the poor memory behavior. We do not include an example of per-object miss rates here in this chapter, however, this could be easily done in Javana.

Because the `shell_sort` method in db seems to suffer the most from poor cache behavior, we focus on that method now. Table 2.4 shows the `shell_sort` method annotated with cache miss information, i.e., L1 and L2 cache miss rates are annotated to each line of source code. Often there are more DL2 accesses then there are DL1 misses; this is explained by the fact that we used write-back, write-allocate caches. Line 13 seems to be the primary source for the high cache miss rate in the `shell_sort` method. The reason is that the `j+gap` index results in a fairly random access pattern into the 61 KB `index` array. It is interesting to note that Hauswirth et al. [45] also identified the `shell_sort` method as a critical method for db.

The number of L1 and L2 misses in Table 2.4 differ from the numbers given Table 2.2; the reason is that the numbers in this table were obtained using the baseline compiler whereas the numbers in Table 2.2 were obtained using the adaptive compiler; the line numbers returned by the adaptive compiler in Jikes are inaccurate due to inlining effects.

2.4.3 Object lifetime

Our third example application computes object lifetimes. In this application we define the object lifetime as the number of memory accesses between the creation and the last use of an object. Knowing the allocation site and knowing where the object was last used can help a programmer to rewrite the code in order to reduce the memory usage of the application or even improve overall performance [70].

Computing object lifetimes without Javana is fairly complicated. First, the virtual machine needs to be extended in order to store the per-object lifetime information. Second, special care needs to be taken so that the computed lifetimes do not get perturbed by the instrumentation code. Finally, all object references need to be traced. This is far from trivial to implement. For example, referencing the object's header is required for accessing the Type Information Block (TIB) or vertical lookup table (vtable) on a method call, for knowing the object's type, for knowing the array's length, etc. Also, accesses to objects in native methods need to be instrumented manually. Implementing all of this in a virtual machine is time consuming, error-prone and will likely be incomplete.

Measuring the object lifetime within Javana on the other hand is easy to do and in addition, it is accurate because it allows for tracking *all* references to a given object. In a Javana instrumentation specification, an

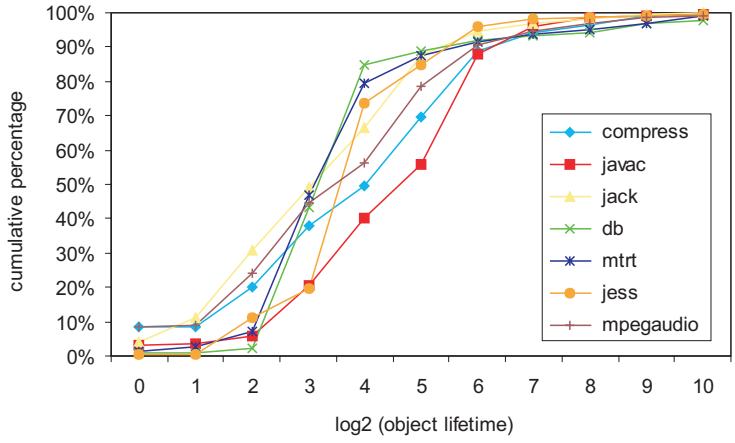


Figure 2.9: The cumulative object lifetime distribution for the SPECjvm98 benchmarks.

object's lifetime can be computed and stored using the per-object `void **userdata` parameter that is available in Javana language. For details, see [58]. As such, computing object lifetimes is straightforward to do in Javana — no more than 50 lines of code. The skeleton of the instrumentation specification is shown in Figure 2.8.

Figure 2.9 shows the lifetime distribution for the SPECjvm98 benchmarks computed using the Javana system. The horizontal axis on these graphs is given on a log-scale; the vertical axis shows the cumulative percentage objects in the given lifetime bucket. We observe that the object lifetimes are fairly small in general, i.e., most objects are short-lived objects. For most benchmarks, the object lifetime typically is smaller than 16 data memory accesses between the creation of an object and its last use. Some benchmarks have a relatively larger object lifetime, see for example javac, compress and mpegaudio, however the object lifetime is still very small in absolute terms, i.e., the object lifetime is rarely more than 64 data memory accesses.

In order to evaluate the accuracy of object lifetime computations without Javana, we have set up the following experiment. We compute the object lifetimes under two scenarios. The first scenario computes the object lifetime when taking into account all memory accesses as done using out-of-the-box Javana. The second scenario computes the object lifetime while excluding all object accesses from non-Java code; this excludes all the object accesses from native JNI functions. This second scenario emulates current practice of building an object lifetime measurement tool

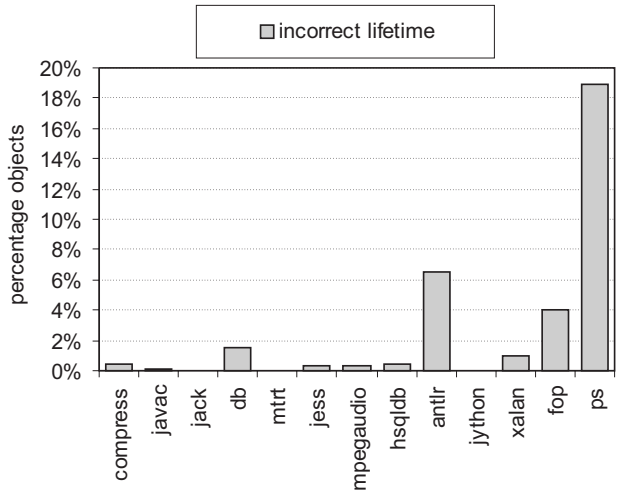


Figure 2.10: The accuracy of object lifetime computations without Javana: the percentage objects for which a non-Javana instrumentation results in incorrect lifetime computations.

within the virtual machine, without Javana. The results are shown in Figure 2.10. The graph shows the percentage of objects for which an incorrect lifetime is computed in current practice, i.e., when not including accesses to objects through JNI functions. We observe large error percentages for a couple of benchmarks, namely fop (4%), antlr (6.5%) and ps (19%). We conclude that current practice of computing object lifetime without Javana can yield incorrect results, and this could be misleading when optimizing the code based on these measurements.

2.5 Related work

We now discuss related work. We first discuss bytecode-level instrumentation tools. Next, we detail on existing vertical profiling approaches using hardware performance counters.

2.5.1 Bytecode-level profiling

A number of Java bytecode-level profiling tools have been presented in the recent literature. These bytecode-level profiling tools differ from the Javana system in that Javana allows for building vertical profiling tools, whereas bytecode-level profiling tools are limited by the fact that they

can't instrument the JVM or any of the native libraries. We discuss two bytecode-level profiling tools.

First, Dufour et al. [32] studied the dynamic behavior of Java applications in an architecture-independent way. To do so, they built a tool called *J [33] that uses the Java Virtual Machine Profiling Interface (JVMPI) to collect a wide set of bytecode-level Java program characteristics. The Java metrics that they collect are related to program size and structure, the occurrence of various data structures (such as arrays, pointers, etc.), polymorphism, memory usage, concurrency and synchronization.

Second, Dmitriev [28] presents a Java bytecode-level profiling tool called JFluid. JFluid can be attached to a running Java application. The attached JFluid then injects instrumentation bytecodes into the methods of the running Java program. The instrumentation bytecodes collect profiling information online.

2.5.2 Vertically profiling Java applications

Some very recent work focused on vertical profiling of Java applications. The purpose of these approaches is to link microprocessor performance to the Java application and the virtual machine. However, they do not allow for building customized vertical profiling tools.

Hauswirth et al. [45] and the earlier work by Sweeney et al. [82] presented a vertical profiling approach that correlates hardware performance counter values to manually inserted software monitors in order to keep track of the program's execution across all layers. The low-level and high-level information is collected at a fairly coarse granularity, i.e., hardware performance counter values and software monitor values are measured at every thread switch. Hauswirth et al. measure various hardware performance metrics during multiple runs yielding multiple traces. And because of non-determinism during the execution, these traces subsequently need to be aligned. Although being much faster than Javana, there are two important limitations with this approach. First, aligning traces is challenging and caution is required in order not to get out of sync [44]. Second, the granularity is very coarse-grained — one performance number per thread switch. This allows for analyzing coarse-grained performance variations but does not allow for analyzing the fine-grained performance issues we target with Javana.

In Chapter 3 of this thesis we will discuss a form of vertical profiling that links microprocessor-level metrics obtained from hardware performance counters to method-level phases in Java. This allows for analyzing Java applications at a finer granularity than the vertical profiling approach by Hauswirth et al. [44, 45], however, the granularity is still much more coarse-grained than the granularity that we can achieve using Ja-

vana.

VTune, a commercially available tool from Intel, also allows for profiling Java applications [29]. VTune samples hardware performance counters to profile an application and to annotate source code with cache miss rate information. However, given the fact that VTune relies on sampling it is questionable whether this allows for fine-grained profiling information with little overhead and perturbation of the results.

All of these vertical profiling approaches rely on a microprocessor's performance counters. This limits the scope of these techniques to evaluating Java system performance on existing microprocessors. These approaches do not allow for building customized vertical Java program analysis tools as the Javana system does.

2.6 Conclusion

Understanding the behavior of Java application is non-trivial because of the tight entanglement of the application and the virtual machine at the lowest machine-code level. This chapter presented Javana, a system for quickly building Java program analysis tools. Javana runs a dynamic binary instrumentation tool underneath a virtual machine. The virtual machine communicates with the dynamic binary instrumentation tool using an event handling mechanism. This event handling mechanism enables the dynamic binary instrumentation layer to build a so called vertical map. The vertical map keeps track of correspondences between high-level language concepts such as objects, methods, threads, etc, and low-level native instruction pointers and memory addresses. This vertical map provides the Javana end user with high-level information concerning every memory access the dynamic binary instrumentation tool intercepts. As a result, Javana is capable of tracking all memory references and all natively executed instructions and to provide high-level information for each of those.

Javana also comes with an easy-to-use Javana instrumentation language. The Javana language provides the Javana user with low-level and high-level information that enables the Javana user to quickly build powerful Java program analysis tools that crosscut the Java application, the VM and the native execution layer.

The first key property of Javana is that Java program analysis tools can be built very quickly. To demonstrate the real power of Javana we presented three example applications: memory address tracing, vertical cache simulation and object lifetime computation. For each of these applications, the core instrumentation specification was only a few lines of code.

The second key property of Javana is that the profiling results are guaranteed to be highly accurate (by construction) because the dynamic binary instrumentation layer tracks every single natively executed instruction. Current practice is typically one of manually instrumenting the virtual machine which is both time-consuming and error-prone. In addition, the accuracy of the profiling results might be questionable because it is hard to instrument a virtual machine in such a way that all memory accesses are tracked, as we have shown through our example applications.

Last but not least, this illustrates that it can be difficult to get deep understanding of the behavior of Java applications and that there is a lot of value in tools that gather more complete profiles or that look at different layers of the execution stack.

Chapter 3

Offline profiling using hardware performance monitors

As shown in Chapter 2, it is useful to look at the behavior of Java applications at the micro-architectural level and to link that information back to the source code of the Java application. While Javana is a great tool for building a wide variety of vertical profiling tools with minimal effort, it suffers from one major drawback that is inherent to the use of a dynamic binary instrumentation tool: it is slow. It typically has a slowdown factor of several hundreds.

In this chapter, we present a technique that measures the performance characteristics of the VM and the application using the hardware performance monitors (HPMs) rather than through a dynamic binary instrumentation tool. This enables gaining insight into what the application's bottlenecks are and where to optimize. To do so with low overhead, it is based on *method-level phase behavior* in Java workloads. A method-level phase is defined as a set of parts of the program execution with similar behavior which do not necessarily need to be temporally adjacent.

Specifically, the goal of our technique is to identify method-level phase behavior, and to use that information to correlate hardware performance characteristics directly to the source code of the application and the virtual machine. This is useful for Java and VM developers during performance analysis of their software.

We have implemented a tool, called MonitorMethod, that implements our technique. MonitorMethod uses an off-line analysis that consists of three steps. In a first step, we determine how much time the Java application spends in different portions or methods of the application. This is

done by instrumenting all methods to read microprocessor performance counter values and to track the amount of time that is spent in each method. The result of this first step is an annotated dynamic call graph. In step 2, we determine the methods in which the application spends a significant portion of its total execution time with the additional constraint that one invocation of the method takes a significant portion of the total execution time as well. This is to avoid selecting methods that are too small. During a second run of the application (step 3), these selected methods are instrumented and performance characteristics are measured. Measuring these performance characteristics is done using the hardware performance monitors provided by the microprocessor. In this step, we measure a number of characteristics such as branch misprediction rate, cache miss rate, number of retired instructions per cycle, etc. By doing so, we obtain detailed performance characteristics for the major method-level execution phases of the Java application. In addition to the method-level phases, we also measure performance characteristics for major parts of the VM, such as the compiler/optimizer, the garbage collector, the class loader, etc. By attributing information captured by the hardware performance monitors to the methods in the source code, MonitorMethod gives developers additional insight in the behavior of their application.

In addition to the use case presented in this dissertation, there are several other applications for method-level phase behavior. Detecting program execution phases and exploiting them has received increased attention in recent literature. Various authors have proposed ways of exploiting phase behavior. One example is to adapt the available hardware resources to reduce energy consumption while sustaining the same performance [13, 27, 48, 72]. Also, JIT compilers [7, 10] and dynamic optimization frameworks [12, 62] heavily rely on implicit phase behavior to optimize code.

This chapter is organized as follows. The next section details on our experimental setup. Section 3.2 discusses the off-line approach for identifying method-level phase behavior as used by MonitorMethod. The results of our phase analysis are presented in Section 3.3. Section 3.4 discusses related work. Finally, we provide a summary in Section 3.5.

3.1 Experimental setup

In this section we discuss the experimental setup: our hardware platform, the use of performance monitors, Jikes RVM in which all experiments are done, and the Java applications that are used in the evaluation section of this chapter.

3.1.1 Hardware platform

We use an AMD Athlon XP microprocessor for our measurements. The processor runs at 1.33 Ghz, has an L2 cache of 265 KB and 1 GB of RAM.

3.1.2 Hardware performance monitors

Modern processors are often equipped with a set of performance counter registers. These registers are designed to count microprocessor events that occur during the execution of a program. They allow to keep track of the number of retired instructions, elapsed clock cycles, cache misses, branch mispredictions, etc. Generally, there are only a limited number of performance counter registers available on the chip. On the AMD Athlon, there are four such registers. However, the total number of microprocessor events that can be traced using these performance monitors exceeds 60 in total. As a result, these registers need to be programmed to measure a particular event. The events that are traced for this study are given in Table 3.1. These events are commonly used in architectural studies to analyze program execution behavior. For most of the analyses done in this chapter, we use derived performance metrics. These performance metrics are obtained by dividing the number of events by the number of retired instructions. In this way, we use events that occurred per instruction. This is more meaningful than the often-used miss rates. For example, we will use the number of cache misses per instruction instead of the number of cache misses per cache access. The reason is that the number of cache misses per instruction relates more directly to performance than cache miss rate since it also incorporates the number of cache accesses per instruction. Thus, the performance metrics derived from the events shown in Table 3.1, include for example CPI (clock cycles per retired instruction), L1 D-cache misses per retired instruction, etc.

Performance monitors have several important benefits over other characterization methods: they cause less slowdown since measurements happen at native execution speed, they are easy to use, and they are highly accurate. However, there are also a number of issues that need further attention. First, measuring more than four events at a time in our setup is impossible. Consequently, multiple runs are required to measure more than 4 events. Second, non-determinism can lead to slightly different performance counter values when running the same program multiple times. Therefore, we measure each performance counter multiple times and use the average during analysis.

In this study, the performance counter values are accessed through the VM, see the next section. In turn, the VM makes use of the following tools: (i) the `perfctr` Linux kernel patch [68], which provides a kernel module to access the processor hardware, and (ii) Performance API

Event
Elapsed clock cycles during execution
Retired instructions
L1 D-cache misses
L2 D-cache misses
L1 I-cache misses
L2 I-cache misses
L1 load misses
L1 store misses
L2 load misses
L2 store misses
Instruction TLB misses
Data TLB misses
Branches mispredicted

Table 3.1: The performance counter events traced on the AMD Athlon XP.

(PAPI) [18], a high-level library presenting a uniform interface to the performance monitors on multiple platforms. The kernel patch allows tracing a single process, maintaining the state of the performance monitors across kernel thread switches. The PAPI library presents a uniform manner for accessing the performance monitors through the kernel module. Not all PAPI defined events are available on every platform, and not all native AMD events can be accessed through PAPI. However, for our purposes, it provides a sufficient set of events.

3.1.3 Virtual machine

As motivated in Section 2.3.1 of this thesis, we use the Jikes Research Virtual Machine (RVM). For our work on MonitorMethod we used the CVS head (development) version from January 2004. We used Jikes RVM's most advanced compilation strategy, namely the adaptive strategy. In this scheme, Jikes RVM compiles each method on its first invocation using the baseline compiler and adaptively optimizes hot methods. Multiple recompilations are possible at higher optimization levels. Furthermore, we used Jikes RVM's built-in *CopyMS collector*, a generational GC with a mark-sweep strategy to clean the mature space.

Current implementations of the Jikes RVM include support for hardware performance monitors on both the IA-32 and PowerPC platforms. On the IA-32 platform, access to the processor hardware is done through the PAPI library as discussed above, see also Figure 3.1. The Hardware Performance Monitor (HPM) subsystem of the Jikes RVM defines a set of

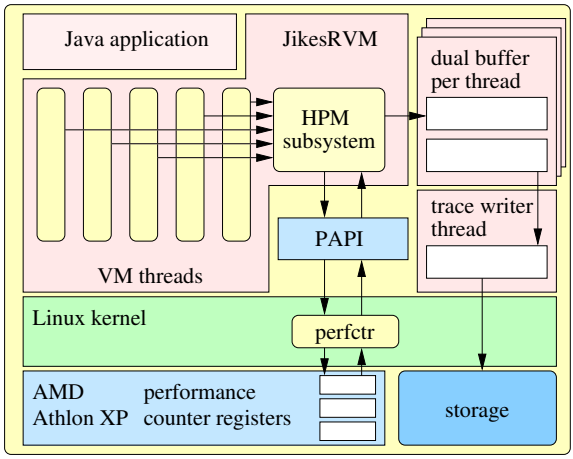


Figure 3.1: An overview of the Jikes RVM tracing system

methods to access the PAPI functions, such as starting, stopping, and resuming the event counters, as well as reading the counter values. Keeping track of the events in the Jikes RVM is done as follows. Essentially, each Java thread keeps track of the performance counter events that occur while it is the executing thread on the virtual processor. Each time the VM schedules a virtual context switch, the removed thread reads the counter values, accumulates them with its existing values, and resets the counters. Hence, a scheduled thread only observes counter values for the events that occur while it is executing. This mechanism for reading performance counter values is the standard implementation within the Jikes RVM. For a more detailed description on this, we refer to [82]. In Section 3.2, we will detail how we extended this approach for measuring performance counter values on a per-method basis.

