Online Optimizations Using Hardware Performance Monitors

Florian T. Schneider

DISS. ETH Nr. 18077

Online Optimizations Using Hardware Performance Monitors

ABHANDLUNG
zur Erlangung des Titels

DOKTOR DER WISSENSCHAFTEN

der

ETH ZÜRICH

vorgelegt von

Florian Thomas Schneider
Dipl. Ing. ETH

geboren am 31. Juli 1979

von Österreich

Angenommen auf Antrag von
Prof. Dr. Thomas R. Gross, Referent
Prof. Dr. Timothy Roscoe, Korreferent
Dr. Vijay S. Menon, Korreferent
Prof. Dr. Matthias Hauswirth, Korreferent

2009

Abstract

Feedback-directed optimization is becoming more and more important in modern execution environments. On one hand modern CPUs impose additional difficulties for the compiler to generate efficient code: Features like multi-core and complex cache architectures make things even harder for the compiler.

On the other hand, modern programming languages are often not well suited for classic compiler optimizations (due to dynamic class loading, polymorphism, etc.). JIT compilation with feedback-guided optimization can provide significant benefits for such programs since the compiler/runtime is able to adapt to the running program and does not require complex static analysis like in an ahead-of-time compiler. Programming languages like Java or C# make intensive use of references and generally deal with a large number of small objects. As a result, data access pattern are often irregular and difficult (if not impossible) to analyze statically.

This thesis focuses on how to gather feedback provided from the hardware platform via hardware performance monitors (HPM) and use it for program optimizations in a managed runtime for Java programs. We present HPM information as another source of feedback – in addition to traditional profile-guided optimization. We work in a dynamic compilation environment and all optimizations are done as the program runs. When doing online optimizations the runtime overhead of performance monitoring is of special importance.

We look at what kind of information is available and useful for optimizations. We discuss the different problems and challenges that come with collecting HPM information. The information has to be precise and unbiased. Different hardware platforms offer different ways of obtaining HPM information. We look at two platforms in more detail: the P4 (IA-32) and the Itanium Montecito (IPF) processor

Since the HPM data collection runs concurrently with the application it should have a very low execution time overhead. Our solutions adaptively adjusts the sampling period for event sampling to limit the amount of data that has to be processed by the JVM. We present a solution to efficiently collect HPM information in an online setting with a consistently small overhead of less than 1% on average.

The raw HPM information has to be mapped back to the source program to be useful for performance analysis or compiler optimizations. We extended an existing JVM to generate the meta-information necessary to perform this mapping similar to a symbolic debugger. Detailed instruction-level information is required to achieve a reliable mapping.

We present different applications of fine-grained (instruction-level) HPM information:

We perform a detailed performance analysis of Java applications. HPM information can be used to identify performance-critical parts of a program at the instruction-level. Our system is also able to measure data address profiles that allow to map HPM samples about cache misses back to data structures on the heap. We show how to use data address profiles to compare the performance behavior of different generational garbage collection algorithms.

As an example of how to use HPM feedback for optimizations, we present two examples: We show how HPM information can be used to perform object co-allocation – an online optimization that improves data locality at GC time in a Java VM. In the best case the execution time is reduced by 16% (3% on average) and L1 cache misses by 28%.

We also demonstrate how HPM feedback can be used to improve loop unrolling in a high-performance static compiler. We perform per-application and per-loop adaptation of loop unrolling parameters using data about stall cycles from the PMU. In the best case we achieve a 39% speedup (6% average) over the default heuristic at the highest optimization level.

Zusammenfassung

Feedback-gesteuerte Optimierungen werden in modernen Laufzeitumgebungen immer wichtiger. Auf der einen Seite haben moderne Prozessoren immer mehr Funktionen, die es einem Compiler schwer gestalten effizienten Code zu generieren. Beispiele dafür sind Multi-Core-Prozessoren und immer komplexere Speicherhierarchien.

Auf der anderen Seite haben moderne dynamische Progammiersprachen Eigenschaften, die fuer klassische Compiler-Optimierungen nicht gut geeignet sind (dynamisches Class-Loading, Polymorphismus, etc.). JIT Compiler haben hier den Vorteil, da sie durch Feeback zur Laufzeit mehr Informationen über ein laufendenes Programm verwenden können und so adaptiv optimieren können ohne komplexe statische Analyse zu benötigen. Programmiersprachen wie Java oder C# verwenden viele Referenzen und eine grosse Zahl von kleinen Objekten. Die Speicherzugriffsmuster sind daher oft unregelmässig und schwierig (wenn nicht unmöglich) statisch zu analysieren.

Diese Dissertation konzentriert sich darauf, wie Informationen, die vom Prozesso durch Hardware-Performance-Monitore (HPM) zur Verfügung gestellt werden, effizient gesammelt und in einer Java Laufzeitumgebung zur Programmoptimierung genutzt werden können. Informationen von den HPM ist eine weitere Quelle von Feedback neben traditioneller Profiling-Information. Unser System ist in eine Java VM mit JIT compiler eingebettet, wobei alle Optimierungen während der Laufzeit ausgeführt werden. Deshalb ist es extrem wichtig, die Feedback-Daten möglichst effizient zu sammeln.

Zuerst betrachten wir welche Art von Informationen zur Verfügung stehen und welche für Optimierungen benutzt werden können. Um HPM-Informationen zu bekommen müssen verschiedene Probleme gelöst werden: Die Informationen müssen genau und unverzerrt sein. Je nach Hardware-Architektur gibt es verschiedene Möglichkeiten das zu erreichen. Diese Arbeit stellt zwei Architekturen genauer vor und beschreibt wie HPM-Informationen genutzt werden könnnen: den P4 Prozessor und den Itanium Montecito-Prozessor.

Da die HPM-Datensammlung und -verarbeitung parallel zur Programmausführung passieren, müssen die zusätzlichen Kosten minimal gehalten werden. Unsere Lösung verwendet ein adaptives Sampling-Intervall um die Datenmenge zu begrenzen und zuverlässig einen geringen Overhead zu erreichen ($< 1\%$ im Durchschnitt).

Dann müssen die gesammelten rohen HPM-Daten zurück mit dem Source-Programm verbunden werden, um sie in der Java VM weiterzuverwenden. Wir erweitern eine existierende JVM, indem wir Meta-Informationen über jede Applikation im JIT Compiler

speichern, ähnlich wie ein symbolischer Debugger. Um die genaue Instruktion im Source-Programm zu finden ist es notwendig, dass die HPM-Daten auf eine Maschineninstruktion genau sind.

Als nächstes stellen wir verschiedene konkrete Anwendungen der erstellten Infrastruktur vor: Zuerst führen wir eine detailierte Performance-Analyse einiger Java-Programme durch. Die HPM-Daten helfen dabei performance-kritische Load-Instruktionen zu finden.

Mit unserem System können wir auch HPM-Daten mit Addressen im Speicher assozieren (d.h. Speicheraddress-Profile zu erstellen). Damit ist es möglich Datenstrukturen eines Programms zu finden, die für die Gesamt-Performance entscheidend sind. In unseren Experimenten verwenden wir Speicheraddress-Profile um die Cache-Performance verschiedener Garbage-Collector-Algorithmen im Detail zu vergleichen.

Zum Schluss zeigen wir zwei Optimierungen um zu zeigen, wie HPM-Informationen als Feedback verwendet werden kann. Die erste Optimierung ist Objekt-Koallokation, eine Online-Optimierung für bessere Cache-Performance beim Speicherzugriff auf Java-Objekte. Im besten Fall erreichen wir damit eine Beschleunigung um bis zu 16% (3% im Durchschnitt) und reduzieren die Anzahl der Cache-Misses um bis zu 28%.

Die zweite Optimierung ist Loop-Unrolling. Hier demonstrieren wir wie HPM-Informationen die existierenden Heuristiken für Loop-Unrolling in einem statischen Compiler verbessern kann. Im besten Fall resultiert eine Verbesserung um bis zu 39% (6% im Durchschnitt) verglichen mit der Standard-Heuristik auf dem hoechsten Optimierungs-Level.

Acknowledgments

Thanks to my PhD advisor Prof. Thomas Gross for the support, for his feedback that helped improving my work and for sharing his invaluable knowledge about how to pursue research as an independent researcher.

Thanks to the second readers Vijay Menon, Mathias Hauswirth and Timothy Roscoe for reviewing my thesis and for the helpful comments.

Thanks to my colleages in no specific order: Valery Naumov, my former office mate, for the friendship and the countless fruitful discussions. Niko Matsakis, my current office mate, with whom I had very good conversations about our research. Yang Su, with whom I exchanged many ideas while writing up the thesis. Mathias Payer, who did his master thesis under my supervision and helped with important parts of the implementation. I'd also like to thank all my other dear colleagues not mentioned here explicitly for the good times spent together and for the feedback and helped me improving my thesis defense presentation.

Finally, thanks to my family. Without my parents I would not be where I am now.

Contents

1

Introduction

Object-oriented programming languages like Java or C# allow changes to an executing program at runtime, e.g., through the use of a dynamic class loader. At the same time, modern processor architectures are difficult compiler targets if the compiler aims to optimize a program for speed of execution; features like prefetching and branch prediction are (sometimes) difficult to model in a compiler. So a code generator is faced with two difficulties: the dynamic nature of the target program complicates analysis of program properties (e.g., it is difficult to determine pointer aliasing or to analyze the memory referencing patterns), and important performance aspects (e.g., number and location of cache misses) are only evident at runtime.

Fortunately, programs written in such an object-oriented language are usually executed in a virtual machine that includes a JIT (dynamic) compiler. The dynamic compiler has the opportunity to immediately make use of information obtained at runtime. We distinguish between two kinds of information about an application that can be obtained at runtime:

- information that is independent of the execution platform like the execution frequency of methods, basic blocks or instructions; often the term *profiles* is used for this kind.

- machine-level information, i.e. performance data about the hardware level of the execution platform. Examples for this type of information are cache misses, TLB misses, or branch prediction failures. This kind of information is hard to model without feedback from the hardware, and therefore such information can be very valuable for a compiler to optimize memory system performance [49].

Profiles are a useful input to the code generator (not only in a JIT compiler but also in an ahead-of-time compiler). However, many previous optimizations (static and dynamic) focus only on the platform-independent information and did not include direct feedback from the hardware level [74, 62]. Yet most modern CPUs (like the Pentium 4, Itanium, PowerPC) have a performance measurement unit (PMU) to obtain performance-related information and therefore could provide input to a dynamic code generator that optimizes a program for a specific hardware platform. Using low-level hardware feedback information is becoming more and more important, especially on platforms like the Itanium

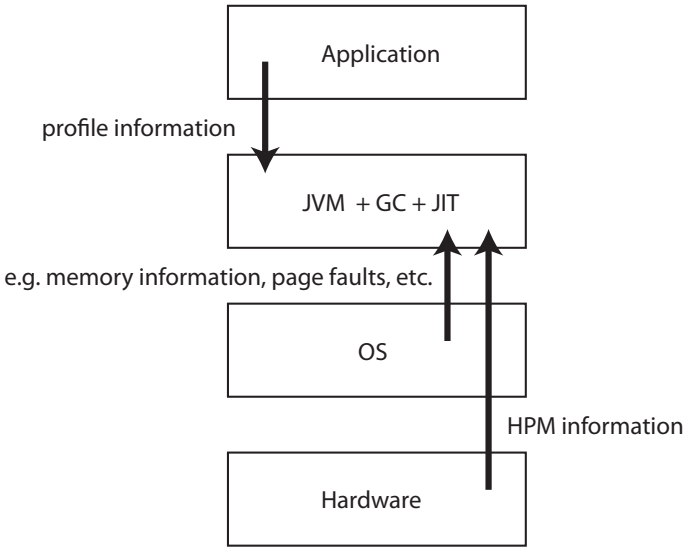


Figure 1.1: Different sources of feedback information that a JVM can use for program optimization.

processor family (IPF) [5, 7] that rely even more on the compiler for achieving good performance.

Of course, an application may also use feedback from other sources to optimize execution on a specific platform: E.g., it could use feedback from the OS about the virtual memory system to optimize memory management [54]. Figure 1.1 shows a high-level view of the different forms of feedback that can be useful for program optimization in a managed runtime environment. From the different sources of feedback that a VM/JIT compiler may use for optimization this dissertation focuses on how hardware-level information can be used for optimization.

To be useful for an optimizing JIT compiler and associated runtime system[1] the collected performance information must be *accurate* enough and *cheap* to obtain at run-time. There are a couple of requirements for a module that makes information from the hardware performance monitors available in such an execution environment:

- The interface between the VM and the performance monitoring hardware should hide machine-specific details where possible.
- The module should be flexible to allow obtaining different execution metrics.
- The overhead to collect the data should be as low as possible, and the system should not perturb executed applications too much.

[1]We consider the JIT compiler, the virtual machine (VM), and the runtime system as one unit since all components must cooperate to perform most interesting optimizations.

- The information must be accurate enough to be useful for online optimization. Often the granularity of a method or even a basic block is too coarse to infer which operation is responsible for some event (e.g. cache misses).

- The platform should work for off-the-shelf VMs, with only small or no changes to the core VM code. Otherwise the effort to port the infrastructure to another VM or to a new release would be prohibitively large.

In this dissertation we show how a Java system can benefit from using machine-level performance data. The approach and results are in general not tied to the Java programming language. Of course, any compiler that uses platform-specific information may also use profile information, e.g., to decide where and when to exploit the results obtained from the performance measurement unit, but this aspect is not discussed further here.

We describe and evaluate a module to feed fine-grained performance data from the hardware performance measurement (HPM) unit of a modern processor into the Java system. Our infrastructure is built on top of the Jikes RVM [22, 21], a freely available open-source research VM implemented in Java, and the perfmon HPM interface [56] running on a Linux kernel.

In our system we exploit special features of the P4 processor, called precise event-based sampling (PEBS), that allow to correlate measured events to single instructions and to the source program (in our case Java bytecode). The overhead of the runtime performance monitoring is reasonably low and stable ($<$1% avg) for a large number of benchmark programs. We achieve this using adaptive event-based sampling.

We also show how the collected data can be used for detailed performance analysis in a Java runtime environment using instruction address profiles and data address profiles.

As an example application of our infrastructure we present a garbage collector that is guided by online hardware feedback and report the results for a selection of standard Java benchmarks. The garbage collector improves data locality of Java programs automatically by co-allocating heap objects using information about data cache misses. The principal idea is to identify those objects and references that "produce" the largest number of cache misses. The garbage collector uses these hints to adapt its behavior for better data locality. Our system is, however, not aimed just at data locality optimizations in the GC. Instead machine-level performance data should be thought of an additional feedback for the whole runtime environment. We chose this optimization to demonstrate that the overhead of the approach is low in practice to allow a code generator/runtime system to deal with memory performance – one of the difficult areas for a compiler for object-oriented programs.

1.1 Thesis statement

The key contribution of this thesis is to show that it is possible to collect detailed performance data from the hardware during runtime with a low-enough overhead so that it can be directly applied for optimization by a managed runtime environment.

1.2 Organization of this dissertation

Chapter 2 describes the hardware and software platform we used and gives an overview of the performance monitoring features of different hardware architectures.

Chapter 3 describes the online performance monitoring infrastructure we built in detail and discusses problems and challenges when doing low-overhead online performance monitoring. We show how we achieve robust low-overhead performance monitoring across different types of programs, how we reduce or avoid biased measurements when doing event-based sampling, and how we map raw HPM data back to the source program.

Chapter 4 shows how our performance monitoring infrastructure can be used for fine-grained performance analysis of Java programs. It discusses the use of instruction address profiles and data address profiles to track down performance-critical instructions and data structures in a program. We also compare the performance of different garbage collection algorithms in terms of cache performance.

In Chapter 5 we present two optimizations that use hardware feedback to improve performance. The first application of online performance monitoring presented is *object co-allocation*, a fully-automatic online optimization driven by hardware feedback. We describe our approach and show how co-allocation improves data locality and speeds up applications by up to 18% in a Java VM using a generational garbage collector. As a second application of HPM information we evaluate how loop unrolling can be improved using hardware feedback. We performed experiments in a static C/C++ compiler and achieve an average speedup between 5 and 6% over the default heuristic at the highest optimization level.

Finally, Chapter 6 gives a survey over related work and Chapter 7 gives a summary of our results and concluding remarks.

2

Background

This chapter gives an overview of the platforms that we can use for performance monitoring. We discuss different approaches of how to localize performance bottleneck in applications in Section 2.1 and show what features modern CPUs offer for performance analysis in Section 2.2.

Section 2.3 compares the functionality found on the performance monitoring unit (PMU) of the two architectures covered in this chapter: the IA-32 and the Itanium processor family.

Section 2.4 describes precise event-based sampling (PEBS), a feature of the IA-32 PMU which allows to collect instruction-level performance data in detail. Our system uses PEBS for performance monitoring.

Finally, we shortly describe the software components that our system is based on in Section 2.5. There we also discuss the motivation behind the platform choices made during the implementation of our system.

2.1 Localizing performance bottlenecks

This section gives an overview of different techniques to track down performance bottlenecks in applications. There are different ways to characterize the performance of an application. Some techniques are only suitable for offline profiling and analysis. Others are also usable in an online optimization setting. Table 2.1 shows the different approaches to measuring and characterizing application performance.

1. Instrumentation [81]: The compiler inserts code sequences that count certain events in the application code. In most cases it is used to generate edge profiles for feedback-guided optimization. It usually requires recompiling existing code. In a dynamic runtime environment instrumentation is often used during the initial compilation [24, 79, 16] or interpreted execution [42]. In the following (optimizing) recompilation there is usually no instrumentation since it slows down execution considerably.

2. Timer-based sampling: The Jikes RVM [24] uses timer-interrupt-based sampling of the application's calling stack to determine frequently executed methods. This is a

	Instrumentation	Timer-based	HPM sampling	Simulation
Runtime overhead	medium to high	very low	low to very low	very high
Resolution	basic block, method	method	basic block, instruction	instruction
Accuracy	medium to high	low	medium to high	high
Invasiveness	medium to high	very low	very low	N/A

Table 2.1: Overview over different techniques for performance profiling.

very light-weight approach, but it is in general limited in resolution to method-level information. For more fine-grained data (e.g., at the basic block level) the interrupt frequency would be prohibitively high. The main advantage of sampling compared to instrumentation is that there is no additional code that has to be inserted into the program, and therefore the runtime overhead is much lower.

3. HPM event-based sampling [32]: Event-based sampling can be used to identify locations where performance-critical events occur. It also provides platform-specific data not available with software-only methods, but requires hardware support. Fortunately, most modern CPUs support event-based sampling. The problem is that the precision of the data often varies a lot depending on the underlying hardware platform and the exact data source. Also, the available hardware documentation often is not detailed enough about these issues so that a compiler implementors can sometimes not be sure which data actually could be used for hardware feedback-guided optimization.

4. Simulation [69]: Simulation can give very precise (and also platform-specific) information about performance behavior of applications. To get useful information the simulator needs a detailed model of the simulated architecture. One downside is that detailed simulation is usually very expensive in terms of execution time (up to 20-100x slowdown), so it is not applicable for dynamic optimization. Approaches that do not perform a full simulation, but instead approximate the precise behavior may be possible even during runtime ([91]).

Overall HPM sampling provides a good compromise for dynamic compilation systems which are the main target platform for feedback-guided optimizations. Instrumentation and simulation have their applications, but are either limited in the type of information that can be obtained, or too expensive for use in a dynamic runtime.

2.2 Overview over hardware performance monitors

Almost all modern general-purpose CPUs offer special hardware support for performance monitoring. In this work we focus mainly on two platforms: the IA-32 represented by the Intel P4 and the Core 2 microarchitecture and, for some experiments, the Itanium 2 Montecito from the Itanium processor family (IPF).

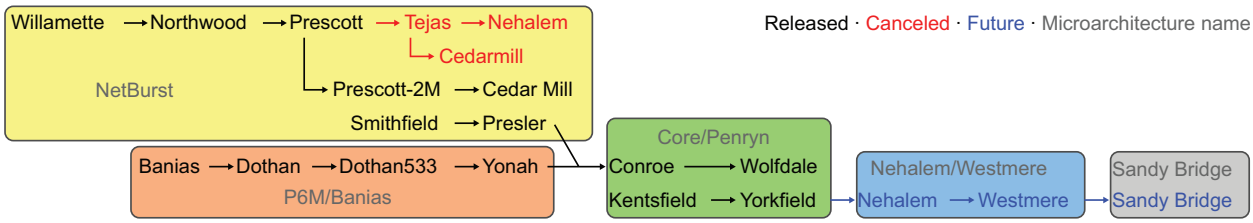


Figure 2.1: Intel's IA-32 processor roadmap [60].

Figure 2.1 [60] shows the roadmap of the Intel IA-32 architectures. Processors using the Netburst microarchitecture are those with a yellow background, Core and Core 2 CPUs have a green background. Names in red are cancelled processors.

For our experiments with IA-32 we are using Prescott P4 processors. Current processors belong to the Core 2/Penryn architecture which developed out of the P6M (Pentium 3, Pentium M) and the Netburst microarchitectures. Even though the Netburst family has been completely replaced by newer Core 2 architecture processors many features of the P4 are still present in the newer architectures. This means that results taken from that CPU generation are in many cases still valid on today's CPUs. Especially, the PMU architectures are very similar. Core 2 has some extensions, but supports basically all PMU features of the P4 that we will discuss in the following sections. The differences of the P4 and Core 2 with regard to performance monitoring are covered in Section 2.2.2. Section 2.4.1 discusses the mapping of HPM events that are important for this work from the P4 to Core 2-based CPUs in more detail.

In the next sections we will describe the hardware performance monitors of the these architectures in more detail.

2.2.1 P4

The P4 offers a large variety of performance events for counting [6, 78, 77].

In total it has 18 counter registers each 40 bit wide. Each of the counter registers has a counter configuration control register (CCCR) associated with it.

Event selection control registers (ESCR) are used for selecting which events to be monitored with the counter registers. There are 45 ESCRs on the Pentium 4 corresponding to 45 events from various parts of the system. (This number may be larger in newer processor models). Each counter register has several ESCRs associated with it to select which events should be counted in that counter register

The P4's performance monitoring unit supports three modes of operation:

- Event counting: The performance counters are configured to count events detected by the CPU's event detectors. A tool can read those counter values after program execution and reports the total number of events. This mode can be used to obtain numbers like cache miss rate, total execution cycles, etc.) More fine-grained information (e.g., on a method level) can be obtained by instrumenting the program

for reading counter values. An application of this measurement mode would be to evaluate the overall effect of program transformations.

- Imprecise event-based sampling (IEBS): A performance counter register is configured to count an event and whenever a certain number of events p has occurred (when the counter overflows), the CPU generates an performance monitoring interrupt. For a sampling period p the user sets up the counter register to

$$2^w - p$$

where w is the width of the counter register. When this register overflows the CPU issues an interrupt. The interrupt service routine records the return address (program counter where the overflow happened), resets the counter and restarts the counter. From the return address it is possible to estimate the location of an event, but the precision of this information heavily depends on the underlying microarchitecture (out-of-order execution, pipeline length, etc.) since the execution of the interrupt service routine is usually delayed by several instructions. We analyze event-based sampling in more detail in Section 2.3.1.

- Precise event-based sampling (PEBS): This type of monitoring is similar to non-precise event-based sampling, but it uses a predefined memory buffer to save the architectural state (all register contents) whenever the counter overflows. PEBS reports the exact instruction where the sampled event happened using special hardware support. In contrast, normal EBS can only measure an approximate location for sampled events due to the super-scalar design and out-of-order execution. With IEBS the sampled program pointer may be up to 5 dynamic basic blocks away from the actual source instruction [45]. Another difference is that PEBS can be used to count only a subset of the available events and only one PEBS event can be counted at a time. We discuss PEBS in more detail in the next section.

2.2.2 Core 2

This section gives a brief overview of the Core 2 architecture [11, 12, 13, 14, 15] with focus on the differences to earlier architectures that concern performance monitoring. There are many architectural differences between older processors based on the Netburst microarchitecture (P4) and the Core 2 microarchitecture. Most of them do not affect the performance monitoring unit significantly. Core 2 also employs aggressive out-of-order execution like the P4 which makes precise sampling equally challenging.

Precise event-based sampling (PEBS) as found on the P4 is available also on Core 2. The set of events is mostly equivalent, but of course there are small changes. The programming of the PEBS features also changes slightly. The exact differences (e.g. which registers to program) can be found in the corresponding Intel architecture reference manuals [15].

The events available on the Core 2 microarchitecture are adapted to the changed cache hierarchy: E.g., there is also no trace cache for micro-ops with Core 2, but instead a normal L1 instruction cache, so the Core 2 PMU offers events for L1 I-cache instead of events for the trace cache. We discuss specific issues about HPM events that must be addressed when porting our system to Core 2 in Section 2.4.1.

The multi-core nature of Core 2 also adds more HPM metrics that can be measured: For some events we can choose to use per-core counters or measure cumulatively for all cores. This can be done by setting a mask value to filter out event only from one core, or count events from all cores. However, in this work we do not deal with events that are specific to multi-processor systems.

2.2.3 IPF

This subsection gives a short overview of Itanium-specific features for performance monitoring [5, 7, 8, 9, 10]. We focus on the newer generation of Itanium processors (Montecito) which we are using in some of our experiments.

The Itanium2 has 12 48-bit wide counter registers that can be programmed to count events. In total there are around 600 different events to measure. The IPF also allows to filter events by instruction address range, data address range, privilege level (supervisor vs user mode) or by op-code. In contrast, the IA-32 can only filter events by privilege level.

The IPF also supports the two modes of measurement present on IA-32: simple event counting and event-based sampling. Additionally it has three features not present on IA-32: the Event Address Register (EAR) the branch trace buffer (BTB) and stall cycle accounting which we will describe shortly:

1. Stall cycle accounting: This measurement mode allows to attribute stall cycles in the CPU to different functional units of the CPU. This is makes it possible to find out where stall cycles come from. The different stall categories are: front end stalls, execution unit stalls, register stack engine stalls, L1 data cache stalls, and flush stalls.

2. Event Address Register (EAR): The EAR can be used for event-based sampling. It is restricted to certain events and it records the exact data (or instruction) address related to an event sample. The IPF does not have a mode comparable to PEBS on IA-32. Instead the EAR is used to gather precise information. Events that support EAR are TLB misses, D-cache/I-cache misses and ALAT misses. It also allows to filter miss events by latency to distinguish short versus long latency misses.

3. Branch Trace Buffer (BTB): The branch trace buffer is also a feature not available on IA32. It is used in combination with event sampling. When a sample is taken (e.g. on an I-cache miss, instruction retired, etc.), the BTB records a history of up to the last 8 branches[1] plus additional information about each branch (address, target,

[1] On the newer Itanium 2 Montecito processors the BTB records 16 branches.

taken/not-taken, predicted correctly/incorrectly). For example, it can be configured to record branch history on each sampled I-cache miss.

Instructions on the IPF are grouped into bundles of 3 instructions 41 bits wide each. Each bundles is 128 bit long and has a 2 bits that indicate the type of bundle.

Instructions are issued to the execution units in instruction groups of up to 6 instructions. All instructions of a group are issued in parallel which means they must not have any dependencies between each other. The compiler has to make sure it inserts stop bits between instruction groups properly to separate instructions that have data dependencies.

As a consequence the HPM unit not only reports the instruction address of a sampled instruction, but also the number of the bundle in the instruction group (0 or 1) and the slot number of the instruction within that bundle (0, 1 or 2). This way the exact source instruction of an event can be found.

For events that do not support the EAR it is only possible to create instruction address profiles (like with IEBS on IA-32). The reported instruction address only provide limited precision and can be off by several bundles depending on which event is measured. In our own experiments we found that the instructions reported the imprecise event-based sampling are often around 5-6 bundles (=15-18 instructions) away from the correct instruction address. For some events we even observe a delay of up to 20 bundles. As a consequence these events can only be located at a very coarse grained level without any further analysis (e.g. method-level).

2.3 Comparison of different HPM architectures

This section compares the features, advantages and disadvantages of two HPM architectures: the P4 as an example of the IA-32 architecture and the Itanium2 Montecito (IPF).

2.3.1 Event-based sampling

We performed an experiment to compare the information that can be obtained using normal event-based sampling on both platforms. In this mode the CPU issues a performance monitoring interrupt (PMI) whenever a preset counter overflows. The location of the event can be estimated from the return address given to the interrupt service routine.

When performing event-based sampling we can see the difference between an in-order-execution processor (IPF Montecito) and an out-of-order platform (P4) in the measurement results.

Figure 2.3 shows the histogram of instruction addresses delivered to the performance monitoring interrupt routine on an P4 and an Itanium Montecito processor. Here, we setup the system to measure L1 cache misses. The C source code of the example program is given in Figure 2.2. It iterates over a large array ($>$ 16KB, the size of the L1 data cache) multiple times so that the array load at line number 6 produces a capacity miss whenever the L1 data cache becomes full.

The left side of Figure 2.3 shows the relevant parts of the loop body in pseudo-code assembly. It consists of one load instruction per iteration. The load instruction sequentially loads an integer value from a large array which is larger than the size of the L1 cache). After the load we manually inserted a few hundred NOP instructions to observe the delay of the program counter reported via the performance interrupt service routine. We iterate over the array a large number of times so that we can expect a large enough number of L1 capacity misses.

On the Montecito all samples are reported on the same address. We see one peak in the histogram at a distance of 42 instructions (14 bundles) between the load instruction and the reported address[2].

In contrast, on the P4 we see two peaks at 173 and 176 NOP instructions where each contains 50% of the sampled events [3]. The reason for multiple peaks is that the P4 is an out-of-order processor whereas the Montecito is an in-order architecture.

The distance between the event source address and the reported address can vary considerably depending on how many instructions can be fetched and processed in between. Also, the delay can be different in a normal application where there are no artificially inserted NOP instructions. The experimental results from our setup serve as an illustration for the problems and challenges when doing event-based sampling rather than a guideline to calibrate other measurements. In general we can assume the the distance between an event source instruction and the reported instruction address is unknown within some lower and upper limit that can be approximated experimentally.

If we assume that many basic blocks consist of 10 or less instructions, the reported location may more than 4 (on the Montecito) up to or 17 (on the P4) dynamic basic blocks away from the event source instruction. This large distance (which may not be known in advance in general) makes is especially hard to map such events back to the source program. On the P4 there is the additional difficulty of event samples that are dispersed over multiple addresses. But it remains hard even on IPF where there is a more predictable distance between the source instruction and the instruction reported by the PMU.

Even though the example workload is very simple compared to real-world applications, it illustrates the difficulties when trying to identify performance bottlenecks in a program reliably.

It also shows that out-of-order processors make things even more complicated so that it is almost impossible to obtain precise instruction-level information without special hardware support like PEBS or the EAR.

[2]This distance can be different for other events. In our experiments with microbenchmarks we observed delays from 5 to 20 bundles (up to 60 instructions).

[3]For other combinations of instructions and events we observed different (but in general not predictable) number of peaks in the histogram (up to to 3).

```
1   long  k = 0;
2   long* A = malloc(10000*sizeof(long));
3   int i;
4
5   for (i=0; i<10000000; i++) {
6     k += A[i % 10000];
7     asm("nop"
8         "nop"
9         "nop"
10        ...
11
12        ...
13        "nop"
14        "nop"
15        "nop");
16  }
```

Figure 2.2: C source code of the example program.

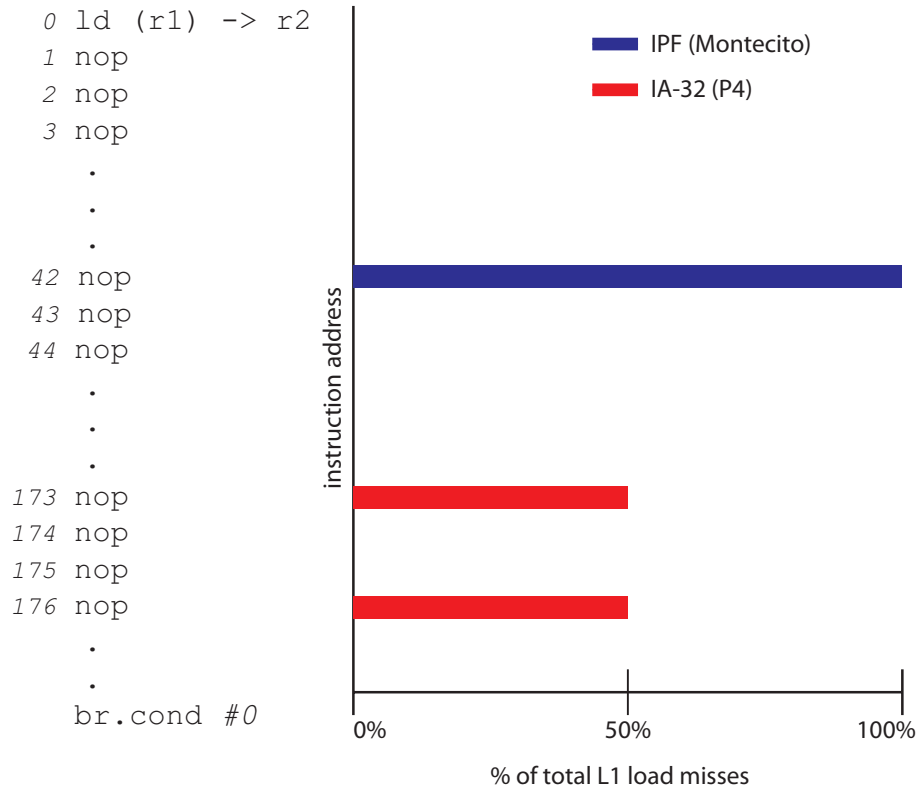


Figure 2.3: Instruction address histogram obtained with event-based sampling on the IPF and the IA-32 platform.

2.3.2 Precise instruction-level information

Since normal event-based sampling is too imprecise to provide fine-grained information, both platforms, IA-32 and IPF, have special hardware support for obtaining precise instruction-level HPM information for a limited number of events:

1. PEBS: All IA-32 processors from the Pentium P4 on (including the newer Core 2 generation CPUs) support Precise Event Based Sampling (PEBS) to accurately sample performance events. A small subset of events can be measured in PEBS mode. The hardware takes care of tracking the exact source instruction for a sampled event and reports all register contents when that event happened. One drawback is that only a very small number of events can be measured with PEBS (e.g., I-cache events are not available). For all others the user has to fall back to normal event-based sampling (EBS) which offers only imprecise instruction-level information. Also, only one PEBS-enabled event can be measured at a time.

2. EAR: The IPF architecture offers the Event Address Registers (EAR) to pinpoint the instruction address or the data address related to an event. It is used together with event-based sampling and can be used to derive the exact origin of an event since the program counter delivered to the interrupt service routine on a counter overflow can be up to 20 instruction bundles away from the event-producing instruction. The EAR can only be used with a limited set of events, but the EAR supports a wider range of events compared to PEBS events on IA-32 (e.g. L1/L2 data cache, instruction cache, TLB, and ALAT events)

2.4 PEBS

We will discuss the PEBS features of the P4 processor in more detail since our system heavily relies on them for performance monitoring.

In PEBS mode the HPM unit performs the collection and storage of the samples autonomously once configured. Figure 2.4 shows the format of a PEBS sample. Each sample contains the CPU state (the program counter EIP plus all register contents) after the sampled instruction. The register values contained in a PEBS sample are used later to recover more high-level information about a PMU event.

The CPU provides a set of registers that specify a buffer called debug store area (DS area). This buffer is allocated by the OS (i.e. the perfmon kernel module). The client (in our case the Java VM) specifies the size of the DS area, and a threshold value that determines at which point the CPU will generate an performance monitoring interrupt (PMI) (e.g., when the DS area is 90% full). After calling the interrupt service routine, the HPM unit is reset to start filling the DS area from the beginning.

Setting up the hardware for monitoring requires the following three steps in case of PEBS[4]:

[4]Normal imprecise event-based sampling only requires the first two steps.

```
1     typedef struct {
2             unsigned long eflags;
3             unsigned long ip;
4             unsigned long eax;
5             unsigned long ebx;
6             unsigned long ecx;
7             unsigned long edx;
8             unsigned long esi;
9             unsigned long edi;
10            unsigned long ebp;
11            unsigned long esp;
12    } pfm_pebs_p4_smpl_entry_t;
```

Figure 2.4: One PEBS record on the P4 contains the instruction pointer (EIP) and all register contents (total 40 bytes).

- Setting the ESCR. It selects the event to be monitored plus event-specific options and determines if we count events in user-level and/or kernel-level code.
- Setting the CCCR that corresponds to the counter register used. It selects the corresponding ESCR that we initialized in the first step.
- When using PEBS we need to initialize two special registers (PEBS_MATRIX_VERT and PEBS_ENABLE) to set up PEBS specific options.

The exact details and values for the setup of the individual registers can be found in the Intel documentation [6] and from the example programs included with perfmon [56].

2.4.1 PEBS support on newer IA-32 processors

This section discusses the support for precise sampling (PEBS) on newer IA-32 processors based on the Core 2 microarchitecture. To make the techniques presented in this thesis applicable to those processors we need to have precise sampling support for the HPM events that we use on the P4.

Fortunately, all the important PEBS events provided by the PMU on the P4 processor have corresponding a event on the newer IA-32 Core 2 architecture. The Core 2 processor family offers a larger set of events which mostly backward compatible to previous CPU generations except for small variations.

Table 2.2 shows equivalent events for different IA-32 architectures: the P4 (Netburst architecture) and the Core 2 architecture. Here we only discuss events concerning the memory hierarchy. The table shows the exact event descriptor for each event type. In most cases each metric is composed of a high-level event (e.g. FRONT_END_EVENT)

	Architecture	
HPM event	P4 (Netburst)	Core 2
L1D load miss	REPLAY_EVENT.L1_LD_EVENT	MEM_LOAD_RETIRED.L1D_MISS
L1I miss	n/a[a]	n/a[b]
L2D load miss	REPLAY_EVENT.L2_LD_EVENT	MEM_LOAD_RETIRED.L2_MISS
L2I miss	n/a[c]	n/a[c]
L3 miss	n/a[b]	n/a[b]
DTLB load miss	REPLAY_EVENT.DTLB_LD_MISS	MEM_LOAD_RETIRED.DTLB_MISS
DTLB store miss	REPLAY_EVENT.DTLB_ST_MISS	n/a[b]
DTLB miss	REPLAY_EVENT.DTLB_ALL_MISS	n/a[b]
ITLB miss	n/a[b]	n/a[b]
memory load	FRONT_END_EVENT.TAGLOADS	INSTR_RETIRED.ANY_P[d]
memory store	FRONT_END_EVENT.TAGSTORES	INSTR_RETIRED.ANY_P[d]

[a] The P4 has a micro-op trace cache instead of a normal L1 I-cache.
[b] No corresponding PEBS event available on this architecture.
[c] L2 cache is unified on both architectures.
[d] This event requires filtering out memory operations since it records *all* retired instructions.

Table 2.2: Equivalent precise sampling events for different CPU architectures.

and a mask value (e.g. TAGLOADS) which selects sub-category of the high-level event. So FRONT_END_EVENT.TAGLOADS would select all load instructions that are seen by the CPU's front end. There are more selection criteria with corresponding selection mask values (e.g. counting speculative vs. non-speculative instructions) which are omitted here for conciseness.

We can see that all events used on the P4 can be also measured on the newer Core 2-based processors. Each P4 PEBS event of interest can be mapped to a corresponding event on Core 2. However, there are a few subtle differences for individual metrics:

- In some cases like for DTLB cache misses, the P4 can capture load and store misses whereas the Core 2 can only measures loads.

- Another difference is that for measuring memory loads or stores the P4 counts operations seen at the front-end whereas on Core 2 only retired instruction are counted. Instructions executing speculatively may not retire, but are discarded at the back-end. However, this discrepancy can be resolved by filtering out speculative instructions by configuring the P4 PMU accordingly.

In general small differences like this do not pose a fundamental problem in applying our techniques on newer architectures. Therefore, our techniques remain applicable for current IA-32 CPUs. Also, the requirements to program PEBS on the P4 and Core 2-based CPUs are very similar. The exact differences are described in Chapter 18 of the IA-32 architecture reference manual[15].

2.5 Runtime platform

This section shortly describes the components involved in our system. It also presents the motivation and the reasons why we picked a particular component for this dissertation. It covers:

- the Linux kernel module for performance monitoring,
- Java VM, the
- JIT compiler and the
- memory management/GC system.

2.5.1 Perfmon

Perfmon [56] is a Linux kernel module that gives the user access to the CPU's performance monitoring features. Perfmon was originally developped at HP labs and is now available as open source.

We use the perfmon2 kernel patch and the corresponding libpfm library [55] to access the hardware performance monitors.

The main reason for using perfmon was that it supports all modern PMUs and offers support for many advanced features like PEBS on the IA-32. It is also actively maintained by the community and seems to become the standard for tools and application of performance monitoring on Linux.

The kernel module provides system calls to read and write the performance monitoring registers of the CPU. The second important feature of perfmon is a virtual per-process view of the hardware performance counters. By default the hardware counts events system-wide (i.e. events of all processes and the OS kernel). It provides a so-called performance monitoring context that can be attached to a Linux process and that contains all HPM information on a per-process basis. Other functions exist for initializing, starting and stopping counters.

Libpfm is a user-space shared library (libpfm) that provides functions common to all platforms. Events that are existent on all hardware platforms (cycle counting, retired instructions) have special platform-independent calls for setup.

However, libpfm has some critical limitations: It does not directly support platform-specific features like PEBS. When measuring platform-specific events the user has to setup the PMU registers manually. For that purpose the proper values for the HPM registers have to be determined from the hardware documentation directly. This limitations lead us to implement our own user-space library. We only reuse libpfm for the parts that do not deal with PEBS. Our library makes the PEBS events available to user applications in a convenient way.

For experiments with normal binaries (e.g., compiled from C/C++) perfmon offers also a command-line interface to perform off-line measurements and profiling.

When compiling with debug information (-g) events can be attributed to functions and source line numbers. To get more detailed information it is usually necessary to perform further analysis manually using a symbolic debugger (e.g., gdb) and disassembling the relevant part of the program.

2.5.2 Jikes RVM

Our implementation is done with the IBM Jikes RVM (version 2.4.2) [22, 21], a high performance Java virtual machine written mostly in Java. It includes an optimizing compiler with an adaptive optimization system (AOS) [24]: First, each method is compiled with a simple and quick baseline compiler. This first compilation does not include any local or global optimizations. Basically, the baseline compiler just concatenates templates for each Java bytecode instruction. The resulting machine code is not very efficient, but it is just used either for infrequently executed methods, or for collecting profile information at runtime for frequently executed methods which will be recompiled later by the optimizing compiler.

Methods that are executed frequently enough are recompiled and optimized using the optimizing compiler. The optimizer has three optimization levels (-O0, -O1 and -O2). The VM uses a static cost model with profiled execution frequencies to decide which optimization level to apply for a method.

When compiling a method with the optimizing compiler Jikes RVM initially translates Java bytecode into a high-level IR (HIR). It is almost equivalent to the Java bytecode except that it is not stack-based, but uses virtual registers to store temporary values.

After performing high-level optimizations (e.g., loop transformations, inlining, devirtualization, redundant load elimination, CSE, constant/copy propagation, etc.) the IR is lowered into the low-level IR (LIR) on which the optimizer performs a set of low-level optimizations (e.g., instruction scheduling, CSE, constant/copy propagation, etc.).

Finally, the compiler generates the machine-level IR (MIR) which closely resembles the final machine code. It does another pass of transformations (register allocation, peephole optimizations) before it generates the executable machine code.

To estimate the execution frequency for methods the VM samples the call stack in regular time intervals and records which methods are on top of the call stack. This is a very efficient way to approximate method execution frequencies without the need for instrumentation [5].

Jikes RVM provides its own thread implementation. It forks one OS process per physical CPU called “virtual” processors. The thread scheduler schedules all Java threads (including the GC and the compilation thread) on these “virtual” processors. It is a quasi-

[5] Alternatively, the baseline compiler can instrument each method with a counter to measure execution frequency.

preemptive m-to-n threading model [20] similar to Java “green threads”. However, this model has implications on implementing I/O in the VM: if one methods executes a blocking system call the whole virtual processor is blocked until the call returns.

We specifically avoid blocking I/O calls when communicating with the kernel module (perfmon) to avoid unnecessary blocking of the virtual processors.

Jikes RVM comes with a flexible module for memory management, the Jikes Memory Management Toolkit (MMTk) [28]. It allows to specify different garbage collector and allocation policies at compile time. All basic GC algorithms like mark-and-sweep, reference counting, semi-space copying and generational GC are implemented in MMTk. In our experiments we mainly use different variants of the generational collectors.

There are several reasons why Jikes RVM was picked as a implementation platform in this thesis:

- Implementation and development: For a full-featured JVM the Jikes RVM is still relatively simple and small compared other JVMs. The fact that it is implemented in Java also makes extending and debugging the VM more convenient than when using a VM implemented in C or C++.
- Performance: For a research compiler Jikes RVM offers reasonable performance compared to production JVMs.
- Availability: Jikes RVM is open-source and actively developed by the community.

2.6 Summary

Modern micro-architectures offer an increasing amount of hardware performance monitoring facilities. For a compiler and runtime system that wants to perform feedback-guided optmizations it is important to be able to obtain precise instruction-level data. Almost all newer CPUs have support for precise event sampling: IA-32 offers Precise Event Based Sampling (PEBS), IPF has the Event Address Register (EAR).

The HPM events (especially the PEBS events) available on newer IA-32 processors like the Core 2 are compatible to the events of the P4 processors we are using in this work. As a result, the infrastructure presented here still can be used on newer platforms.

In our implementation we are using Linux and the perfmon kernel patch. Perfmon seems to become the standard module for performance monitoring and fully supports all platforms we are dealing with (IA-32, IPF)

As a Java runtime system we use Jikes RVM, a research Java virtual machine. It is relatively easily extensible (compared to a full-size production JVM), but still offers competitive performance which makes it a good candidate for a research prototype.

3

Hardware Performance Monitoring in a Java VM

This chapter discusses the design and the implementation of a HPM measurement module for a Java VM. The goal is to collect 'real-time' hardware performance data at runtime. Such a system has additional requirements to meet compared to an off-line profiling tool. We will describe the different design parameters to make efficient online performance monitoring possible.

First, we present an overview over the system in Section 3.1. Section 3.2 discusses the implementation and the changes done to the affected components: the VM, the JIT compiler and optimizer and the memory management.

We will the go on with a discussion on the various problems and challenges that arise when performing online performance monitoring and present a solution to each them. Some of these issues like mapping raw HPM data to the original program source or avoiding biased measurements are a general problem when collecting HPM data, others, like how to limit the worst-case measurement overhead, are specific for online monitoring and do not occur in an off-line setting. There are three main challenges to overcome so that such a system can be used in practice:

1. Mapping HPM data to source code (Section 3.3): The hardware only delivers raw addresses and register contents. The system needs additional meta-information about the source program to correctly map the raw HPM data back to the source code. In our case the source code consists of Java bytecode. For every HPM event we have to find the Java method and the bytecode instruction that is the origin of that event.

2. Monitoring overhead (Section 3.4): There are two objectives concerning runtime overhead to meet. First, achieving a low overhead is crucial in an online setting. The application that runs at the same time as the monitoring code should not experience a significant slowdown. Secondly, in a sampling-based system like ours, the runtime overhead of online HPM measurements should be stable across different applications. Different programs generate different amounts of HPM events. The monitoring module must adapt to changing event rates.

3. Unbiased measurement (Section 3.5): The measurement should accurately reflect the application's performance behavior. Since our system is based on sampling there is always the problem of avoiding a prohibitively large bias. We performed various experiments with micro-benchmarks to quantify the bias when doing sampling, and we will discuss how to improve the accuracy of sampling-based HPM measurements.

3.1 System overview

We use the precise event-based sampling (PEBS) feature of the P4 processor [6] to measure events like cache misses. In principle the system can be implemented on any system that offers the kind of precise instruction-level HPM events like PEBS does. As discussed in Chapter 2 this is the case for the newer Core 2 architecture, but also for the Itanium platform. The PEBS mechanism has two advantages that make it especially useful for monitoring applications during runtime:

The first advantage is that PEBS reports the exact instruction (program counter plus all register contents) for the sampled events. This allows the compiler to recover higher-level information about the collected events, e.g., method, bytecode instruction, or field variable accessed.

Secondly, the CPU collects event samples on its own using a microcode routine and stores them in the Debug Store (DS) area – a buffer supplied by the OS kernel module. An interrupt is generated only when the DS area is filled to a specified threshold.

The P4 has a small number of events that can be selected for PEBS (e.g. L1, L2 cache misses and DTLB misses). It also allows only one PEBS event to be measured at a time. For other events that do not support PEBS, but only Imprecise event-based sampling (IEBS), it is not possible to map the reported addresses back to the source program on a instruction- or basic block-level. Method-level information could be still recovered to a certain extent, but very small methods may not be represented correctly since the displacement of IEBS samples can several basic blocks from the real event source instruction. Section 2.3.1 discusses the problems arising when using IEBS in more detail.

Figure 3.1 summarizes the system and how the different components interact with each other. The system consists of three main parts:

1. Perfmon loadable kernel module [56][1]: This kernel module is part of the Perfmon infrastructure and is developed at HP. It offers the functions to access the performance counter hardware for a variety of hardware platforms. The kernel module hides the platform-specific details from the JVM. It also provides the interrupt handler that is called by the sampling hardware when the CPU buffer for the samples is full.

[1] Available for download at http://www.hpl.hp.com/research/linux/perfmon/

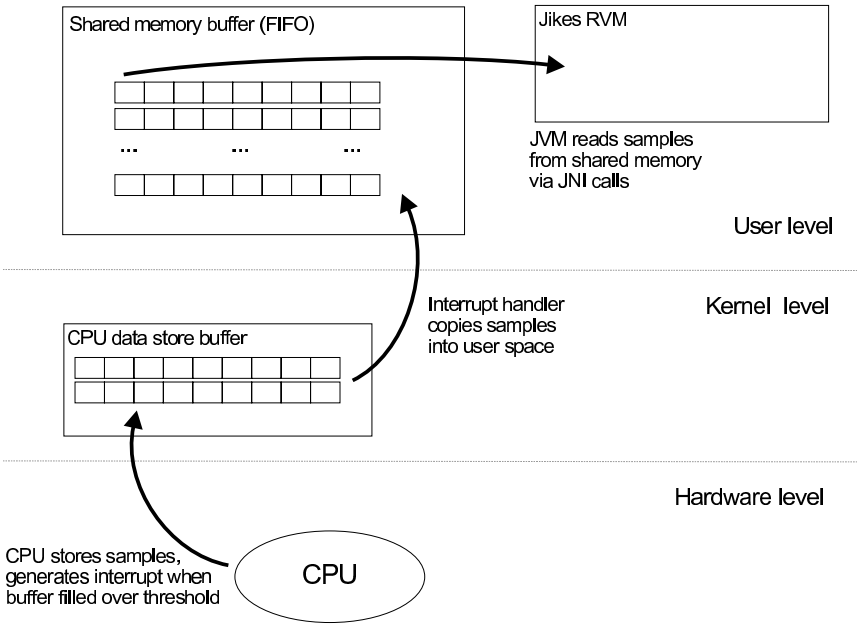


Figure 3.1: Overview of the monitoring system.

2. Native shared library (C) [74]: Since we cannot call device drivers directly from Java or from the Jikes RVM we developed a native library to provide an interface to the kernel functions and access it via the Java Native Interface (JNI). This makes porting the system to other Java VMs (that support JNI) easier. We could not reuse libpfm (the user-space library provided with perfmon) because it does not support monitoring a process using PEBS like we do in our system.

3. Collector thread (Java): We use a separate Java thread that performs all monitoring tasks inside the JVM. It calls the native library for communication with the PMU and uses it to transfers HPM data from the kernel space into the JVM.

3.2 Implementation and design issues

Since we are using Jikes RVM which is almost completely written in Java, most of our performance monitoring module is also implemented in Java except for the native library that provides the JNI interface to perfmon.

The copying of samples into user-space is necessary to allow the use of a different hardware platform with very few changes to the user-space library. The library is not limited to Java and can also be used from other runtime environments. Basically, the system can also be integrated with other Java VMs that support JNI. In principle it is also possible to monitor programs written in C or C++. This aspect, however, is not central to this work.

3.2.1 User-space library

This section gives a short overview of the PEBS library that serves as an interface from the VM to the perfmon kernel module. The library allows to read samples from the kernel module. The challenge here is to make the data exchange between Java and the kernel as efficient as possible.

In our system we could not use libpfm which is provided by HP with perfmon since it does not support the special PEBS features needed for our online performance monitoring system. Since libpfm does not include this functionality we need to access the HPM registers directly via the kernel module. For this purpose we implemented our own user-space library to give the Java VM access to the PEBS HPM facilities.

The main functions of the library are:

- Functions for startup and initialization of the hardware counters.
- Copying monitoring data (samples) from kernel space (DS area filled directly by the CPU) to the user(VM)-supplied buffer. The samples are returned from the native library in a Java `int[]`.
- Calls for changing interval/monitoring during runtime: e.g. sampling interval or event type.
- Stopping monitoring and shutting down the HPM module.

There are two possibilities to transfer data from kernel space to user space (VM address space):

1. Transfer data via JNI: This is safe, but slow. JNI data transfers work via Java reflection. This has the advantage that it does not require coordinating with other VM activities like the GC thread.
2. Copy data directly into the Java object (array) without JNI interaction: This is much faster than using JNI calls to copy the data, but we have to make sure that the garbage collector does not interfere by moving the object that the native code is writing to. We provide a pre-allocated array to the native code. The library function then copies all collected samples into this array directly without any JNI calls. We have to disable the GC manually for the short period of time while samples are copied from kernel space. As a consequence the native code must not allocate any new Java objects, since this may trigger a garbage collection which is not possible while GC is disabled. In practice we did not find this to be a significant limitation.

In our native library we use the second approach (copying samples directly) since it reduces the overhead of processing the HPM samples significantly.

The HPM unit of the P4 supports only a limited number of events to be measured in PEBS mode. Since we rely on PEBS for an accurate mapping of event samples back

to the program source, our library is limited to this set of events. The following events (followed by their perfmon event qualifier) can be measured with using our performance monitoring library:

- L1 data load misses (REPLAY_EVENT:L1_LD_EVENT)
- L2 data load misses (REPLAY_EVENT:L2_LD_EVENT)
- DTLB load misses (REPLAY_EVENT:DTLB_LD_MISS)
- DTLB store misses (REPLAY_EVENT:DTLB_ST_MISS)
- All DTLB misses (REPLAY_EVENT:DTLB_ALL_MISS)
- Memory loads/stores completed (FRONT_END_EVENT:TAGLOADS/TAGSTORES)

Note that the absolute numbers obtained when measuring these events do not always directly correlate with the real program behavior due to hardware limitations. E.g., the Intel P4 documentation [6] states that not all L2 misses are captured by the HPM unit. Another example concerns counting completed memory loads: Our own measurements show that when sampling completed loads, only about 50% of all loads are considered for sampling. However, this is still ok as long as we can ensure that the samples taken are not biased too much.

3.2.2 Modifications to the VM

At VM startup we load our shared library that provides access to the HPM hardware. Next, we initialize it using user-defined command-line parameters. The initialization takes the event identifier, an initial sampling interval and the sample buffer size as input parameters. If not specified further the system provides useful default values for missing parameters.

Then, the JVM starts a monitoring thread. We mark this thread it as a daemon thread. This means that the VM automatically kills it after all user threads have terminated and the VM shuts down (after the main application thread terminates). Figure 3.2 shows the activities of the monitoring thread in pseudo code. The monitoring thread wakes up periodically and polls for new samples. The polling interval is set to 100ms by default. After it read all available samples from the PEBS buffer it starts filtering and processing the raw data. In this step the system tries map each sample back to the original source code (Java bytecode). We discuss this processing step in more detail in Section 3.3. By processing samples in batches the cost of invoking the monitoring thread is amortized over a large number of collected samples.

The PEBS sample buffer should be allocated large enough so that it does not overflow between two invocations of the monitoring thread. We found that a size of 20K samples (800 Kbytes) is large enough for even the most demanding monitoring applications. During normal operation usually not more than 20% of the buffer is occupied. If an overflow

```
1       while(true) {
2         int[] samples = Read_PEBS_Samples();
3         if (samples.length > 0)
4           Process_PEBS_Samples();
5         Sleep(100ms);
6       }
```

Figure 3.2: Main loop of the monitoring thread running in the VM.

still occurs the HPM hardware stops collecting data temporarily and restarts only after the samples have been read out and processed. In this case the system just does not collect HPM data until the next polling interval.

3.2.3 Modifications to the compiler

The raw performance monitoring data can only be useful for optimizations if we can map the raw data back to the original "source" program (Java bytecode in our case). For this purpose we need to store additional meta-information about the compiled code in the JIT compiler. This meta-information is necessary to identify the Java method and the bytecode instruction within the method for a given HPM sample. The next section (3.3) describes the mapping of HPM data back to the Java bytecode in detail.

3.3 Mapping HPM data to the source program

This section will present our approach to mapping HPM events back to the Java bytecode so that the information can be used for dynamic optimization in a JVM. We show the modifications necessary in the JVM to make this mapping possible.

Depending on the underlying platform there are different difficulties in mapping the samples to the source code. For example, on the IA32 the reported program counter of a sample points to the instruction *after* the event-producing instruction. It is not trivial to navigate back one instruction on a variable instruction length architectures (CISC) like the IA32. Having a JIT compiler is clearly an advantage here: All meta-information about the machine code is available at run-time, and we can navigate in machine code on a variable instruction length architecture.

One sample on the P4 platform has a size of 40 bytes. It contains the program counter (EIP) where the sampled event occurred and the values of all registers at that time. To actually use the raw data for optimization, we need to obtain higher-level information about each sample. For identifying the Java bytecode instruction of a sample we only have to analyze the EIP register. The data register are only used if we are generating data address profiles as described later in Section 3.3.3.

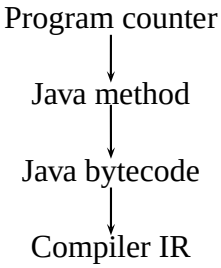


Figure 3.3: Process of finding the IR instruction from the program counter (EIP) given by the raw sample data.

Figure 3.3 shows the high-level process. First the collector thread extracts the samples that are of importance for the VM. By looking at the program counter register (EIP on IA-32), addresses outside the VM address space (e.g., from kernel space or native libraries) are filtered out immediately since we are only interested in events that occur in the machine code generated by the JIT compiler.