3.1.4 Java applications

We use the SPECjvm98 and PseudoJBB (i.e. the SPECjbb200 derivative) benchmark suites for the experiments in this chapter. For the SPECjvm98 benchmarks we used the `s100` input set, and the JVM was set to use a 64 MB heap. For PseudoJBB, we use 8 warehouses and the JVM is configured to use a 384 MB heap. More information about these benchmarks can be found in Section 2.3.3.

3.2 Method-level phases

Previous work has shown that applications exhibit phase behavior as a function of time, i.e. programs go from one phase to another during execution [13, 48, 51]. In this chapter, we study *method-level phase behavior*. We consider a method-level phase to be the execution of a Java method including the execution of all its callees. This includes the execution of all the methods called by the selected method, i.e. all the methods in the call graph of which the selected method is the root. There are two motivations for doing so. First of all, the granularity of a method call including its callees is not too small to introduce too much overhead during profiling. Second, it is sufficiently fine-grained to identify phases of execution. Previous work on phase classification [51] has considered various methods for identifying phases, ranging from the instruction level, the basic block level, the loop level up to the method-level. From this research, Lau et al. conclude that phase classification using method-level information is fine-grained enough to identify phases reliably, especially when the application executes many small methods. This is generally the case for applications written in object-oriented languages, of which Java is an example. The use of method calls as a unit for phase classification was also studied by Balasabrumonian et al. [13] and Huang et al. [48].

The following issues are some of the more specific goals we want to reach in order to make MonitorMethod useful:

- We want to gather information from the start to the end of the program's execution. We want maximal coverage without gaps.
- The overhead when profiling method calls should be small enough not to interfere with normal program execution. This means that tracing all executed methods is not a viable option. Also, we want the volume of the trace data to be acceptable.
- We want to gather as much information as possible. At a minimum, the collected information should be sufficiently fine-grained such that transitions in Java performance characteristics can be readily identified. Such transitions can be caused by thread switches, e.g. the garbage collector is activated, or because the application enters a method that shows different behavior from previously executed methods.

To reach these goals, we use the following off-line phase analysis methodology. During a first run of the Java application, we measure the number of elapsed clock cycles in each method execution. This information is collected in a trace file that is subsequently used to annotate a dynamic call graph. A dynamic call graph is a tree that shows the various method invocations during a program execution when traversed in

depth-first order. In a second step of our methodology, we use an off-line tool that analyzes this annotated dynamic call graph and determines the major phases of execution. The output of this second step is a Java source file that describes which methods are responsible for the major execution phases of the Java application. In the third (and last) step, we dynamically link a compiled version of this Java source file to the VM and execute the Java application once again. The Java class file that is linked to the VM forces the VM to measure performance characteristics using the hardware performance monitors for the selected methods. The result of this run is a set of detailed performance characteristics for each method-level phase.

Obviously, to identify recurring phases, static phase analysis has the advantage over dynamic phase analysis as it can look at the 'future' by looking ahead in the trace file. A dynamic approach would have to anticipate phase behavior and that could result in suboptimal phase identification. In addition, the resources that are available during off-line analysis are much larger than in case of on-line analysis, irrespective whether phase classification is done in software or in hardware.

3.2.1 Mechanism

This section details on how a Java workload is profiled with MonitorMethod. We first discuss how the methods in the Java application are instrumented. This mechanism will be used during two steps of our methodology: when measuring the execution time of each method execution during the first run (step 1), and when measuring the performance characteristics of the selected methods (step 3). The only difference between both cases is that in step 1 we instrument all methods. In step 3, we only profile the selected methods. In the second subsection, we detail on what information is collected during profiling. Subsequently, we address profiling the components of the VM.

Instrumenting the application methods

Methods compiled by the VM compilers consist of three parts: (i) the prologue, (ii) the main body of the method, and (iii) the epilogue. The prologue and epilogue handle the calling conventions, pushing and popping the callee's stack frame, yielding at a thread switch, etc. The goal is to capture as many of the generated events during the execution of a method. To achieve this, we add our instrumentation to the method's prologue and to the beginning of the method's epilogue. Methods are instrumented online by all the Jikes RVM compilers, i.e. the baseline compiler as well as the optimizing compiler.

Extending the baseline compiler to instrument methods is quite

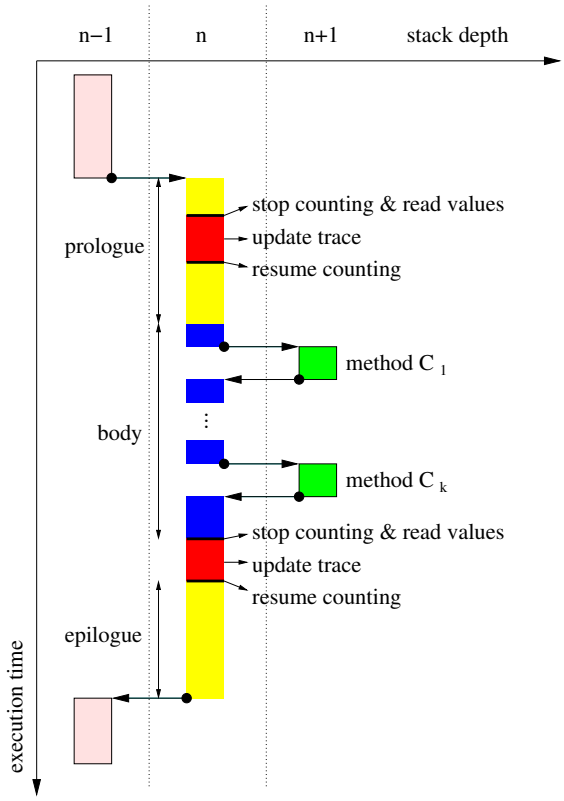


Figure 3.2: Tracing the performance counter events at the prologue and epilogue of a method call.

straightforward. It involves emitting the machine code to call the instrumentation functionality in the Jikes RVM run-time facilities. Calling the instrumentation functionality is done using the Jikes RVM HPM API. Adding calls to baseline compiled methods introduces new yield points. As each yield point is a potential GC point (or *safe point*), it is necessary to update the stack maps accordingly. If not, referenced objects might not be reachable for the GC and risk being erroneously collected.

For the optimizing compiler, things are slightly more complicated. Optimizations are directed by the optimization planner, and involve multiple layers, from a high-level representation to a machine code level representation. Our instrumentation involves adding an extra compiler phase to the compiler plan in the High Intermediate Representation (HIR) optimization set. Basically, we walk through the control flow graph of each method and add similar call instructions to the prologue and epilogue as we did for baseline compiled methods.

Next to the baseline and optimizing compiler, Jikes RVM also employs an on-stack replacement (OSR) scheme [37]. OSR allows the machine code of methods that are executing to be replaced by an optimized version of that machine code, or even by a baseline compiled version. This is especially useful for long-running methods, as the VM does not need to wait until the method finishes executing to replace the code that is currently executing with the newer code. For this, certain OSR safe-points are available in the compiled code. At these points, the OSR mechanism can interrupt the thread executing the code and replace it with the optimized version. Our implementation also supports OSR.

Regardless of the compiler that was used to generate the executable machine code, we call our tracing framework as soon as the new frame for the callee has been established, see Figure 3.2. At this point, the thread executing the method updates its counter values, and suspends counting through the Jikes RVM HPM interface. In this way, no events are counted during the logging of the counter values. When the trace values have been stored into a trace buffer, counting is resumed. To enhance throughput, the trace data is stored in a buffer. We maintain a per-thread cyclic list, which contains two 128 KB buffers. To ensure that these buffers are never touched by the garbage collector, they are allocated outside of the Java heap. Each time one of the buffers for a thread is full, it is scheduled to be written to disk, and the other buffer is used to store the trace data of that thread. A separate thread[1] stores the full buffer contents to disk in a properly synchronized manner. The same action sequence occurs at the method epilogue, just before the return value is loaded into the return register(s) and the stack frame is popped. When a method frame is popped because of an uncaught exception, we also log the counter values at that point. In summary, this approach reads the performance monitor values when entering the method and when exiting the method. The difference between both values gives us the performance metric of the method including its callees. For computing the performance metrics of the method itself, i.e. excluding its callees, the performance metrics of the callees need to be subtracted from the caller method.

From Figure 3.2, it can be observed that the events occurring before reading the counter values in the prologue and the events in the epilogue of a method are attributed to the calling method. However, this inaccuracy is negligible for our purposes.

MonitorMethod pays special attention to exceptions which can be thrown either implicitly or explicitly. The former are thrown by the VM itself whereas the latter are thrown by the program. In both cases, whenever an exception is thrown, control must be transferred from the code that caused the exception to the nearest dynamically-enclosing exception

[1]This is an OS-level POSIX thread, not a VM thread. This ensures that storing the trace data does not block the Virtual Processor POSIX thread on which the Jikes RVM executes.

handler. To do so, Jikes RVM uses stack unwinding: stack frames are popped one at a time until an exception handler is reached. When a frame is popped by the exception handling mechanism, the normal (instrumented) epilogue is not executed, i.e. there is a mismatch in prologue versus epilogue. To solve this problem, we instrumented the exception handling mechanism as well to assure that the trace always contains records for methods that terminate because of an uncaught exception.

Logging the trace data

An instrumented run of our application results in multiple traces, one with the IDs for the compiled methods (baseline, optimized and OSR-compiled), and the others with the relevant counter information per thread. Each record in the latter requires 37 bytes at most, and provides the following information:

- A 1-byte tag, indicating the record type (high nibble) and the number of counter fields (1 up to 4) used in the record (low nibble). The record type denotes whether the record concerns data for a method entry, a method exit, a method exit through an exception, an entry into the compiler, a virtual thread switch, etc.
- Four bytes holding the method ID. This should prove more than sufficient for even very large applications.
- Eight bytes per counter field in the record. We can measure up to four hardware performance monitor values at a time.

It is possible to use a scheme in which the traces for each thread are written to a single file. In this case, we add extra synchronization to ensure the order of the entries in the trace is the same as the execution order. The disadvantage here is that there occurs a serialization during the profiling run, which can be bothersome when using multiple virtual processors on a multi-processor system. Also, in this case, each record will contain two extra bytes for the thread ID.

The total trace file size is thus a function of the number of method invocations, the number of virtual context switches and the number of traced events. Again, for clarification, the same structure is used for both the first step of our methodology (measuring execution times for each method) and the third step (measuring performance characteristics for the selected phases). However, for the first step we apply a heuristic so that we do not need to instrument all methods; this reduces the run-time overhead and prevents selecting wrapper methods as the starting point of a phase. A method is instrumented if the bytecode size of its body

is larger than a given threshold (50 bytes), or if the method contains a backward branch, i.e. can contains a loop.

Instrumenting VM routines

As mentioned earlier, a VM consists of various components, such as the class loader, the compiler, the optimizer, the garbage collector, the thread scheduler, etc. To gain insight in Java workload behavior, it is thus of primary importance to profile these components. For most of these, this is easily done using the available Jikes RVM HPM infrastructure since they are run in separate VM threads. This is the case for the garbage collector and the optimizer (which uses six separate threads). To be able to capture the behavior of the compiler, we had to undertake special action since calling the baseline compiler is done in the Java application threads. In case of the optimizing compiler, the method is queued to be optimized by the optimizer thread. These two cases were handled in our modified Jikes RVM implementation by manually instrumenting the `runtimeCompiler` and `compile` methods from `VM_Runtime`.

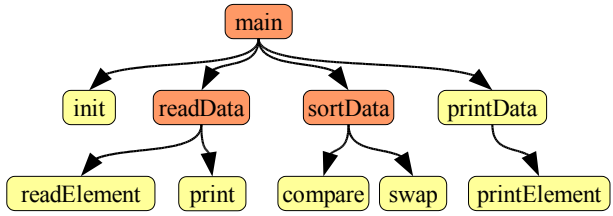
3.2.2 Phase identification

This section discusses how MonitorMethod identifies phases using the off-line phase identification tool. MonitorMethod takes a trace file with timing information about method calls and thread switches (see Section 3.2.1) as input, analyzes it, and outputs a list of unique method names that represent the phases of the application.

To select method-level phases, we use the algorithm proposed by Huang et al. [48] which requires two parameters, called θ_{weight} and θ_{grain} . The basic idea of this algorithm is to select methods in which the program spends a substantial portion of its total execution time (i.e. a fraction of the total execution time that is larger than θ_{weight}), and in which the program spends a sufficiently long period of time on each invocation (i.e. this eliminates short methods and is realized through θ_{grain}).

To illustrate the phase identification algorithm, consider the call graph in Figure 3.3. It depicts a call tree that is the result of analyzing the trace file of a fictive sort program. The sort program reads the data to be sorted, prints an intermediate status message to the screen, sorts the data, and finally prints the sorted data before terminating. For simplicity, abstract time units are used. The table in Figure 3.3 also shows the total time spent in each method, as well as the time spent per invocation.

To identify program phases, our tool first computes the total and average execution times spent in each method. For all methods, these times include the time spent in their callees. In order for a method to be selected



Method name	Total time	Time/call	Calls
`main`	1800	1800	1
`init`	30	30	1
`readData`	300	300	1
`readElement`	200	4	50
`print`	30	30	1
`sortData`	1300	1300	1
`compare`	600	2	300
`swap`	500	2	250
`printData`	170	170	1
`printElement`	150	3	50

Figure 3.3: A fictive phase identification example.

as a program phase, its total execution time needs to be at least a fraction θ_{weight} of the program's total execution time, and the average execution time should take at least a fraction θ_{grain} of the program's total execution time on average. In our running example, $\theta_{\text{weight}} = 10\%$ and $\theta_{\text{grain}} = 5\%$, would select methods whose total execution time is more than 180 and whose average execution time is more than 90 — `main`, `readData` and `sortData`, respectively.

3.2.3 Statistical evaluation

To verify that the phase-detection mechanism described above actually works, we used several statistical techniques to show that our technique is capable of reliably discriminating method-level phases. We used Coefficient of Variation (CoV) to quantify the variability within a phase, and we used an ANOVA test to quantify the variability between different phases. The results show that a larger variability is observed between the phases than within the phases. This asserts that our technique is capable of reliably discriminating method-level phases, and that MonitorMethod can use this information. A detailed overview of these results

Benchmark	Configuration		Overhead	
	θ_{weight} (%)	θ_{grain} (%)	Estimated (%)	Measured (%)
compress	8×10^{-6}	6×10^{-6}	1.84	1.82
jess	1.0	1.0	1.22	1.27
db	8×10^{-6}	6×10^{-6}	7.17	5.61
javac	2×10^{-2}	6×10^{-3}	2.61	2.11
mpegaudio	2×10^{-2}	2×10^{-3}	10.75	3.52
mtrt	10^{-2}	10^{-3}	24.68	7.83
jack	1.0	10^{-2}	3.98	4.28
PseudoJBB	2×10^{-1}	2×10^{-4}	3.69	6.65

Table 3.2: Summary of the selected method-level phases for the chosen θ_{weight} and θ_{grain} values: the estimated overhead and the real overhead..

can be found in our OOPSLA'04 paper [41].

3.3 Results

3.3.1 Identifying method-level phases

Tracing all methods at their entry and exit points is very intrusive. Thus, it is important to determine a set of method-level phases such that the incurred overhead is relatively low, but such that we still get a detailed picture of what happens at the level of the methods being executed. This is done by choosing appropriate values for θ_{weight} and θ_{grain} . These values depend on three parameters: (i) the maximum acceptable overhead, (ii) the required level of information, and (iii) the application itself. The off-line analysis tool aids in selecting values for θ_{weight} and θ_{grain} by providing an estimate for both the overhead and the information yielded by each possible configuration. The graphs at the top of Figure 3.4 and Figure 3.5 present the number of selected method-level phases as a function of θ_{weight} and θ_{grain} . Figure 3.4 presents the data for jack, and Figure 3.5 presents the data for PseudoJBB). Out of the eight benchmarks we analyzed the remaining six benchmarks showed similar results; these have graphs have been omitted from this dissertation.

The graphs at the bottom Figure 3.4 and Figure 3.5 show the corresponding estimated overhead which is defined as the number of profiled method invocations (corresponding to method-level phases) divided by the total number of method invocations. Note that this is not the same as coverage, since selected methods also include their callees. The coverage is always 100% in our phase classification method because the main

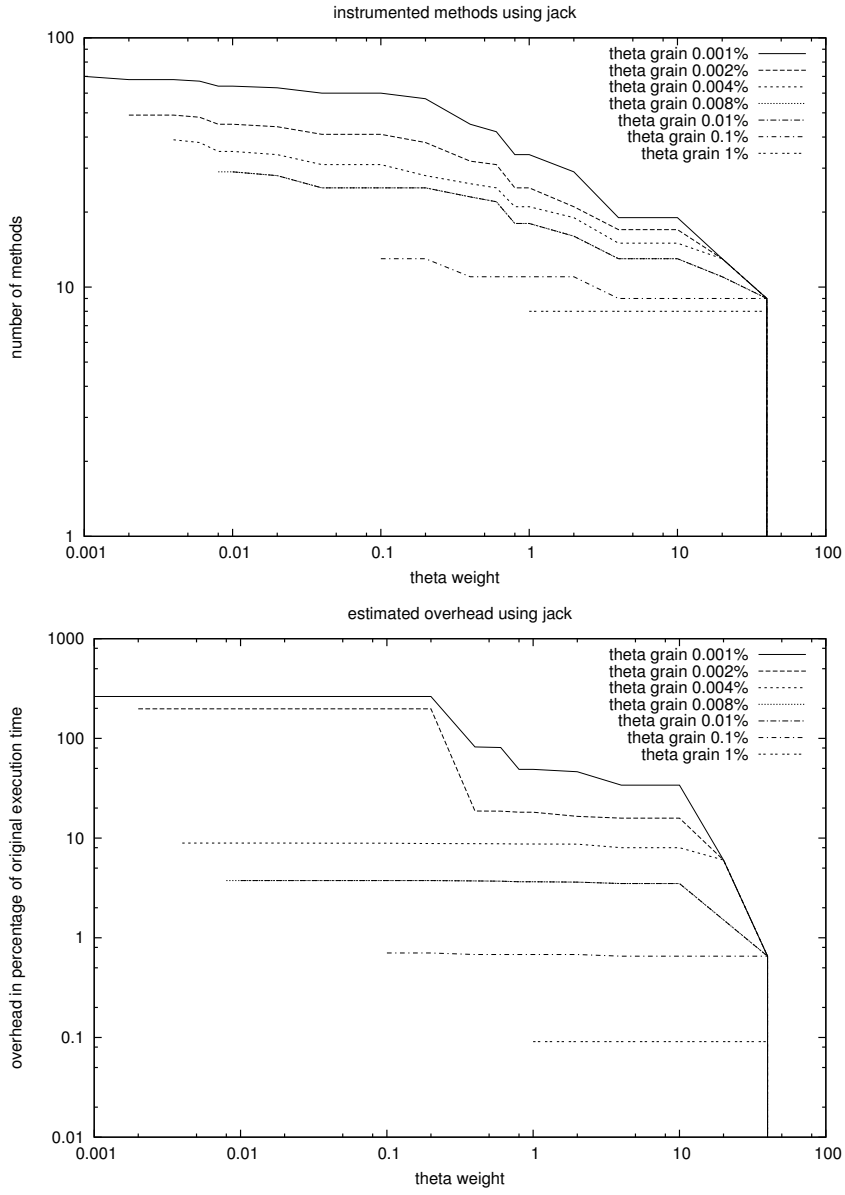


Figure 3.4: Estimating the overhead as a function of θ_{weight} and θ_{grain} for jack. The figure at the top presents the number of selected method-level phases; the figure at the bottom presents the estimated overhead.