3.3.1 Method lookup

We implemented a function to quickly find the Java method where an hardware event occurred. Given a program counter of an event sample this function returns the name of a Java method.

For each method that compiled in the JIT we know the start and end address of the compiled code. Since the number of compiled methods is often very large ($>$ 20K methods in many programs) and we need to perform the operation of identifying the method for each sample delivered by the PMU, this search function has to be very fast [2].

We chose a two-level table lookup combined with a binary search within the methods contained in one 4K page. Figure 3.4 shows how we search for the method given the program counter (EIP): The 32-bit program counter is split up: The upper 8-bit indexes the first-level table where each entry points to the second-level table which covers 4096 memory pages of 4 KB (mid 12-bit). All tables are allocated on demand to minimize memory usage. The lowest 12 bits of the program counter finally represent the offset inside a 4K-page. For each 4K-page we record the methods that have code inside that page. The third step in the lookup function searches for the method that corresponds to that offset using binary search. In Figure 3.4 there are only three methods m1, m2 and m3 in the identified 4K memory page.

In the best case (if the identified 4K-page contains only code of one method), the lookup procedure takes a constant 3 table lookups. If there are many small methods within one 4K page the last step - searching all entries within one page - may take more iterations ($O(2 + log(n)$ where n is the number of methods per 4K page).

[2] In normal operation we collect around 1000-2000 samples per second

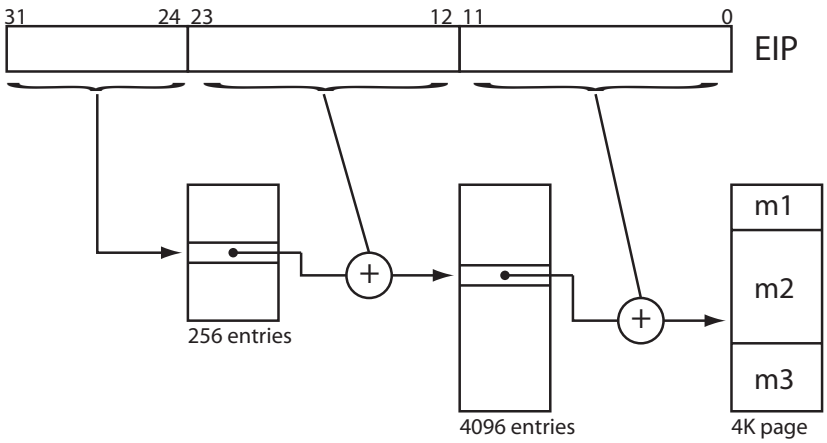


Figure 3.4: Finding the Java method for a given program counter (EIP).

On average over all JVM98 programs, the assembly code size of a JIT-compiled method is about 13 KB. This means that we get constant lookup time on average because most 4K pages contains only code of one method.

On each method compilation we have to update the mapping tables: Whenever a new method compiled we insert an entry, if a method re-compiled by the optimizing compiler we delete the old entry and insert a new one representing the new version of the method. This simple data structure has the advantage that it can be updated efficiently.

To further reduce the execution time overhead we also modified the allocation of the method objects in the JIT compiler: We allocate all method objects in the so-called "immortal" space of the heap. We do this because when using a copying GC the objects that store the machine code for each method will be moved in memory so that our system would need to re-computing the method lookup tables after each GC.

By default, those objects stay resident until the VM terminates. The resulting space overhead due to stale method objects is however reasonably small because in our setup only a small fraction of methods are re-compiled and replaced by the optimizing compiler. Also, the total amount of memory consumed by the binary code is very small compared to the overall memory consumption of a typical Java application. The small additional space overhead is not a problem in practice. It could be eliminated completely by allocating method objects in a separate GC-managed space where object are not moved.

3.3.2 Bytecode lookup

After we identified the method where a hardware event happened we need to to find the exact Java bytecode of that event to obtain higher-level information such as the object type.

The JIT compiler knows the start address of each machine instruction it generated. We build up a table that contains an information record for each machine instruction: It stores the index of the Java bytecode instruction where the machine instruction originated from. Basically, it maps machine instruction offset (from the start of the method) to Java bytecode index. The VM keeps the Java bytecode of each method in memory by default because it may be needed for recompilation of a method. Therefore, we do not need any additional memory except for the machine code maps.

Since we reported program counter points to the instruction following the event-producing instruction we have to navigate backward in the machine code. This is not trivial on a variable instruction length architecture like the IA32, but since the JIT compiler "knows" the start addresses of each machine instruction it generated we can use it to recover that information.

We extended the instruction mapping information that the compiler keeps for each method: Basically, we store a machine code mapping like a source-level debugger (from machine code addresses to Java bytecode in this case). All map entries are sorted by the instruction address and linked in a double-linked list. To walk backward one instruction we just find the map entry for the reported address via a hash map lookup and navigate to the previous map entry from there.

This mapping is already performed for methods that are compiled with the baseline compiler. For opt-compiled methods however the compiler only stores this information for the GC points. We extended the optimizing compiler so that it generates the machine code mapping not only for GC points, but for all generated assembly instructions.

Figure 3.5 shows code generated by the JIT with its associated Java bytecode instructions. Some Java bytecodes require multiple assembly instructions like the INVOKEVIRTUAL in Figure 3.5. Usually each bytecode maps to $>=1$ machine code instructions (1:n mapping).

There may be cases when multiple Java bytecodes are translated into a single machine instruction. In this case we cannot fully recover the original Java bytecode that triggered an event. The system just returns the first matching bytecode in such a situation. However, these cases occur only very rarely, so that they do not affect the measurements significantly.

Also, some instructions generated by the JIT compiler do not correspond to any Java bytecode (e.g. method prologue/epilogue code, native libraries, JNI methods). From the memory map of the JVM process we can identify the module where these events happened (e.g. the name of the native shared library), but we cannot retrieve any more fine-grained information. Currently, events that occur at such instructions are just ignored by the monitoring module and not processed further.

From that point on we are able to count events for each Java bytecode instruction. Every intermediate representation (IR) instructions also contains its corresponding Java bytecode index. This way the compiler can map the events to the internal IR instructions. This requires to keep the IR data structures in memory after method compilation. We found that the additional space overhead does not affect application performance signifi-

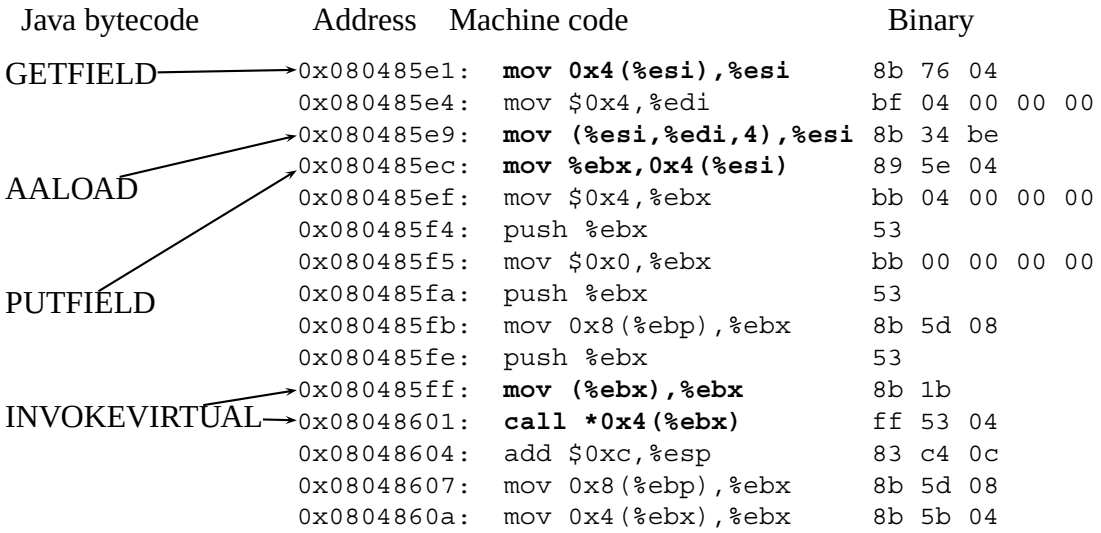


Figure 3.5: JIT-ted code fragment with associated Java bytecodes. One bytecode can have multiple machine code instructions associated with it.

cantly.

3.3.3 Data address profiles

A data address profile associates locations in memory HPM events like cache misses or TLB misses. This information is very useful if we want to know which data structures in a program are responsible for a potential performance bottleneck.

Some CPUs offer a way to find out the data address associated with certain events: The IPF platform provides a special register called Event Address Register (EAR) [5, 9] to capture this information for cache misses, TLB misses and ALAT [67, 63] events.

On the IA32, creating a data address profile is also possible using the PEBS mode for measuring events. However, obtaining the data address of an event is more complicated than on the IPF and requires additional steps because there is no such extra register (such as the EAR). Instead we need to calculate the data address from the register contents and the memory operand of the precise instruction of the event. We get this information from the PEBS sample.

Note that we have to rely on PEBS to find the precise address of the event-producing instruction. With the normal imprecise sampling (IEBS) obtaining a data address profile is not possible since the reported EIP is too far away from the original instruction that caused the event (see Section 2.3.1. The calculation of the data address depends on identifying the exact machine instruction.

Since PEBS reports the program counter (EIP) *after* the event source instruction we have to go back 1 instruction from the reported instruction which is in general not trivial on a variable instruction length set architecture like the IA32. Fortunately, the meta-

information that we store in the JIT compiler at code generation time helps here since we can look up the start offset of each machine instruction that was generated in the JIT compiler. From there we can navigate backward to the correct instruction.

Once we pinpointed the correct instruction we calculate the data address from the register contents contained in each PEBS sample. For this purpose we implemented a JNI interface to XED [3] in our native performance monitoring library to decode an instruction and to extract the memory operand plus its components:

The most complex memory addressing mode on IA32 uses 2 register operands and 2 immediate operands (constants) and is mostly used for array access. (e.g.,

`MOV ebx, [eax + 4*edx + 12]`).

The data address d is calculated as follows:

$$d = base + scale * index + offset$$

where $base$ and $index$ are always registers, $scale$ is a constant with a value of 1, 2, 4 or 8 and $offset$ is also a constant immediate operand. In our previous example `eax` would be the base, `edx` the index, `4` the scale, and `12` the offset.

All simpler IA-32 addressing modes can be expressed using the most complex addressing mode by just setting the unused components to zero.

We can use the calculated data address to obtain further information like the region on the heap where the hardware event happend (stack or heap). For field and array operation it is also possible to find the object header on the heap to obtain the runtime type of the Java object that was accessed. For the IPF platform this was done in previous work [17]. Our system basically provides the functionality of the IPF's Event Address Register (EAR) for the IA-32 platform.

3.4 Runtime overhead of online performance monitoring

In a production environment with online optimization it is very important to keep the runtime overhead as small as possible. In this section we show how expensive the runtime monitoring infrastructure is in terms of execution time and space overhead. Both must be reasonably low to make optimization using runtime monitoring possible.

A second requirement is that the overhead should be stable and predictable across a large variety of user applications. When doing event sampling every application generates a different amount of performance data which results in different overhead for performance monitoring. When doing online performance monitoring the data collection happens in parallel to application execution. On one hand we need HPM data fine-grained enough to get a representative picture of the performance behavior, on the other hand we should not collect too much data since this will slow down program execution. To be useful in a production JVM, the approach should be fully automatic (i.e., no manual tuning of sampling parameters, etc.).

3.4.1 Space overhead

The systems needs to allocate additional memory for gathering detailed source-level performance data. First, there are buffers for temporary storage of the samples collected. The user-space library keeps sets up a buffer for 20K PEBS samples and the VM data collection thread stores the raw data in an `int[]` array of the same size.

As described in the previous sections we need additional tables in the VM to resolve raw addresses to Java methods and bytecode. The space overhead of the additional metadata in the VM is shown in Table 3.1. The second column (machine code) shows the size of the machine code generated by the compiler in KBytes. Column 4 (MC maps) shows the size of the machine code maps that are needed to resolve raw samples. For comparison, we show the size of the GC maps alone in Column 3. The last row shows the total size and the map sizes of the Jikes boot image. The boot image maps are pregenerated at compile-time and do not contribute to execution time. We can see that the machine code maps are 4 to 5 times as large as the GC maps, but the total sizes of the maps for an application are tiny compared to the maps that are contained in the boot image.

We consider only library and application classes and leave out VM internal classes at the moment because we do not consider them for optimization. Including these would just make the boot image larger but would not influence application performance significantly. Currently, the whole boot image is about 9MB bigger than the original (increase of 20% from 45M to 54M).

The maps for application classes take up to 5x the space needed for the GC maps. However, in absolute numbers the size of the maps generated is moderate (up to 1870K bytes for jython). Adding the larger boot image, this results in an total increase of 11MB in memory usage. This is still small compared to the maximal heap size of our target Java programs which are typically long-running server applications.

There is potential for improving the space efficiency of the machine code mapping to reduce the size of the boot image. We reused the existing implementation for GC maps and it would be possible to custom-tailor the data structure for our needs. But the runtime overhead of using the existing data structures is low enough to use it for our purpose.

3.4.2 Runtime overhead with a fixed sampling period

This section discusses the runtime overhead of online hardware performance monitoring: We measure the monitoring overhead present the numbers for fixed and adaptive sampling intervals. In practice using a fixed interval is only useful for measurement purposes since it is easier to compare different runs of a program. When evaluating online optimizations we always use the adaptive interval.

Figure 3.6 shows the execution time compared to the original VM configuration without runtime event sampling using different sampling intervals from 25K to 100K. In this experiment we configured the system to monitor L1 cache misses. We measured the execution time for different sampling intervals (25K, 50K, 100K) to evaluate the relation

program	machine code	GC maps only	MC maps
compress	12	6	28
jess	20	12	43
db	7	4	20
javac	55	30	140
mpegaudio	71	31	168
mtrt	46	26	120
jack	40	22	111
pseudojbb	316	164	948
antlr	38	26	90
bloat	77	46	247
fop	8	4	16
hsqldb	117	67	290
jython	685	422	1870
luindex	119	58	316
lusearch	93	46	239
pmd	64	43	174
boot image	14975	10380	8260

Table 3.1: Space overhead: Size of machine code maps in KB.

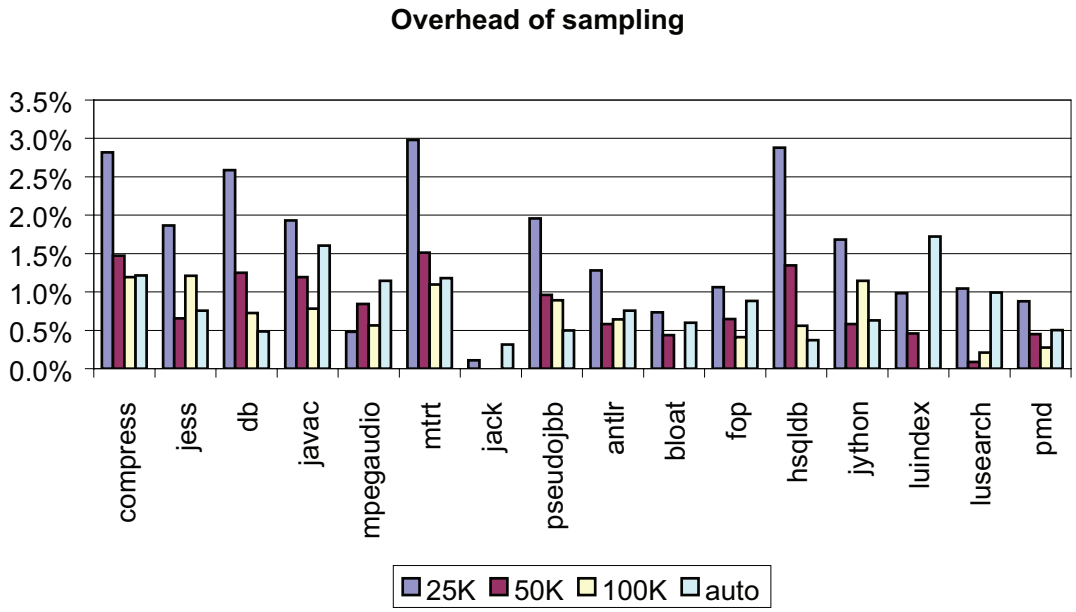


Figure 3.6: Execution time overhead when monitoring L1 cache misses with different fixed sampling intervals.

between sampling rate and execution time overhead.

The reported numbers for execution time are averages over 5 executions of each program, and they include all overhead from mapping raw sample data. The heap size here is fixed to 4x the minimum size for each program so that the GC overhead does not interfere with our measurement in an unpredictable way. In general, the runtime overhead is mostly independent of the heap size since the space overhead of the monitoring very low as shown in Section 3.4.1.

For most programs the time overhead is proportional to the sampling rate (e.g. db and pseudojbb). A smaller sampling interval means higher sampling frequency and thus more data to be processed by the monitoring module. For others (e.g., mpegaudio) the constant portion of the overhead dominates. The absolute number of samples is not very high in these cases. The worst case is an increase of almost 3% for mtrt, compress and hsqldb with the smallest interval(25K).

We can also see that also the overhead when measuring L1 cache misses varies significantly between different programs. The overhead ranges from close to 0% for some programs (jack) to around 3% for mtrt, compress and hsqldb.

The "auto" configuration shown for comparison here has a variable sampling interval. The runtime overhead varies less between the different benchmarks and is still low on average ($<$1%). Section 3.4.3 will introduce our approach to solve the problem of reducing the variance of the runtime overhead across different applications by constantly adaptive the sampling period at runtime.

3.4.3 Limiting the monitoring overhead: Adaptive sampling period

Since the cost of monitoring is proportional to the number of event samples generated, we can limit the runtime overhead by setting an appropriate sampling period. However, each application "produces" events at a different rate, and even within one application there may be large variations in the number of events occurring (i.e. in different program phases).

Traditionally it is the responsibility of the user to manually set a sampling period that results in a small runtime overhead, but still gives enough samples for a representative picture of the program behavior. The sampling period has to be set individually for each event and for each application. Previous systems [17] also follow this manual approach. However, this is not practical in a production system where we would like to have a fully automatic setting of an appropriate sampling period. Instead we implement a fully automatic *adaptive sampling period* to solve that problem.

We limit the runtime cost by trying to measure on average not more than a certain number of events per time period. This requires changing the sampling period during program execution.

Changing the sampling period, however, requires reconfiguring the HPM unit of the CPU. Since this is quite expensive in terms of execution time on the P4 (it requires a call to the shutdown/startup functions of perfmon) we only adapt the sampling period in

longer intervals (e.g., every 1/2 second) [3].

Whenever the monitoring thread delivers new samples we record the number of samples collected (s_i for the ith measurement period). One measurement period is 100ms in our implementation. We calculate a moving average M of reported events over the last 2 seconds ($n = 20$ in this case). By adjusting the sampling period we try to limit the number of samples that need to be processed in the future to L samples/measurement period. We calculate the next sampling period p_{new} as follows:

$$p_{new} = \frac{p * M}{L}$$

where p is the current sampling interval. The moving average M here is calculated as

$$M = \frac{s_{i-1} + s_{i-2} + ... + s_{i-n}}{n}.$$

When decreasing the sampling period (=more samples) we make sure that the buffer is large enough and we also adjust the polling interval of the monitoring thread so that no samples are dropped due to a buffer overflow.

Figure 3.7 shows the adaptation of the sampling interval for the JVM98 db benchmark during runtime. The left y-axis shows the collected number of L1 cache miss samples per second (in units of 1K samples/second) and the right y-axis shows the sampling interval set in the HPM module. The x-axis shows time in milliseconds. Since the number of cache misses varies throughout the execution time, the sampling interval adapts continuously. We can see a sharp increase in the number of events at around 20'000 ms. As a consequence the sampling interval is adjusted from 5'000 to around 25'000. After some warm-up time the sampling interval stays between 40K and 45K. The average event rate settles around the preset value of 400 (0.4K) samples per second.

Runtime overhead with adaptive sampling period

Figure 3.8 shows the monitoring overhead with fixed and adaptive sampling rates. For this evaluation we choose to measure DTLB misses because they vary more across different applications than L1 and L2 cache misses: Programs with a very small memory footprint (e.g. mpegaudio, jack) produce almost an order of magnitude less DTLB misses than program with a larger working set (e.g. db, mtrt).

When using a fixed sampling period the monitoring overhead across the JVM98 programs varies widely from 5% (mpegaudio) to 23% (db). On average the overhead is 14%.

When using the adaptive period, we set the target value so that the program with the previously lowest overhead (mpegaudio) has approximately the same overhead with the adaptation enabled. The overhead for the adaptive sampling rate stays between 2%

[3]The IPF PMU allows for a more efficient implementation of changing the sampling interval at runtime where we do not need to stop and restart the sampling in perfmon.

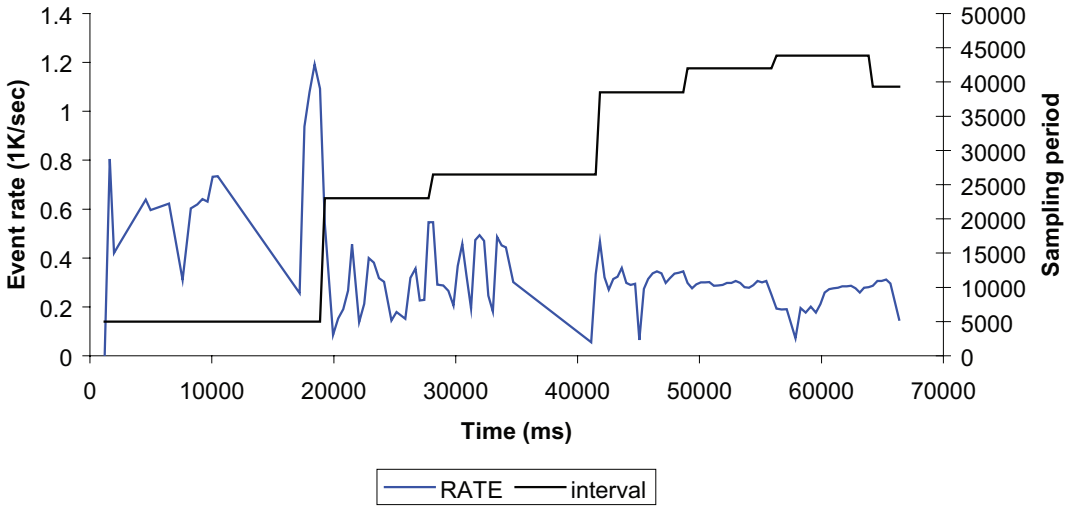


Figure 3.7: Adaptive sampling period and rate of L1 misses for SPEC JVM98 db.

(compress) and 7% (jess). The average overhead is 5%, and we do not experience a large variation as with the fixed sampling period.

Note that we collect more samples than usual for this experiment to illustrate the effect of monitoring more clearly. In normal operation (when doing online optimizations) the target value for the HPM event rate is set to limit the monitoring overhead to 1% instead of 5% as shown here.

Still, the overhead is not perfectly constant throughout all programs. Some benchmarks react more sensitive to the additional activity of the monitoring thread that runs in parallel with the application threads.

These results are a strong argument for using an adaptive sampling rate in a production system since it avoid excessively high overhead for memory-intensive applications. Instead it provides much more stable runtime overhead across different applications.

Figure 3.9 shows the average monitoring overhead with targeted sampling rates of 0 (no sampling) to 32000 samples/sec for all SPEC JVM98 programs. The plot also includes the 95% confidence intervals for each data point. The baseline is the default VM configuration without any monitoring.

When collecting few samples (less than 2000 samples/sec) the constant portion of the overhead (invocation of the monitoring thread, etc.) is still significant and the standard deviation is almost as large as the overhead itself: The 95% confidence interval at 2000 samples/sec is almost 0.6%. Up to 4000 samples/sec the overhead is below 1% on average. The 95%-confidence interval for the remaining data points is between 0.6% and 0.75%. We can see a clear proportional relationship between the number of samples collected and the monitoring overhead. At 32K samples/sec the average overhead reaches almost 4%.

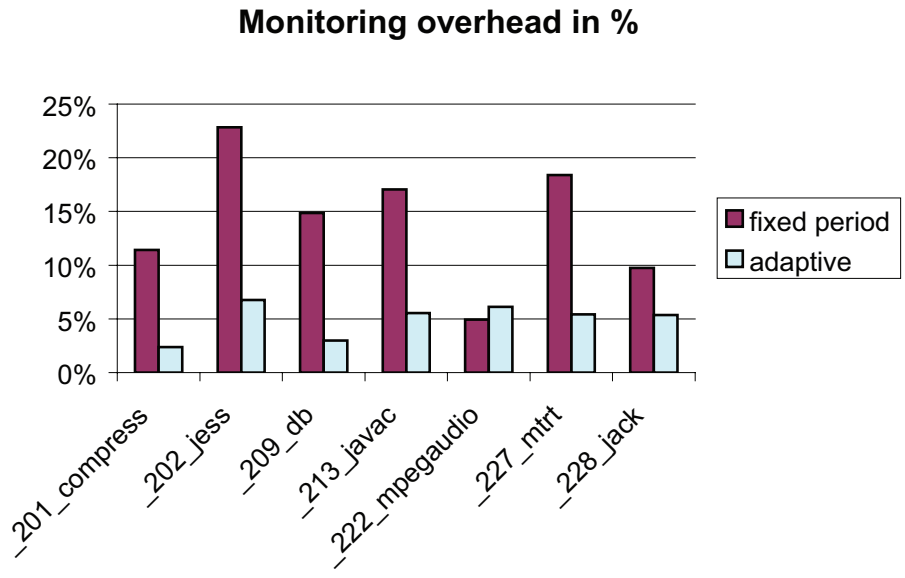


Figure 3.8: Monitoring overhead for DTLB misses for the SPEC JVM98 programs using fixed and adaptive sampling rates.

Collecting more than 2000 samples/sec is rarely useful in practice. More samples than that do not give any additional information about the performance behavior of a program. Here we present the numbers for up to 32000 samples/sec to illustrate the relationship between the amount of collected performance data and the performance monitoring overhead. The numbers show that we can achieve a monitoring overhead of around 1% or less when we sample less than 4000 samples/sec.

3.5 Biased event sampling

Event sampling is always an approximation of the real behavior. In reality event sampling is often biased. This means some parts of a program may be over- or underrepresented by event sampling. We show factors that may contribute to biased measurements and present possible solutions.

Since event-based sampling is only an approximation we would like to make sure that this approximation is as good as possible. This is especially important when we use the information gathered from the HPM unit is used for optimization: If the optimizer draws wrong conclusions from the data, the result of an optimization may be not as expected.

The reason for biased measurements is that the sampling process is not purely random as it should ideally be. The amount of bias depends on how the hardware selects events for sampling.

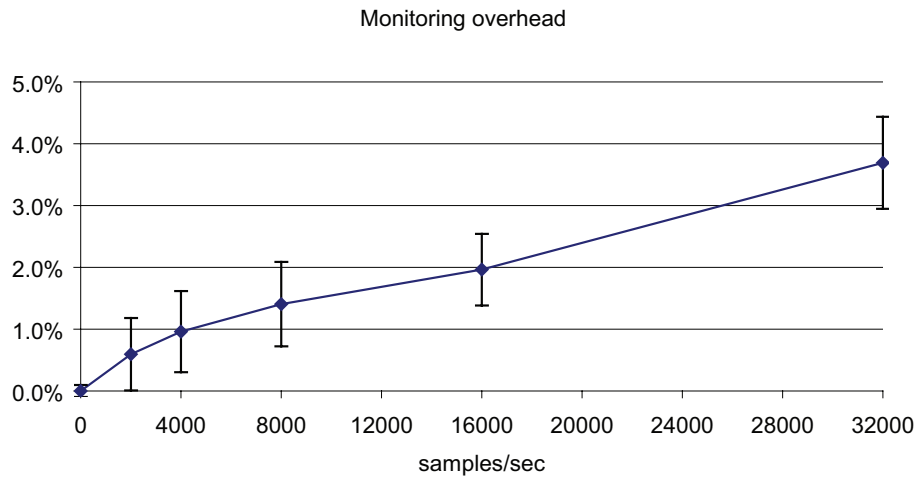


Figure 3.9: Average monitoring overhead across all JVM98 programs with an adaptive sampling interval.

We perform experiments with two platforms to evaluate the amount of bias when doing event-based sampling: the P4 (IA-32) and the Montecito (IPF) processors.

To illustrate the problem of biased sampling we perform a simple experiment with a micro-benchmark. We set up event sampling to measure array loads and stores in a tight loop. Figure 3.10 shows the corresponding Java code. Each iteration performs a loads and a store at each array element. The event sampling is set up to count memory loads/stores using PEBS.

In an ideal world (with purely random sampling) we would expect that all array element are sampled equally frequently. In reality the distribution is never completely uniform, but it should not deviate too much from perfectly random sampling.

3.5.1 IA-32

Figure 3.11(a) and 3.11(b) show the distribution of samples on the 8 array elements when using a fixed sampling interval without randomization. Each column represents the number of samples at an array access. The sampling interval is varied by +/-1 (399'999 and 400'000) in the two plots. We clearly see that certain array loads are almost never recorded, whereas others are overrepresented. The histograms also show that the exact sampling period has big impact on the sample distribution. Varying the sampling interval by +/-1 returns a totally different sample distribution. In general the distribution are very far away from the ideal uniform distribution. For each of the two sampling intervals we see peaks at different array elements.

One solution to this problem would be to chose a purely random sampling interval after each event that was sampled. Unfortunately this is not possible on the IA-32 HPM

```
1       int[] array = new int[8];
2
3       for (int i=0; i<10000; i+=1) {
4         array[0]++;
5         array[1]++;
6         array[2]++;
7         array[3]++;
8         array[4]++;
9         array[5]++;
10        array[6]++;
11        array[7]++;
12      }
```

Figure 3.10: Example program with a manually unrolled loop to measure sampling bias when counting integer array stores.

architecture: When using PEBS the sampling interval can only be adjusted after a performance monitoring interrupt occurred (i.e., when the PEBS buffer is full). This means that we cannot chose a random interval after each sampled event. By design this may lead to biased measurements. By periodically varying the sampling interval slightly by a random amount we try to avoid or at least reduce such biased sample distributions.

Figure 3.12 shows the sample distribution with the adaptive sampling interval and randomization enabled. We set up the system so that the average number of samples is the same as with the fixed interval. After each PEBS interrupt the interval is changed by a random amount (+/-128).

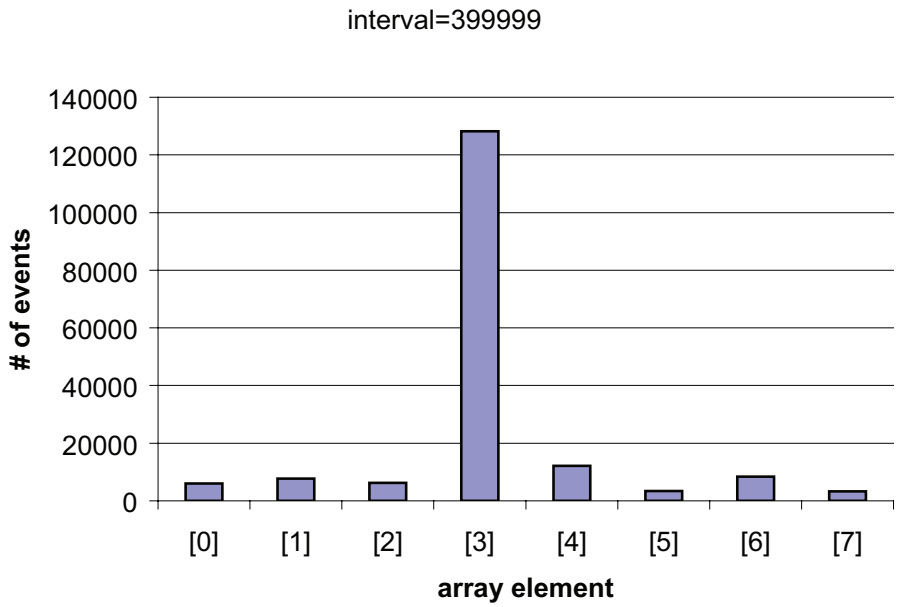
Qualitatively, we see that randomization of the sampling interval helps. With the fixed interval the difference between the smallest and the highest frequency is almost 100x where with the randomized interval this difference decreases to around 4x. The standard deviation with a fixed interval is 3-4x larger than with the randomized interval.

Still, the histogram from Figure 3.12 looks very different from a perfectly uniform distribution. One remaining problem is that currently we only change the sampling interval approximately 1/2 second (because it requires the expensive procedure of stopping/restarting the HPM in perfmon). This problem is specific to the IA32 architecture and is not present on IPF.

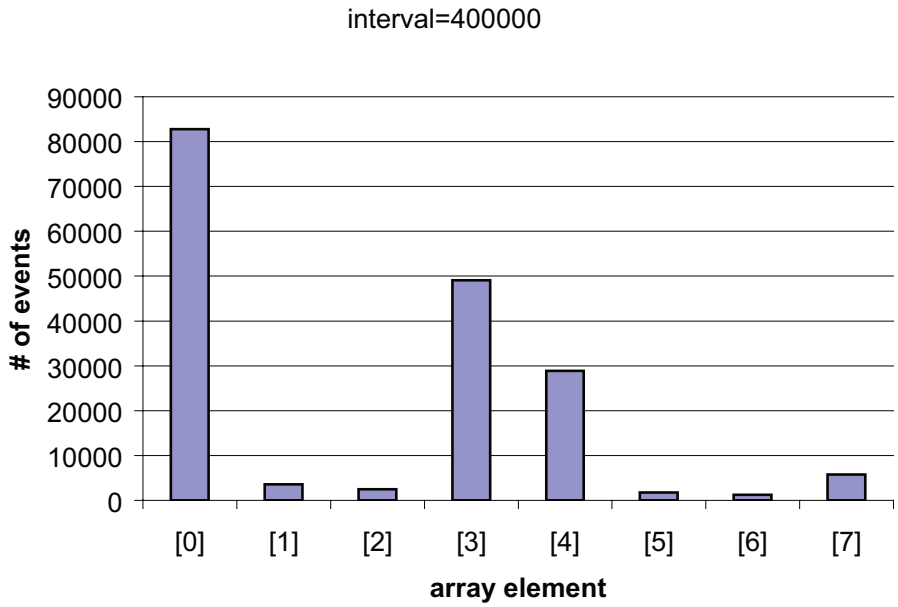
3.5.2 IPF

For comparison we performed the same experiment on an IPF machine: This platform allows more freedom in changing the sampling interval since it can change the interval after each taken sample.

We use the same program from Figure 3.10 as in the experiments with the P4 described before. Since we cannot run Jikes RVM on IPF we performed the measurements using



(a) Sampling interval 399999



(b) Sampling interval 400000

Figure 3.11: Event sample distribution for array stores in an integer array of length 8 with slightly varying different sampling intervals.

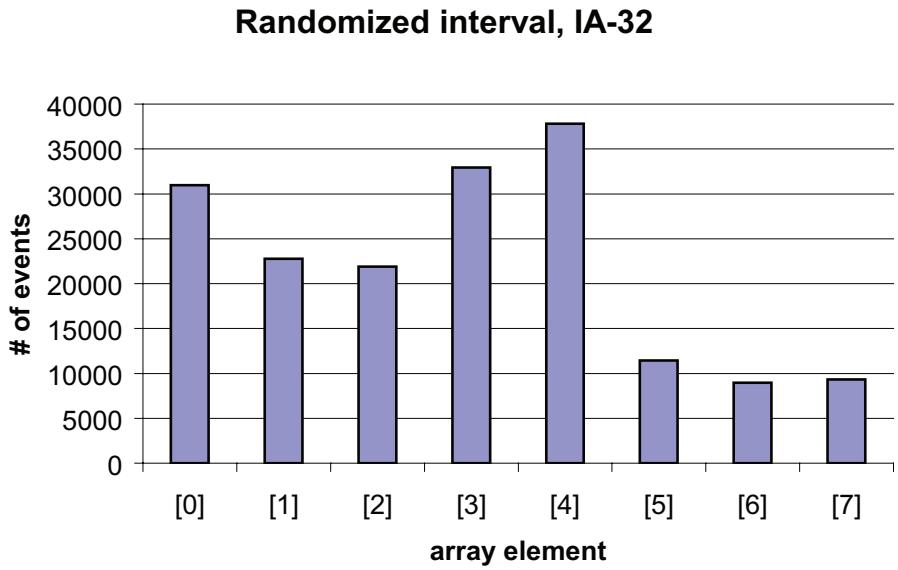


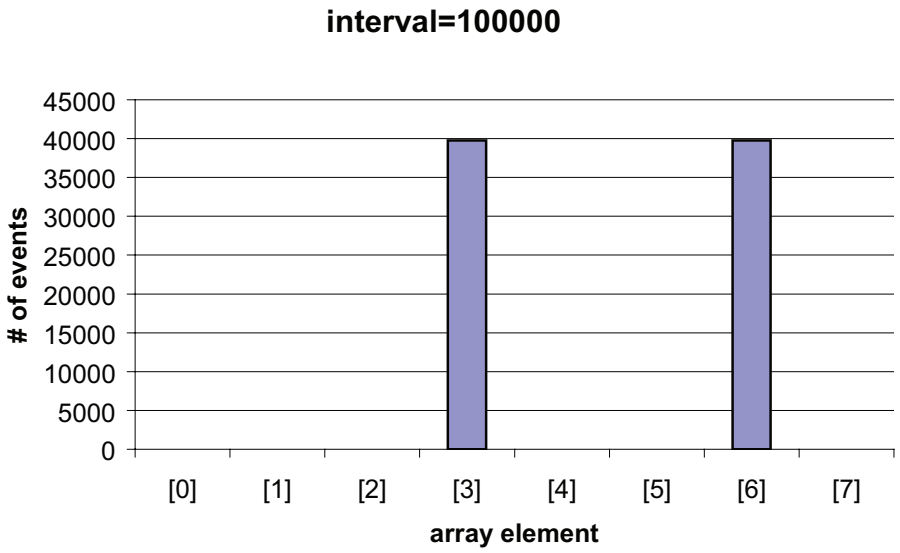
Figure 3.12: Event sample distribution for array stores in an integer array of length 8 using an adaptive randomized sampling interval.

the static Java compiler gcj [2]. This is acceptable since the program we are analyzing is very simple (only one hot method), and we are only interested in the evaluating different sampling techniques.

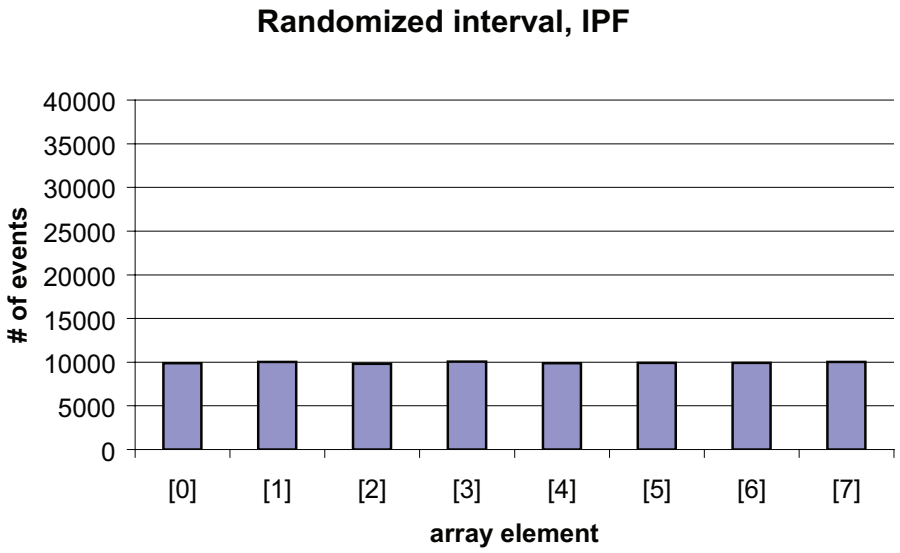
Figure 3.13(a) shows the results without randomization. The results look very similar to the results on the IA-32 except that the bias is even more pronounced than with Jikes RVM on IA-32: Almost all events are measured at element 3 and 6. The number of events for the other array elements is negligibly small ($<$10) so that they are not visible in the plot. One reason for the uneven distribution that is that when using a static compiler there are no background activities (like GC) that may "disturb" the measurement and provide some additional non-determinism.

Figure 3.13(b) shows the number of events measured when using a randomized sampling interval. It almost perfectly matches a uniform distribution across all 8 array elements. This is possible because the sampling interval can be reset to a new randomized value after each event whereas on IA-32 there are longer periods with a fixed interval between interval resets.

We perform a second experiment to illustrate a problem that arises when we obtain biased measurements with sampling. In Figure 3.14(a) we plot the number of events for an individual array element from the program in Figure 3.10 across different fixed sampling intervals from 99980 to 100020. Depending on the sampling interval we either count a lot of events ($>$40000) or almost none ($<$10).



(a) Fixed sampling interval



(b) Randomized sampling interval

Figure 3.13: Event sample distribution for array stores in an integer array of length 8 on an IPF platform.

Also note that there is a certain periodicity in the number of events: The pattern repeats every 8th interval. This can be easily explained by the fact that we perform exactly 8 array stores (one into each array element) in every loop iteration.

Such a biased measurement over- or underestimates the number of events at a specific location and we may not find the real hot-spots in an application.

In contrast, Figure 3.14(b) shows the result with randomization. We configured the monitoring to randomly assign the lower 8 bits of the sampling interval: In our example of 100'000 (0x186A0) the sampling interval would be randomly chosen between 99'840 (0x18600) and 100'095 (0x186FF) with an average of 99'967.5.

Here, the number of event is almost constant around 10'000 for all sampling intervals. Using randomization make the measurement process robust and unbiased.

In general the same observation holds for the both platforms - the IA-32 and the IPF: Randomization is absolutely necessary to obtain unbiased measurements and to correctly identify hot-spots in an application.

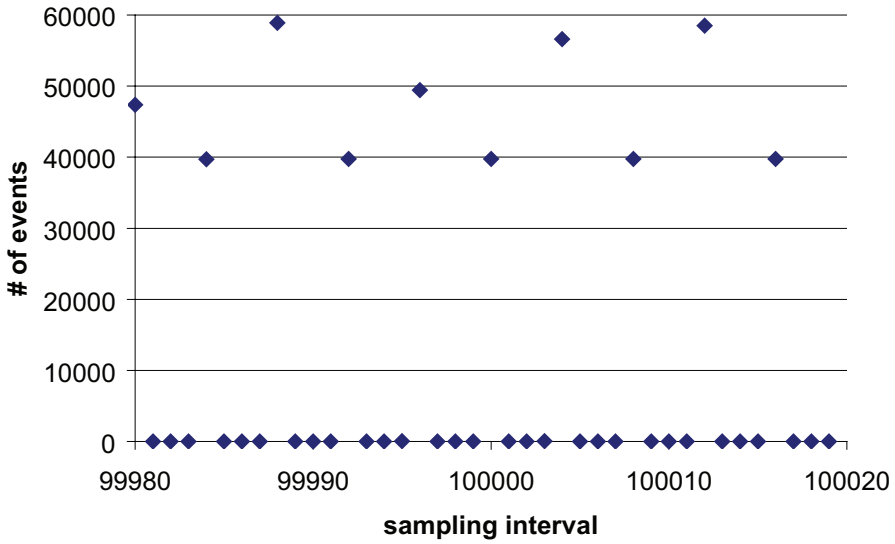
3.6 Summary

There are several challenges when doing performance monitoring using HPMs in an online setting:

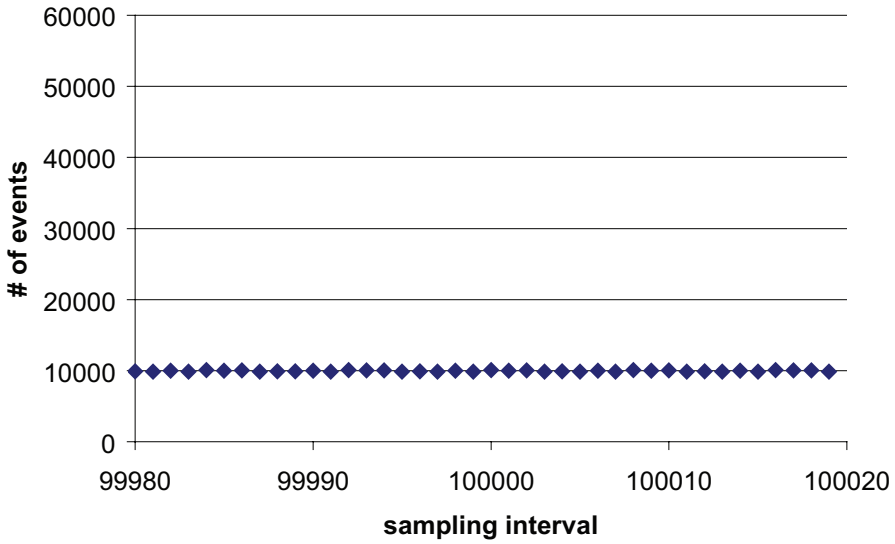
First, the gathered information has to be accurate at the instruction-level to map relevant performance behavior back to the source program. We generate meta-information in the JIT compiler that is later used to map the raw HPM information (event samples including instruction and data addresses) back to Java bytecode because only high-level information is useful for the optimizer or the GC to guide its optimization decisions. Since the PMU of the P4 processor reports the program counter of the instruction following the instruction that caused an event, we also need the machine code map generated by the JIT to navigate backward in machine code with a variable-length instruction set.

The PEBS feature also makes it possible to obtain data address profiles using our infrastructure. We decode the machine instruction of a PEBS sample, extracting its memory operand and calculating the memory address from the register contents contained in each PEBS sample.

We showed that online performance monitoring can be done with a very low overhead so that application performance is only minimally impacted: For a setting of 1000 samples/sec with an adaptive interval or a fixed sampling interval of 100K the average execution time overhead is below 1% across a large number of benchmark programs – a value that is low compared to software-only profiling techniques. The sampling interval adaptation is necessary to compensate for the varying performance behavior of Java applications: The amount of events produced varies significantly across different applications, but also in different execution phases within the same application. By constantly adjusting the sampling interval so that the VM receives a pre-set number of events per time period we can limit the worst-case overhead that an application may incur.



(a) Fixed sampling interval



(b) Randomized sampling interval

Figure 3.14: Number of events counted for a single array element with different sampling intervals

The space overhead of the meta-information used for processing the raw HPM data in the VM is usually small compared to the overall heap size of typical Java applications. We see a maximum value of 11 MB required for the machine instruction mapping information which is low enough so that overall performance is not impacted significantly.

Our experiments with data locality optimizations indicate that 1000 samples/sec offers a reasonable resolution and is sufficient to obtain enough coverage. We can conclude that online monitoring can be performed in a robust way with less than 1% overhead on average which makes it a very attractive addition to software-only approaches.

Finally, we have to make sure that the measurement process does not introduce any bias. If the monitoring is too biased we cannot get a correct picture of the real performance behavior of an application. By approximating a random sampling procedure we try to avoid biased sampling. We showed that we can achieve an unbiased measurement with proper randomization on the IPF. On IA-32 the bias for PEBS sampling can be reduced considerably by randomizing the sampling interval after each performance monitoring interrupt, but it cannot be completely avoided due to hardware limitations.

4

Measuring application performance behavior using HPM

After presenting the performance monitoring infrastructure in Chapter 3 we show now how such a system can be used in a Java managed runtime environment to measure and characterize the performance behavior of applications. This section presents different performance characteristics of Java applications that can be obtained with fine-grained hardware performance monitoring data.

We first show how to HPM data can be used to to locate performance-critical load instructions (Section 4.1). For a JIT compiler it is necessary to identify locations in a program where optimizations should be applied to get the maximum benefit. The time budget in a JIT compiler is limited because the compiler runs at the same time as the application. As a consequence it has to focus on the hot spots in an application when performing expensive optimizations to get the maximum performance benefit at low compilation time cost [24].

The second part of this chapter in Sections 4.2 and 4.3 shows how we can do detailed performance analysis using data address profiles. As presented in Section 3.3.3 a data address profile attributes events like cache misses or TLB misses to memory addresses (instead of the instruction address where an event happened). Data address profiles prove to be a very useful tool for analyzing memory performance. Finally, in Section 4.4 we show how to use data address profiles to compare the performance characteristics of different GC algorithms in Jikes RVM.

4.1 Distribution of data cache misses on load instructions

The precise event-based sampling of the P4 allows us to measure the distribution of cache misses over all memory load instructions in the program. Identifying hot loads is useful to target optimizations in the compiler at the most significant portions of the program. Insertion of prefetch instructions [17] is one example of an optimization that need precise information about candidate load instructions that may benefit from prefetching.

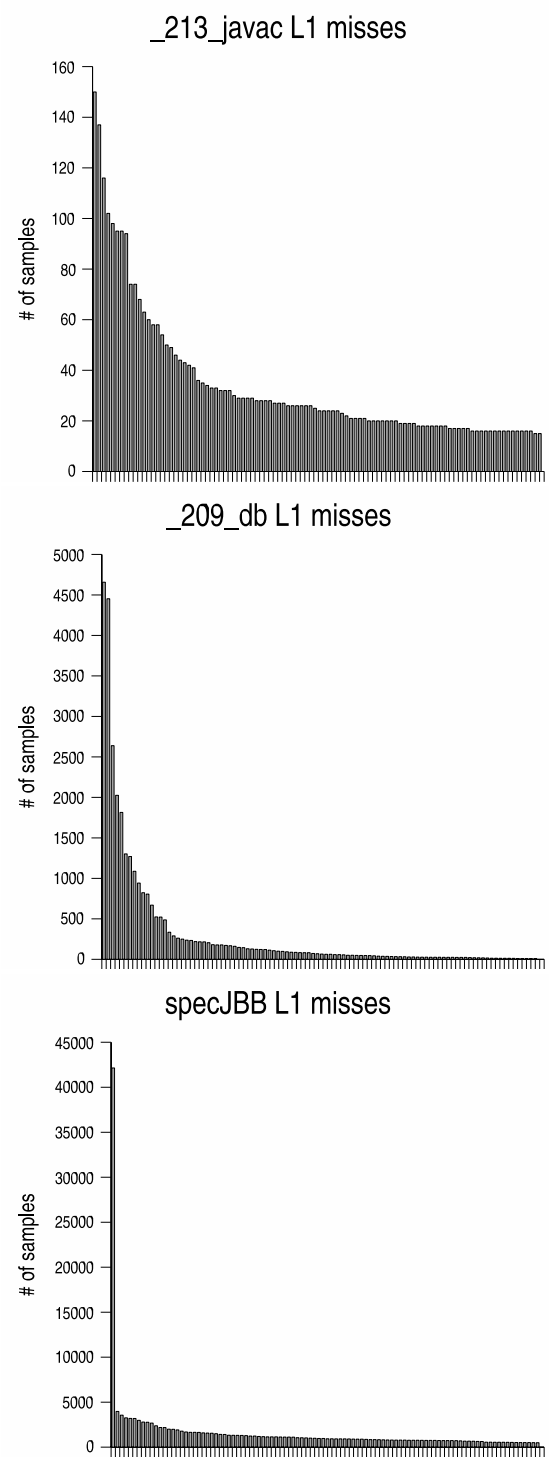


Figure 4.1: Histograms of L1 cache misses (100 most contributing load instructions).

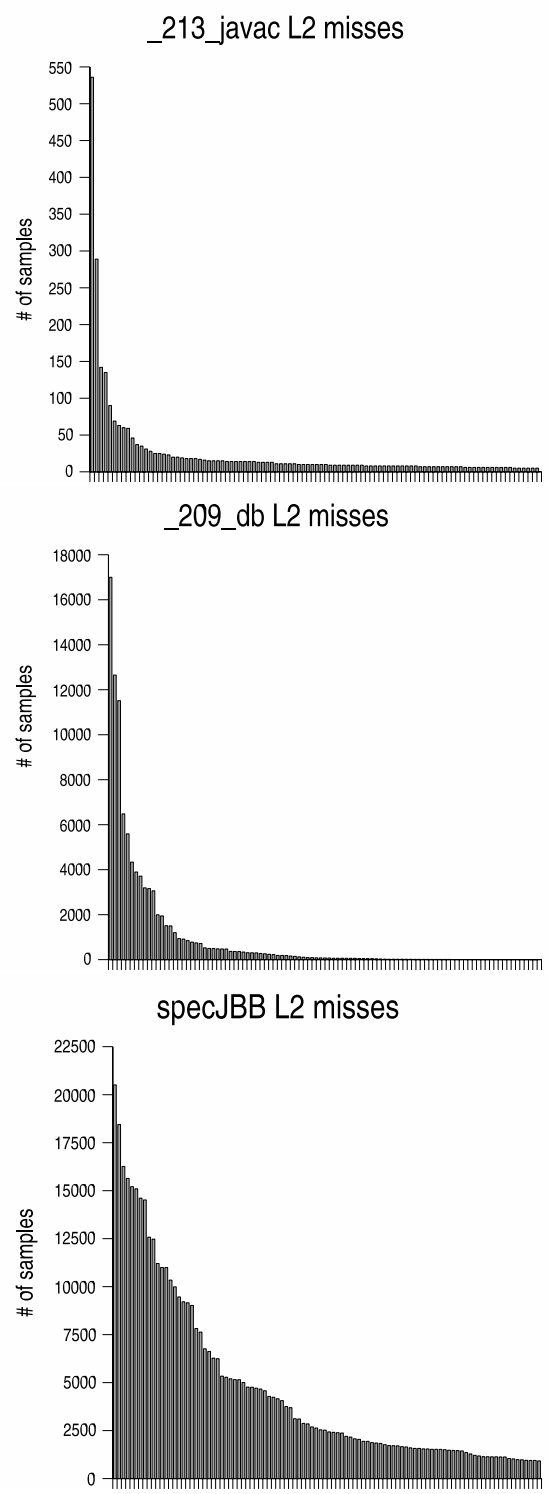


Figure 4.2: Histograms of L2 cache misses (100 most contributing load instructions).

We selected three programs with different performance characteristics for this experiment: *db* and *javac* from the JVM98 benchmarks [83] and the *specjbb2000* benchmark [84].

For our measurements we use an sampling interval of 1000 events for L2 misses and 10000 events for L1 misses. For *db, javac*, and *specJBB* we measure the frequency of L1- and L2-misses. Figure 4.1 and Figure 4.2 show the histogram of the 100 most contributing load instructions for L1 and L2 cache misses.

The absolute number of events differs very much between these benchmarks: *javac* has by far the lowest number of cache misses. This benchmark seems not limited by memory bandwidth. *db* and *specJBB* exhibit roughly 2 orders of magnitude more cache misses. Note that here we use a fixed sampling interval to be able to compare the number of samples between programs.

The 100 most contributing load instructions produce 37% of the L1 misses in *javac*, 98% in *db*, and 55% in *specJBB*. The distribution of L1 cache misses is much more uniform for *javac*. This type of application are in general harder to optimize for a JIT compiler since it is not clear where to perform expensive optimizations.

The picture is different for long latency L2 cache misses. Figure 4.2 shows the same information for L2 misses. There, the 100 most contributing loads are responsible for 74%, 99% and 85% of the events.

For *db* the distribution of L1 and L2 misses is quite similar – there are very few hot loads. In *javac*, on the other hand, the L1 misses are generally distributed over the whole program, whereas the L2 misses are slightly more localized. *specJBB* is somewhere in-between, but more closer to *db* (except for the top instruction in *specJBB* that produces an order of magnitude more L1 misses than the second most significant instruction). Especially in *javac*, the cache misses are spread out over a large number of load instruction in different methods.

To further analyze the distribution of cache misses we computed the 80% quantiles for each program and event.

Table 4.1 shows the total number of contributing load instructions (column 100%) and the 80%-quantile of the distribution of cache misses. The third column for each program shows the percentage of loads producing 80% of all cache misses relative to the total population of load instructions.

We can clearly see that the programs fall into two categories: *db* and *specjbb* have a high concentration of cache misses on relatively small number of load instructions: 4% resp. 6% contribute to 80% of all L1 cache misses. The picture is similar with L2 misses (4% and 3%). *javac* has a much more wide-spread distribution with a long, heavy-weight tail. Here, 41% of all load instructions produce 80% of all L1 cache misses (23% for L2 misses).

For programs of the first category (*db, specjbb*) the JIT compiler can usually afford to focus expensive optimizations on the few hot spots in the program. Program of the second category (like *javac*) are in general hard to optimize in a dynamic compilation

HPM event	db			javac			specjbb		
	100%	80%	percent	100%	80%	percent	100%	80%	percent
L1 cache misses	571	21	4%	3172	1296	41%	8526	477	6%
L2 cache misses	295	13	4%	672	153	23%	2361	76	3%

Table 4.1: 80% quantiles for L1 and L2 miss distribution on load instructions.

environment because it is hard to get achieve a good trade-off between compilation time and optimization benefit.

In general, this distinction is more critical for client applications where start-up performance it critical. In long-running server programs the compiler would can usually spend much more effort on optimizations without considering compilation time cost.

Previous research [24] shows a similar picture when looking at execution time spent in individual method: *javac* has a much larger working set of hot methods than *db* or *specjbb*. As a consequence it benefits much less (if at all) from the adaptive optimization system found in Jikes RVM. In this work we are measuring cache misses instead of execution time spent, but in general we make the same observation that programs with few hot loads are easier to optimize for data locality and show a larger benefit than programs with a more wide-spread distribution of cache misses.

Chapter 5 will discuss the impact of object co-allocation, an optimization to reduce cache misses, on these programs in detail.

4.2 Data address distribution of memory loads

In Section 3.3.3 we showed how we can collect data address profiles with our system. One application of the data address profiles is to determine where on the heap an application performs its memory loads and stores or where cache misses do occur. There are many possible properties we can count with data address profiles using HPM information. Examples are:

- Counting access to stack versus heap variables.
- Counting access to short-lived and long-lived objects in a generational garbage collector.

Some of these metrics like counting loads or stores could also be obtained with software-only technique by instrumenting code, but using HPM information has the advantage that it does not require any instrumentation and offers a low measurement overhead so that it is better applicable for online optimizations.

Here we use data address profiles to find out in which parts of the Java heap cache miss and DTLB miss events occur most frequently.