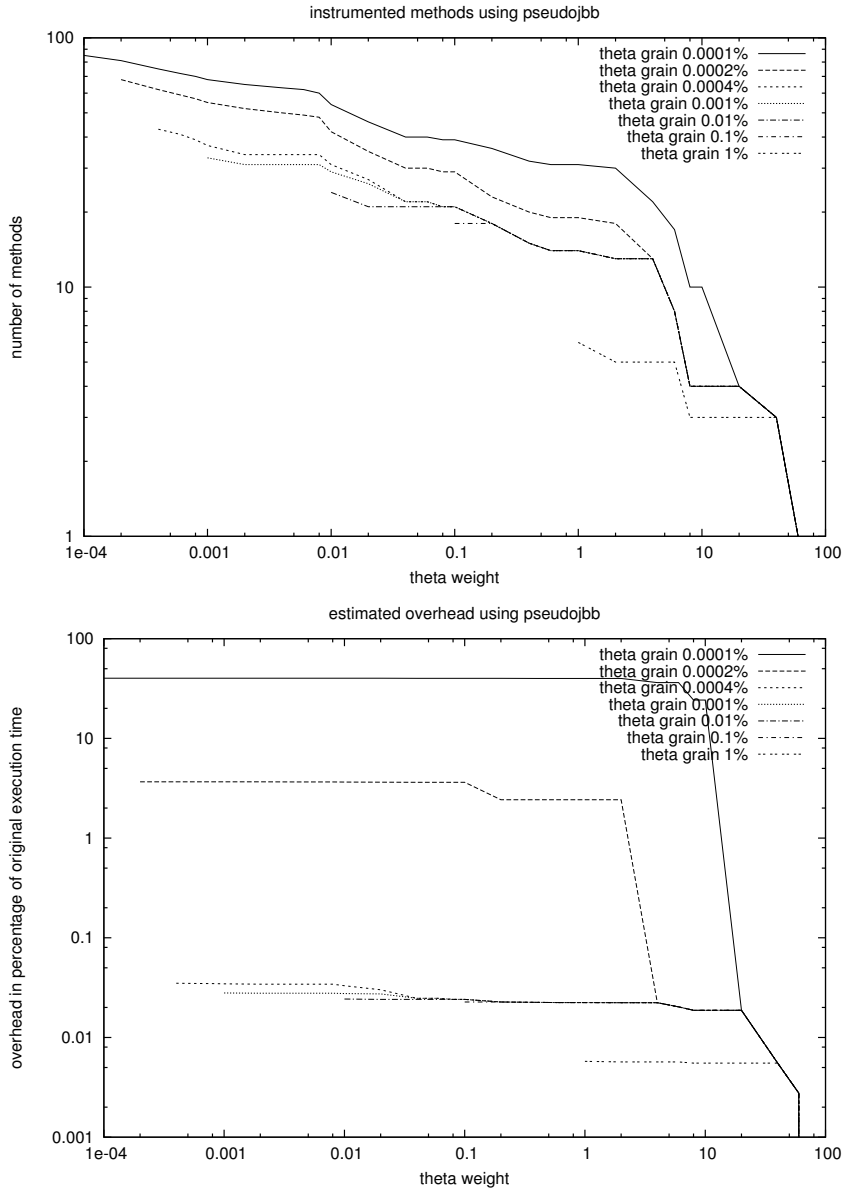


Figure 3.5: Estimating the overhead as a function of θ_{weight} and θ_{grain} for PseudoJBB. The figure at the top presents the number of selected method-level phases; the figure at the bottom presents the estimated overhead.

method of the application is always selected.

Using the plots in Figure 3.4 and 3.5 we can now choose appropriate values for θ_{weight} and θ_{grain} for each benchmark. This is done by inspecting the curves with the estimated overhead. We choose θ_{weight} and θ_{grain} in such a way that the estimated overhead is smaller than 1%, i.e. we want less than 1% of all method invocations to be instrumented. The results for each benchmark are shown in Table 3.2. Note that the user has some flexibility for determining appropriate values for θ_{weight} and θ_{grain}; this allows the user to determine the number of selected method-level phases according to his interest.

So far, we have used an estimated overhead which is defined as the number of profiled method invocations versus the total number of method invocations of the complete program execution. To validate these estimated overheads, i.e. to compare with the actual overheads, we proceed as follows. We measure the total execution time of each benchmark (without any profiling enabled) and compare this with the total execution time when profiling is enabled for the selected methods. The actual overhead is defined as the increase in execution time due to adding profiling. Measuring the wall clock execution time is done using the GNU/Linux `time` command. Table 3.2 compares the estimated overhead and the actual overhead. We observe from these results that the actual overhead is usually quite small and mostly tracks the estimated overhead very well. This is important since determining the estimated overhead is more convenient than measuring the actual overhead. In two cases the estimate is significantly larger than the measured overhead, i.e. for mpegaudio and for mtrt. The latter can be explained by the fact that the formula for estimating the overhead is somewhat naive.

Benchmark	Configuration		Number of phases	
	θ_{weight} (%)	θ_{grain} (%)	Static (total)	Dynamic (total)
compress	8×10^{-6}	6×10^{-6}	49 (54)	2,664 (19,726,311)
jess	1.0	1.0	10 (211)	23 (22,693,249)
db	8×10^{-6}	6×10^{-6}	52 (57)	32,223 (1,484,605)
javac	2×10^{-2}	6×10^{-3}	29 (503)	9,864 (23,388,699)
mpegaudio	2×10^{-2}	2×10^{-3}	23 (191)	40,064 (29,338,068)
mtrt	10^{-2}	10^{-3}	30 (94)	88,719 (14,859,306)
jack	1.0	10^{-2}	18 (182)	2,528 (4,292,580)
PseudoJBB	2×10^{-1}	2×10^{-4}	52 (381)	29,599 (16,224,804)

Table 3.3: Summary of the selected method-level phases for the chosen θ_{weight} and θ_{grain} values: the number of static and dynamic phases.

For completeness, Table 3.3 presents the number of static method-level

Benchmark	Configuration		Trace file size (KB)
	θ_{weight} (%)	θ_{grain} (%)	
compress	8×10^{-6}	6×10^{-6}	211
jess	1.0	1.0	68
db	8×10^{-6}	6×10^{-6}	2590
javac	2×10^{-2}	6×10^{-3}	871
mpegaudio	2×10^{-2}	2×10^{-3}	753
mtrt	10^{-2}	10^{-3}	2,680
jack	1.0	10^{-2}	244
PseudoJBB	2×10^{-1}	2×10^{-4}	2,766

Table 3.4: Summary of the selected method-level phases for the chosen θ_{weight} and θ_{grain} values: the size of the trace file.

phases as well as the number of phase invocations or dynamic phases. For reference, the total number of static methods as well as the total number of dynamic method invocations are shown. Table 3.4 presents the file size of the trace file obtained from running the Java application while profiling the selected method-level phases. Recall that besides application method records, the trace also contains data regarding thread switches, GC, and compiler activity.

3.3.2 Analysis of method-level phase behavior

A programmer analyzing application behavior will typically start from a high-level view of the program. Two of the first things one wants to analyze are where the time is spent, and whether potential performance bottlenecks are due to the application or the VM. In the first subsection, we look at the high-level behavior of Java applications and compare it with the behavior of the VM (GC, compiler, etc.). Once the high-level behavior is understood, the next logical step is to investigate parts of the application into more detail. The subsequent subsection shows how the programmer can use the information collected by our framework to gain insight about the low-level behavior of his program, and how our data can help identify and explain performance bottlenecks.

VM versus application behavior

Figure 3.6 (top) shows the number of events occurring in the application versus the VM. This is done here for PseudoJBB. We observe that most of the events occur in the application and not in the VM. Indeed, the total program execution spends 73% of its total execution time in the

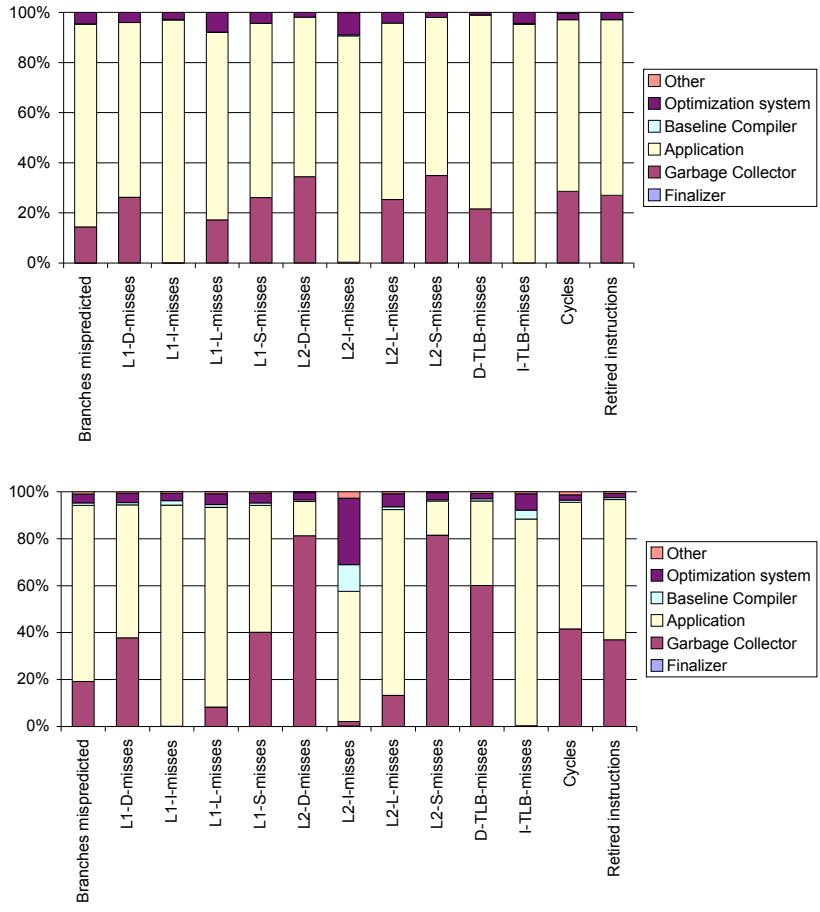


Figure 3.6: The performance characteristics for the application versus the VM components for PseudoJBB (top) and jack (bottom).

application; the remaining 27% is spent in the VM. The time spent in the VM is partitioned in the time spent in the various VM components: compiler, optimizer, garbage collector and others. We observe that the most dominant part of the VM routines is due to the optimizer (3.5%) and the garbage collector (23%). In Section 4.4.2 we will analyse the overhead of the optimization system in more detail. This graph reveals several interesting observations. For example, although the optimizer is responsible for only 3.5% of the total execution time, it is responsible for a significantly larger proportion of the L1 D-cache load misses (6.3%) and L2 I-cache misses (13.7%). The garbage collector on the other hand, accounts for significantly more L2 D-cache misses (28%) than it accounts for execution time (21%). Another interesting result is that the garbage collector accounts for a negligible fraction of the L1 and L2 I-cache and I-TLB misses. This is due to the fact that the garbage collector has a small instruction footprint while accessing a large data set.

Figure 3.6 (bottom) presents a similar graph for jack. The percentage of the total execution time spent in the application is 54%. Of the 46% spent in the VM, 41.5% is spent in the garbage collector, 2.3% in the optimizing compiler, 0.9% in the baseline compiler and 1.3% in other VM routines, such as the thread scheduler, class loader, etc. These results confirm the specific behavior of the garbage collector previously observed for PseudoJBB: low L1 and L2 I-cache and I-TLB miss rates and high L2 D-cache and D-TLB miss rates (due to writes). The baseline compiler and the optimizer show high L2 I-cache miss rates.

Table 3.5 presents the time spent in the application versus the time spent in the VM components for the SPECjvm98 and PseudoJBB benchmarks. The time spent in the application varies between 54% and 92% of the total execution time; the time spent in the garbage collector varies between 7% and 42%, and the time spent in the optimizer varies between 0.4% and 5.8%. The execution time in the other VM components is negligible. We conclude that Java workloads spend a significant fraction of their total execution time in the VM, up to 46% for the long-running applications included in our study. For short-running applications, for example SPECjvm98 with the s1 input set, this fraction will be even larger [35]. It is interesting to note that the three benchmarks (compress, db, and mpegaudio) for which the total execution time spent in the application is significantly larger than the average case (89%, 92% and 85%, respectively), were labeled as 'simple' benchmarks by Shuf et al. [74].

In this chapter we use a 64 MB heap size for the SPECjvm98 benchmarks and a 384 MB heap size for the PseudoJBB benchmark. It is important to note that the time spent collecting garbage depends largely on the heap size; for larger heap sizes, less time will be spent collecting garbage. Hence, Figure 3.6 and Table 3.5 present a single data point only. Measuring the execution time spent in garbage collection with different

Benchmark	Application (%)	Garbage collector (%)	Baseline compiler (%)	Optimizing compiler (%)	Other (%)
compress	89.136	9.377	0.205	0.696	0.580
jess	56.835	39.641	0.919	1.914	0.688
db	92.211	6.991	0.128	0.455	0.213
javac	65.463	28.987	0.940	3.618	0.984
mpegaudio	85.000	7.999	0.559	5.821	0.620
mtrt	65.802	28.039	0.485	4.687	0.982
jack	53.905	41.556	0.941	2.317	1.265
PseudoJBB	73.348	22.974	0.091	3.532	0.063

Table 3.5: The time spent in the application and the different VM components.

heap sizes falls outside the scope of this work.

Application performance bottleneck analysis

Profilers provide a means for programmers to perceive the way their programs are performing. Our technique provides an easy way to programmers to gain insight about the performance of their application at the micro-architectural level. That is, hardware performance monitors can be linked to the methods in the source code. Conventional methods on the other hand, are much more labor-intensive and error-prone for the following reasons: (i) the huge amount of data gathered from a profiling run, and (ii) the presentation of this huge amount of data usually prevents quick insight.

This section shows how method-level phases can help answer three fundamental questions programmers might ask when optimizing their application: (i) what is the application's performance bottleneck, (ii) why does the performance bottleneck occur, and (iii) when does the performance bottleneck occur?

To answer the first question (what is the performance bottleneck?), we ordered the phases by CPI. Tables 3.6, 3.7 and 3.8 present the major bottleneck phases for both the SPECjvm98 benchmarks and for PseudoJBB. We show the phases for which the total execution time takes more than 1% or 2% of the program execution time and for which the CPI is above the average CPI, or which otherwise display bad behavior for a shown characteristic. Methods whose CPI is worse than the average CPI, are potential bottlenecks.

To answer the second question (why does the performance bottleneck occur?), this table also shows the cache miss rates and the branch misprediction rate. This information is helpful in identifying why these phases suffer from such a high CPI. For example, high D-cache miss rates suggest that the programmer should try to improve the data memory behavior for the given phases. We can make the following interesting observations—these are just a few examples to clarify the usefulness of linking microprocessor level information to source-code level methods.

- The `Compressor.compress` method in compress suffers from high D-cache miss rates. Optimization of the data memory behavior can be achieved by applying prefetching.
- From all the benchmarks, mtrt has a method with the highest fraction mispredicted branches: `Scene.RenderScene`. This method contains two nested loops, iterating over all pixels in the scene to be rendered. Inside the loop there are a number of conditional branches and a call to for example `Scene.Shade`. In turn, the latter shows bad branch behavior due to numerous (nested) tests that

Phase	Time (%)	CPI	L1-D	L1-I	L2-D	L2-I	Branches mispredicted
compress							
FileInputStream.read	2.1517	5.194	8.41	56.74	4.39	8.31	32.13
garbage collector	9.3773	1.7778	4.04	0.02	2.59	0.01	4.39
Compressor.compress	58.3916	1.7447	22.91	0.01	4.36	< 0.01	7.28
Decompressor.decompress	25.2042	0.9242	2.53	< 0.01	0.12	< 0.01	4.83
Benchmark average	n/a	1.4830	12.25	0.10	2.01	0.04	5.69
jess							
Jesp.parse	1.1021	1.8701	4.52	5.91	1.04	1.71	14.55
garbage collector	39.6413	1.7647	4.02	0.02	2.58	0.01	4.33
Rete.Run	53.8732	1.1796	4.92	0.51	0.45	0.05	4.51
Benchmark average	n/a	1.3959	4.66	0.68	1.12	0.17	4.73
db							
Database.shell_sort	85.5593	5.1134	26.42	0.02	18.01	0.01	4.87
Database.remove	4.5821	2.7155	11.33	0.10	6.28	0.06	1.36
garbage collector	6.9912	1.7989	3.92	0.03	2.45	0.01	4.23
Database.set_index	2.3873	1.5749	5.74	0.08	3.38	0.04	0.08
Benchmark average	n/a	3.9847	4.71	0.05	3.13	0.01	1.07

Table 3.6: The method-level phases from SPECjvm98. The L1 and L2 I-cache miss rates, L1 and L2 D-cache miss rates and the branch misprediction rate are given as the number of events per 1,000 instructions.

Phase	Time (%)	CPI	L1-D	L1-I	L2-D	L2-I	Branches mispredicted
javac							
SourceClass.check	7.0644	2.2501	8.06	13.13	1.74	1.75	23.2
garbage collector	28.9874	1.8059	4.48	0.02	2.55	0.01	4.76
SourceClass.compileClass	26.0361	1.7408	5.14	6.53	0.98	0.85	16.26
Parser.parseClass	22.6849	1.4789	2.93	5.52	0.54	0.44	18.26
Benchmark average	n/a	1.6747	4.28	4.48	1.32	0.66	13.26
mpegaudio							
garbage collector	7.9994	1.7795	4.15	0.03	2.62	0.01	4.71
lb.read	20.7281	0.8602	1.41	1.17	0.01	< 0.01	6.38
t.O	75.1119	0.8430	0.82	0.49	< 0.01	< 0.01	2.29
Benchmark average	n/a	0.8157	1.07	0.47	0.03	0.02	3.25
mtrt							
Scene.RenderScene	1.9640	2.3249	12.32	18.74	0.21	0.25	35.98
garbage collector	28.0391	1.7829	3.73	0.02	2.40	< 0.01	4.47
Scene.GetLightColor	23.7782	1.4919	9.74	3.29	0.68	0.03	8.95
Scene.Shade	36.2558	1.3496	7.21	2.84	0.42	0.05	11.42
Scene.ReadPoly	2.4275	1.2909	1.52	3.32	0.12	0.10	8.88
Benchmark average	n/a	1.5389	8.27	2.66	1.03	0.07	7.18

Table 3.7: The method-level phases from the SPECjvm98 (continued) benchmark suite. The L1 and L2 I-cache miss rates, L1 and L2 D-cache miss rates and the branch misprediction rate are given as the number of events per 1,000 instructions.