The Java heap in Jikes RVM consists of several logical spaces where objects are allocated. The number and the configuration of the different spaces depends on the garbage collector algorithm used. Here we only show the setting for the default generational collector:

- VM space (*boot*): The core VM objects + code resides in this space. It is initialized from the bootstrap image loaded at startup and contains VM internal code and data structures only.
- Meta space (*meta*): All meta-data that is used by the memory management and the GC is allocated in the meta space.
- Immortal space (*immortal*): It contains global VM-internal objects that are allocated at run-time. They are not collected by the GC and reside in memory until the VM terminates.
- Large object space (*los*): All large objects ($>$ 8 KB by default) are allocated there. It it managed by a separate GC optimized of large objects. The default GC is not as efficient for large objects.
- Nursery object space (*nursery*): The generational GC initially allocates all objects in the nursery space. By default this space is variable in size can grow until the maximal heap size is exhausted. When this occurs the system triggers a nursery GC). After a nursery GC cycle the remaining live objects are copied into the mature object space.
- Mature object space (*ms*): After one garbage collection the live objects from the nursery space are copied into the mature object space. It contains all objects that are live for at least one GC cycle. Once the mature space exhaust the maximum heap size, the GC starts a full-heap GC cycle. If the full-heap collection is done with a semi-space copying collector (instead of the default mark-and-sweep GC) the mature space is divided into two semi-spaces *ss0* and *ss1*.

Each of the object spaces are placed at a fixed virtual address range, but they occupy a variable amount of physical memory pages (up to the maximal heap size) depending on how many objects are allocated. By looking at an object's header address we can easily determine in which space an object resides.

Figure 4.3 shows the distribution of memory loads on the Java heap. We see that compress has almost no memory loads in the mature space (ms). On the other hand, db and mpegaudio have a significant portion of loads from mature objects.

Note that an access to a stack variable (a local variable or a method parameter) appears in the large object space, since each thread's stack is allocated as a Java object (byte array) in the LOS (default stack size is $>$ 200 KB). It is possible to filter out stack variable accesses by determining the address ranges of all stacks to divide the LOS into stack and heap regions.

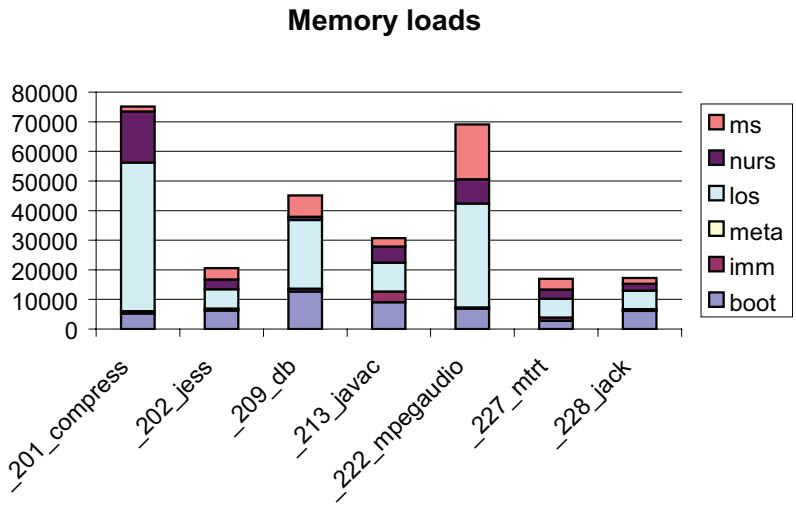


Figure 4.3: Distribution of memory loads in the different VM memory spaces.

This explains the large number of loads in the LOS for many programs. Only compress actually deals with large arrays in the application code. For the other programs most of those loads can be attributed to stack variables.

In our analysis and optimization we only focus on object in the normal heap (i.e. not in the LOS). Therefore, we do not need to distinguish accesses to local variables.

4.3 Distribution of DTLB and cache miss addresses

Using the data address profile we can determine how many cache misses occur in each of the Jikes RVM heap spaces. Figures 4.4 and 4.5 show the distribution of L1 and L2 cache misses for the JVM98 programs. DTLB misses shown in Figure 4.6 have a distribution very similar to L1 cache misses. They y-axis shows the absolute number of events sampled. We used a fixed sampling period for each measurement (10K for L2 misses, 20K for L1 and DTLB misses and 200K for memory loads) to be able to compare the numbers across different programs.

L2 cache misses are not significant for all other JVM98 programs except db. The db benchmark exhibits by far the largest number of cache misses (L1 + L2). Only 16% of all memory loads occur in the mature space (see Figure 4.3, but those 16% contribute 95% of the total L1 cache misses, 99% of the L2 cache misses and 95% of the DTLB misses. This indicates an exceptionally high cache miss rate for mature objects for db.

mtrt shows a similar, but less pronounced trend: Here, the mature space loads make up 21% of all loads and cause 82% of all L1 cache misses, 75% L2 cache misses and 66% of the DTLB misses.

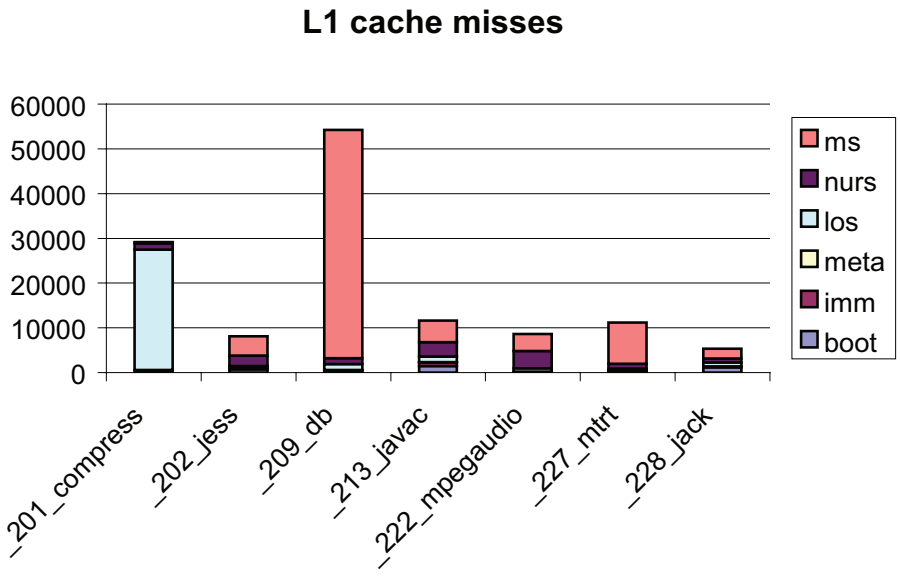


Figure 4.4: Distribution of L1 cache misses in different heap spaces.

The number of cache and DTLB misses in the mature space and the nursery can be used to guide optimization decisions: A runtime system could improve data locality at the place on the heap where most cache misses occur. Also, these numbers can be used to use an appropiate allocation and garbage collection algorithm.

Our own experiences with data locality optimizations confirm the picture shown by the distribution of cache misses: Section 5.1 covers object co-allocation which optimizes data locality at GC time for objects in the mature space. The programs with a high miss rate there also show most benefit from co-allocation.

4.4 Analysis of data cache misses with different GC algorithms

In this section we look at two GC algorithms and study the performance difference between the two on the example of a memory-intensive benchmarks. We also show how cache misses are distributed on the heap when using different garbage collection algorithms.

We compare the two best performing generational collectors in the Jikes RVM: A generational mark-and-sweep collector (*GenMS*) and a generational semi-space copying collector (*GenCopy*). Now we take a closer look at the generational GC framework of Jikes RVM. As described in Section 4.2 each of these two collectors divides the heap into two spaces:

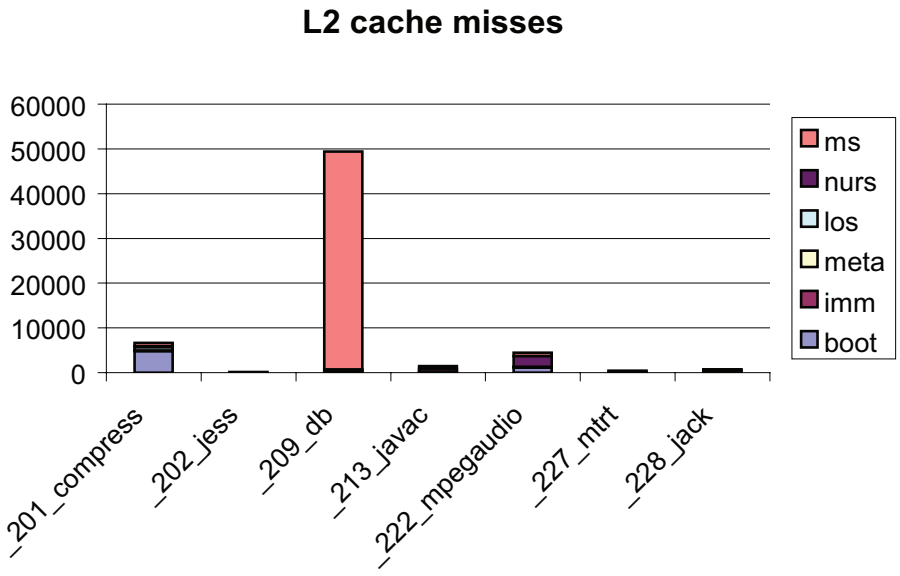


Figure 4.5: Distribution of L2 cache misses in different heap spaces.

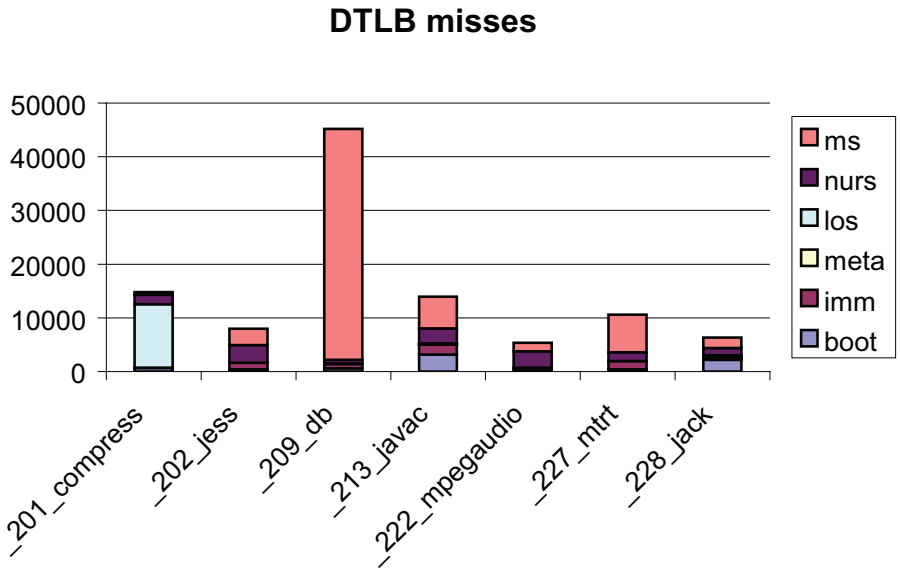


Figure 4.6: Distribution of DTLB misses in different heap spaces.

1. Young object space (aka. nursery)

2. Mature object space

The difference between the two GCs is the algorithm used for full-heap collections:

- GenMS uses mark-and-sweep for full-heap collections. On a full-heap GC it copies objects from the nursery into the mature space which is managed by a free-list allocator using different size classes for objects. This is the default "production" setting in Jikes RVM. In general this is the most efficient configuration for small applications and for small heap sizes.

- GenCopy uses a semi-space copying collector for full-heap collections. It allocates mature objects continuously in the same way as young objects. This collector usually has more GC overhead (due to the additional space overhead of copying). On the other hand copying the objects may provide a better performing data layout. GenCopy is usually performing better when the heap size is large enough so that only few GC cycles occur.

For this experiment we run JBB2000 which is a very memory-intensive benchmarks and allocates a large number of objects. We can expect that GC has a significant impact on overall performance for JBB2000.

First, we compare execution time of these two GC configurations. We break the total execution time down into GC time and application time. Figure 4.7 shows total execution time of JBB2000 broken down into time spent in the GC and time spent in application code for the GenCopy and the GenMS collector. Here, we use a fixed maximum heap size of 512 MB. Each result is the average of 5 runs. The 95% confidence intervals for the execution times are less than 1s for all measurements. Overall execution with GenCopy is on average 5.5 seconds (5.4%) faster than with GenMS.

Figure 4.8 shows the total execution time of JBB2000 with varying heap sizes. For smaller heap sizes (up to 320M) GenMS outperforms GenCopy. Figure 4.9 shows the GC overhead of both collectors: In general GenMS is faster at heap sizes $<$ 320M. At 240M the GC overhead of GenCopy is 4x higher than with GenMS. (The data point for GenCopy at 160M is missing since it did not finish due to an `OutOfMemoryError`. The cross-over point for total execution time is at around 320M where GenCopy becomes faster. GC time alone is similar at large heap sizes: From 400M heap size GenCopy is between 9% and 15% faster than GenMS.

To find out where this performance difference comes from we look at the time spent in actual application code and the time spent in the GC. Figure 4.7 shows the time time spent in the application and in GC code in seconds.

Since data address profiles allow us to measure the direct impact of GC strategies not only on the number but also on the distribution of cache misses we can directly observe the characteristics of GC algorithms. Previous work indicates that copying GC algorithms

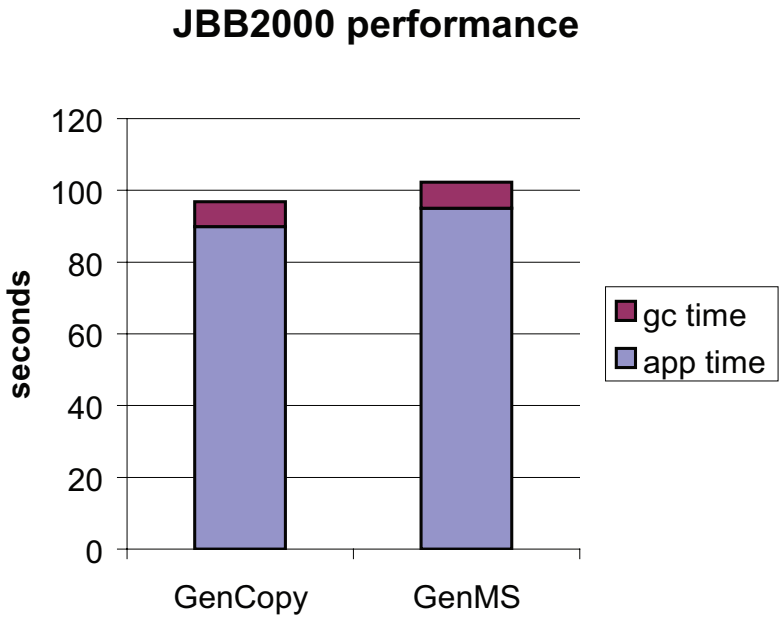


Figure 4.7: Total execution time, GC time and application time for JBB2000 with the GenMS and the GenCopy collector.

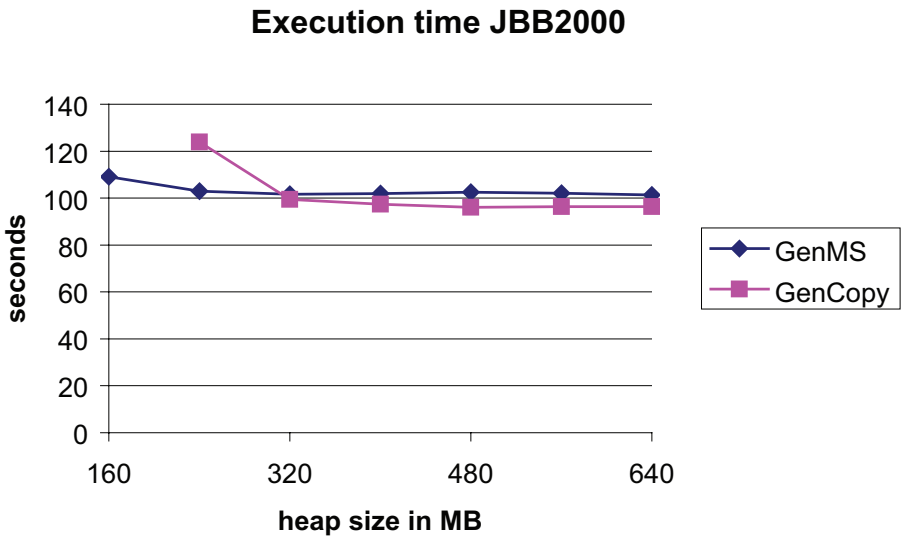


Figure 4.8: Total execution time for JBB2000 with varying heap size.

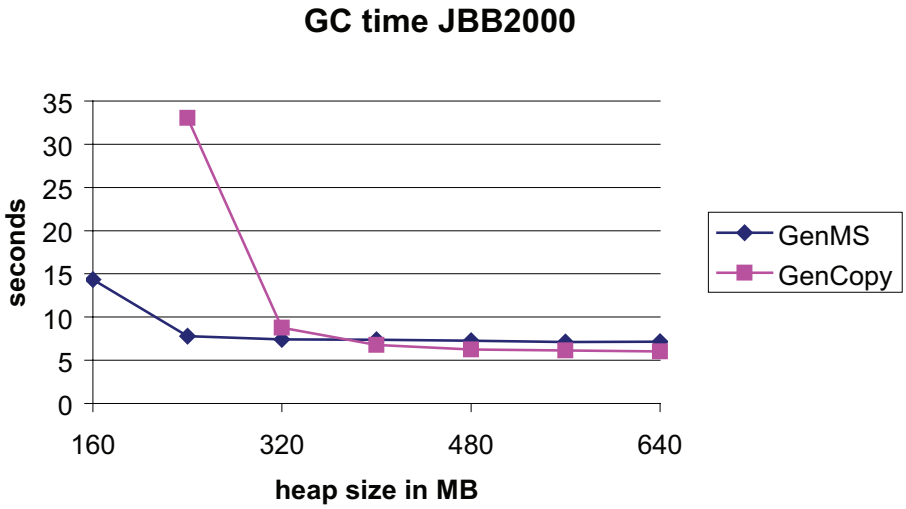


Figure 4.9: GC time for JBB2000 with varying heap size.

provide better cache performance than others [27]. We are using our HPM-generated data address profiles to confirm these findings.

With a 512M heap GenMS spends on average around 7.3 seconds (7.2% of total time) in the GC whereas GenCopy takes 6.9 seconds (7.2% of total). This means that even though both collectors perform very similar with only 0.4s difference, the overall performance of GenCopy is better by around 5.5s. One obvious explanation is the improved data layout of objects on the heap due to the semi-space copying algorithm. Since we have the possibility to directly attribute cache misses to memory addresses we can use our HPM infrastructure to confirm this explanation.

Figures 4.10 and 4.11 show the distribution of L1 and DTLB misses with two different generational garbage collectors with a fixed heap size of 512M. The total number of L1 cache misses with the GenCopy collector is about 6.5% smaller than with the GenMS collector. The fraction of misses occurring in the mature space (ms) is very similar for both configurations: 71.8% for GenCopy and 70.2% for GenMS.

The difference between the two garbage collectors is more obvious when looking at DTLB misses[1]: In total, GenCopy has 18% less DTLB misses than GenMS. From the stacked columns in Figure 4.11 we can see that almost all of these additional DTLB misses with GenMS occur in the mature space: In total 65% of all DTLB misses occur in mature space with GenMS, whereas only 59% for GenCopy.

Of course limiting the total heap to a smaller size helps the GenMS collector because it has less space overhead compared to GenCopy, but this aspect is not central to this

[1]We also experimented with L2 cache misses, but due to hardware limitations of the P4 PMU the absolute numbers for L2 misses were not meaningful for a direct comparison.

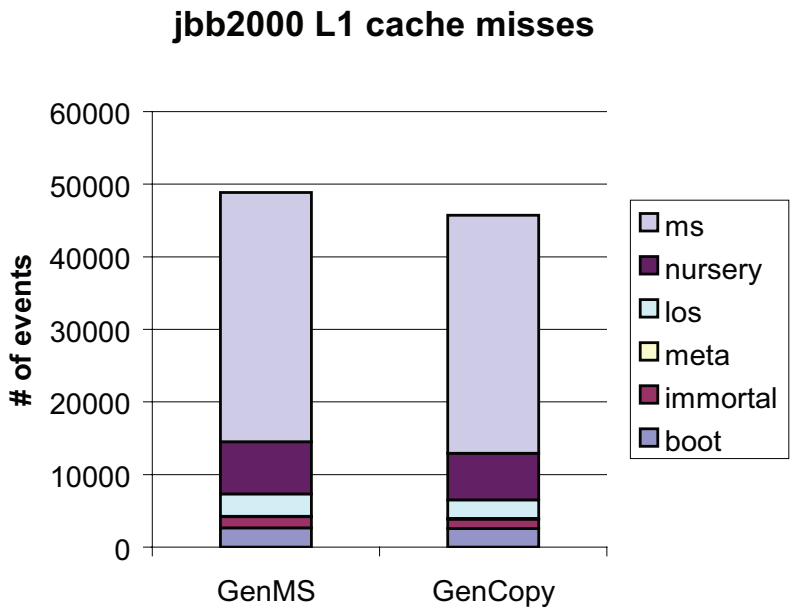


Figure 4.10: Comparison between GenMS and GenCopy collectors: L1 cache misses (absolute number of samples).

experiment where we assume plenty of free main memory. This is the reason we use a fixed heap size of 512M for the experiments here.

These numbers clearly confirms the observation that the copying garbage collector improves data locality in the mature space over the mark-and-sweep collector [27]. The explanation for the increase of DTLB misses is that GenMS allocates mature objects in size classes to limit fragmentation. Different size classes always start at separate 4K pages and therefore objects that are connected by a reference and are accessed consecutively may end up in different pages.

4.5 Summary

Method- or basic-block-level precision is often not enough to characterize the performance behavior of complex Java application. We present how our HPM measurement infrastructure can be used for detailed instruction-level performance analysis. First, we are able to identify performance-critical load instructions that miss the cache frequently. Applications show a large variation in the distribution of cache misses over the program code. Some exhibit few hot-spots on which the compiler can focus optimizations. Other applications show a more uniform distribution and are hard to optimize for a JIT compiler that has to weigh optimization cost versus performance benefit.

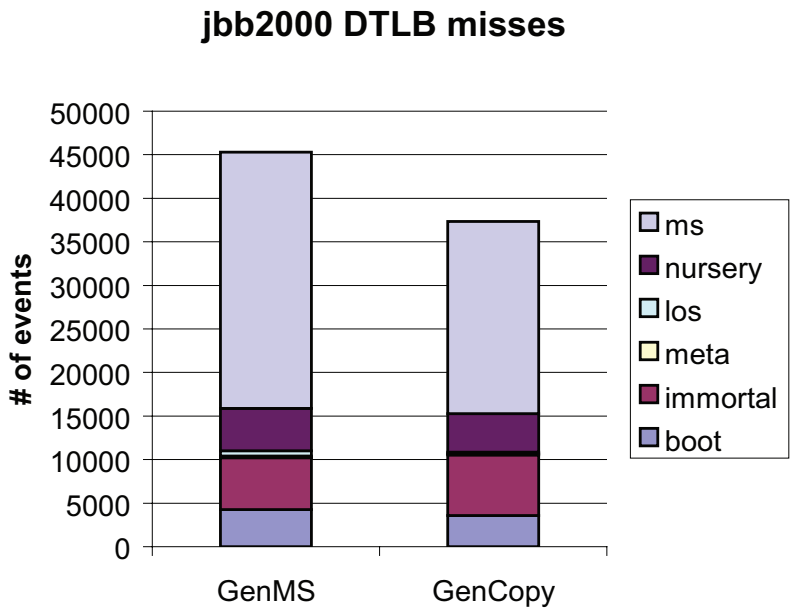


Figure 4.11: Comparison between GenMS and GenCopy collectors: DTLB misses (absolute number of samples).

Using HPM event sampling we can not only collect instruction-address profiles, but also - if the hardware allows - data address profiles: Using the infrastructure presented in Chapter 3 we measured data address profiles for different Java applications on an IA-32 (P4) platform. Data address profiles allow to relate HPM events to memory addresses. This enables us to locate HPM events in memory and to identify performance-critical data structures in a program. Data address profiles are a very useful tool to characterize the performance behavior of a program at a detailed level. We show which metrics can be obtained using data address profiles and how events (memory loads, data cache misses, DTLB misses) are distributed in the different regions the Java heap.

One important application of performance analysis using data address profiles is to predict how successful a data locality optimization will be on a given application. For instance, if the an optimization targets objects in the mature space and the vast majority of cache misses occur in the nursery space or vice versa, we cannot expect a substantial improvement in performance.

Finally, we present a detailed performance analysis of different GC algorithms in Jikes RVM and confirm earlier results [27] by directly observing the effect of GC strategies on data locality using data address profiles. The key difference to previous work is that we can actually measure cache misses within different regions of the Java heap. Previous work could only speculate about the speedup or slowdown of a specific GC algorithm due to data locality.

5

Optimizations using HPM feedback

This chapter describes two optimizations that use HPM feedback to guide optimization decisions. First, Section 5.1 presents object co-allocation, an online optimization to improve data locality in a Java VM.

Section 5.3 shows how HPM data can be used to improve loop unrolling heuristics in a high-performance static compiler. We use an off-line approach to show the potential of using HPM feedback for loop unrolling. The primary goal of this study is to show how much static heuristics can be improved by using HPM feedback.

5.1 Coallocation guided by HPM feedback

In this section we present object co-allocation as an example optimization for data locality that applies the gathered HPM performance data in a modified generational garbage collector [85].

Java applications often suffer from irregular memory access pattern due to the use of many objects and references. Typical Java programs contain many access path expressions (like `a.b.c`). Following such chains of references can cause performance problems if the objects involved are distributed on the heap in an unfavorable way.

Object co-allocation tries to improve the situation by detecting the most critical reference chains and re-ordering objects toward a more cache-friendly data layout.

5.1.1 Analysis

Objects that have a reference between them are often accessed at the same time. A frequently occurring example is a `String` object with its internal representation contained in a separate `char[]` array object. If such two objects are placed in different cache lines or different memory pages there may be an additional performance penalty due to cache or TLB misses when performing some operation on these objects.

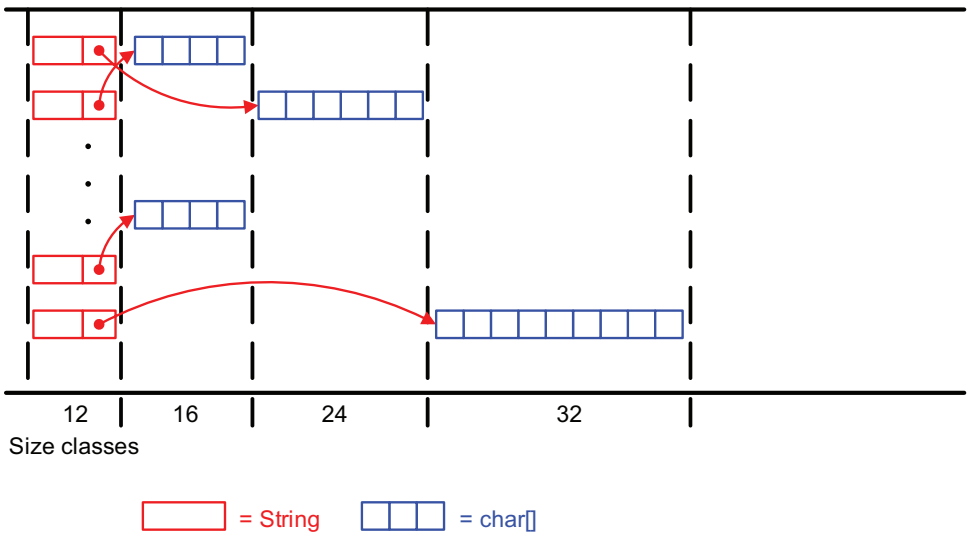


Figure 5.1: Heap structure without coallocation.

The Jikes RVM uses a generational mark-and-sweep (GenMS) algorithm for GC. This approach is usually fast and has low space overhead compared to a pure copying collector.

One drawback of the GenMS algorithm is that mature objects are often placed in a sub-optimal layout with respect to cache performance. Objects are allocated in size classes to limit heap fragmentation, and they are not moved as in a copying GC.

Figure 5.1 shows objects on the heap using the default allocation algorithm. The downside of this allocation strategy is often poorer cache performance: Objects that are accessed consequently in time may end up allocated in different size classes (and therefore, in different memory pages).

Section 4.4 showed that the GenCopy algorithm provides better data locality whereas the GenMS collector has less GC time overhead. From this observation we can conclude that a combination of these two properties may yield a speedup over both approaches (i.e. best of the two). Object coallocation tries to achieve this goal by placing object which are frequently missed in the cache adjacently in memory.

5.1.2 Approach

Co-allocation allocates selected pairs of objects together in one size class. It allocates two objects o_1 and o_2 where o_1 has a reference field pointing to o_2 ($o_1.x == o_2$) in one contiguous memory block.

A typical Java access path expression $o_1.x.y$ is translated into two *getfield* operations by the bytecode-to-IR translator:

```
I1: t1 = o1.x;
I2: t2 = t1.y;
```

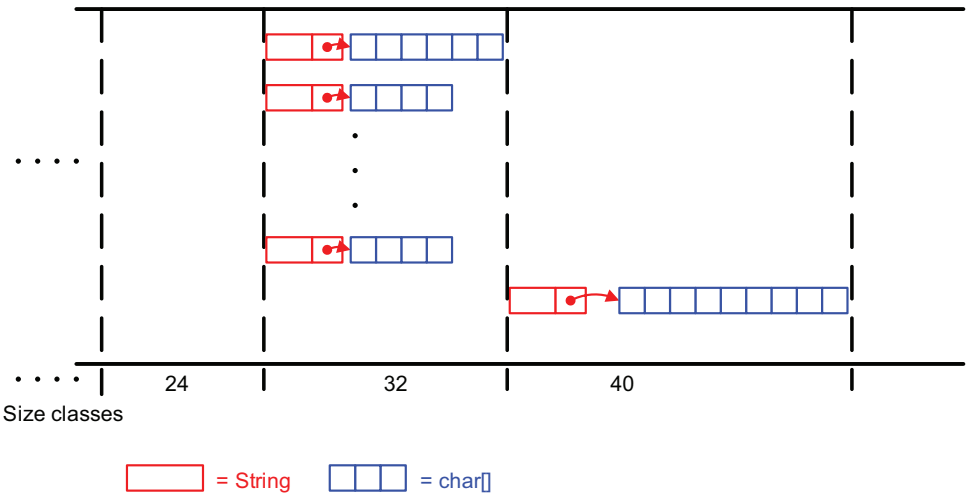


Figure 5.2: Heap structure with coallocation.

If both indirect loads access the same cache line we can avoid a cache miss on the second indirect load `I2`. This indicates that coallocation works best for small enough objects where

$$size(o_1) + size(o_2) < size(cacheline).$$

There is less or no benefit if the object's size exceeds the size of a cache line ($size(o_1) > size(cacheline)$ or $size(o_2) > size(cacheline)$). Coallocation may or may not improve performance in this case depending on the alignment of the objects. Also, objects referenced by an array cannot be co-allocated. The compiler can in general not know which array elements to co-allocate.

Figure 5.2 shows an abstract model of the heap after performing coallocation. Objects that are accessed consequently and cause a large number of cache misses are allocated together in the same size class. Since the CPU fetches whole cache lines from main memory we get an implicit prefetching of the co-allocated object.

For larger objects and arrays, software prefetching [17] would be a promising candidate to hide the potential load latency since there is no restriction on the size or on the layout of the objects.

The GenMS GC in our system does bump-pointer allocation for young objects and copies matured objects into the mark-and-sweep collected mature object space. Tenured objects are managed using a free-list allocator that allocates objects into 40 different size classes up to 8 KBytes (=VM default setting) to minimize heap fragmentation. Larger objects are handled in a separate portion of the heap called the large object space (LOS).

The GenMS GC offers better space efficiency than a pure copying GC (no copy reserve needed). On the other hand a copying GC is known to generally enhance data locality as we have already shown in Section 4.4. The goal of our co-allocation is to combine those two advantages, i.e. having a space-efficient GC that provides good data locality automatically by using feedback from the hardware.

We use cache misses as a simple metric to decide which objects to co-allocate. This is only an approximation since not every cache miss actually degrades performance: The compiler and/or the hardware may be able to completely overlap the miss latency with executing other instructions. Unfortunately, it is not possible to collect detailed information about actual data stalls on the P4.

The online optimization consists of three parts:

1. Filtering of instructions of interest at method compilation time
2. Monitoring cache misses for individual classes and references
3. Nursery tracing algorithm that support co-allocation

The first part is performed for each method compiled by the opt-compiler. As a consequence the monitoring system does not consider instructions in non-optimized methods which are only compiled by the baseline compiler. However, this is not a major limitation since those methods are rarely executed (otherwise they would be selected for recompilation by the JIT). Part two is done concurrently to the execution of the application. The sample collector thread periodically invokes the monitoring module that performs the bookkeeping and translates the raw data. The third part is implemented in the garbage collector where the cache miss data about field references are used to guide co-allocation. The following sections describe each of these three parts in more detail.

5.1.3 Mapping cache misses to object references

For each method that is compiled with the opt-compiler (as selected by the AOS) the sample collector thread performs an additional pass to filter out instructions that must be monitored for cache misses in the HPM module: we are interested in reads/writes to objects that are referenced from another heap object.

From the raw HPM samples alone we can find out the type of object (e.g. `char[]`) where a cache miss occurred. To be useful for optimization we need to retrieve more context information about these load instructions. For example, if we measure many cache misses on `char[]` we would like to know which array was actually accessed.

Initially, the compiler creates a mapping of instruction pairs: For each heap access instruction `S` it checks if the target address is loaded from a field variable `f` (also located on the heap). If yes, it saves a tuple (`S`, `f`). The motivation is that co-allocating the parent object with the child object increases the chance that both objects lie in the same cache line. This way the child object is implicitly prefetched when accessing the parent object. The opt-compiler computes these tuples by walking the use-def edges [19] upward from heap access instructions (field/array access, virtual calls and object-header access).

Figure 5.3 shows an example access path expression with its Java bytecode. Our analysis would create a mapping with instruction and field y (`I3`, `A::y`). For illustration we show the bytecode here - internally we actually use the actual high-level IR instructions

```
class A {              void foo() {
  A x;
  A y;                   ... = p.y.i;
  int i;
}                      }
```

```
I1: aload_2        // Local var p
I2: getfield   y; // Load field y
I3: getfield   i; // Load field i
```

Figure 5.3: Example bytecode for expression `p.y.i`.

that correspond to the bytecode. If we encounter a miss on `I3` (load of field `i`), we increase the event count for associated reference field (`A::y`). We keep a per-reference event count which tells the runtime system how many misses occurred when dereferencing the corresponding access path expressions.

In general, for all indirect memory loads we would like to find where the load address (reference) originated from. The compiler can obtain this information by looking at the IR and the control flow graph and computing the reaching definitions of the reference operand of the indirect load in question. Since our IR is already in SSA form [43], we just have to search backward in the SSA graph to identify the definitions of the reference operand and examine the defining instruction. There are 4 categories of instructions that can define an reference operand:

1. `aload`: Load from local variable or parameter
2. `getfield`, `[iabjc]aload`: Heap object/array load
3. `getstatic`: Load from global (=static) variable
4. ϕ-function: In case of multiple reaching definitions.

For co-allocation we are only interested in loads that have their address defined by a `getfield` operation. In case we encounter a ϕ-function, we recursively process the ϕ-operands to search for all normal definitions.

5.1.4 Assigning weights to references

Once we identified all candidate loads we have a set of tuples that contain all references expressions that are candidates for co-allocation. This is done at compile-time.

During run-time we process HPM samples and assign cache misses the candidate expressions. On each cache miss we distribute a weight of 1 onto all matching candidate expressions.

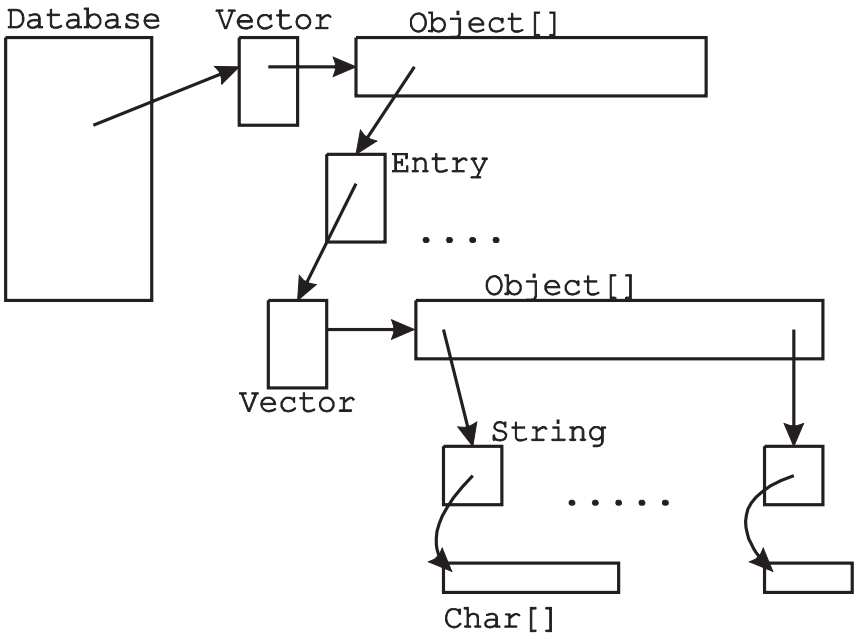


Figure 5.4: Heap structure in the SPECJVM98 db benchmark.

For determining which reference field a cache miss should be attributed to, we have to analyze the program code and look at potential candidate load instructions. A cache miss sample is attributed (fully or partially, in case there are multiple definitions reaching the candidate load instruction) to a reference fields `f` if the source instruction `S` is among the instructions of interest (i.e. a mapping (`S`, `f`) exists).

Figure 5.4 shows a part of the heap for the JVM98 db benchmark. Each of the edges represents a reference field. Cache misses are attributed to these references. Basically, each edge is annotated with a weight corresponding to the number of cache misses occurring when accessing that object.

Figure 5.6(a) shows an indirect load a_1`.z` and the single definition of its address operand a_1. When a cache miss occurs at that load, the compiler assigns it to the reference `x` that points to the missed object a_1`.z`.

Figure 5.6(b) shows an indirect load a_3`.z` where the base address `a` is defined by an indirect load (either `p.x` or `q.y`). Here, the compiler cannot attribute the cache miss at `a.z` uniquely to one reference because there are more than one definitions of the base address of the target object `a`.

In this case we assign fractional weights to each of the references according to the edge profile information. In Figure 5.6, the weight for `p.x` would be

$$n_1/(n_1 + n_2).$$

For `q.y` we have

$$n_2/(n_1 + n_2).$$

AssignWeights(LoadInst s)

$a \leftarrow$ AddressOperand(s)
AssignWeightsRecursive(a, 1)
return

AssignWeightsRecursive(Opnd p, float f)

$d \leftarrow$ def(p)
if d is normal def **and** d is getfield **then**
  Weight[FieldDesc(d)] += f
else if d is ϕ-function **then**
  for all p in ϕ-operands(d) **do**
    AssignWeightsRecursive(p, $f * freq(p)/(freq(p_1) + ... + freq(p_n))$)
  end for
else
  {Don't care for other defs (e.g., parameters, locals, statics vars)}
end if
return

Figure 5.5: General algorithm for assigning weights to field references using SSA form and the existing edge profile information.

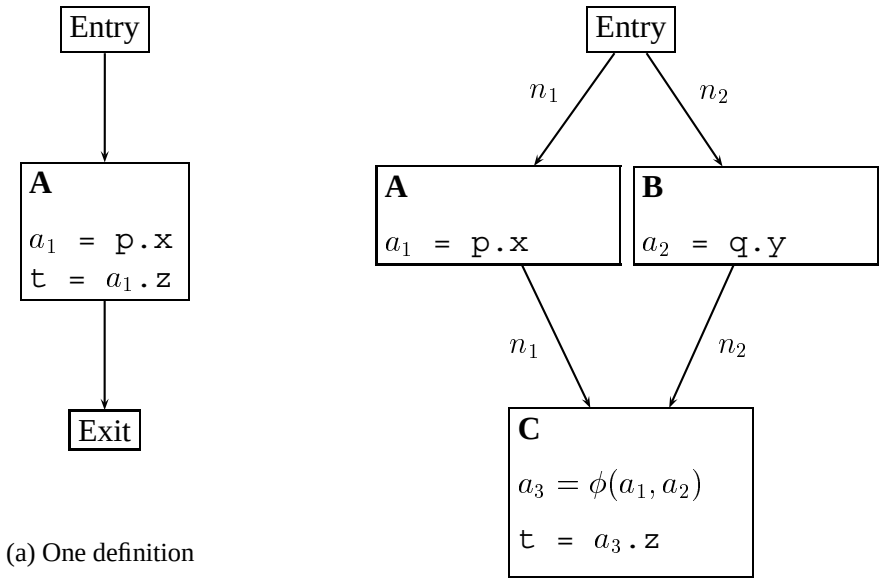
If no edge frequencies are available, we fall back to default heuristics: e.g. give weight 0.5 for each path in an if-else-region or assume a loop body is executed a fixed number of times.

Figure 5.5 shows the general of assigning a weight to all candidate expressions in pseudo-code: Initially we invoke the function $AssignWeightsRecursive$ with a weight of 1. If an indirect load address has $>= 2$ definitions (i.e. it is defined by a ϕ-function) we have to recursively follow all use-def edges backward and assign fractional weights according to the corresponding CFG edge frequencies.

The auxiliary function $AddressOperand$ returns the address operand of a load instruction. Def returns the defining instruction of an operand, and $FieldDesc$ gives the Java field descriptor for a Java *getfield* instruction.

However, in reality the case of multiple (non-phi) definitions is rare. When looking at all SPECJVM98 [83] programs, only 5% of the indirect loads have $>= 2$ definitions of the load address. 95% have only a single reaching definition of their address operand.

Since we can only compute reaching definition within one method, this approach is limited to method boundaries. This means, that the source of reference operands defined by method parameters cannot be identified. If the definition of an address operand is unknown, the system just skips monitoring cache misses for this load instruction. Aggressive inlining as performed at the highest optimization level in many JIT compilers may improve the situation in such a case.



(a) One definition

(b) Two definitions

Figure 5.6: CFG with an indirect load instruction where the base has one (a) and two definitions (b). In Figure (b), the edged are annotated with their execution frequencies n_1 and n_2 taken from the edge profile information.

Class	Field	Type	Weight
Vector	elementData	Object[]	1866
String	value	char[]	1852
Entry	items	Vector	1111
Database	index	Entry[]	853
Vector$1	this	Vector	37
...	...	...	...

Table 5.1: Sampled weights for field references in SPECJVM98 db when monitoring L1 cache misses.

5.1.5 Online monitoring

Currently, we set the system up to monitor events in application classes only and exclude events occurring inside VM code[1]. This is not a limitation of the monitoring system itself, but just because the optimization only deals with objects allocated in the user application code.

If there are multiple reference fields in a class that are potential candidates for co-allocation, the system selects the one with the largest weight. To make sure that the weights reach a stable state we require that a certain threshold of cache misses is measured for a reference before co-allocation is enabled. We experimentally determined that a hot-reference threshold of 100 samples per reference field already yields stable results and also gives a short enough "warm-up" period so that coallocation can be applied early during the program execution.

The rate of events for each reference field is measured throughout the execution and this allows detecting phase changes in the execution or checking whether an optimization decision by the JIT or the GC had a positive or a negative impact. On many platforms, the effect of a data locality optimization is difficult to predict in general. A system that includes feedback based on a performance monitoring unit allows an assessment of the effectiveness of an optimization step. If the transformation improved performance, the system can proceed normally. If the transformation reduced performance, either a different optimization step can be performed or it is possible to revert to the old code.

5.1.6 Nursery tracing with co-allocation

When the GC encounters an object that contains reference fields during performing a nursery space collection it checks if it is possible to co-allocate the most frequently missed child object: we have to check if both objects together do not exceed the size limit for the free-list allocator. Object larger than this limit are allocated in a separate large object space. The VM keeps a list the reference fields for each class type sorted by number of associated cache misses.

Figure 5.7 shows the GC tracing procedure with co-allocation in pseudo-code. For simplicity and better understanding we omit details of the copying process and all the synchronization code that is necessary for multi-threaded tracing in the GC. The method `traceObject` traces each object and if necessary copies it from the nursery space. It takes an object reference as input parameter and returns an object reference. If an object is already copied, the function just returns the copy. Otherwise it creates a copy and returns it.

For objects that have a reference to a co-allocated object, we perform three checks:

1. The reference has to be non-null.

[1]Since Jikes RVM is written in Java itself we could analyze the performance of the VM code in the same way as we do for the Java application running in the VM. However, in this work we only focus on the performance behavior of application code.

2. The sum of the sizes of both objects has to fit into the largest size class ($<$ MAX-SIZE)

3. The co-allocated object has to reside in the nursery space since it might be already copied or co-allocated with another object.

If all three checks succeed, the function copies the first object and co-allocates the second object. The helper functions *Copy* and *CopyAndCoallocate* (not shown here) perform the low-level copying operations (memory allocation, address calculations, memory copy). If one check fails, the function just continues and treats the object as a normal object.

The function *SetForwardingPointer* marks an object in the from-space as forwarded and stores a pointer to the new copied object. *GetForwardingPointer* returns the forwarding pointer of a forwarded object.

After an object has been copied it is entered into the tracing queue (using the helper function *Enqueue*) so that its references are scanned later.

When deciding to co-allocate two objects the GC just requests enough space to fit both objects. They will be assigned to the appropriate size class by the free-list allocator that manages the mature space. Without co-allocation the objects may - depending on their size - end up in different size classes.

Note that co-allocation may increase internal fragmentation because there is only a limited number of size classes (40 in the default allocator) that do not cover each size exactly. The actual results depend on how co-allocated objects fit into their assigned size classes, but in general the application will use slightly more memory with co-allocation. For a very tight heap size this may even degrade performance since there may be more garbage collections.

By using the GenMS collector with co-allocation we combine space-efficiency and good data locality. None of the existing collectors provides this combination. Of course an optimized static copying strategy could achieve a similar benefit in many scenarios [76], but adapting to an individual application's memory access pattern proved to be important [58], and it has been shown that data locality optimizations often help in some cases and hurt in others. Detecting those cases at run-time is a strong argument for using performance counters for guidance.

5.2 Experimental evaluation of object co-allocation

In the following section we study the effect of object co-allocation on a set of Java benchmarks. We compare the baseline with our co-allocating GC in different configurations and use cache misses to guide the optimization.

```
1  ObjectReference traceObject(ObjectReference object) {
2
3    ObjectReference newObject = null;
4    if (current space is from-space)
5      return object
6
7    if (isForwarded(object))
8      return GetForwardingPointer(object)
9
10   if (Type(object).hasCoAllocation())
11   {
12     coallocObject = Reference to co-allocation candidate object
13     if (coallocObject != null &&
14         Address(coallocObject) is in nursery-space &&
15         Size(object) + Size(coallocObject) < MAXSIZE)
16     {
17       newObject = CopyAndCoallocate(object, coallocObject)
18       newObjectCoalloc = ObjectReference.fromAddress(Address(newObject)
19                          + Size(object))
20       SetForwardingPointer(coallocObject, newCoallocObject)
21       Enqueue(newCoallocObject)
22     }
23   }
24
25   if (newObject == null)
26     newObject = Copy(object)
27
28   SetForwardingPointer(object, newObject)
29   Enqueue(newObject)
30
31   return{newObject}
32 }
```

Figure 5.7: Nursery tracing procedure with co-allocation.

5.2.1 Experimental platform

We carried out our experiments on a 3 GHz Pentium 4 (Prescott) with 1M L2 cache and 1 GB of main memory. The L1 cache for data is 16K. One cache line contains 128 bytes. The P4 has an out-of-order execution engine and can issue several instructions in parallel. It also includes hardware-based prefetching of data streams.

The system runs a Linux 2.6.16 kernel with the Perfmon2 patches and the corresponding libpfm library (version 3.2).

Our Java virtual machine is Jikes RVM version 2.4.2 [22, 21].

5.2.2 Methodology

The baseline to which we compare our measurements is the "FastAdaptiveGenMS" build configuration of Jikes RVM which is in general the most efficient configurations (default "production" build). It includes the adaptive optimizing JIT compiler [24] and a generational garbage collector with an Appel-style variable size nursery [23]. The mature space is managed by a mark-and-sweep collector. This collector is included with MMTk [28] - the garbage collection framework that comes together with the Jikes RVM.

All timing results are averages over 5 executions. Within each execution we perform 3 runs of a benchmark and report the timing of the third iteration. We choose this scheme to exclude compilation time overhead because the large majority of methods are compiled and optimized during the first iteration of a program execution. Running more iterations (> 3) did not yield any further convergence of execution times for our set of programs. In general we follow the measurement protocols employed in previous work with Java virtual machines [58, 68].

JIT compiler configuration

In our experiments we use two compiler configurations: the pseudo-adaptive configuration and the adaptive configuration which is the default configuration of Jikes RVM.

We use the pseudo-adaptive configuration for the Jikes JIT compiler for all experiments except where explicitly noted differently. Each program runs with a pre-generated compilation plan. This technique, also known as "replay compilation" [58], ensures that the compiler optimizes exactly the same methods and the variations due to the adaptive optimization system are removed. This setting almost completely eliminates non-determinism introduced by the adaptive optimization system. With this configuration the standard deviation in our measurements is negligibly small. As a baseline we choose the compilation plan with the fastest execution time of the unmodified JVM out of 5 executions.

For the evaluation of the total speedup we use the adaptive configuration. It does not use replay compilation, but instead uses the adaptive optimization system [24] to decide which methods should be recompiled during execution. Therefore it is non-deterministic

in contrast to the pseudo-adaptive configuration. The variance of the resulting execution times is significantly larger here, and we show confidence intervals for these measurements. This setting tries to present results as close to a real-world scenario as possible [30]. We use it to validate results previously obtained with the pseudo-adaptive configuration.

Sampling period

We use the adaptive sampling interval as shown in Section 3.4.3 for evaluating execution time. When measuring the number of events (like L1 cache misses) we use a fixed sampling interval to be able to compare the absolute number of samples between program runs.

With a sampling approach the choice of an appropriate sampling interval is critical. The interval should be fine-grained enough to give a statistically representative picture of the program behavior. But, since we are performing the sampling during program execution the overhead should also be reasonably low. As shown in Chapter 3 the runtime overhead is proportional to the number of samples collected by the VM. From the results in Section 3.4.3 we can see that the monitoring overhead up to 2000 samples/sec is below 1% which is reasonably low for an online optimization. For our measurements we choose 2000 samples/sec as the maximum sampling rate [2].

The sampling interval for the online performance monitoring is randomized by periodically changing the lower order bits randomly (8 bits in our configuration). As shown in Section 3.5, this prevents measuring biased results by sampling some locations more often than with true random sampling.

5.2.3 Benchmark programs

The different benchmarks are listed in Table 5.2.

For the SPEC JVM98 [83] programs we are using the largest input size. We performed 3 iterations of each benchmark during each execution and took the time of the last iteration to approximate steady-state performance.

For the DaCapo programs [26] we used the "default" input size[3]. Like with JVM98 We also perform 3 iterations per run and report the times for the third iteration.

PseudoJBB is a version of SPEC JBB2000 [84] with a fixed number of transactions to measure execution time instead of transactions per second. For pseudoJBB we perform report the average total execution times because this benchmark uses a different measurement harness for execution. Since this benchmark has an execution time of over 1.5 minutes, the compilation time overhead is negligibly small.

[2] In practice we found that a value of 1000 samples/sec already provides very good accuracy and low overhead for all benchmarks programs.

[3] The programs *chart*, *eclipse* and *xalan* were excluded because they are not compatible with version 2.4.2 of Jikes RVM.

db mtrt compress jess javac mpegaudio jack	Programs from the SPEC JVM98 benchmarks [83] with the largest workload (s=100) repeated 3 times.
antlr bloat fop hsqldb jython luindex lusearch pmd	Programs from the DaCapo benchmark suite (version 10-2006 MR-2) [26].
pseudojbb	This is a version of SPEC JBB2000 [84] with a fixed number of transactions (n=100000, max 6 warehouses).

Table 5.2: Benchmark programs.

5.2.4 Number of co-allocated objects

Figure 5.8 shows the number of co-allocated object for different sampling intervals using a logarithmic scale. There are 2 programs (compress and mpegaudio) where no objects are co-allocated. They allocate mostly large objects which are placed in the separate large-object space by the allocator or only allocate few objects. Therefore, they have no candidate objects for co-allocation.

The programs with a large number of co-allocated objects (db, pseudojbb, hsqldb, luindex and pmd) are less sensitive to the choice of the sampling interval: The largest interval is enough to cover most objects. In the remaining programs the number of co-allocated objects is several orders of magnitude lower, and co-allocation is more sensitive to the choice of the sampling interval.

5.2.5 Performance impact of co-allocation

Figure 5.9 shows the number of L1 cache misses with co-allocation in the GC turned on relative to the baseline using a large heap size. In Figure 5.10 we summarize the impact on application performance using a range of heap sizes (1-4x minimum heap size).

There is a noticeable reduction in L1 cache misses using HPM-guided co-allocation for several programs (jess, db, pseudojbb, bloat and pmd). mpegaudio shows varying

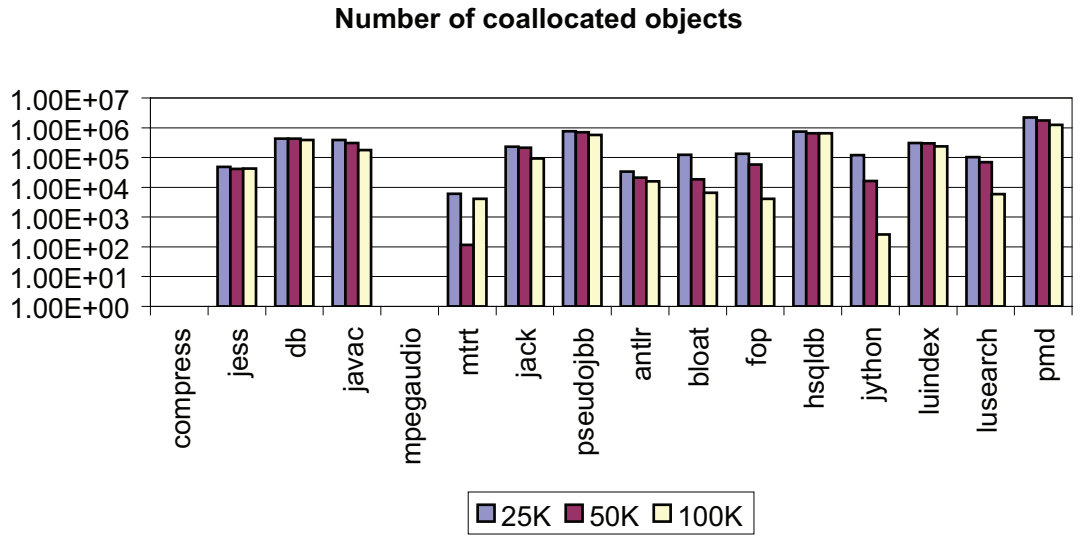


Figure 5.8: Number of co-allocated objects at different sampling intervals (heap size = 4x min heap size).

numbers (from -6% to +5%) that are not due to co-allocation (no candidate objects), but rather show influences from the event monitoring and processing. There is little or no effect on the other programs. From all benchmarks db gets the most benefit: 28% fewer L1 cache misses. This benefit translates into an execution time reduction of up to 14%.

The reduction on L1 misses for jbb, one of the most memory-intensive programs from this suite, is only between 2 and 6%. The resulting speedup is up to 2% for large heaps. Here we observe that there are many frequently missed objects (2.4 million objects were co-allocated) and that the majority of those objects are relatively large (`long[]` arrays with a size of $>$128 bytes). As a consequence, optimizing for reduced cache misses at the cache-line level does not yield a significant benefit for this program. (Using DTLB misses as driver for the optimization decisions does not improve the results since it results in co-allocating the same types of objects.)

As shown in Section 5.2.4 the number of co-allocated objects is rather small (in the order of thousands) for most JVM98 benchmark program. This is the reason why the impact on L1 cache performance is also small (except for db)

This indicates that there are not many mature objects that cause frequent cache misses: These programs have relatively small working sets and/or many young objects that do not benefit from better spatial locality in the mature space.

Overall, three programs (db, pseudojbb, bloat) show a speedup, and 7 programs are slightly slowed down by using dynamic co-allocation. The worst case for large heaps is javac with -2.1% which is in part due to the monitoring overhead (reported in Figure 3.6).

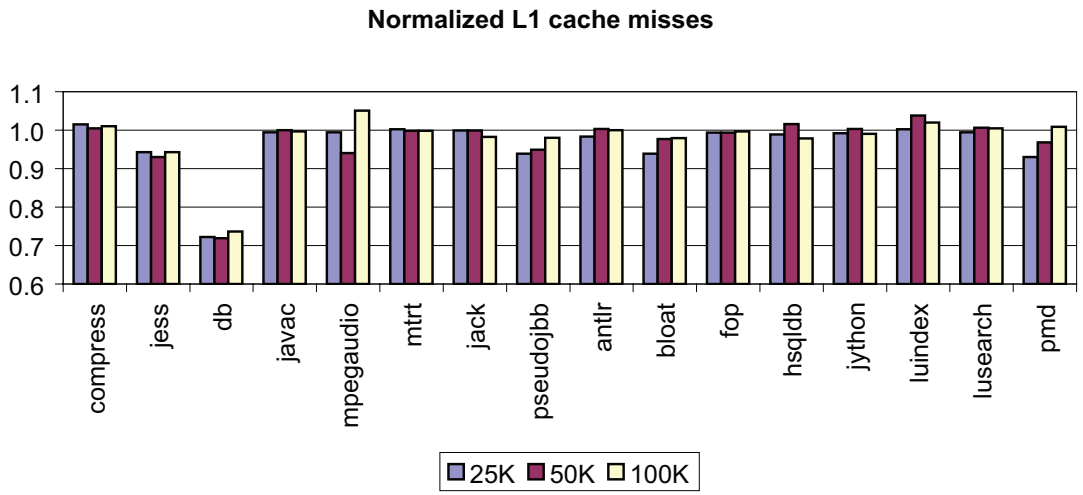


Figure 5.9: L1 miss reduction with co-allocated objects (heap size = 4x minimum heap size).