Phase	Time (%)	CPI	L1-D	L1-I	L2-D	L2-I	Branches mispredicted
jack							
Jack_the_Parser_Generator._Jack3_1	2.34092	1.9655	5.84	5.19	0.57	0.22	11.53
garbage collector	41.5561	1.7741	4.04	0.02	2.59	0.01	4.36
Jack_the_Parser_Generator.production	1.87112	1.7039	4.38	7.41	0.51	0.73	15.16
Jack_the_Parser_Generator.jack_input	2.8412	1.6014	3.59	6.26	0.38	0.56	13.69
Jack_the_Parser_Generator.expansion_choices	20.5098	1.5546	4.46	7.54	0.22	0.23	15.94
Jack_the_Parser_Generator.java_declarations_and_code	19.4081	1.3648	2.70	5.1	0.11	0.09	12.64
Jack_the_Parser_Generator_Internals.db_process	2.78693	1.2737	2.61	1.61	0.59	0.28	4.74
ParseGen.build	2.6689	1.1157	2.27	0.37	0.69	0.06	2.41
Jack_the_Parser_Generator.complex_regular_expression	1.98849	0.5508	3.45	7.45	0.13	0.19	10.46
Benchmark average	n/a	1.5976	3.83	3.58	1.19	0.24	9.58
PseudoJBB							
DeliveryTransaction.process	2.7597	3.0722	8.74	9.95	6.45	2.61	17.32
garbage collector	22.9744	2.1581	5.76	0.03	3.59	< 0.01	4.35
TransactionManager.go	57.9074	2.1219	6.77	7.77	2.91	0.75	11.08
Warehouse.loadStockTable	7.7381	1.4245	1.25	1.61	0.14	0.02	8.26
Benchmark average	n/a	2.046	6.02	5.02	2.69	0.57	9.13

Table 3.8: The method-level phases from the SPECjvm98 (continued) and SPECjbb2000 (as observed in PseudoJBB) benchmark suites. The L1 and L2 I-cache miss rates, L1 and L2 D-cache miss rates and the branch misprediction rate are given as the number of events per 1,000 instructions.

are conducted to decide on the color of the pixel that is being rendered. This behavior can be optimized by changing the code layout to improve the branch predictability.

- Poor I-cache behavior can be observed for the `expansion_choices` method in jack.

- For the SPECjvm98 benchmarks, the garbage collector shows a very consistent behavior, with a CPI that remains around 1.77. Also, the garbage collector shows very good I-cache behavior both on L1 and L2. This is (i) due to the fact that garbage collection can take quite some time and (ii) because the garbage collector code is usually quite compact.

Finally, to answer the last question (when does the performance bottleneck occur?), one can use region information to relate phases to the time behavior of an application. Figure 3.7 shows the CPI over time for javac. Region one corresponds to the garbage collector, region 2 to `Parser.parseClass` and region 3 with `SourceClass.check` and `SourceClass.compileClass`.

3.4 Related work

The first subsection details on related work done on Java performance analysis. In the second subsection, we discuss phase classification and detection techniques.

3.4.1 Java performance analysis

Cain et al. [21] characterize the Transaction Processing Council's TPC-W web benchmark which is implemented in Java. TPC-W is designed to exercise the web server and transaction processing system of a typical e-commerce web site. They used both hardware execution (on an IBM RS/6000 S80 server with 8 RS64-III processors) and simulation in their analysis.

Karlsson et al. [49] study the memory system behavior of Java-based middleware. To this end, they study the SPECjbb2000 and SPECjAppServer2001 benchmarks on real hardware as well as through simulation. For the real hardware measurements, they use the hardware counters on a 16-processor Sun Enterprise 6000 multiprocessor server. They measure performance characteristics over the complete benchmark run and make no distinction between the VM and the execution phases of the application.

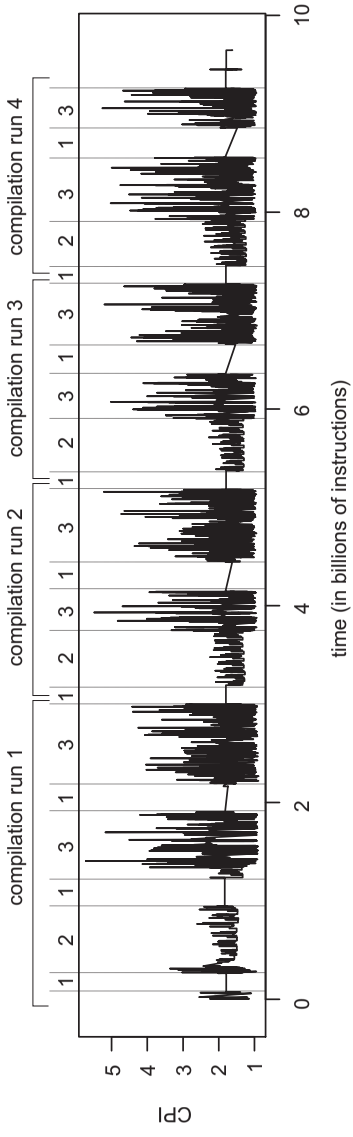


Figure 3.7: The region information in a javac -s100 run.

Luo et al. [57] compare SPECjbb2000 versus SPECweb99, VolanoMark and SPEC CPU2000 on three different hardware platforms: the IBM RS64-III, the IBM POWER3-II and the Intel Pentium III. All measurements were done using performance counters and measure aggregate behavior.

Eeckhout et al. [35] measure various performance counter events and use statistical data analysis techniques to analyze Java workload behavior at the microprocessor level. One particular statistical data analysis technique that is used in that paper is principal components analysis which allows to reduce the dimensionality of the data set. This reduced data set allows for easier reasoning. In that work, the authors also measured aggregate performance characteristics and made no distinction between execution phases.

Dufour et al. [32] present a set of architecture-independent metrics for describing dynamic characteristics of Java applications. All these metrics are bytecode-level program characteristics and measure program size, the usage frequency of various data structures (arrays, pointers, floating-point operations), memory use, concurrency, synchronization and the degree of polymorphism.

Dmitriev [28] presents a bytecode-level profiling tool for Java applications, called JFluid. During a typical JFluid session, the VM is started with the Java application without any special preparation. Subsequently, the tool is attached to the VM, the application is instrumented, the results are collected and analyzed on-line, and the tool is detached from the VM. The instrumentation is done by injecting instrumentation bytecodes into methods of a running program. In JFluid, the user needs to specify which call subgraph, called a 'task' by Dmitriev, from an arbitrary root method is to be instrumented. This method has two major differences with our approach: (i) we do not operate at the bytecode level but at the lower microprocessor level and (ii) we provide a means to automatically detect these 'tasks'. The advantage of our approach is that it relieves the user from manually selecting major tasks of execution. The disadvantage of our approach is that a preparatory run is required.

Sweeney et al. [82] present a system to measure microprocessor level behavior of Java workloads. To this end, they generate traces of hardware performance counter values while executing Java applications. This is done for each Java thread and for each microprocessor on which the thread is running. The latter can be useful in case of a multiprocessor environment. The infrastructure for reading performance counter values used by Sweeney et al. is exactly the same as the one we use—using HPM in the Jikes RVM—except for the fact that our measurements are done on an IA-32 ISA platform opposed to the PowerPC ISA platform. Sweeney et al. read the performance counter values on every virtual context switch in the VM. This information is collected for each virtual processor and for each Java thread, and written in a per virtual processor

record buffer. Sweeney et al. also present a tool for graphically exploring the performance counter traces. The major difference between the work by Sweeney et al. and our work, is that we collect performance counter values on a per-phase basis as opposed to the timing-driven approach of taking one sample on every virtual context switch. The benefit of measuring performance counter values on a per-phase basis is that performance counter values can be easily linked to the code that is executed in the phase. We believe this is particularly useful for analysis in general, and for application and VM developers in particular. Moreover, our approach is more general than the approach by Sweeney et al. since the information we obtain can be easily transformed to behavioral information over time. This can be done by ordering our information on a time-line basis. The benefit of the approach by Sweeney et al. is that they use an on-line technique, while we essentially perform an offline analysis.

In later work [44, 45], Hauswirth et al. extend this work [82]. They found that using hardware performance monitors was not enough for a complete understanding of certain performance phenomena, and concluded that they also needed information from higher layers in the execution stack. They propose an extended approach, called *vertical profiling*, that enables this level of understanding by correlating behavioral information about multiple layers of the execution stack. In their extended work, they still use timer-driven sampling whereas our work used phase behavior.

VTune [29] is a tool from Intel that provides a framework for gathering performance counter data over time. Our work differs in that we are using instrumentation instead of sampling. However, unlike our technique, VTune does not require a preparatory run.

3.4.2 Program phases

Several techniques that have been proposed in the recent literature to detect program phases divide the total program execution in fixed intervals. For each interval, program characteristics are measured during program execution. When the difference in program characteristics between two consecutive intervals exceeds a given threshold, the algorithm detects a phase change. These approaches are often referred to as *temporal techniques*.

Balasubramonian et al. [13] compute the number of dynamic conditional branches executed. A phase change is detected when the difference in branch counts between consecutive intervals exceeds a threshold. This threshold is adjusted dynamically during program execution to match the program's execution behavior. Sherwood et al. [71, 72] use basic block vectors (BBVs) to identify phases. A BBV is a vector in which the elements count the number of times each static basic block is executed in

the fixed interval. These BBVs are weighted by the number of instructions in the given basic block. A phase change is detected when the Manhattan distance between two consecutive intervals exceeds a given threshold. They consider both static and dynamic methods for identifying phases in [71] and [72], respectively. In a follow-up study, Lau et al. [51] study several structures for classifying program execution phases. They study approaches using basic blocks, loops, procedures, opcodes, register usage and memory address information. In contrast to the previously mentioned approaches which all use micro-architecture-independent characteristics—i.e. the metrics are only dependent on the instruction set architecture (ISA) and not on the micro-architecture—Duesterwald et al. [31] use micro-architecture-dependent characteristics to detect phases. The metrics used by them are the instruction mix, the branch prediction accuracy, the cache miss rate and the number of instructions executed per cycle (IPC). These metrics are measured using performance monitors over fixed intervals of 10 milliseconds.

Next to temporal phase detection approaches, there exist a number of approaches that do not use fixed intervals. Balasubramonian et al. [13] consider procedures to identify phases. They consider non-nested and nested procedures as phases. A non-nested procedure is a procedure that includes its complete call graph, i.e., including all the methods it calls, as is done in MonitorMethod. A nested procedure does not include its callees. They concluded that non-nested procedures are better performing than nested procedures. Huang et al. [48] also use procedures to identify phases. The method used in our work to identify method-level phases of execution—using θ_{weight} and θ_{grain} —is based on the approach proposed by Huang et al. Next to this static approach, they also propose a hardware-based call stack mechanism to identify program phase changes. Our work differs from the one by Huang et al. for at least three reasons. First, we explore the technique for detecting phases in more detail by quantifying the overhead and coverage as a function of θ_{weight} and θ_{grain} . Huang et al. chose fixed θ_{weight} = 5% and θ_{grain} = 1,000 cycles in their experiments. Second, we study Java workloads whereas Huang et al. studied SPEC CPU2000 benchmarks. Java workloads provide several additional challenges over C-style workloads because of the managed runtime environment. Third, the focus of the work by Huang et al. was on exploiting phase behavior for energy-efficient computing. The focus of our work is on using phase behavior to increase the understanding during program performance analysis.

Huang et al. [48] found that program behavior tends to be fairly homogeneous across different invocations of the same procedure. More recently, Lau et al. [50] found that this extends to loops as well. Lau et al. used this observation to develop an automated profiling technique to identify code locations (branches, procedure calls, returns, loop entries,

etc) whose executions correlate to phase changes. To do so, they build a combined procedure and loop graph, and they use this graph to identify the start of phases. The main difference with our work is that we only consider procedures.

Nagpurkar et al. [63] present a framework that can be used to develop a wide range of on-line phase detection algorithms. The input of the framework is an execution profile and the output of the framework is a series of states that indicate whether the execution is in a phase or in a transition between phases. These states could, for example, be used by a dynamic optimization system to perform specializing optimizations when the behavior is stable or it can consider optimization decisions when the behavior changes. An interesting property of their framework is that it detects these phase changes based on a user-provided similarity model. Nagpurkar et al. did not explore using hardware performance monitors as the input to this similarity model, however we see no reason why this should not be possible. An other important difference with our work is that their technique is on-line.

3.5 Conclusion

Java applications closely interact with the VM. In addition, applications themselves are becoming increasingly complex. Because of this, automatic tools for characterizing and understanding such software systems are becoming paramount for effective performance analysis. We have shown this in Chapter 2, and we have shown it again in this chapter.

The purpose of the work in this chapter is to take advantage of method-level phase behavior of Java applications so that we can relate processor-level performance characteristics to application source code with acceptable overhead.

The technique presented consists of three steps. In the first step, we measure the execution time for each method invocation using hardware performance monitors which are made available through the Jikes RVM HPM API. The second step analyzes this information using an off-line tool and selects a number of phases. These phases are subsequently characterized in the third step using performance monitors. This characterization includes measuring a number of microprocessor events such as cache miss rates, TLB miss rates, branch misprediction rates, etc. Particularly novel compared to existing work is the fact this technique can link the microprocessor-level information to the methods in the Java application's source code.

Using this framework, we investigated the phase behavior of both the SPECjvm98 and SPECjbb2000 benchmark suite. In a first set of experiments, we compared the characteristics of the Java application versus the

various VM components. We conclude that Java workloads spend a significant portion of their total execution time (up to 46%) in the VM, more specifically in the garbage collector. In a second set of experiments, we focused on the method-level phase behavior of the Java application itself. We have shown that the overhead incurred due to profiling is small.

Finally, we showed how this information can be used by developers to answer three fundamental questions about the performance of their application: (i) what are the application's bottlenecks, (ii) why do these bottlenecks occur and (iii) when do these bottlenecks occur?

Chapter 4

Online profiling using hardware performance monitors

In the previous chapters, we looked at *offline* profiling techniques that link information captured at different layers of the execution stack. In this chapter, we will look at an *online* vertical profiling technique. The goal of online profiling techniques is to collect profile information during the execution of the application and to consume that information within the same run.

To achieve high performance, production Java virtual machines contain at least two modes of execution: (i)) *unoptimized* execution, using interpretation [61, 67, 80] or a simple dynamic compiler [7, 15, 24, 47] that produces code quickly, and (ii) *optimized* execution using an optimizing dynamic compiler. Methods are first executed using the unoptimized execution strategy. An online profiling mechanism is used to find a subset of methods to optimize during the same execution. Many systems enhance this scheme to provide multiple levels of optimized execution [7, 61, 80], with increasing compilation cost and benefits at each level. A crucial component to this strategy is the ability to find the important methods for optimization in a low-overhead and accurate manner.

Two approaches that are commonly used to find optimization candidates are method *invocation counters* [24, 61, 67, 80] and *timer-based sampling* [7, 15, 61, 80, 86]. The counters approach counts the number of method invocations and, optionally, loop iterations. Timer-based sampling records the currently executing method at regular intervals using an operating system timer.

Although invocations counters can be used for profiling unoptimized

code, their overhead makes them a poor choice for use in optimized code. As a result, VMs that use multiple levels of optimization rely exclusively on sampling for identifying optimized methods that need to be promoted to higher levels. Having an accurate sampler is critical to ensure that methods do not get stuck at their first level of optimization, or in unoptimized code if a sample-only approach is employed [7, 15].

Most VMs rely on an operating system timer interrupt to perform sampling, but this approach has a number of drawbacks. First, the minimum timer interrupt varies depending on the version of the OS, and in many cases can result in too few samples being taken. Second, the sample-taking mechanism is untimely and inaccurate because there is a variable delay between the timer going off and the sample being taken. Third, the minimum sample rate does not change when moving to newer, faster hardware; thus, the effective sample rate (relative to the program execution) continues to decrease as hardware performance improves.

This work advocates a different approach, using the hardware performance monitors (HPMs) on modern processors to assist in finding optimization candidates. This HPM-sampling approach measures the time spent in methods more accurately than any existing sample-based approach, yet remains low-overhead and can be used effectively for both optimized and unoptimized code. In addition, it allows for more frequent sampling rates compared to timer-based sampling, and is more robust across hardware implementations and operating systems.

This chapter makes the following contributions:

- We describe and empirically evaluate the design space of several existing sample-based profilers for driving dynamic compilation;
- We describe the design and implementation of an HPM-sampling approach for driving dynamic compilation; and
- We empirically evaluate the proposed HPM approach in Jikes RVM, demonstrating that it has higher accuracy than existing techniques, and improves performance by 5.7% on average and up to 18.3%.

To the best of our knowledge, no production VM uses HPM-sampling to identify optimization candidates to drive dynamic compilation. This work illustrates that this technique results in significant performance improvement and thus has the potential to improve existing VMs with minimal effort and without any changes to the dynamic compiler.

This chapter is organized as follows. Section 4.1 provides further background information for this work. Section 4.2 details the HPM-sampling approach we propose. After detailing our experimental setup in Section 4.3, Section 4.4 presents a detailed evaluation of the HPM-sampling approach compared to existing techniques; the evaluation in-

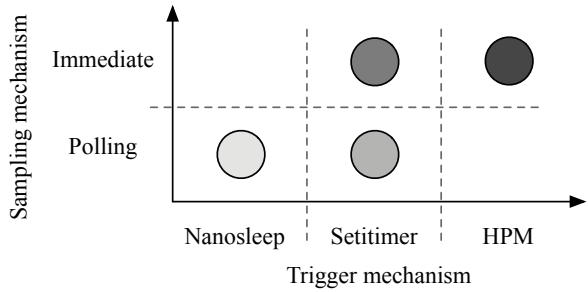


Figure 4.1: The design space for sampling profilers: sampling versus trigger mechanism.

cludes overall performance, overhead, accuracy, and robustness. Section 4.5 compares our contribution to related work and Section 4.6 provides a summary.

4.1 Background

This section describes background for this work. Specifically, it describes the design space of sampling techniques for finding optimization candidates and discusses the shortcomings of these techniques; gives relevant details of Jikes RVM; and summarizes hardware performance monitor facilities, focusing on the particular features we employ in this work. This background is important to help understand why it is useful to feed micro-processor level information into the JVM and how it helps to collect more complete profiles.

4.1.1 Sampling design space

Two important factors in implementing any method sampling approach are (i) the trigger mechanism and (ii) the sampling mechanism. Figure 4.1 summarizes this 2-dimensional design space for sampling-based profilers. The horizontal axis shows the trigger mechanism choices and the vertical axis shows the sampling mechanism choices. The bullets in Figure 4.1 represent viable design points in the design space of sampling-based profilers – we will discuss the nonviable design points later. Section 4.4 compares the performance of these viable design points and shows that HPM-immediate is the best performing sampling-based profiling approach.

Trigger mechanisms

The trigger mechanism is the technique used to periodically initiate the sample-taking process. All production JVMs that we are aware of that use sampling to find optimization candidates use an operating system feature, such as nanosleep or setitimer, as the trigger mechanism for finding methods to optimize [7, 15, 80, 86, 61]. The *nanosleep* approach uses a native thread running concurrently with the VM. When the timer interrupt goes off, this native thread is scheduled to execute and then set a bit in the VM. When the VM next executes and sees the bit is set, a sample is taken. The *setitimer* approach does not use a native thread and thus has one less level of indirection. Instead, the interrupt handler for the VM sets the bit in the VM when the timer interrupt goes off. The third option, HPM, will be described in Section 4.2.

Timer-based sampling (using nanosleep or setitimer) is a low overhead profiling technique that can be used on all forms of code, both unoptimized and optimized code. It allows for the reoptimization of optimized code to higher levels, as well as for performing online profile-directed optimizations, when the profile changes, such as adaptive inlining [7].

However, timer-based sampling does have the following limitations:

Not enough data points: The timer granularity is dependent on the operating system settings, e.g., a mainstream operating system, such as Linux 2.6, provides a common granularity of 4 ms, which means that at most 250 samples/second can be taken. Other operating systems may not offer such a fine granularity. For example, in earlier versions of Linux the granularity was only 10 ms, resulting in at most 100 samples per second. Furthermore, the granularity is not necessarily something that can be easily changed because it will likely require rebuilding the kernel, which may not be feasible in all environments and will not be possible when source code is not available.

Lack of robustness among hardware implementations: Timer-based sampling is not robust among machines with different clock speeds and different microarchitectures because a faster machine will execute more instructions between timer-interrupts. Given a fixed operating system granularity, a timer-based profiler will collect fewer data points as microprocessors attain higher performance. Section 4.4 demonstrates this point empirically.

Not timely: There is a variable delay from when the operating system discovers that a thread needs to be notified about a timer event and when it schedules the thread that requested the notification for initiating the sample taking.

As we will see in Section 4.2, the HPM approach addresses all of these shortcomings.

Sampling mechanisms

Once the sample-taking process has been initiated by some trigger mechanism, a sample can be taken. Two options exist for taking the sample (the vertical axis in Figure 4.1): *immediate* and *polling*. An immediate approach inspects the executing thread(s) to determine which method(s) they are executing. The polling approach sets a bit in the VM that the executing threads check at well-defined points in the code, called *polling points*. When a polling point is executed with the bit set, a sample is taken.

Polling schemes are attractive for profiling because many VMs already insert polling points into compiled code (sometimes called *yieldpoints*) to allow stopping the executing thread(s) when system services need to be performed. The sampling profiler can piggyback on these existing polling points to collect profile data with essentially no additional overhead.

Polling points are a popular mechanism for stopping application threads because the VM often needs to enforce certain invariants when threads are stopped. For example, when garbage collection occurs, the VM needs to provide the collector with the set of registers, globals, and stack locations that contain pointers. Stopping threads via polling significantly reduces the number of program points at which these invariants must be maintained.

However, using polling as a sampling mechanism does have some shortcomings:

Not timely: Although the timer expires at regular intervals, there is some delay until the next polling point is reached. This is in addition to the delay imposed by timer-based profiling as described in the previous section. In particular, the trigger mechanism sets a bit in the VM to notify the VM that a sample needs to be taken. When the VM gets scheduled for execution, the sample is not taken immediately. The VM has to wait until the next polling point is reached before a sample can be taken.

Limited accuracy: Untimely sampling at polling points may also impact accuracy. For example, consider a polling point that occurs after a time-consuming operation that is not part of Java, such as an I/O operation or a call to native code. It is likely that the timer will expire during the time-consuming operation so unless the native code clears the sampling flag before returning (or the VM somehow ensures that it was never set), the next polling point executed in Java code will have an artificially high probability of being sampled. Es-

sentially, time spent in native code (which generally does not contain polling points) may be incorrectly credited to the caller Java method.

An additional source of inaccuracy is that some VMs, such as Jikes RVM, do not insert polling points in all methods and thus some methods cannot be sampled. For example, native methods (not compiled by the VM's compilers), small methods that are always inlined (polling point elided for efficiency reasons), and low-level VM methods do not have polling points.

Overhead: Polling requires the executing code to perform a bit-checking instruction followed by a conditional branch to sampling code. This bit-checking code must always be executed, regardless of whether the bit has been set. Reducing the number of bit-checking instructions can reduce this overhead (as Jikes RVM does for trivial methods), but it will also reduce the accuracy as mentioned above.

To avoid these limitations we advocate an immediate approach as described in Section 4.2.

4.1.2 Jikes RVM

As explained in Section 2.3.1, methods running on Jikes RVM are initially compiled by a baseline compiler, which produces unoptimized code quickly. An optimizing compiler is used to recompile important methods with one of its three optimization levels: 0, 1, and 2 [7, 8].

Jikes RVM uses timer-based sampling with polling for finding methods to be considered for optimization. Specifically, a timer-interrupt is used to set a bit in the virtual machine. On our platform, Linux/IA32, the default Jikes RVM system does this every 20 ms, which was the smallest level of granularity on Linux when that code was written several years ago. However, current Linux 2.6 kernels allow for a finer granularity of 4 ms by default.[1] Therefore, we also compare our work to an improved default system, where the timer-interrupt is smaller than 20 ms. Section 4.4 discusses the performance improvements obtained by reducing this interrupt value. This illustrates an important shortcoming of an OS timer-based approach: as new versions of the OS are used, the sampling code in the VM may need to be adjusted.

Jikes RVM provides two implementation choices for timer-based sampling: (i) nanosleep-polling, and (ii) setitimer-polling. The first strategy, which is the default, spawns a auxiliary, native thread at VM startup.

[1]The granularity provided by the OS is a tradeoff between system responsiveness and the overhead introduced by the OS scheduler. Hence, the timer granularity cannot be too small.

This thread uses the nanosleep system call, and sets a bit in the VM when awoken before looping to the next nanosleep call. The polling mechanism checks this bit. The second strategy, setitimer, initiates the timer interrupt handler at VM startup time to set a bit in the VM when the timer goes off. Setitimer does not require an auxiliary thread. In both cases, the timer resolution is limited by the operating system timer resolution.

When methods get compiled, polling instructions called *yield-points* are inserted into the method entries, method exits, and loop back edges. These instructions check to see if the VM bit is set, and if so, control goes to the Jikes RVM thread scheduler to schedule another thread. If the VM bit is not set, execution continues in the method at the instruction after the yield-point. Before switching to another thread, the system performs some simple profiling by recording the top method on the stack (for loop or exit yield-points) or the second method on top of the stack (entry yield-points) into a buffer. When N method samples have been recorded, an organizer thread is activated to summarize the buffer and pass this summary to the controller thread, which decides whether any recompilation should occur. The default value for N is 3.

The controller thread uses a cost/benefit analysis to determine if a sampled method should be recompiled [7, 8]. It computes the expected future execution time for the method at each candidate optimization level. This time includes the cost for performing the compilation and the time for executing the method in the future at that level. The compilation cost is estimated using the expected compilation rate as a function of the size of the method for each optimization level. The expected future execution time for a sampled method is assumed to be the amount of time the method has executed so far, scaled by the expected speedup of the candidate optimization level.[2] For example, the model assumes that a method that has executed for N seconds will execute for N more seconds divided by the speedup of the level compared to the current level. The system uses the profile data to determine the amount of time that has been spent in a method.

4.1.3 Hardware performance monitors

As explained in Section 3.1.2, modern processors are usually equipped with a set of performance event counter registers also known as hardware performance monitors (HPMs). The HPM hardware can be configured to count elapsed cycles, retired instructions, cache misses, etc. Besides simple counting, as used for MonitorMethod in Chapter 3, the hardware performance counter architecture can also be configured to generate an interrupt when a counter overflows. This interrupt can be converted to

[2]This speedup and the compilation rate are constants in the VM. They are currently obtained offline by measuring their values in the SPECjvm98 benchmark suite.

a signal that is delivered immediately to the process using the HPMs. This technique is known as event-based sampling. The HPM-sampling approach that we propose uses event-based sampling using the elapsed cycle count as its interrupt triggering event.

4.2 HPM-immediate sampling

This section describes our new sampling technique. It first discusses the merits of immediate sampling and then describes HPM-based sample triggering. When combined, this leads to HPM-immediate sampling, which we advocate in this work.

4.2.1 Benefits of immediate sampling

An advantage of using immediate sampling is that it avoids an often undocumented source of overhead that is present in polling-based profilers: the restrictions imposed on yield-point placement.

VMs often place yield-points on method prologues and loop backedges to ensure that threads can be stopped within a finite amount of time to perform system services. However, this placement can be optimized further without affecting correctness. For example, methods with no loops and no calls do not require a yield-point because only a finite amount of execution can occur before the method returns.

However, when using a polling-based profiler, removing yield-points impacts profile accuracy. In fact, yield-points need to be placed on method epilogues as well as prologues to make a polling-based sampler accurate; without the epilogue yield-points, samples that are triggered during the execution of a callee may be incorrectly attributed to the caller after the callee returns. For this reason, Jikes RVM places epilogue yield-points in all methods, except for the most trivial methods that are always inlined.

There are no restrictions on yield-point placement when using an immediate sampling mechanism. All epilogue yield-points can be removed, and the prologue and backedge yield-point placement can be optimized appropriately as long as it maintains correctness for the runtime system. The experimental results in Section 4.4 include a breakdown to show how much performance is gained by removing epilogue yield-points.

4.2.2 HPM-based sampling

HPM-based sampling relies on the hardware performance monitors sending the executing process a signal when a counter overflows. At VM

```
input          : CPU context
output         : none
begin
    registers ← GetRegisters (CPU context);
    processor ← GetJikesProcessorAddress ();
    if isJavaFrame (processor, registers) then
        stackFrame ← GetFrame (processor);
        methodID ← GetMethodID (stackFrame);
        sampleCount ← sampleCount + 1;
        samplesArray [sampleCount ] ← methodID;
    end
    HPMInterruptResumeResetCounter ()
end
```

Algorithm 1: HPM signal handler as implemented in JikesRVM, where the method ID resides on the stack.

startup, we configure a HPM register to count cycles, and we define the overflow threshold (the sampling interval or the reciprocal of the sample rate). We also define which signal the HPM driver should send to the VM process when the counter overflows.[3] Instead of setting a bit that can later be checked by the VM, as is done with polling, the VM acquires a sample immediately upon receiving the appropriate signal from the HPM driver. Several approaches can be used to determine the executing method. For example, the program counter can be used to determine the method in the VM's compiled code index. Alternatively, if the method ID is stored on the stack, as is done in Jikes RVM, the method ID can be read directly from the top of the stack. In our implementation, we take the latter approach, as illustrated in Algorithm 1: the state of the running threads is checked, the method residing on the top of the stack is sampled, and the method ID is copied in the sample buffer.

Because the executing method can be in any state, the sampler needs to check whether the stack top contains a valid Java frame; if the stack frame is not a valid Java frame, the sample is dropped. On average, less than 0.5% of all samples gathered in our benchmark suite using an immediate technique are invalid.

4.2.3 How HPM-immediate fits in the design space of sampling-based profiling

Having explained both immediate sampling and HPM-sampling, we can now better understand how the HPM-immediate sampling-based pro-

[3] Our implementation uses SIGUSR1; any of the 32 POSIX real-time signals can be used.

filing approach relates to the other sampling-based approaches in the design space. HPM-immediate sampling shows the following advantages over timer-based sampling: (i) sample points can be collected at a finer time granularity, (ii) performance is more robust across platforms with different clock frequencies and thread scheduling quanta, and (iii) it is more timely, i.e., a sample is taken immediately, not at the next thread scheduling point. Compared to polling-based sampling, HPM-immediate sampling (i) is more accurate and (ii) incurs less overhead.

Referring back to Figure 4.1, there are two design points in sampling-based profiling that we do not explore because they are not desirable: nanosleep-immediate and HPM-polling. The nanosleep-immediate approach does not offer any advantage over setitimer-immediate: to get a sample, nanosleep incurs an even larger delay compared to setitimer, as explained previously. The HPM-polling approach is not desirable because it would combine a timely trigger mechanism (HPM-sampling) with a non-timely sampling mechanism (polling).

4.2.4 HPM platform dependencies

HPMs are present on almost all modern microprocessors and can be used by applications if they are accessible from a user privilege level. Not all microprocessors offer the same HPM events, or expose them in the same way. For example, on IA-32 platforms, low overhead drivers provide access to the HPM infrastructure, but programming the counters differs for each manufacturer and/or processor model. One way in which a VM implementor can resolve these differences is by encapsulating the HPM subsystem in a platform-dependent dynamic library. In this way, HPM-sampling is portable across all common platforms.

Standardizing the HPM interfaces is desirable because it can enable better synergy between the hardware and the virtual machine. In many ways it is a chicken-and-egg problem; without concrete examples of HPMs being used to improve performance, there is little motivation for software and hardware vendors to standardize their implementations. However, we hope this and other recent work [1, 69, 79] will show the potential benefit of using HPMs to improve virtual machine performance.

Furthermore, collecting HPM data can have different costs on different processors or different microarchitectures. As our technique does not require the software to read the HPM counters, but instead relies on the hardware itself to track the counters and to send the executing process a signal only when a counter overflows, the performance of reading the counters does not affect the portability of our technique.

4.3 Experimental setup

Before evaluating the HPM-sampling approach, we first detail our experimental setup: the virtual machine, the benchmarks, and the hardware platforms.

4.3.1 Virtual machine

For the work in this chapter, we used the CVS version of Jikes RVM from April 10th, 2006. A motivation for the use of Jikes RVM is provided in Section 2.3.1. To ensure a fair comparison between HPM-immediate sampling with the other sampling-based profilers described in Section 4.1, we replace only the sampling-based profiler in Jikes RVM. The cost/benefit model for determining when to optimize to a specific optimization level and the compilers itself remain unchanged across all sampling-based profilers. In addition, we ensure that Jikes RVM's thread scheduling quantum remains unchanged at 20 ms across different sampling-based profilers with different sampling rates.

The experiments are run on a Linux 2.6 kernel. We use the `perfctr` tool version 2.6.19 [68] for accessing the HPMs.

4.3.2 Java applications

We use the SPECjvm98 benchmark suite [77], the DaCapo benchmark suite [17], and the PseudoJBB benchmark [76]. We run all SPECjvm98 benchmarks with the largest input set (`-s100`). For the DaCapo benchmark we use release version 2006-10. We use only the benchmarks that execute properly on our baseline system, the April 10th, 2006 CVS version of Jikes RVM. We consider 35 K transactions as the input to PseudoJBB.

Table 4.1 gives an overview of the benchmark applications that we used along with some additional information required to interpret the results in this chapter. The second column in Table 4.1 shows the number of application threads. The third column gives the number of application methods executed at least once; this does not include VM methods or library methods used by the VM. The fourth column gives the running time on our main hardware platform, the Athlon XP 3000+, using the default Jikes RVM configuration.

We consider two ways of evaluating performance, namely *one-run* performance and *steady-state* performance, and use the statistically rigorous performance evaluation methodology as described by Georges et al. [40]. For one-run performance, we run each application 11 times each in a new VM invocation, exclude the measurement of the first run, and then report average performance across the remaining 10 runs. We use a

Benchmark	Application threads	Methods executed	Running time (seconds)
compress	1	189	6.1
jess	1	590	2.9
db	1	184	12.1
javac	1	913	6.5
mpegaudio	1	359	5.8
mtrt	3	314	3.9
jack	1	423	3.3
antlr	1	1419	5.6
bloat	1	1891	14.2
fop	1	2472	6.1
hsqldb	13	1277	7.3
jython	1	3093	21.2
pmd	1	2117	14.8
PseudoJBB	1	812	6.6

Table 4.1: The benchmark characteristics for the default Jikes RVM configuration on the Athlon XP 3000+ hardware platform.

Student t-test with a 95% confidence interval to verify that performance differences are statistically meaningful. For SPECjbb2000 we use a single warehouse for measuring one-run performance.

For steady-state performance, we use a similar methodology, but instead of measuring performance of a single run, we measure performance for 50 consecutive iterations of the same benchmark in 11 VM invocations, of which the first invocation is discarded. Running a benchmark multiple times can be done easily for the SPECjvm98 and the DaCapo benchmarks using the running harness.

4.3.3 Hardware platforms

We consider two hardware platforms, both are AMD Athlon XP microprocessor-based computer systems. The important difference between both platforms is that they run at different clock frequencies and have different memory hierarchies, see Table 4.2. The reason for using two hardware platforms with different clock frequencies is to demonstrate that the performance of HPM-sampling is more robust across hardware platforms with different clock frequencies than other sampling-based profilers.

Processor	Frequency (Ghz)	L2 (KB)	RAM (GB)	Bus (Mhz)
1500+	1.33	256	1	133
3000+	2.1	512	2	166

Table 4.2: An overview of the hardware platforms used.

4.4 Evaluation

This section evaluates the HPM-immediate sampling profiler and compares it against existing sampling profilers. This comparison includes performance along two dimensions: one-run and steady-state, as well as measurements of overhead, accuracy, and stability.

4.4.1 Performance evaluation

We first evaluate the performance of the various sampling-based profilers — we consider both one-run and steady-state performance in the following two subsections.

One-run performance

Impact of sampling rate. Before presenting per-benchmark performance results, we first quantify the impact of the sample rate on average performance. Figure 4.2 shows the percentage average performance improvement on the Athlon XP 3000+ machine across all benchmarks as a function of the sampling interval compared to the default Jikes RVM, which uses a sampling interval of 20 ms. The horizontal axis varies the sampling interval from 0.1 ms to 40 ms for the nanosleep- and setitimer-sampling approaches. For the HPM-sampling approach, the sampling interval varies from 100 K cycles up to 90 M cycles. On the Athlon XP 3000+, this is equivalent to a sampling interval varying from 0.047 ms to 42.85 ms. Curves are shown for all four sampling-based profilers; for the immediate-sampling methods, we also show a version including and excluding epilogue yield-points to help quantify the reason for performance improvement. Because an immediate approach does not require any polling points, the preferred configuration for the immediate sampling approach is with no yield-points.