Figure 5.10: Execution time relative to the baseline for different heap sizes (heap size from 1-4x min heap size) with pseudo-adaptive compilation.

In Section 4.1 we showed that javac has its cache misses spread over a large number of instructions. Our results with co-allocation seem to confirm that these types of programs are harder to optimize than programs like db or pseudojbb that exhibit few hot load instructions.

Note that monitoring is turned on throughout the whole execution even when no candidate objects are found. The performance for the programs where no objects are co-allocated could be improved reduced by turning off monitoring after no candidate object have been found.

For small heaps sizes the picture looks different. db is the only program that still shows a speedup at minimum heap size (9.3%). Generally, co-allocation yields better results at larger heaps because it may increase the load on the garbage collector in a tight heap scenario: The larger the allocated chunks the more internal fragmentation exists due to the limited number of size classes in the free-list allocator. This space overhead may result in additional GC cycles to be performed. When running at the minimum heap size this space overhead factor gets more dominant and almost all programs are either slowed down or have a smaller speedup (e.g., db) with co-allocation.

This additional GC cost plus the monitoring overhead is mostly responsible for the slowdown of the programs where co-allocation does not succeed.

Effect of adaptive compilation

We also measured performance without pseudo-adaptive replay compilation to assess a more real-world scenario and confirm that replay compilation provides a realistic picture of the performance behavior.

For this experiment we show the numbers for the SPEC JVM98 benchmarks and pseudojbb benchmarks[4]. We also invoke each benchmarks multiple times within one JVM invocation to separate startup and steady-state performance. We also show the 95% confidence intervals in this plot since the variations in execution time are noticeable when not using replay compilation. The confidence intervals allows to judge which results are statistically significant [50].

Figure 5.11 shows the resulting relative execution time speedup. The average speedup over all programs is close to 3%, but only for db and pseudojbb the confidence intervals of base and co-allocation do clearly not overlap, which indicates that there is a significant speedup for these benchmarks.

For compress, mtrt and jack average speedups between 1% and 5%, but for these programs also the confidence interval is almost as large as the average speedup. For jess, javac and mpegaudio we measure no effect to up a slowdown of 2% in the worst case (mpegaudio). Also here, the confidence interval is larger than the average difference. In general, the adaptive JIT compilation increases the variance of the execution time considerably. For the programs with such a large variance, we cannot clearly determine the

[4]Jikes RVM 2.4.2 had some stability problems when running the Dacapo programs with the adaptive configuration

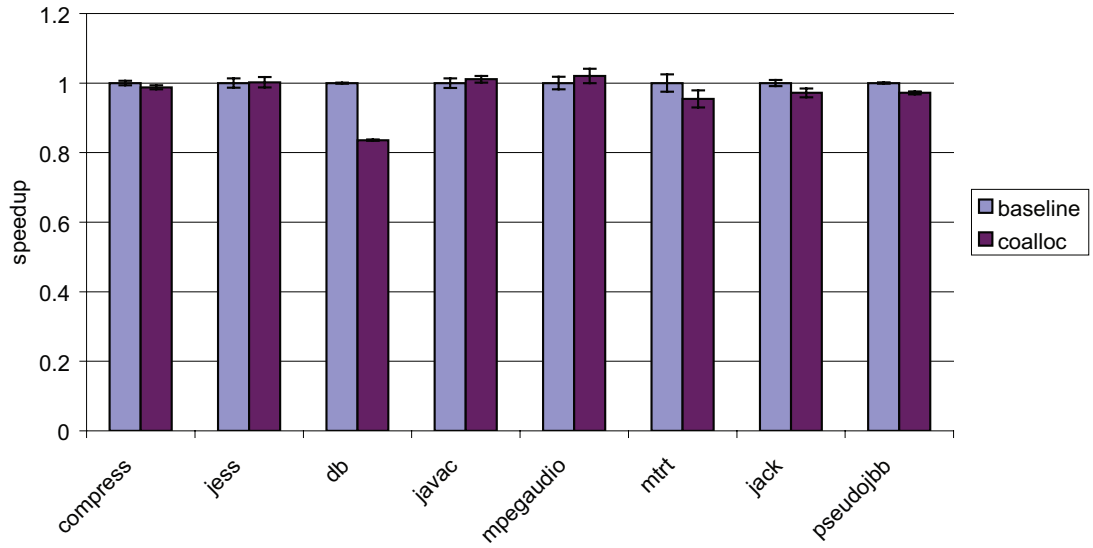


Figure 5.11: Execution time relative to the baseline with the adaptive configuration (default heap size and no compilation plan

effect of co-allocation, and it should be considered inconclusive.

However, the general trend of the results is the same as in the pseudo-adaptive configuration: We definitely see a 3% speedup for pseudojbb and a 16% speedup for for db. Those numbers are slightly different than the results with the pseudo-adaptive configuration (14% for db, 2% for pseudojbb). The average performance of the adaptive configuration is slightly better than the pseudo-adaptive configuration because we use the best-performing compilation plan as a baseline in Figure 5.10.

This confirms that the use of replay compilation shows realistic performance behavior. The results from Section 5.2.5 match the numbers with adaptive recompilation closely, but exclude the large variations in execution time found with the adaptive compilation configuration.

Co-Allocation vs. Copying GC

Figure 5.12 analyzes the performance of db in more detail. Now we compare a generational copying (GenCopy) collector versus the generational mark-and-sweep (GenMS) with object co-allocation. As described in Section 4.4 the GenCopy collector generally improves spatial data locality in the mature space over a non-moving collector - on the other hand it has a larger GC cost at small heap sizes [27]. The performance numbers for db confirm this observation.

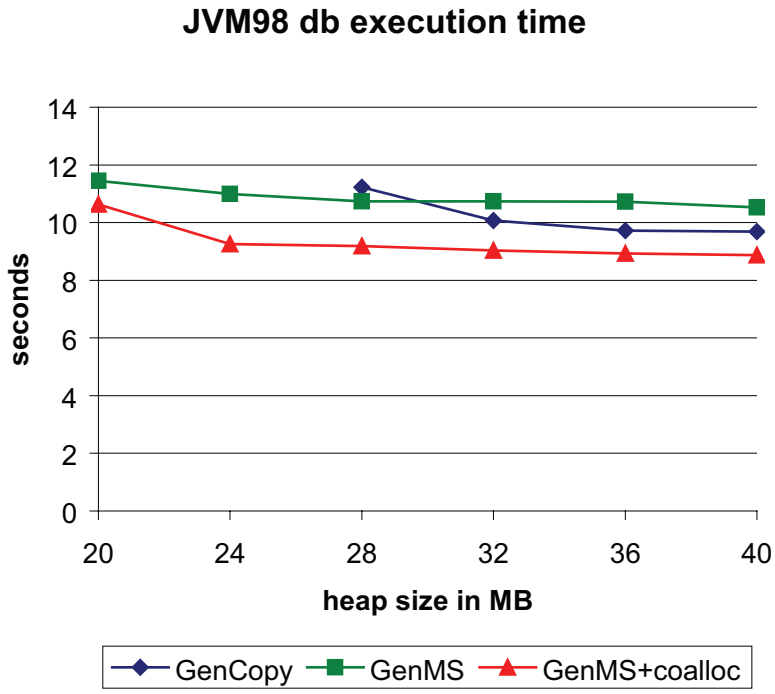


Figure 5.12: GenCopy vs GenMS with co-allocation

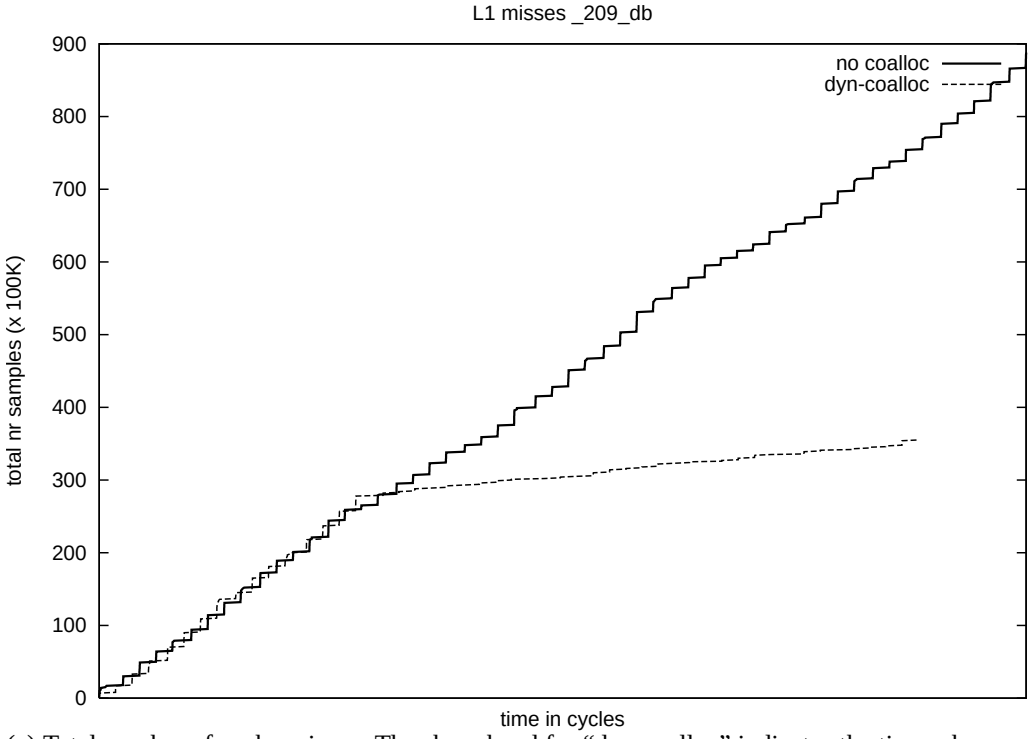
At a heap size of 28MB (smallest possible for GenCopy[5]) we see a speedup of 18% when enabling co-allocation with GenMS versus GenCopy alone. The default GenMS collector (without co-allocation) is in between with 4% speedup vs GenCopy.

The larger the heap gets, the better GenCopy performs relative to the other collectors. From a heap size of 32M up to 40M GenCopy becomes 6% to 9% faster than GenMS. But the interesting observation here is the co-allocating GC consistently outperforms both others (by 8% and 16% at 40M). This indicates that the GenMS collector with object co-allocation can combines good data locality with space-efficiency in one system.

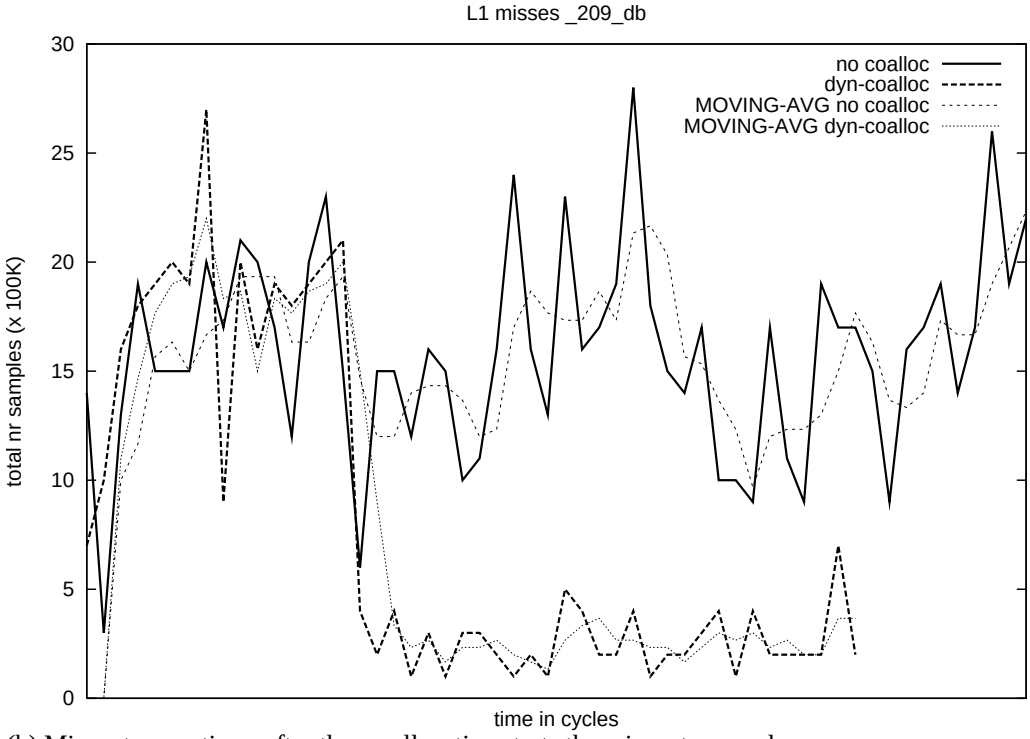
5.2.6 Runtime feedback

To actually guide optimization automatically a VM needs accurate feedback. Figure 5.13 depicts two types of data that we collect for programs, here shown for the db benchmark: Figure 5.13(a) shows the cumulative total count of L1 cache misses when dereferencing the field `String::value`. The sharp bend for "dyn-coalloc" occurs exactly when the co-allocation is switched on. The stepwise-constant shape of the measurement is caused by our batch-processing of samples in the monitoring module.

[5]The data points for at 20 and 24MB are missing for GenCopy because it needs at least 28MB to execute whereas GenMS only requires a heap size of 20MB.



(a) Total number of cache misses: The sharp bend for “dyn-coalloc” indicates the time when co-allocation kicks in



(b) Miss rate over time: after the co-allocation starts the miss rate goes down

Figure 5.13: Effect of co-allocation: Cache misses sampled for `String` objects db

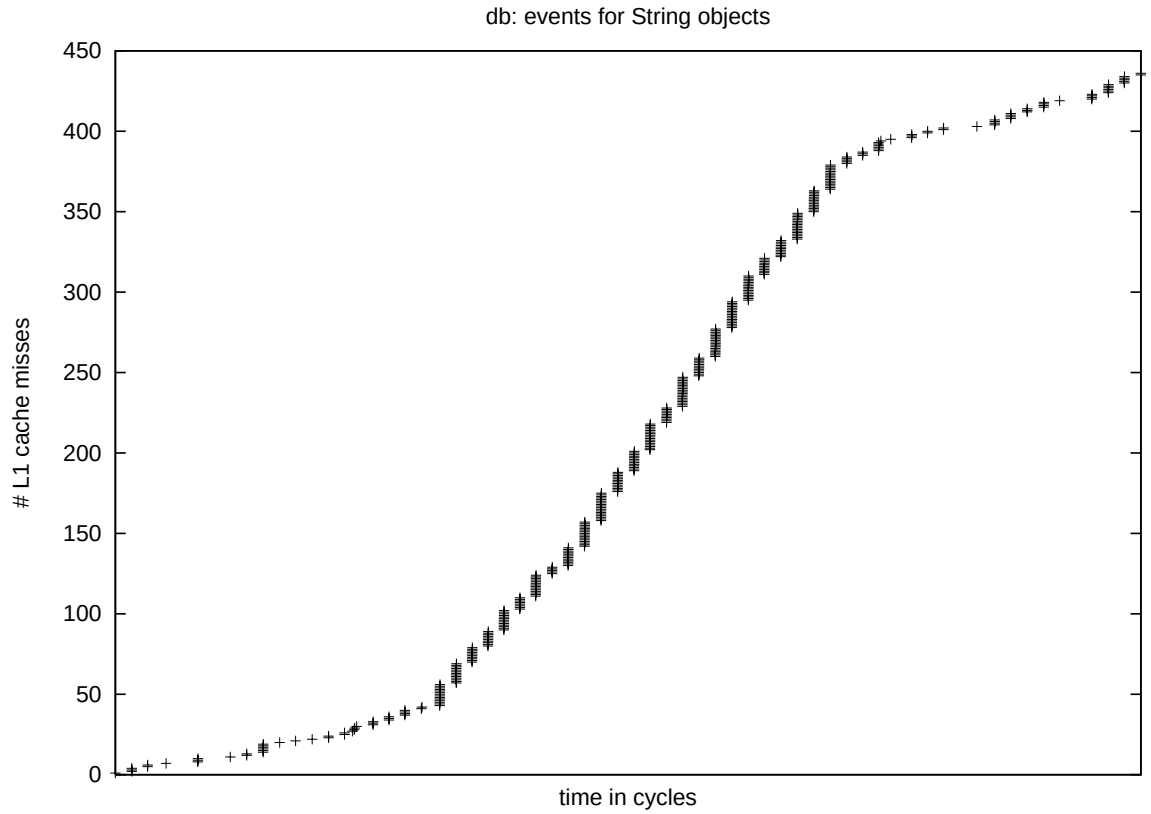


Figure 5.14: Cache misses sampled for `String` objects db with an poorly performing locality “optimization”

Figure 5.13(b) shows the L1 cache miss rate over time. It is locally quite volatile (in part also due to our monitoring infrastructure), but we can see the drop in the miss rate at the same time as in Figure 5.13(a) when co-allocation becomes active after the "warm-up" phase. The bold lines show the actual measured values. In addition we plot the moving average over the last 3 periods for both versions as thin lines. This metric follows the general trend without heavy local fluctuations. The precise association of the miss events with object types and references allows the VM to assess the effect of individual optimization decisions: in this case the internal `char[]` was co-allocated with the `String` object which resulted in a total reduction of misses on those objects by around 60%.

For long-running application the VM also needs to detect when an optimization has a negative effect on overall performance. To illustrate such a situation we show the cache misses over time when the GC happens to perform a poorly performing optimization in a controlled setting. For this experiment, we set up the system to perform the "opposite" of co-allocation: instead of co-allocated objects we explicitly allocate them in different cache lines.

Figure 5.14 shows the cache misses over time for `String` objects in db starting out with a good allocation order. We then instructed the GC manually to place up to 128 bytes (the size of a cache line) of empty space) between the `String` and the `char[]` objects - effectively undoing the originally well performing setting. Monitoring the cache miss rate for individual classes allows the system to discover that this transformation does not improve or degrades performance. After several measurement periods it triggers a switch back to the original configuration. Currently, a simple heuristic is used to determine when to switch, and we are still investigating suitable settings.

Also, mature objects that are already co-allocated remain in place - only newly promoted objects will follow the new copying policy. Figure 5.14 shows the effect on the miss rate after switching back to the original allocation policy; the miss rate returns to its old value. We did not see such a situation where undoing co-allocation was necessary during our experiments with co-allocation – this may be more important for other optimizations.

5.2.7 Summary

Object co-allocation is a technique to improve the data locality of objects on the heap. We show how to use together with HPM feedback to optimize access to objects in the mature object space of a generational garbage collector.

The HPM samples are mapped to individual Java bytecodes. From there we can deduce higher-level information such as the object type involved. We perform a data-flow analysis on the CFG to find pairs of heap accesses that qualify as candidates for co-allocation.

When collecting data from the HPM we annotate those candidates with cache miss information and use this information to decide to co-allocate the objects where many

cache misses occur If edge profile information is present in the IR, it is also used to correctly assign weights to different candidate instructions when necessary,

Pure copying GCs provide better good data layout, but they suffer from high GC overhead (especially for small to medium heap sizes). We use a hybrid (generational mark-and-sweep) collector extended with co-allocation. By combining co-allocation with a space-efficient generational GC we achieve the best of two worlds: low GC overhead and improved data locality.

We show that instruction-level HPM information can be used to guide online object co-allocation in a generational GC. With this combination we can combine a space-efficient GC with good data locality.

For a variety of benchmarks we see a speedup of up to 16% (average 3%). The number of co-allocated objects varies a lot between different programs. Only programs with a significant number of objects in the mature object space have a large number of candidates for co-allocation.

For several programs we see a significant reduction of cache misses, but they do not show a corresponding speedup in execution time. We do not have a conclusive answer for this phenomenon, but there several possible reasons it: First, the latency of some of the misses may be hidden by the out-of-order execution hardware. Another reason is that the benefit of co-allocation is compensated by the slightly increased GC activity. This effect gets larger as the heap size becomes smaller as we see on the measurements with different heap sizes.

Not all applications are good candidates for co-allocation since this optimization only improves data locality for long-living objects in the mature object space. If a program mostly allocates young objects, co-allocation will little chance of improving performance. On the other hand such an application also will not experience a noticeable slowdown because the overhead of online performance monitoring is really small with around 1% on average.

The advantage of online performance monitoring is that we detect cases where co-allocation will not be successful: In Chapter 4 we presented how to use HPM data to analyze the performance behavior of Java applications in detail with data address profiles. This allows us to predict if co-allocation will be successful in reducing cache misses by counting the cache misses in the mature object space. Co-allocation could be disabled for applications with few cache misses in the mature object space. A more general application would be to dynamically select suitable optimizations for individual applications in the JVM.

Another advantage of online monitoring is that it allows the VM to observe the effect of an optimization. In some cases we see that co-allocation reduces the number of cache misses occurring at some objects (e.g., `char[]` by up to 60%. However, if the application experiences a negative effect as a result, the VM can decide to switch off badly a performing optimization. This system is a step into an performance-aware runtime environment that can judge which optimizations actually bring benefits and which do not.

5.3 Loop unrolling using HPM feedback

In this section we look at a different kind of information from the hardware performance monitors. Object co-allocation presented in Section 5.1 uses data cache misses as a source of feedback. The second case study presented here shows how to use front-end stall cycle events (caused by instruction cache misses or branches) to improve loop unrolling heuristics.

5.3.1 Background

Loop unrolling [44, 73, 89] is a common program transformation found in almost all modern optimizing compilers. Loop unrolling is very effective since many programs spend a large portion of their execution time inside loops There are two main benefits from loop unrolling:

- The number of branch instructions in the unrolled loop is reduced. Also, the control variable is modified fewer times than in the original loop.
- The unrolled loop body exposes more opportunities for instruction level parallelism (ILP).

On modern architectures the second benefit is much more important than the first one because all modern general purpose architectures (like IA-32, IPF, PowerPC) have features designed to exploit instruction-level parallelism. They also include sophisticated branch prediction logic and perform prefetching to reduce the penalty due to branches.

Most compilers implement simple heuristics of when and how to perform loop unrolling. Even with simple rules, the performance improvement can be quite significant - often in the range of 10 to 30% [44].

Still, many compilers apply loop unrolling in a conservative way because if done too aggressively there also may be drawbacks due to loop unrolling. Two examples of potential problems if is applied too aggressively are:

- Increase in code size: If a loop body gets too large due to unrolling, this may adversely affect instruction cache performance [86].
- More registers needed by the unrolled code: The unrolled loop body may contain much larger basic blocks than the original code. To exploit potential ILP the compiler needs to assign more registers to the unrolled code which may adversely affect performance [44, 73].

Since we do not have a high-performance Java VM for the IPF architecture available in source code, we perform our experiments with loop unrolling using a high-performance static compiler (Open64 C/C++ compiler[1]).

The goal of this study is to evaluate the potential of using HPM data to guide loop unrolling decisions. For some programs we found that the default heuristics are too conservative. More aggressive loop unrolling resulted in a speedup of up to 39% compared to the fastest default configuration which uses the highest optimization level available (-O3).

5.3.2 Runtime platform

For this study we choose the Itanium 2 Montecito processor (IPF platform) because it offers more useful HPM events with information about the instruction cache and also allows accounting stall cycles in the execution pipeline.

Instead of a Java compiler we extended the Open64 C/C++ compiler [1]. The Open64 compiler can generate code for IA-32 and IPF. Open64 has a highly-optimizing code generator which is especially important for the IPF platform and includes all major global optimizations. In order to show that HPM feedback can be useful for improving existing loop unrolling heuristics we need a competitive baseline.

Loop unrolling in Open64

Loop unrolling is one of the many loop optimizations built into Open64. It is is controlled mainly by two parameters in Open64:

1. An absolute upper value for the unroll factor (*unroll_times_max*) limiting the number of times that the loop body is replicated.
2. The maximal size of the loop body after unrolling (*unroll_size_max*) limiting the final size of the unrolled loop.

In general, Open64 can only unroll single basic block loops. For loops bodies with multiple basic blocks the optimizer performs if-conversion and tries to convert them to a single block using predication.

For nested loops, Open64 only considers the inner-most loops as loop unrolling candidates. Loops in outer nesting levels are not changed.

The size of the loop body is measured in number of operations in the code generator's IR. If a loop is fully unrolled the unroll factor is equal to the trip count of the loop. Otherwise, it is always set to a power of two by default.

The final unroll factor for each loop is determined the function *Determine_Unroll_Factor*. This function distinguishes different cases:

- Loops with a constant trip count that may be fully unrolled: If a loop has a constant trip count and its size when fully unrolled is smaller than the upper limit - as specified by the *unroll_size_max* parameter - the compiler will fully unroll the loop.
- Loops that should not be fully unrolled: This case occurs when either of the two parameters *unroll_times_max* or *unroll_size_max* would be exceeded by fully unrolling.

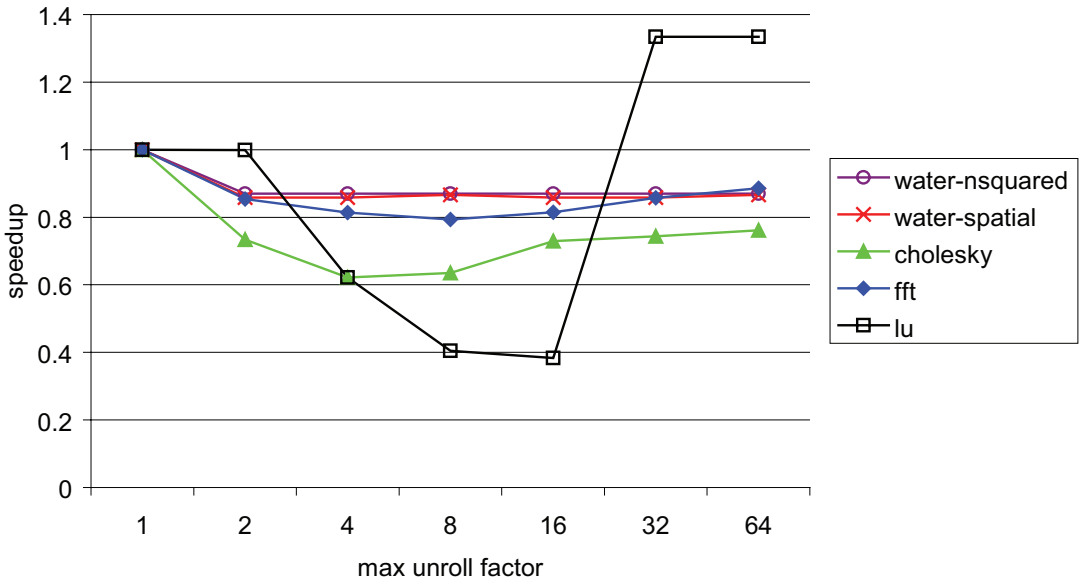


Figure 5.15: Execution time of selected SPLASH-2 programs relative to the "no unrolling"configuration.

In this case the unroll factor is set to the largest power of 2 so that the unrolled size and the unroll factor are still smaller than the upper limits (*unroll_size_max* and *unroll_times_max*). [6]

5.3.3 Approach

To use HPM feedback for improving loop unrolling heuristics we need to identify HPM events that correlate well with the overall performance of a loop. For optimizing the performance of a whole application we can just measure total execution time to determine the optimal parameters. One reason to use HPM events instead for predicting the performance is the possibility to determine specific per-loop heuristics. This is not possible using total execution time. A similar problem arises when translating our off-line approach into an online optimization: We need to obtain an accurate estimation of each individual loop's performance at runtime. Profiling execution time at runtime on a per-loop basis may be difficult or not possible at all with a high enough precision.

Performance impact of loop unrolling

Figure 5.15 shows the relative executions time of five selected SPLASH-2 programs. For conciseness, we only show the numbers for those benchmarks where loop unrolling has a maximum speedup of at least 10% in this plot. Since we are running statically compiled C programs, the standard deviation of the execution time is negligible for these programs.

There are three programs that clearly show a single optimal unrolling factor: fft, cholesky and lu. For each of these programs the optimal value is different (4 for cholesky, 8 for fft and 16 for lu) This observation shows that a single heuristic is not good enough to capture characteristics of different applications.

The remaining two programs (water-spatial, water-nsquared) exhibit a different behavior: Loop unrolling already provides its maximal benefit with a unrolling factor of two with around 14%. This speedup does not change significantly with increasing unrolling factors.

As a next step we look at HPM events to find out where the performance differences when doing loop unrolling with different parameters comes from. Our experiments show that stall cycles are a good predictor for the performance of an application. The IPF platform offers a large range of events that count many different types of stall cycles. We use the FE_BUBBLE_ALL event which counts all front-end stall cycles to estimate the performance of a loop. There are several sub-categories of this event to distinguish stalls by their source (e.g., stalls due to I-cache misses, branches, full instruction buffer, etc.) We use this event because many of these sub-types of front-end stalls are directly affected by loop unrolling.

Figure 5.16 shows the number HPM events relative to the configuration with no unrolling. Loop unrolling has an effect on the I-cache and on the number of branches executed: We see that stalls due to branches decrease constantly with an increasing unrolling factor. I-cache misses, on the other hand, exhibit a sharp increase after unrolling 32 or more times. The total number front-end stall cycles correlates very well to the total execution time.