We make several observations from Figure 4.2. First, comparing the setitimer-immediate versus the setitimer-polling curves clearly shows that an immediate sampling approach outperforms a polling-based sampling mechanism on our benchmark suite. The setitimer-immediate curve achieves a higher speedup than setitimer-polling over the en-

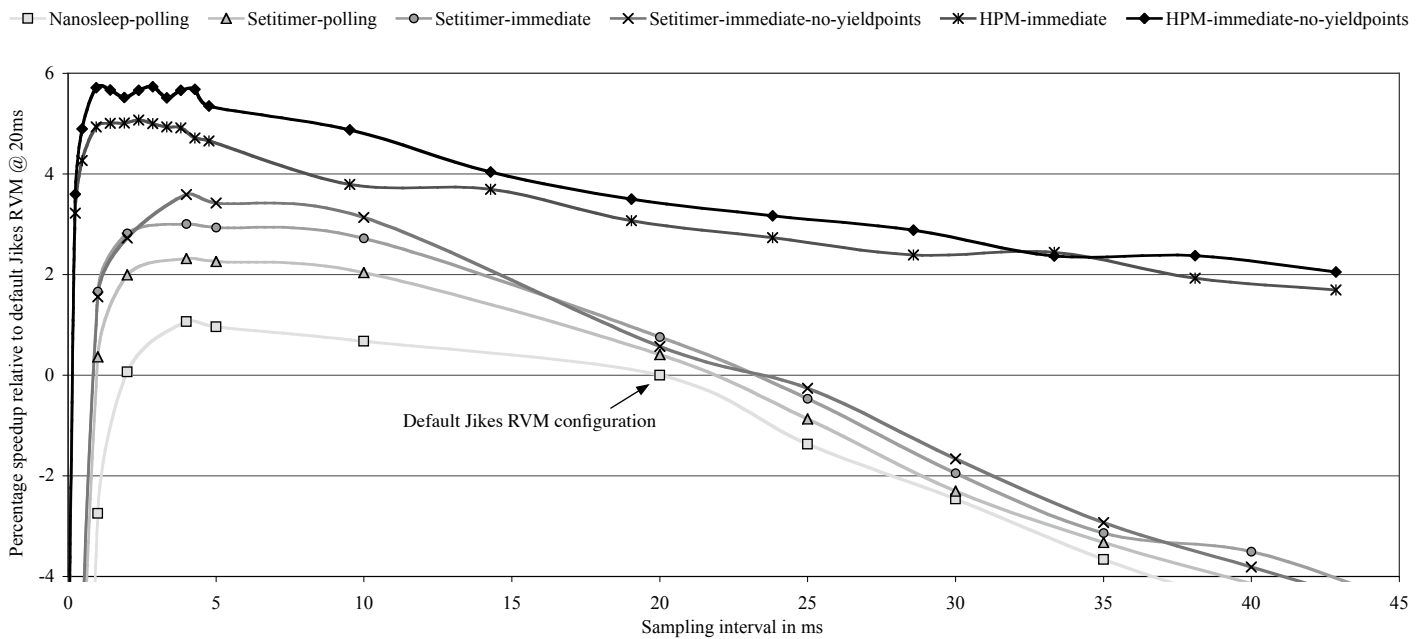


Figure 4.2: The percentage average performance speedup on the Athlon XP 3000+ machine for the various sampling profilers as a function of sample rate, relative to default Jikes RVM, which uses a sampling interval of 20 ms.

tire sample rate range. Second, HPM-based sampling outperforms OS-triggered sampling — the HPM-immediate curve achieves higher speedups than setitimer-immediate. Third, removing epilogue yield-points yields a slight performance improvement for both the HPM-based and OS-triggered immediate sampling approaches. So, in summary the overall performance improvement for the HPM-sampling approach that we advocate comes from three sources: (i) HPM-based triggering instead of OS-triggered sampling, (ii) immediate sampling instead of polling-based sampling, and (iii) the removal of epilogue yield-points.

Each sampling-based profiler has a best sample rate for our benchmark suite. Values below this rate result in too much overhead. Values above the rate result in a less accurate profile. We use an interval of 9 M cycles (approximately 4.3 ms) for the HPM-immediate approach in the remainder of this chapter.[4] Other sampling-based profilers achieve their best performance at different sample rates — in all other results presented in this chapter, we use the best sample rate *per sampling-based profiler*. For example, for the setitimer-immediate approach with no yield-points the best sampling interval on our benchmark suite is 4 ms. The default Jikes RVM with nanosleep-polling has a sampling interval of 20 ms.

Per-benchmark results. Figure 4.3 shows the per-benchmark percentage performance improvements of all sampling profiler approaches (using each profiler's best sample rate) relative to the default Jikes RVM's nanosleep-polling sampling approach, which uses a sampling interval of 20 ms. The graph at the top in Figure 4.3 is for the best sample rates on the Athlon XP 1500+ machine. The graph at the bottom is for the best sample rates on the Athlon XP 3000+.

The results in Figure 4.3 clearly show that HPM-immediate sampling significantly outperforms the other sampling profiler approaches. In particular, HPM-immediate results in an average 5.7% performance speedup compared to the default Jikes RVM nanosleep-polling approach on the Athlon XP 3000+ machine and 3.9% on the Athlon XP 1500+ machine. HPM-immediate sampling results in a greater than 5% performance speedup for many benchmarks (on the Athlon XP 3000+): antlr (5.9%), mpegaudio (6.5%), jack (6.6%), javac (6.6%), hsqldb (7.8%), jess (9.6%) and mtrt (18.3%).

As mentioned before, this overall performance improvement comes from three sources. First, immediate sampling yields an average 3.0%

[4] As Figure 4.2 shows, we explored a wide range of values between 2 M and 9 M for the HPM-immediate approach. An ANOVA and a Tukey HSD post hoc [40, 64] test with a confidence level of 95% reveal that in only 1.5% of the cases (7 out of 468 comparisons), the execution times differ significantly. This means one can use any of the given rates in [2 M; 9 M] without suffering a significant performance penalty. Therefore, we chose 9 M as the best value for HPM-immediate.

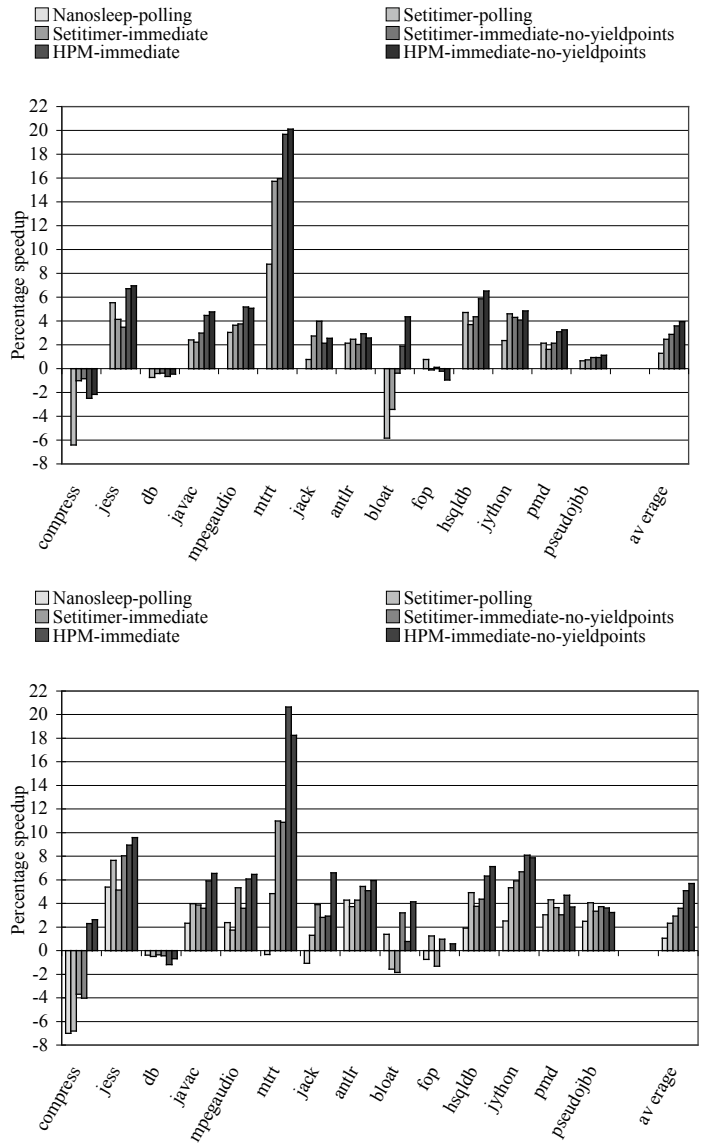


Figure 4.3: The percentage performance improvement relative to the default Jikes RVM configuration (nanosleep-polling with a 20 ms sampling interval). Each configuration uses the single best sampling interval as determined from the data in Figure 4.2 for all benchmarks. The graphs give improvement on the Athlon XP 1500+ with a 20 M sample interval (top) and the Athlon XP 3000+ with a 9 M sample interval (bottom). On the former, the nanosleep-polling bars show no improvement, because the default configuration rate performed best for that particular technique; as a result, the left graph has one bar less.

speedup over polling-based sampling. Second, HPM-sampling yields an additional average 2.1% speedup over OS-triggered sampling. Third, eliminating the epilogue yield-points contributes an additional 0.6% speedup on average. For some benchmarks, removing the epilogue yield-points results in significant performance speedups, for example jack (4.1%) and bloat (3.4%) on the Athlon XP 3000+ machine.

Statistical significance. Furthermore, these performance improvements are statistically significant. We use a one-sided Student t-test with a 95% confidence level following the methodology proposed by Georges et al. [40] to verify that HPM-immediate-no-yieldpoints does result in a significant performance increase over the non-HPM techniques. For each comparison, we require one test where the null hypothesis is that both compared techniques result in the same execution time on average, the alternative hypothesis is that HPM-immediate-no-yieldpoints has a smaller execution time. The null hypothesis is rejected for 10 out of 14 benchmarks when we compare to nanosleep-polling; it is rejected for 8 out of 14 benchmarks when we compare to setitimer-immediate-no-yieldpoints; and it is rejected for even 5 out of 14 benchmarks when we compare to HPM-immediate. This means that HPM-immediate-no-yieldpoints outperforms the best execution times compared to the other techniques.

Robust performance across machines. The two graphs in Figure 4.3 also show that the HPM-immediate sampling profiler achieves higher speedups on the Athlon XP 3000+ machine than on the 1500+ machine. This observation supports our claim that HPM-sampling is more robust across hardware platforms, with potentially different clock frequencies. The reason is that OS-triggered profilers collect relatively fewer samples as clock frequency increases (assuming that the OS time quantum remains unchanged). As such, the profile collected by an OS-triggered profiler becomes relatively less accurate compared to HPM-sampling when clock frequency increases.

Steady-state performance

We now evaluate the steady-state performance of HPM-sampling for long-running Java applications. This is done by iterating each benchmark 50 times in a single VM invocation. This process is repeated 11 times (11 VM invocations of 50 iterations each); the first VM invocation is a warmup run and is discarded [40].

Figure 4.4 shows the average execution time per benchmark iteration across all VM invocations and all benchmarks. Two observations can be

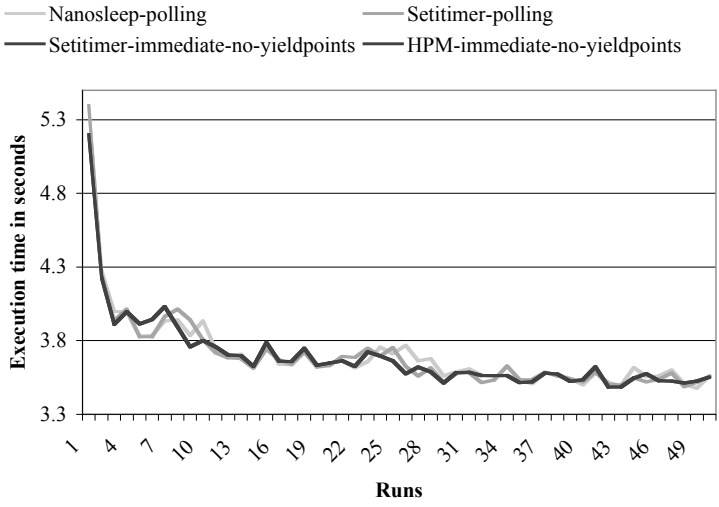


Figure 4.4: Quantifying steady-state performance of HPM-immediate-no-yieldpoints sampling: average execution time per run for 50 consecutive runs.

made: (i) it takes several benchmark iterations before we reach steady-state behavior, i.e., the execution time per benchmark iteration slowly decreases with the number of iterations, and (ii) while HPM-immediate is initially faster, the other sampling mechanisms perform equally well for steady-state behavior. This suggests that HPM-immediate is faster at identifying and compiling hot methods, but that if the hot methods of an application are stable and the application runs long enough, the other mechanisms also succeed at identifying these hot methods. Once all important methods have been fully optimized, no mechanism has a significant competitive advantage over the other, and they exhibit similar behavior. In summary, HPM-sampling yields faster one-run times and does not hurt steady-state performance. Nevertheless, in case a long-running application would experience a phase change at some point during the execution, we believe HPM-immediate will be more responsive to this phase change by triggering adaptive recompilations more quickly.

4.4.2 Analysis

To get better insight into why HPM-sampling results in improved performance, we now present a detailed analysis of the number of method recompilations, the optimization level these methods reach, the overhead of HPM-sampling, the accuracy of a sampling profile, and the stability of

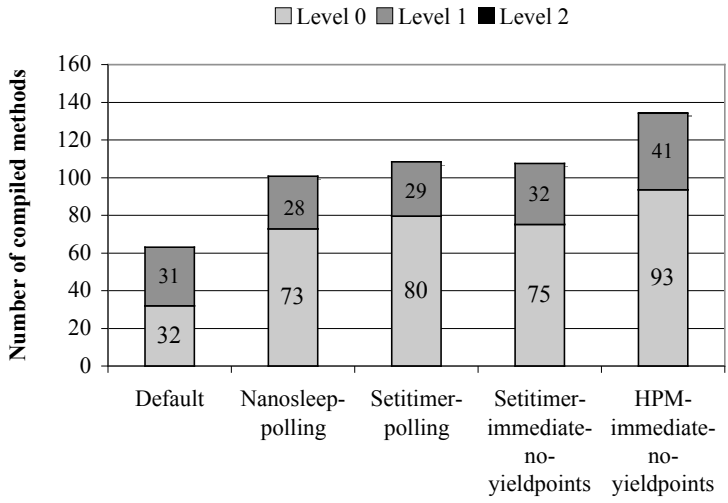


Figure 4.5: The average number of method recompilations by optimization level across all benchmarks on the Athlon XP 3000+.

HPM-sampling across multiple runs.

Recompilation activity

Table 4.3 shows a detailed analysis of the number of methods sampled, the number of methods presented to the analytical cost/benefit model, and the number of method recompilations for the default system and the four sampling-based profilers. The difference between default and nanosleep-polling is that the default system uses 20 ms as the sleep interval, whereas nanosleep-polling uses the best sleep interval for our benchmark suite, which is 4 ms. Table 4.3 shows that HPM-sampling collects more samples (2066 versus 344 to 1249, on average) and also presents more samples to the analytical cost/benefit model (594 versus 153 to 408, on average) than OS-triggered sampling. This also results in more method recompilations (134 versus 63 to 109, on average).

Figure 4.5 provides more details by showing the number of method recompilations by optimization level. These are average numbers across all benchmarks on the Athlon XP 3000+ as reported by Jikes RVM's logging system. This figure shows that HPM-immediate sampling results in more recompilations, and that more methods are compiled to higher levels of optimization, i.e., compared to nanosleep polling, HPM-immediate compiles 27% more methods to optimization level 0 and 46% more meth-

	Default			Nanosleep-polling			Setitimer-polling			Setitimer-immediate			HPM-immediate		
Benchmark	S	P	C	S	P	C	S	P	C	S	P	C	S	P	C
compress	258	162	16	1010	173	15	1190	195	17	977	288	19	1536	437	23
jess	126	82	35	425	191	45	499	216	47	416	199	49	714	299	55
db	530	56	9	1950	87	9	2340	92	9	1959	159	9	3349	189	10
javac	251	147	80	893	447	139	1048	510	159	1004	426	150	1647	620	195
mpegaudio	250	199	58	900	453	75	1065	526	80	906	634	81	1489	908	98
mtrt	128	98	33	460	204	54	529	230	58	538	281	63	847	382	82
jack	140	56	27	510	162	41	585	192	47	451	110	22	758	185	32
antlr	249	120	60	855	339	100	1040	402	112	750	269	105	1263	403	138
bloat	601	150	60	2145	404	111	2595	490	117	2255	329	111	3733	486	144
fop	273	60	37	995	199	63	1175	230	70	576	185	74	957	275	103
hsqldb	247	180	91	869	478	133	995	525	138	1117	500	152	1816	747	181
jython	928	426	203	3294	1079	342	3813	1220	357	3169	1145	350	5178	1667	426
pmd	565	244	93	1995	623	154	2355	721	165	2355	704	158	3938	989	206
PseudoJBB	271	161	85	950	451	129	1114	521	146	1014	489	152	1697	729	190
Average	344	153	63	1232	378	101	1453	433	109	1249	408	107	2066	594	134

Table 4.3: Detailed sample profile analysis for one-run performance on the Athlon XP 3000+: 'S' stands for the number of method samples taken, 'P' stands for the number of methods that have been proposed to the analytical cost/benefit model and 'C' stands for the number of method recompilations.

ods to optimization level 1.

Figure 4.3 showed that HPM-immediate sampling resulted in an 18.3% performance speedup for mtrt. To better understand this speedup we performed additional runs of the baseline and HPM-immediate configurations with Jikes RVM's logging level set to report the time of all recompilation events.[5] We ran each configuration 11 times and the significant speedup of HPM, relative to the baseline configuration, occurred on all runs, therefore, we conclude that the compilation decisions leading to the speedup were present in all 11 HPM-immediate runs. There were 34 methods that were compiled by the HPM-immediate configuration in all runs, and of these 34 methods, only 13 were compiled on all baseline runs. Taking the average time at which these 13 methods were compiled, HPM compiles these methods 28% sooner, with a maximum of 47% sooner and a minimum of 3% sooner. Although this does not concretely explain why HPM-immediate is improving performance, it does show that HPM-immediate is more responsive in compiling the important methods, which most likely explains the speedup.

Overhead

Collecting samples and recompiling methods obviously incurs overhead. To amortize this cost, dynamic compilation systems need to balance code quality with the time spent sampling methods and compiling them. To evaluate the overhead this imposes on the system, we investigate its two components: (i) the overhead of collecting the samples and (ii) the overhead of consuming these samples by the adaptive optimization system. As explained in Section 4.1.2, the latter is composed of three parts implemented as separate threads in Jikes RVM: (a) the organizer, (b) the controller, and (c) the compiler.

To identify the overhead of only collecting samples, we modified Jikes RVM to discard samples after they have been captured in both an HPM-immediate configuration and the baseline nanosleep-polling configuration. In both configurations no samples are analyzed and no methods are recompiled by the optimizing compiler. By comparing the execution times from the HPM-immediate configuration with those of the default Jikes RVM configuration that uses nanosleep-polling with the 20 ms sample interval, we can study the overhead of collecting HPM samples. Figure 4.6 shows this overhead for a range of HPM sample intervals averaged across all benchmarks; at the sampling interval of 9 M cycles, the overhead added by HPM-immediate sampling is limited to 0.2%.

To identify the overhead of processing the samples we look at the time spent in the organizer, controller, and compiler. Because each of these

[5] This logging added a small amount of overhead to each configuration, but the speedup remained about the same.

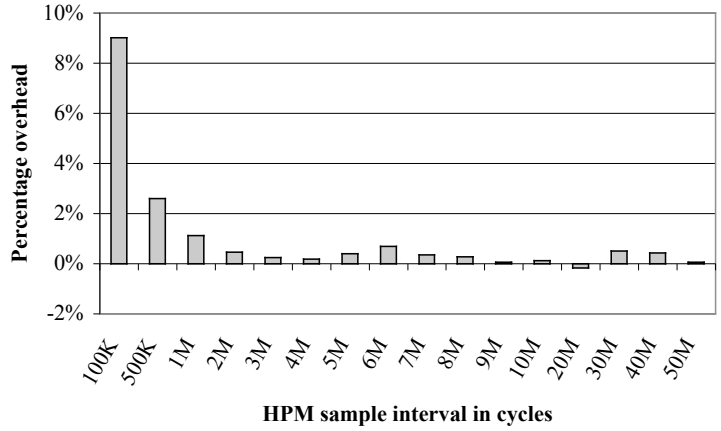


Figure 4.6: The average overhead through HPM-immediate-no-yieldpoints sampling for collecting samples at various sample intervals.

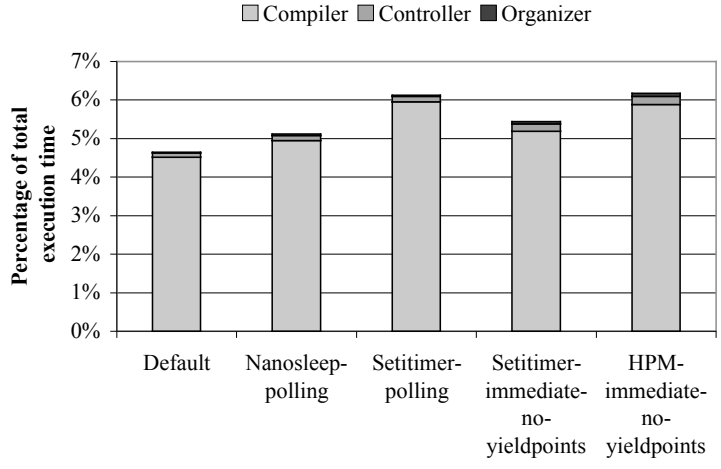


Figure 4.7: The average overhead of consuming the samples across all benchmark. The default systems uses 20 ms as the sample interval, where as the other systems use their best sample intervals for our benchmark suite.

subsystems runs in their own thread, we use Jikes RVM's logging system to capture each thread's execution time. Figure 4.7 shows the fraction of the time spent in the organizer, controller, and the compiler for all sampling profiler approaches averaged across all benchmarks. The results in Figure 4.7 are consistent with the results from MonitorMethod in Table 3.5. Based on Figure 4.7, we conclude that although HPM-immediate collects many more samples, the overhead of the adaptive optimization system increases by only roughly 1% (mainly due to the compiler). This illustrates that Jikes RVM's cost/benefit model successfully balances code quality with time spent collecting samples and recompiling methods — even when presented with significantly more samples as discussed in the previous section. We believe this property is crucial for the applicability of HPM-immediate sampling.

Accuracy

To assess the accuracy of the profile collected through HPM-immediate sampling, we would like to compare the various sampling approaches with a perfect profile. A perfect profile satisfies the property that the number of samples for each method executed is perfectly correlated with the time spent in that method. Such a profile cannot be obtained, short of doing complete system simulation. Instead, we obtain a detailed profile using a frequent HPM sample rate (a sample is taken every 10 K cycles) and compare the profiles collected through the various sampling profiler approaches with this detailed profile.

We determine accuracy as follows. We run each benchmark 30 times in a new VM instance for each sampling approach (including the detailed profile collection) and capture how often each method is sampled in a profile. A profile is a vector where each entry represents a method along with its sample count. We subsequently compute an average profile across these 30 benchmark runs. We then use the metrics described below to determine the accuracy for a sampling approach by comparing its profile with the detailed profile. We use both an unweighted and a weighted metric.

Unweighted metric. The unweighted metric gives the percentage of methods that appear in both vectors, regardless of their sample counts. For example, the vector $x = ((a,5),(b,0),(c,2))$ and the vector $y = ((a,30),(b,4),(c,0))$ have corresponding *presence vectors* $p_x = (1,0,1)$ and $p_y = (1,1,0)$, respectively. The *common presence vector* then is $p_{\text{common}} = (1,0,0)$. Therefore, p_{common} has a score of .33, which is the sum of its elements divided by the number of elements.

This metric attempts to measure how many methods in the detailed profile also appear in the profiles for the sampling approach of interest.

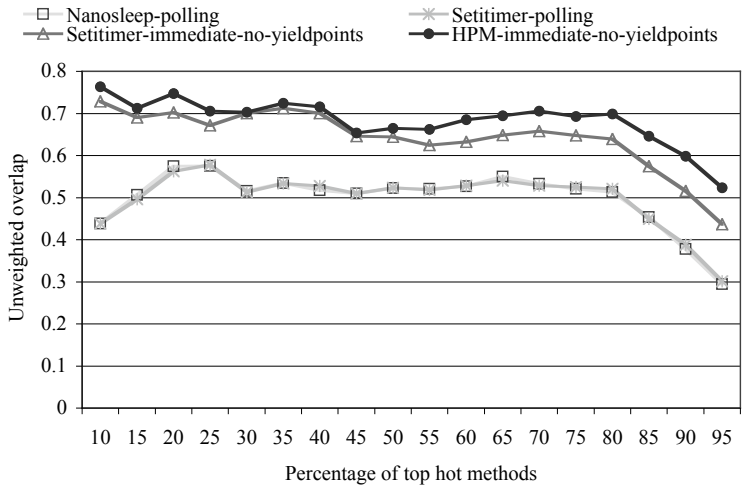


Figure 4.8: The sampling accuracy using the unweighted accuracy metric: the average overlap with the detailed profile is shown on the vertical axis for the top N percent of hot methods on the horizontal axis.

Because the metric ignores how often a method is sampled in the detailed profile, it seems appropriate to consider only the top N percent of most frequently executed methods in the detailed profile.

Figure 4.8 shows the average unweighted metric for the various sampling-based profilers for the top N percent of hot methods. The horizontal axis shows the value of N. The vertical axis shows the unweighted metric score, so higher is better for this metric. For example, the accuracy of the profilers finding the top 20% of methods found in the detailed profile is 57.4%, 56.3%, 70.2%, and 74.5% for nanosleep-polling, setitimer-polling, setitimer-immediate-no-yieldpoints, and HPM-immediate-no-yieldpoints, respectively. Immediate sampling techniques are clearly superior to polling-based techniques on our benchmark suite.

Weighted metric. The weighted metric considers the sample counts associated with each method and computes an overlap score. To determine the weighted overlap score, we first normalize each vector with respect to the total number of samples taken in that vector. This yields two vectors with relative sample counts. Taking the element-wise minimum of these vectors gives the weighted *presence* vector. The score then is the sum of the elements in the *presence* vector. For example, for x and y as defined above, the score is 0.71. Indeed, the only element that has samples in both vectors, a, scores $\frac{5}{5+0+2} = 0.71$ in x and $\frac{30}{30+4+0} = 0.88$ in y.

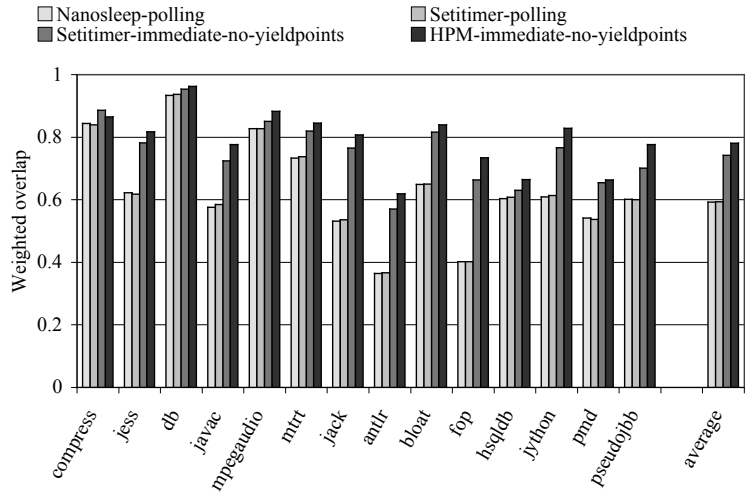


Figure 4.9: The accuracy using the weighted metric of various sampling-based profilers compared to the detailed profile.

Figure 4.9 demonstrates the accuracy using the weighted metric for all benchmarks. This graph shows that polling-based sampling achieves the lowest accuracy on average (59.3%). The immediate techniques score much better, attaining 74.2% on average for setitimer. HPM improves this even further to a 78.0% accuracy on average compared to the detailed profile.

Stability

It is desirable for a sampling mechanism to detect hot methods in a consistent manner, i.e., when running a program twice, the same methods should be sampled (in the same relative amount) and optimized so that different program runs result in similar execution times. We call this *sampling stability*. To evaluate stability, we perform 30 runs holding the sampling mechanism constant. We use the weighted metric described in the previous section to pairwise compare all the vectors for the different benchmark runs with the same sampling mechanism. We report the stability score as the mean of these values.

For example, given three vectors

$$x = ((a, 5), (b, 1), (c, 4))$$
$$y = ((a, 6), (b, 0), (c, 3))$$
$$z = ((a, 5), (b, 2), (c, 3))$$

that correspond to three benchmark runs of a particular configuration, the stability is computed as follows. First, we normalize the vectors and take the element-wise minimum of all the different combinations of vectors. Comparing x and y yields $((a, \frac{1}{2}), (b, 0), (c, \frac{1}{3}))$, comparing x and z yields $((a, \frac{1}{2}), (b, \frac{1}{10}), (c, \frac{3}{10}))$ and comparing y and z results in $((a, \frac{1}{2}), (b, 0), (c, \frac{3}{10}))$. Next, we compute the sum of the elements in each of the vectors and compute the final stability score as the mean of these values. That is, $\frac{0.83+0.9+0.8}{3} = 0.84$.

Figure 4.10 compares the stability of the following five configurations: (i) nanosleep-polling, (ii) setitimer-polling, (iii) setitimer-immediate, (iv) HPM-immediate, and (v) the HPM configuration with a sample rate of 10 K cycles, i.e., the detailed profile. The detailed profile is very stable (on average the stability is 95.1%). On average, nanosleep-polling and setitimer-polling have a stability score of 75.9% and 76.6%, respectively. The average stability for setitimer-immediate is 78.2%, and HPM-immediate has the best stability score (83.3%) of the practical sampling approaches.

4.5 Related work

This section describes related work. We focus mostly on profiling techniques for finding important methods in language-level virtual machines. We briefly mention other areas related to online profiling in dynamic optimization systems.

4.5.1 Existing virtual machines

As mentioned earlier, Jikes RVM [7] uses a compile-only approach to program execution with multiple recompilation levels. All recompilation is driven by a polling-based sampler that is triggered by an operating system timer.

BEA's JRockit [15, 73] VM also uses a compile-only approach. A sampler thread is used to determine methods to optimize. This thread suspends the application threads and takes a sample at regular intervals. Although full details are not publicly available, the sampler seems to be triggered by an operating system mechanism. It is not clear if the samples

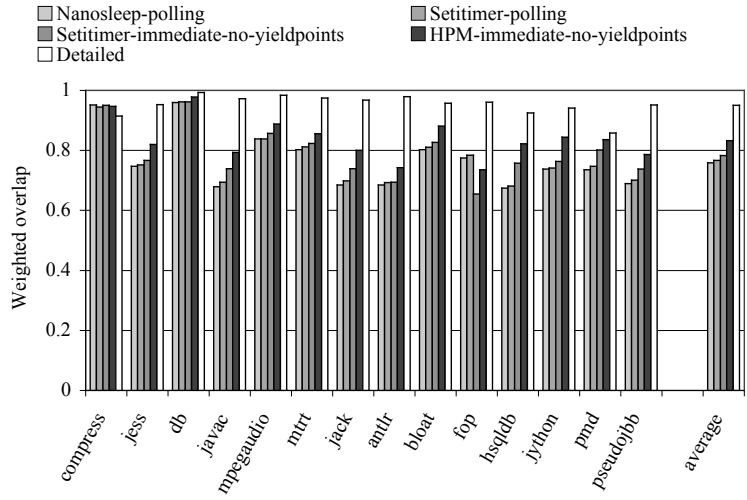


Figure 4.10: Quantifying sampling stability: higher is better.

are taken immediately or if a polling approach is used. Friberg's MS thesis [39] extends JRockit to explore the use of HPM events on an Itanium 2 to better perform specific compiler optimizations in a JIT. The thesis reports that using HPM events improves performance by an average of 4.7%. This work also reports that using HPM events to drive only recompilation (as we advocate in this work) does not improve performance, but does compile fewer methods. In a workshop presentation, Eastman et al. [34] report similar work to Friberg's work. The slides claim to extend JRockit to use an HPM-immediate approach using Itanium 2 events to drive optimization. Unlike Friberg they do not report how the new approach compares to existing approach for finding optimization candidates, but instead focus on how HPM events improve performance when used by compiler optimizations.

Whaley [86] describes a sampling profiler that is implemented as a separate thread. The thread awakes periodically and samples the application threads for the purpose of building a dynamic call graph. He mentions that this sampling thread could be triggered by operating system or processor mechanisms, but used an operating system sleep function in his implementation. No performance results are reported comparing the various trigger approaches. The IBM DK for Java [80] interprets methods initially and uses method entry and back edge counters to find methods to optimize. Multiple optimization levels are used. A variant of the sampling technique described by Whaley (without building the dynamic call graph) is used to detect compiled methods that require further optimiza-

tion.

IBM's J9 VM [61] is similar to the IBM DK for Java in that it is also interpreter-based with multiple optimization levels. It uses a sampling thread to periodically take samples. The implementation of the sampler is similar to Jikes RVM in that it uses a polling-based approach that is triggered by an operating system timer.

Intel's ORP VM [24] employs a compile-only strategy with one level of recompilation. Both the initial and the optimizing compiler use per-method counters to track important methods. A method is recompiled when the counter passes a threshold or when a separate thread finds a method with a counter value that suggests compiling it in the background. Optimized methods are not recompiled. No sampling is used.

In a technical report, Tam et al. [83] extend Intel's ORP to use HPMs to trigger dynamic code recompilation. They instrument method prologues and epilogues to read the HPM cycle counter. The cycle counter value is read upon invocation of a method and upon returning from the method, and used to compute the time spent in each method. These deltas are accumulated on a per method basis, and when a method's accumulated time exceeds a given threshold (and that method has been executed at least two times), the method is recompiled at the next optimization level. They report large overheads, and conclude that this technique cannot be used in practice. Our approach is different in that it does not use instrumentation and relies on sampling to find candidate methods.

Sun's HotSpot VM [67] interprets methods initially and uses method entry and back edge counters to find methods to optimize. No published information is available on what, if any, mechanism exists for recompiling optimized methods and if sampling is used. However, HotSpot was greatly influenced by the Self-93 system [47], which used a compile-only approach triggered by decayed invocation counters. Optimized methods were not further optimized at higher levels.

In summary, no production VMs use HPMs as a trigger mechanism for finding optimization candidates. Two descriptions of extending JRockit to use HPMs do exist in the form of MS thesis and a slides-only workshop. In contrast to our work, neither showed any improvement using this approach and no comprehensive evaluation was performed.

4.5.2 Other related work

Lu et al. [55] describe the ADORE dynamic binary optimization system, which uses event-sampling of HPMs to construct a path profile that is used to select traces for optimization. At a high level this is a similar approach to what we advocate, HPM profiling for finding optimization candidates, but details of the optimization system (a binary optimizer versus

a virtual machine) are quite different.

Adl-Tabatabai et al. [1] used HPMs as a profiling mechanism to guide the placement of prefetch instructions in the ORP virtual machine. We believe prefetching instructions are inserted by forcing methods to be recompiled. Although this work may compile a method more than once based on HPM information, it does not rely on HPMs as a general profiling mechanism to find optimization candidates. Su and Lipasti [79] describe a hybrid hardware-software system that relies on hardware support for detecting exceptions in specified regions of code. The virtual machine then performs speculative optimizations on these guarded region. Schneider et al. [69] show how hardware performance monitors can be used to drive locality-improving optimizations in a virtual machine. Although these works are positive examples of how hardware can be used to improve performance in a virtual machine environment, neither address the specific problem we address: finding candidates for recompilation.

Ammons et al. [4] show how HPMs can be incorporated into an offline path profiler. Andersen et al. [5] describe the DCPI profiler, which uses HPMs to trigger a sampling approach to understand program performance. None of these works use HPMs to find optimization candidates.

Zhang et al. [87] describe the Morph Monitor component of an online binary optimizer that uses an operating system kernel extension to implement an immediate profiling approach. Duesterwald et al. [30] describe a low-overhead software profiling scheme for finding hot paths in a binary optimization systems. Although both works are tackling the problem of finding optimization candidates, neither use HPMs.

Merten et al. [62] propose hardware extensions to monitor branch behavior for runtime optimization at the basic block and trace stream level. Conte et al. [26] describe a hardware extension for dedicated profiling. Our work differs in that we use existing hardware mechanisms for finding hot methods.

4.6 Conclusion

To our knowledge, this is the first work to empirically evaluate the design space of several sampling-based profilers for dynamic compilation. We describe a technique, HPM-sampling, that leverages hardware performance monitors (HPMs) to identify methods for optimization. In addition, we show that an immediate sampling approach is significantly more accurate in identifying hot methods than a polling-based approach. Furthermore, an immediate sampling approach allows for eliminating epilogue yield-points. We show that, when put together,

HPM-immediate sampling with epilogue yield-point elimination outperforms all other sampling techniques. Compared to the default Jikes RVM configuration, we report a performance improvement of 5.7% on average and up to 18.3% without modifying the compiler. Moreover, we show that HPM-based sampling consistently improves the accuracy, robustness, and stability of the collected sample data.

These result support the central thesis of this PhD dissertation; in order to improve the runtime performance of Java applications, the ability to capture more complete profiles is key. By taking advantage of information captured at the lowest level of the execution stack, we can collect profile information that is more accurate and faster to obtain.

Chapter 5

Conclusion

Virtual execution environments are here to stay. As a result, an increasing number of applications run on top of execution stacks that have multiple layers. The objective of this thesis was to investigate how information collected at different levels of an execution stack is helpful in understanding and optimizing program performance. This is important, because the abstractions and virtualizations introduced by virtual execution environments, come with some trade-offs. In this work, we focus on the Java programming language and its Java Virtual Machine. Two major trade-offs of virtual execution environments have been identified.

First, understanding the behavior of Java applications is difficult because it involves a complex interaction between the different components of the VM (interpreter, run-time compiler, garbage collection), the application, and the underlying hardware. Furthermore, the applications themselves are growing in size and complexity and often run on top an application framework. So while it is great that virtual execution environments like Java shield developers from as much details as possible, it makes it increasingly difficult to conduct thorough program analysis.

Second, abstraction and virtualization layers are often at odds with the ability to achieve high performance. For example, the fact that a JVM operates on machine-independent instructions rather than native machine instructions, makes it an interesting challenge to achieve high performance.

In this thesis, we show that both problems benefit from advances in profiling techniques. Better offline profiling tools help us provide better insight in the behavior of Java applications, and novel online profiling mechanisms can help us improve performance of Java applications. The thesis of this PhD dissertation is that in order to understand and optimize the performance of Java applications, the ability to capture complete profiles is key.

Specifically, this PhD dissertation presented three novel profiling techniques.

First, we presented Javana, a framework that makes it easy to build vertical profiling tools. We illustrated how Javana combines information gathered at different layers of the execution stack to help application developers and researchers gain better understanding of Java applications.

Next, we presented MonitorMethod, a novel profiling tool that exploits phase behavior to link microprocessor-level information to the methods in a Java application. We showed how this information can be used by application developers to identify and explain performance bottlenecks in terms of method-level phases.

Finally, in Chapter 4 we proposed HPM-based sampling, a technique that leverages hardware performance monitors to identify methods for optimization. We showed that by propagating information captured by the hardware to the JVM, we can identify hot methods faster and more accurately than a polling-based approach implemented solely in software. Compared to existing profiling techniques implemented in the JVM, our HPM-based sampling technique improves performance by 5.7% on average and up to 18.4%.

The central attribute of these contributions is that we link information gathered at the microprocessor-level to other layers in the execution stack to gather more complete profiles. By taking advantage of information captured at the lowest level of the execution stack, we can collect profile information that is more accurate, faster to obtain, or that was otherwise not available.

5.1 Future work

We believe there are a number of potentially interesting directions for future research. Our work has demonstrated that there is potential for better synergy between the hardware and virtual machines. Both try to exploit a program's execution behavior, but little synergy has been demonstrated to date. This is partially due to the narrow communication channel between hardware and software, as well as the lack of cross-subdiscipline innovation in these areas. We hope this work will encourage others to explore this fruitful area of greater synergy between hardware and software (and system software like VMs in particular).

First, for our HPM-based sampling technique we used the cycle counter event to sample methods. Other HPM events may also be useful for identifying methods for optimization, such as cache miss count events and branch misprediction count events. Second, it may be interesting to apply the improved accuracy of HPM-based sampling to other parts of

the adaptive optimization system, such as dynamic call graph profiling used for method inlining.

Second, our work on MonitorMethod uses an offline phase detection step. If we'd be able to detect method-level phase behavior online – for example with dedicated hardware – we could eliminate the required training run and make MonitorMethod an online tool. Being able to detect online phase behavior is useful not only for gaining better insight in the behavior of Java applications, but can potentially be used to improve performance of Java applications. Specifically, we believe it might be useful to link different compilation plans to phases with different behavior as observed at the microarchictectural level. Similarly, our work on Javana could also benefit from being able to offload some of the book keeping or computational work to hardware.

Appendix A

Other contributions

In addition to the work that is covered in this thesis, we also contributed to the following papers.

A.1 Statistically rigorous Java performance evaluation

Java performance is far from being trivial to benchmark because it is affected by various factors such as the Java application, its input, the virtual machine, the garbage collector, the heap size, etc. In addition, non-determinism at run-time causes the execution time of a Java program to differ from run to run — and the variability can be fairly high.

There exist a wide variety of Java performance evaluation methodologies used by researchers and benchmarkers. All of these methodologies differ from each other in a number of ways. Some report average performance over a number of runs of the same experiment; others report the best or second best performance observed; yet others report the worst. Some iterate the benchmark multiple times within a single VM invocation; others consider multiple VM invocations and iterate a single benchmark execution; yet others consider multiple VM invocations and iterate the benchmark multiple times. Some eliminate the non-determinism due to JIT compilation by fixing the compiler work for each benchmarking experiment.

We showed that prevalent methodologies can be misleading, and can even lead to incorrect conclusions. We presented a survey of existing Java performance evaluation methodologies and discussed the importance of statistically rigorous data analysis for dealing with non-determinism. We advocated approaches to quantify startup as well as steady-state perfor-

mance, and, in addition, we provided JavaStats, software to automatically obtain performance numbers in a rigorous manner.

This work is described in:

1. *Statistically Rigorous Java Performance Evaluation*, Andy Georges, Dries Buytaert, Lieven Eeckhout. In Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '07), Montreal, Canada, October 2007.

A.2 Garbage collection in hard- and software

During codesign of a system, one still runs into the impedance mismatch between the software and hardware worlds. We identified the different levels of abstraction of hardware and software as a major culprit of this mismatch. For example, when programming in high-level object-oriented languages like Java, one has objects, methods, memory management, etc. that facilitates development but these have to be largely abandoned when moving the same functionality into hardware.

As a solution we proposed a design that is able to bridge the gap by providing the same capabilities to hardware components as to software components. This seamless integration is achieved by introducing an architecture and protocol that allow reconfigurable hardware and software to communicate with each other in a transparent manner, i.e. no component of the design needs to be aware whether other components are implemented in hardware or in software.

Part of the design is a novel technique that allows reconfigurable hardware to manage dynamically allocated memory. This is achieved by allowing the hardware to hold references to objects and by modifying the garbage collector of the virtual machine to be aware of these references in hardware.

This work is described in:

1. *FPGA-aware garbage collection in Java*, Philippe Faes, Mark Christiaens, Dries Buytaert, Dirk Stroobandt. In Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'05), Tampere, Finland, August 2005

A.3 Garbage collection hints

We showed that a popular class of garbage collectors often make suboptimal decisions both in terms of *when* and *how* to collect. We argue that garbage collection should be done when the amount of live bytes is low

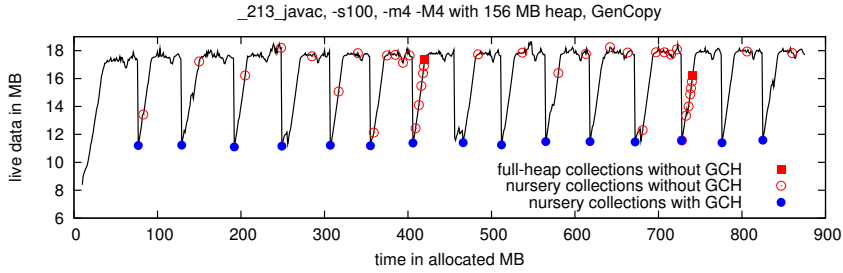


Figure A.1: The garbage collection points with and without garbage collection hints.

(in order to minimize the collection cost) and when the amount of dead objects is high (in order to maximize the available heap size after collection). In addition, we observe that this class of collectors sometimes trigger a nursery collection in cases where a full-heap collection would have been better.

Based on these observations, we propose *garbage collection hints (GCH)* which is a profile-directed method for guiding garbage collection. Offline profiling is used to identify favorable collection points in the program code. In those favorable collection points, the garbage collector dynamically chooses between nursery and full-heap collections based on an analytical garbage collector cost-benefit model. By doing so, GCH guides the collector in terms of *when* and *how* to collect. Experimental results using the SPECjvm98 benchmarks and two generational garbage collectors show that substantial reductions can be obtained in garbage collection time (up to 29×) and that the overall execution time can be reduced by more than 10%. In addition, we also show that GCH reduces the maximum pause times and outperforms user-inserted forced garbage collections.

Figure A.1 illustrates why GCH actually works for the javac benchmark. This graph shows the number of live bytes as a function of the number of allocated bytes. The empty circles denote nursery collections and the squares denote full-heap collections when GCH is not enabled. Without GCH, GC is triggered at points where the number of live bytes is not necessarily low. In fact, the maximum GC time that we observed on our platform for these GC points is 225 ms; and 12 MB needs to be copied from the nursery to the mature generation. The GC time for a full-heap collection takes 330 ms. When GCH is enabled (see the filled circles in Figure A.1), garbage gets collected when the amount of live bytes reaches a minimum, i.e., at an FCP. The GC time at an FCP takes at most 4.5 ms since only 126 KB needs to be copied. From this example, we observe two

key features why GCH actually works: (i) GCH preferably collects when the amount of live data on the heap is low, and (ii) GCH eliminates full-heap collections by choosing to perform (cheaper) nursery collections at more valuable points in time.

This work is described in:

1. *Garbage collection hints*, Dries Buytaert, Kris Venstermans, Lieven Eeckhout and Koen De Bosschere. In Lecture Notes in Computer Science Volume 3793, Springer-Verlag. In Proceeding of the International Conference on High Performance Embedded Architectures and Compilers (HIPEAC'05), Barcelona, Spain, November 2005.

2. *GCH: Hints for Triggering Garbage Collections*, Dries Buytaert, Kris Venstermans, Lieven Eeckhout and Koen De Bosschere. Transactions on High-Performance Embedded Architectures and Compilers, 1(1):52-72, June 2006

A.4 Hints for refactoring Java applications.

It has long been observed that many objects in languages with garbage collection have a short life-time [46, 53, 78]. In other words, many objects become unreachable and garbage shortly after they have been created.

We investigated if we can prevent these short-lived objects from being created in the first place. We conjecture that most of these objects are created merely to communicate data from one location in the program to another. When there is a short time between the object's creation and its last use, we expect that a simple program refactoring might be possible to establish the communication between the two program locations without creating an object. Here, we understand refactoring to mean "changing the internal structure of the software, without changing its functionality" [38]. Whereas most refactorings focus on optimizing the design of the software, we focus on reducing the rate at which it creates new objects.

By reducing the number of allocated objects, less garbage is created. This results in less work for the garbage collector and a better temporal locality, potentially leading to significant speedups. Furthermore, in a number of cases, time-consuming code executed in object constructors can be avoided, leading to further speedups.

We developed a tool which helps to automatically find the locations in the source code where many short-lived objects are created, the locations where they are used for the last time, and the methods between which these objects are communicated. By intelligently clustering all the objects observed during a profiling run, our tool groups together objects that can

be optimized by the same refactoring. We used our tool to optimize a number of programs from well-known benchmark suites. The optimizations typically require less than an hour of programmer effort, which is measured as the total time required to analyze the output of our tool, understand the source code constructs responsible for generating many short-lived objects, coming up with a suitable refactoring, and applying the required source code changes. In most cases, only a limited number of code lines need to be changed, without breaking the object-oriented design of the program. After refactoring, between 1.5 and 14 times fewer bytes are allocated, resulting in speedups between about 1.1 and 15.

This work is described in:

1. *Hinting Refactorings to Reduce Object Creation In Java*, Dries Buytaert, Kristof Beyls, Koen De Bosschere. In Proceedings of the fifth ACES symposium, Edegem, Belgium, 2005.

Bibliography

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

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[3] B. Alpern, S. Augart, S. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. McKinley, M. Mergen, J. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes Research Virtual Machine project: Building and open-source research community. *IBM Systems Journal*, 44(2), 2005.

[4] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 32(5):85–96, May 1997.

[5] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. tak A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, Nov. 1997.

[6] Apache. BCEL: the bytecode engineering library. http://jakarta.apache.org/bcel.

[7] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, 35(10):47–65, Oct. 2000.

[8] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Architecture and policy for adaptive optimization in virtual machines. Technical Report 23429, IBM Research, Nov. 2004.

[9] M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *ACM Workshop on Dynamic and adaptive compilation and optimization*, pages 52–64. ACM Press, 2000.

[10] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of Java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 111–129. ACM, 2002.

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

[12] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM, 2000.

[13] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *ACM/IEEE international symposium on Microarchitecture (MICRO)*, pages 245–257. ACM, 2000.

[14] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, 1994.

[15] BEA. BEA JRockit: Java for the enterprise technical white paper. http://www.bea.com, Jan. 2006.

[16] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with JMTk. In *International Conference on Software Engineering (ICSE)*, pages 137–146, 2004.

[17] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Eliot, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 169–190. ACM Press, 2006.

[18] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[19] D. C. Burger and T. M. Austin. The SimpleScalar Tool Set. Computer Architecture News, 1997. See also `http://www.simplescalar.com` for more information.

[20] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *ACM conference on Java Grande*, pages 129–141. ACM, 1999.

[21] H. W. Cain, R. Rajwar, M. Marden, and M. H. Lipasti. An architectural evaluation of Java TPC-W. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2001.

[22] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 139–149. ACM Press, 1998.

[23] J.-D. Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *International Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2001.

[24] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. The open runtime platform: A flexible high-performance managed runtime environment. *Intel Technology Journal*, 7(1):5–18, 2003.

[25] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing judo: Java under dynamic optimizations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 13–26. ACM Press, 2000.

[26] T. M. Conte, K. N. Menezes, and M. A. Hirsch. Accurate and practical profile-driven compilation using the profile buffer. In *ACM/IEEE international symposium on Microarchitecture (MICRO)*, pages 36–45, Washington, DC, USA, 1996. IEEE Computer Society.

[27] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *International Symposium on Computer Architecture (ISCA)*, pages 233–244. IEEE Computer Society, 2002.

[28] M. Dmitriev. Selective profiling of Java applications using dynamic bytecode instrumentation. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Computer Society, 2004.

[29] J. Donnell. *Java Performance Profiling using the VTune Performance Analyzer*. Intel, 2004.

[30] E. Duesterwald and V. Bala. Software profiling for hot path prediction: less is more. *International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 35(11):202–211, Nov. 2000.

[31] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, page 220. IEEE Computer Society, 2003.

[32] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 149–168. ACM, 2003.

[33] B. Dufour, L. Hendren, and C. Verbrugge. *J: A tool for dynamic analysis of Java programs. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems and Languages (OOPSLA Companion)*, pages 306–307, Oct. 2003.

[34] G. Eastman, S. Aundhe, R. Knight, and R. Kasten. Dynamic profile-guided optimization in the BEA JRockit JVM. In *3rd Workshop on Managed Runtime Environments (slides only)*, Mar. 2005.

[35] L. Eeckhout, A. Georges, and K. De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 169–186. ACM, 2003.

[36] J. Fenn and A. Lindon. Gartners hype cycle special report. t Technical report. http://www.gartner.com, 2004.

[37] S. J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *International Symposium on Code Generation and Optimization (CGO)*, pages 241–252. IEEE Computer Society, 2003.

[38] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, 2000.

[39] S. Friberg. Dynamic profile guided optimization in a VEE on IA-64, 2004. IMIT/LECS-2004-69.

[40] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*. ACM Press, 2007.

[41] A. Georges, D. Buytaert, L. Eeckhout, and K. D. Bosschere. Method-level phase behavior in Java workloads. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 270–287. ACM Press, 2004.

[42] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Prentice Hall PTR, 2005.

[43] M. Harkema, D. Quartel, B. M. M. Gijsen, and R. D. van der Mei. Performance monitoring of java applications. In *International Workshop on Software and performance (WOSP)*, pages 114–127. ACM Press, 2002.

[44] M. Hauswirth, A. Diwan, P. S. Sweeney, and M. C. Mozer. Automating vertical profiling. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 281–296, Oct. 2005.

[45] M. Hauswirth, P. S. Sweeney, A. Diwan, and M. Hind. Vertical profiling: understanding the behavior of object-oriented applications. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 251–269, Oct. 2004.

[46] M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *International Symposium on Memory Management (ISMM)*, pages 36–39. ACM, 2002.

[47] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):355–400, July 1996.

[48] M. C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: application to energy reduction. In *International Symposium on Computer Architecture (ISCA)*, pages 157–168. ACM Press, 2003.

[49] M. Karlsson, K. E. Moore, E. Hagersten, and D. A. Wood. Memory system behavior of Java-based middleware. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2003.

[50] J. Lau, E. Perelman, and B. Calder. Selecting software phase markers with code structure analysis. In *International Symposium on Code Generation and Optimization (CGO)*, pages 135–146. IEEE Computer Society, 2006.

[51] J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Computer Society, 2004.

[52] H. B. Lee and B. G. Zorn. BIT: A tool for instrumenting Java bytecodes. In *USENIX Symposium on Internet Technologies and Systems*, pages 73–82, 1997.

[53] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[54] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Prentice Hall PTR, 1999.

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

[56] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, June 2005.

[57] Y. Luo, J. Rubio, L. K. John, P. Seshadri, and A. Mericas. Benchmarking internet servers on superscalar machines. *Computer*, 36(2):34–40, 2003.

[58] J. Maebe, D. Buytaert, L. Eeckhout, and K. D. Bosschere. Javana: a system for building customized Java program analysis tools. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 153–168. ACM Press, 2006.

[59] J. Maebe and K. De Bosschere. Instrumenting self-modifying code. In *Workshop on Automated Debugging (AADEBUG)*, pages 103–113, Sept. 2003.

[60] J. Maebe, M. Ronsse, and K. De Bosschere. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *Workshop on Binary Translation (WBT)*, Sept. 2002.

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

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

[63] P. Nagpurkar, C. Krintz, M. Hind, P. F. Sweeney, and V. T. Rajan. Online phase detection algorithms. In *International Symposium on Code Generation and Optimization (CGO)*, pages 111–123. IEEE Computer Society, 2006.

[64] J. Neter, M. H. Kutner, W. Wasserman, and C. J. Nachtsheim. *Applied Linear Statistical Models*. WCB/McGraw-Hill, 1996.

[65] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[66] N. Nethercote and J. Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 89–100. ACM Press, 2007.

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

[68] perfctr. perfctr version 2.6.19 for Linux. http://user.it.uu.se/ mikpe/linux/perfctr.

[69] F. T. Schneider, M. Payer, and T. R. Gross. Online optimizations driven by hardware performance monitoring. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 373–382. ACM Press, 2007.

[70] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 104–113, 2001.

[71] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–57. ACM, 2002.

[72] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *International Symposium on Computer Architecture (ISCA)*, pages 336–349. ACM, 2003.

[73] K. Shiv, R. Iyer, C. Newburn, J. Dahlstedt, M. Lagergren, and O. Lindholm. Impact of JIT JVM optimizations on Java application performance. In *7th Annual Workshop on Interaction between Compilers and Computer Architecture (INTERACT-7)*, pages 5–13, Mar. 2003.

[74] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing the memory behavior of Java workloads: a structured view and opportunities for optimizations. In *International conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 194–205. ACM, 2001.

[75] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 29(6):196–205, June 1994.

[76] Standard Performance Evaluation Corporation. SPECjbb2000 Java Business Benchmark. http://www.spec.org/jbb2000.

[77] Standard Performance Evaluation Corporation. SPECjvm98 Benchmarks. http://www.spec.org/jvm98.

[78] D. Stefanovic and J. E. B. Moss. Characterization of object behaviour in standard ml of new jersey. In *ACM conference on LISP and Functional Programming (LFP)*, pages 43–54. ACM, 1994.

[79] L. Su and M. H. Lipasti. Speculative optimization using hardware-monitored guarded regions for Java virtual machines. In *International conference on Virtual Execution Environments (VEE)*, pages 22–32. ACM Press, 2007.

[80] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. Design and evaluation of dynamic optimizations for a Java just-in-time compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):732–785, July 2005.

[81] Sun Microsystems. Java Virtual Machine Profiler Interface. http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html.

[82] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of Java applications. In *Third Virtual Machine Research and Technology Symposium*, 2004.

[83] D. Tam and J. Wu. Using hardware counters to improve dynamic compilation. Technical Report ECE1724, Electrical and Computer Engineering Department University of Toronto, Dec. 2003.

[84] The DaCapo Project. DaCapo Benchmark Suite. http://www-ali.cs.umass.edu/DaCapo/gcbm.html.

[85] D. Ungar and R. B. Smith. Self. In *ACM SIGPLAN Conference on History of programming languages (HOPL)*, pages 9–1–9–50. ACM Press, 2007.

[86] J. Whaley. A portable sampling-based profiler for Java virtual machines. In *ACM conference on Java Grande*, pages 78–87. ACM Press, June 2000.

[87] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. System support for automatic profiling and optimization. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 15–26. ACM Press, 1997.