Figure 5.17 shows the speedup of using the optimal loop unrolling as found by profiling front-end stalls over the whole application relative to Open64's default configuration. For many programs the default heuristic is not aggressive enough and tuning the maximum unroll factor allows a speedup of up to 38% over the baseline for *lu.* Several other programs show improvements between 2% and 5%. For the remaining application there is no improvement. The average improvement over all programs is 6%.

The results show that the default heuristic is not aggressive enough for several program. It is important to note that the optimal unroll factor differs between applications. If this would not be the case, just changing the compiler's default setting globally for all programs would achieve the same result. This is also an argument in favor of doing this optimization in an online setting since it is not convenient to perform off-line profiling for

[6]Loops without a trip count (e.g. while-loops) are unrolled up to the maximum size that is smaller than the upper size limit (*unroll_size*). The unroll factor is not rounded down to a power of 2.

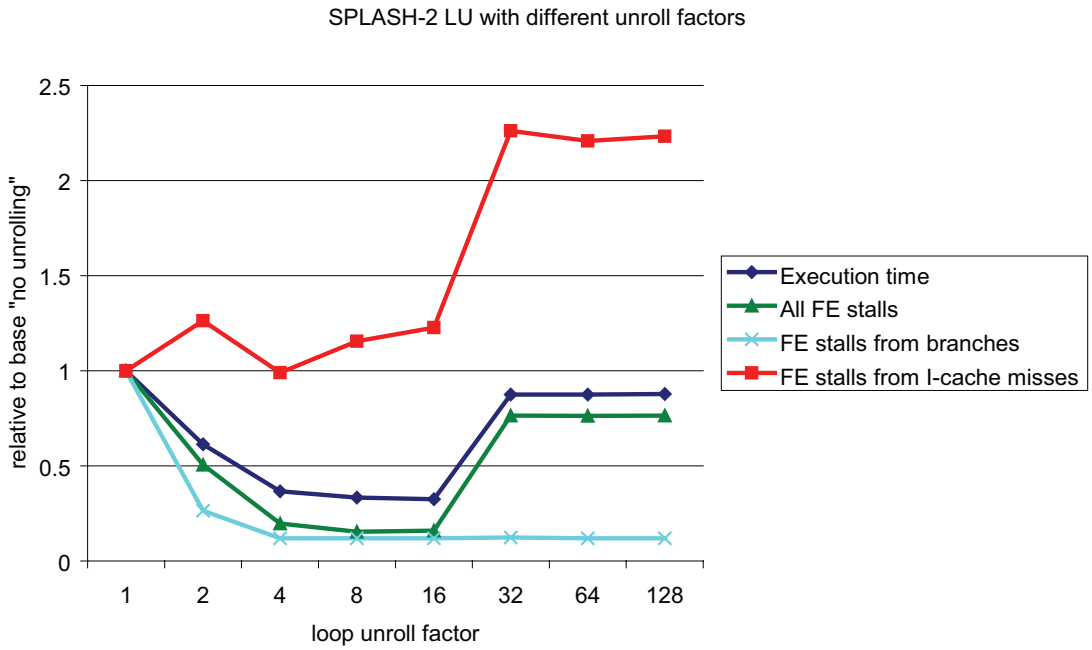


Figure 5.16: Performance behavior of *lu* with different maximum unrolling factors.

each application before compilation.

5.3.4 Computing per-loop unrolling hints

The execution time numbers show that different programs need different loop unrolling parameters for optimal performance. As a next step we evaluate the benefit of tuning the loop unrolling parameters for each loop individually. It may be the case that different loops within one applications also require different optimization parameters for best performance.

To determine the number of stalls for each loop we use a sampling approach. We configure the PMU to sample front-end stall events and attribute the collected samples to the loops in the source program.

The Open64 compiler can generate source line debug information that allows us to map samples to line number in the source program. Each loop covers of a range of source lines.

To find the line number for a sample we use the `addr2line` utility which takes the instruction address and a binary with line number information as input. The output of `addr2line` gives the source file name plus the source line number which we use to identify to which loop a sample belongs to.

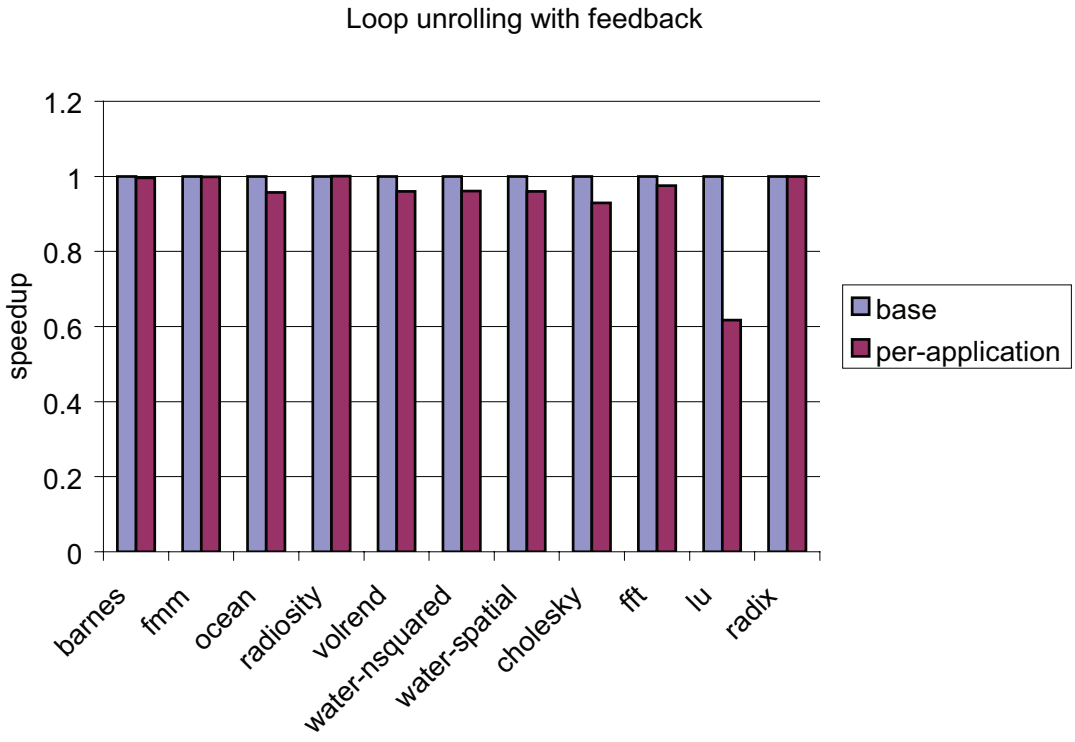


Figure 5.17: Execution time with optimal global unrolling factor.

Max unroll factor	Stalls in loop L_i					Total stalls
	L_1	...	...	...	L_n	
1	$s_{1,1}$	...	...	...	$s_{n,1}$	S_1
2	$s_{1,2}$	...	...	...	$s_{n,2}$	S_2
4	$s_{1,4}$	...	...	...	$s_{n,4}$	S_4
8	$s_{1,8}$	...	...	...	$s_{n,8}$	S_8
16	$s_{1,16}$	...	...	...	$s_{n,16}$	S_{16}

Table 5.3: Matrix with front-end stalls for each loop in the program at different maximum unroll factors.

In a first pass we run the compiled binary monitoring HPM events. After mapping the samples to line numbers and loops we feed the results into Open64 in the a second compilation pass that determines the final unrolling parameters for each loop.

When there are more than one loop with the same nesting level at the same source line we have to equally distribute the samples for that source line across these loops. This approximation may not be 100% accurate, but we found very few cases where such a situation actually occurs in real programs.

For nested loops we only consider the inner-most loops since loop unrolling in Open64 only operates on those loops.

For each program we measure front-end stalls within each loop with different maximum unroll factors. Previous work explored the benefits of loop unrolling and found that unrolling more than 15 times does not offers any significant benefit [44]. We use a maximum unrolling factors from 1 to 16 in our system. Like in the original Open64 configuration we only use powers of two as unrolling factors for loops that are not fully unrolled.

The results the performance monitoring runs are summarized in a matrix as shown in Table 5.3. Every row contains the number of events in every loop for a given maximum unroll factor.

For each loop i we calculate the optimal maximal unroll factor x_i by selecting a power of two between 1 and 16 with the lowest number of front-end stalls:

$$x_i = \min_{j=1,2,4,8,16}(s_{i,j})$$

By choosing a loop-specific heuristic for unrolling we hope to achieve better performance than by selecting one setting for the whole program.

The total number of stalls S_j for a given unrolling factor j is given by the sum across each individual row in the stall matrix:

$$S_j = \sum_{i=1}^{n} s_{i,j}$$

	Stall cycles (prediction)		Real speedup	
Program	app (S_j)	loop (S_{opt})	app	loop
barnes	1	1	1	0.99
cholesky	1	0.99	0.96	1
fft	0.98	0.98	0.98	0.98
fmm	0.98	0.90	1	1
lu	0.51	0.50	0.62	0.61
ocean	0.94	0.87	0.95	0.97
radiosity	1	1	1	1
radix	1	1	1	1.02
volrend	0.98	0.98	0.95	0.97
water-nsquared	0.95	0.95	0.96	0.96
water-spatial	0.94	0.94	0.96	0.96

Table 5.4: FE stall cycles compared with real speedup for per-application and per-loop heuristics.

The expected total number of stalls with optimal per-loop unrolling hints S_{opt} is always smaller or equal than the best single heuristic:

$$S_{opt} <= \min_{j=1,2,4,8,16}(S_j)$$

S_{opt} can be viewed as a prediction on the speedup when using a per-loop unrolling heuristic. The interesting question is by how much the number stalls can potentially be reduced with per-loop hints compared to the best global heuristic. In other words, how much does S_{opt} differ from the minimum value for S_j.

We measured these values for the SPLASH-2 benchmarks[7]. Table 5.4 compares the stall cycles relative to the baseline with the real speedup in execution time for the per-application (column "app") and the per-loop (column "loop") unrolling heuristics.

We can see that for most all programs the global per-application heuristics look as good as the per-loop heuristics. Four programs show of a difference of at least 1% between the per-application and the per-loop optimization: *cholesky*, *lu*, *fmm* and *ocean* with up to 8% less stalls inside loops for *fmm* with the per-loop heuristics. For these programs we can expect some performance improvement due to per-loop optimization.

For the other programs the difference between per-application and per-loop is less than 1%. When we looked at the code of these benchmarks more closely we found that they often have only a handful (in the order of 2-3) of hot loops where most computation is performed. This is one possible explanation why the global heuristics already perform so well. For few loops it is much more likely to find a good common optimal unrolling factor that can be applied globally.

[7] *raytrace* was excluded because it produced a segmentation fault in certain configurations of Open64

Program	Software pipelining	Loop unrolling	None	Total
barnes	15	2	20	37
cholesky	235	103	78	416
fft	9	14	5	28
fmm	60	2	46	108
lu	13	2	8	23
ocean	111	147	12	270
radiosity	24	5	48	77
radix	16	3	7	26
volrend	21	2	29	52
water-nsquared	12	26	18	56
water-spatial	16	19	15	50

Table 5.5: Number of loops optimized with loop unrolling and software pipelining with the compiler's default heuristic.

The last two columns show the real speedup with global per-application heuristic (see also Figure 5.17 and per-loop heuristic. We can see that per-loop heuristics match the performance of the global optimal parameters for most programs. In some cases per-loop is 1-2% faster (barnes, lu). In one case there is a 2% slowdown (radix). The overall speedup compared to the compilers default heuristic for the per-loop heuristic is 5% (6% for per-application).

For some programs the stalls measured for each individual loop would indicate a larger speedup (e.g. *fmm*, *ocean*) for the per-loop heuristics. It is not exactly clear why we do not see a speedup. One reason may be that with loop-specific tuning we only have a local view for each inner loop. We only measure stalls within the inner-most loops, but not all execution time is spent in those loops. There may still be unknown interactions with other optimizations or with other parts of the code. The per-application tuning has the advantage of a global view, but can only apply one heuristic for all loops in the program which may also not be optimal in all cases (e.g. *lu*).

5.3.5 Discussion

Interaction with other loop transformations

It is still not completely clear how different loop unrolling heuristics interact with other transformations like software pipelining.

Open64 decides on a per-loop basis whether to perform software pipelining, loop unrolling or both. In some cases software pipelining is less beneficial when already performing aggressive loop unrolling. There are around a dozen different parameter to control software pipelining in Open64, so may be difficult to find an optimal setting there. Therefore, we first looked at loop unrolling in isolation for this study.

Table 5.5 summarizes the total number of loops in each application and shows how many of the loops that are optimized with loop unrolling and software pipelining. The first column counts loops that are software pipelined (and may be unrolled before). The second column shows the number of loops that are only unrolled. The third column shows the remaining loops which are not optimized for various reasons (e.g., contain multiple basic block even after if-conversion)

The first observation from this table is that in some programs (e.g. barnes, fmm, radiosity) many loops are optimized with software pipelining. For the programs where software pipelining is used heavily it would be advantageous to also investigate the heuristics for software pipelining and tune them using HPM feedback: Including more optimizations into the HPM feedback-based tuning process would be an interesting problem for future work.

Secondly, the number of loops not considered in either optimization shown in column "None" is quite high in some programs (up to 62% for *radiosity*). There may be still potential for optimization when applying loop unrolling also to those loops. The unoptimized loops mainly consist of multi basic block loops that could not be converted into a single basic block by if-conversion. Previous research indicates that there is a significant benefit in optimizing multi basic block loops with unrolling [44] – at least on the hardware available 10 years ago. It would be interesting to investigate in how far these observations still hold on modern platforms like the Montecito.

Translation into an online optimization

The main reason for using Open64 with the IPF architecture in this study is that we needed a highly-optimized code generator to be able to show that HPM feedback can improve a single global heuristic. The IPF platform also offers detailed information about the instruction cache and stall cycles. With a competitive Java VM available for the IPF platform it would be interesting to see how such a system performs in an online optimization setting.

Feedback-guided loop unrolling could be translated into an online optimization suitable for a Java VM. Many JVMs already perform several compilation passes and collect profile information.

In such a system we would need a measure for the performance of each loop at runtime. The number of stall cycles alone is not enough to assess relative performance since we have to compare performance of different versions of a loop at runtime and not after a complete execution like in the off-line scenario.

A metric like instruction-per-cycle (IPC) could be used to compare performance of loops at runtime. If there is still optimization potential (i.e. IPC is less than optimal), the compiler would try a more aggressive heuristic and increase the maximum unroll factor. If the resulting code yields a better throughput (larger IPC value) we keep the newly optimized version and continue with performance monitoring. If performance degrades, we go back to the last version. Such a strategy can work well for programs like *lu* or *fft*

that show a unique optimal unrolling factor.

The target applications for this kind of online optimizations are long-running programs (business server, high-performance scientific computing, etc.) where several recompilation phases are amortized over a long time.

One limitation of current platforms is that they can only do event-based sampling for one event at a time. To calculate the IPC for a region in the code we would need to sample two events at the same time: instructions retired and CPU cycles spent. This limitation may be an obstacle when trying to implement feedback-guided loop unrolling on current hardware. Event multiplexing may be one option to solve that problem. With multiplexing, the system switches between two events periodically. However, it may be difficult to obtain accurate results using event multiplexing.

Even though optimizing unrolling parameters on a per-loop basis does not yield a higher speedup, it is still useful in an online setting: Here the runtime system does not have global a view of the whole application and has to base its optimization decisions on temporary local observations in any case.

5.3.6 Summary

Our experiments with a static compiler show that loop unrolling can be tuned with HPM feedback. Per-application tuning already provides the best results for almost all programs. We measured up to 38% reduction in execution time over the compiler default configuration (-O3) - with an average improvement of 6%.

On top of that optimizing loop unrolling parameter on a per-loop basis only provides up to 2% speedup for one application. There is also a case with a 2% slowdown. For the remaining programs the effect is null or less than 1%. Across all programs we get an average speedup of 5%. Since our benchmark programs contain relatively few hot loops a single global set of parameters per-application seems to be enough for most programs. The local view of the per-loop adaptation may be sub-optimal in some cases compared to the global application-wide tuning.

One open question is why the potential savings in number of stall cycles does not always translate into a speedup. There may be unintended interactions of loop unrolling with other optimizations that need to be explored in more detail.

Another limitation of the current system is that it does not consider unrolling of outer nested loops [73] as this is currently not supported in Open64.

Tuning loop unrolling would be also suitable in an online setting: With an automatic selection of unrolling heuristics there is no need for platform- or application-specific tuning. However, actually implementing such a system as an online optimization may prove difficult on current hardware because of limitations when doing event based sampling. It would be desirable to be able to sample more than one HPM event at a time.

6

Related Work

There are four areas of prior work that we discuss here:

- data gathering techniques using profiles or hardware performance monitors (HPMs),
- data locality optimizations,
- garbage collection and its impact on data locality, and
- online optimizations in general.

For each of these areas we focus mostly on work related to dynamic optimization of Java programs and hardware performance monitors.

6.1 Profiling and Performance monitoring

For data gathering techniques we mainly focus on approaches suitable for dynamic compilation and optimization. There exists a fair bit of prior work about profiling and profile-guided optimization in ahead-of-time compilers (see, e.g., [71, 36, 64]) that is however not central to the topic of this thesis.

Simple CPU time profiling for Java can be performed using the JVM Tool Interface (JVMTI) [80]. JVMTI is an interface where a profiler can subscribe to different events in the JVM (e.g., field access, method start, etc.). It can be used to generate profiles and to identify hot-spots by measuring time spent in each method.

ProfileMe [45] is an combined software/hardware approach to perform instruction-level profiling using hardware performance counters on an out-of-order processor. The problem with traditional event-based sampling - especially on modern out-of-order processors - is that the instruction that caused the sampled event is earlier by an unknown amount, than the instruction reported to the user. To solve this problem ProfileMe samples random instructions instead of events and records all events that were experienced executing the instruction selected for profiling. By obtaining enough instruction samples the frequency of events can be estimated. The approach requires special hardware features (present on the Alpha 21164 [48]) for storing the information about each sampled

instruction. AMD also introduced instruction-based sampling (IBS) [47, 18] in their latest quad-core processor generation (Barcelona). Their approach also randomly samples instructions instead of events and closely resembles the ProfileMe approach.

There are various tools for accessing hardware performance counters. PAPI [31] provides a high-level cross-platform API for measuring events that are common on most hardware platforms under Linux. On Linux, the low-level access to the hardware performance counters is provides by device drivers like perfctr [70], hardmeter [4] or perfmon [56]. which are part of the Linux kernel. Tools like PAPI are based on these device drivers. In our system we use perfmon since it has the best support for advanced features like PEBS on IA-32.

VTune [41] is a performance analysis tool for Intel platforms (IA-32, IPF). It also supports event-based sampling to locate performance bottlenecks and is available for Linux and Windows systems. After running a user program in VTune, it provides an annotated source code view of the user program as a visual help to quickly identify performance hot-spots.

Hardware performance counters are frequently used for off-line performance analysis and characterization of workloads [90, 40, 32, 34].

Buck et al. [33] perform off-line measurements of L2 data cache misses on an Itanium 2 processor for a set of C programs. In their case study they present a tools to relate the raw information from the HPM to the source code level data structures and they manually tune applications using this higher-level information. Our work follows a similar concept, but is fully automatic (i.e. no user interaction or manual tuning required). Also, since we perform the monitoring in an on-line fashion, we have to care a lot about measurement overhead whereas an off-line approach does not have these constraints.

Recently, hardware performance monitors have also been used to study the performance behavior of managed runtime environments as often seen for languages like Java or C#. Hauswirth et al. [53, 52] study the interaction between the Java VM and the lower levels of the execution platform (OS, libraries, hardware). They measure how these layers influence each other by introducing “software performance counters” which capture performance metrics of the software subsystems and correlate them to the information gathered by the hardware performance counters.

To correlate data from the hardware with Java methods Georges et al [51] instrument method entries and exits with reads of the hardware performance counters. Their approach reduces the number of instrumentations significantly by first identifying execution phases and then only instrumenting the start and the end of these phases. This way the high overhead of instrumenting every method can be avoided.

Simulation is an alternative approach to analyze the performance of applications. A full, accurate simulation usually is very expensive in terms of execution time. There exist several binary instrumentation tools [69, 65] that can be used for such analyses, but in general they are not applicable for online optimization due to the prohibitively large slowdown (often by 50-200x) [69].

Zhao et al. [91] use a lower-cost dynamic approach: They partially simulate the memory system using short memory access traces of a program at periodic intervals. Compared to other binary instrumentation approaches their overhead is competitively low with an average of 14% relative to normal program execution.

6.2 Data locality

Several studies show that data locality optimizations can improve the performance of programs with irregular memory access patterns. Field reordering [61, 72] is a technique that targets objects that do not fit into one cache line. It places fields with high temporal affinity together to improve cache utilization. Class splitting [37] achieves a similar effect by splitting data structures into two: a hot (frequently accessed) and a cold part. The hot parts are allocated together to avoid infrequently used data to use up cache memory. Usually these techniques rely on profiling information to approximate a good data layout because it is generally hard to statically optimize data locality in object-oriented programs. The reported speedup due to class splitting ranges from 6% to 18% for a set of five Java 1.0.2 benchmarks. Note that they used a different hardware platform for the evaluation (Sun UltraSPARC) so that the numbers cannot be directly compared with those measured on an IA-32 platform. Class splitting is an optimization that is orthogonal to object coallocation. It could be combined with coallocation so that the hot parts of each objects are coallocated together. It is an interesting open question to find out if we can get an even bigger performance benefit by combining different data locality optimizations than when using one technique alone.

6.3 GC

Recent research shows that garbage collection is not only a useful help for the programmer, but can also improve performance by moving objects in memory to achieve better data locality. There has been a lot of research in garbage collection in recent years, but here we only focus on work about how GC impacts data locality:

Blackburn et al. [29] show that new hybrid GC approaches can combines the advantages of a copying collector with better space efficiency. Their GC algorithm combines copying and marking into a single pass to achieve space efficiency, fast reclamation, and good mutator performance. They compare performace against a set of canonical algorithms where the speedup ranges from 7% to 25%. Versus the default production GC GenMS (the same we are using in our evaluation) the improvements are much smaller with an average of 1% across the whole set of benchmarks. In our work we achieve a similar goal by combining object co-allocation with a space-efficient GC. Our work does not propose a completely new GC algorithm. Instead it can be viewed as an online optimization applied at GC time on top of an existing GC.

Hirzel [57] evaluates a large variety of different data layouts which are generated at GC time. It turns out that almost every layout is optimal for some programs, but has bad

performance for others. This work confirms our observation that it is difficult to statically determine a good layout for all possible applications and that feedback information helps in finding good data layouts.

6.4 Online optimizations

Several high performance JVMs use adaptive optimization based on run-time profiling [21, 39, 79]. The Jikes RVM [24] uses timer-based sampling of the call stack to find frequently executed methods. The frequency profiles are used to determine where to spend the most effort for optimization: The more often a method is invoked, the more expensive optimizations are applied to it. A static cost/benefit model for the different optimizations is used to evaluate whether a method should be recompiled. It also has the ability to use continuous profiling feedback to improve performance of long-running applications [25].

Instead of normal profile information, hardware performance monitors can also be used to guide recompilation decisions in the JIT compiler [35]. They use the CPUs cycle counter event to determine which methods are executed frequently. The HPM-based approach offers several advantages over software-based sampling techniques for identifying hot methods for recompilation: Most importantly, better accuracy at a lower cost compared with polling or based call-stack sampling based on the timer interrupt.

Adl-Tabatabai et al. [17] present a dynamic optimization to eliminate long-latency cache misses. They insert prefetch instructions after dynamically monitoring cache misses using hardware performance monitors of the Itanium processor. The approach exploits the fact that objects that are linked through a reference often have a constant delta between their starting addresses. Software prefetching must be used consciously because fetching the wrong data into the cache may have a negative performance impact. By using hardware performance monitors to guide the prefetching they achieve a speedup of 14% for the SPEC JBB2000 benchmark. For the other programs from the JVM98 benchmark suite there seems little benefit from prefetching where co-allocation gets up to 16% speedup for db. In our system we do not target prefetching, but instead we reorder object instances to reduce the number of cache misses. The two optimizations - inserting prefetches and co-allocating objects - seem to be orthogonal and complement each since they work best for different types of programs.

Online object reordering [58] is a different dynamic locality optimization for Java. It reorders objects at garbage collection time using a copying GC. The heuristic for reordering is determined by profiling field access operations with a light-weight software-only profiling mechanism. The overhead of profiling is around 2-3% in this system. Objects with “hot” fields (frequently accessed) are placed adjacent to their referent objects to increase spatial locality by visiting those references first during the copying process in the GC. This approach requires a copying garbage collector (which is present in many modern VMs) and can match the performance of the best static copying strategy. However, for most applications there is no signficant speedup compared to the most efficient baseline.

Our work takes a related approach, but we do not rely on execution frequencies as a metric for locality. Instead we use direct feedback from the memory hierarchy about cache misses to guide compiler and GC decisions. Also we combine our object co-allocation with a generational mark-and-sweep garbage collector to show that object co-allocation can achieve space-efficient garbage collection with good data locality.

Similar ideas have been used to improve code locality. Dynamic code management [59] is a code reordering algorithm to improve code locality and reduced instruction TLB stalls. The system builds up a call graph at runtime and uses a light-weight reordering heuristic to determine the optimized code layout which results in a speedup of up to 6%.

Shuf et al [75] also use the memory management system to improve data locality and present an object allocation scheme that attempts to place frequently instantiated types that are connected via references close together in memory. They also show that a locality-based heap traversal algorithm can improve GC performance.

Lau et al [62] show how to use direct measures of performance (cycle counts) to guide inlining decisions in a dynamic compiler. The JIT generates two version of each method: one with aggressive inlining and one with the default (more conservative) heuristic. By executing each of the two versions randomly during the measurement phase the compiler collects timing information about each version. After filtering out outliers it can use those timings to decide which version of the method to use in future. Our approach also uses real machine metrics as feedback, but gathers more fine-grained information about the program's interaction with the execution platform (like cache or TLB misses).

Ispike [66] is a post-link optimizer. It dynamically optimizes binary code for the Itanium platform at load time and performs optimizations like instruction and data prefetching. It uses the IPF hardware performance counters to obtain statistical profiles with a lower overhead than by using code instrumentation. Our approach operates on the source language (bytecode) level and can use high-level information about the program for optimization which can be difficult or impossible to get at the binary level.

Chilimbi et al. [38] address memory performance pointer-intensive of C and C++ programs. They implement dynamic profiling using code instrumentation. The system thens extract hot data stream from the load address profile information and adds prefetch instructions to the code. Profiling is turned off periodically to achieve a low overhead for data collection.

Object inlining [46] is another optimization that improves data locality and potentially eliminates memory loads. Wimmer et al. [87, 88] have implemented dynamic object inlining for the Hotspot virtual machine [42]. They profile field access frequencies during interpreted execution and use this profile information to guide object inlining. Additional runtime checks are necessary to de-optimize compiled code in case an inlined objects "escapes" (i.e. its reference is used outside the scope of its enclosing object). It seems that object inlining works also well for the same programs where co-allocation is successful (e.g. *db* with up to 50% improvement). However, the absolute speedup numbers are not directly comparable since they use a different runtime platform.

Feedback from the HPM can also be used in the OS: Tam et al. [82] use information about cross-chip communication bandwidth in a multi-core system to optimize the scheduling decisions of the OS. The general idea is that tasks that share data should be scheduled on the same core (or on cores that share a common cache) whereas independent processes should execute on different processors to minimize overall memory bus traffic.

7 Conclusions

With newer generation of CPUs having more features for performance monitoring built into them we think that systems like presented in this thesis will become more important in the future. This thesis makes contributions in three areas:

- Infrastructure and techniques for online performance monitoring to collect detailed HPM information
- Performance analysis using the collected HPM information
- Optimizations using feedback from the hardware.

The following sections sum up each of the main topics and also point out possible extensions for future systems.

7.1 Online performance monitoring

We presented a system that allows efficient and precise online performance monitoring unit using hardware performance monitors.

The overhead imposed by the performance monitoring is reasonably low ($<$1% avg) so that it does not significantly disturb running applications and can be enabled throughout the whole program execution. Using an adaptive sampling period we achieve a stable overhead across a large variety of different programs.

With appropriate compiler assistance it is possible to map performance-related events to source-level constructs. Using additional meta-information in the compiler we are able to determine the Java bytecode instruction and the data address for a given performance event.

Since the HPM information is inexpensive to collect it can be used by a JVM in addition to the existing profile information almost without any additional cost.

However, there are still limitations on what data can be obtained on many current hardware platforms. Since we think that a dynamic runtime environment like a JVM can benefit from HPM feedback in many places, we would like to see more instruction-level data available on future hardware architectures. Some newer architectures (like the IPF)

already provide more information in their PMU. Online optimizations using hardware feedback will become more pervasive if this trend continues.

7.2 Performance analysis of Java applications

Fine-grained HPM information can be used for detailed performance analysis. We showed how information about cache misses can be used to identify performance-critical load instructions. The distribution of cache misses varies widely between optimizations. Our experience with data locality optimizations indicate that programs with few hot-spots are in general easier to optimize.

We implemented an infrastructure to collect data address profiles on the IA-32 platform using the precise event-based sampling (PEBS) mechansim. Data address profiles help to understand why optimizations are beneficial or unsuccessful for a given program.

We also show how to use data address profiles to describe and accurately explain the performance behavior of different garbage collector algorithms. Our results confirm observations made in previous work. The use of data address profiles allows us to precisely quantify the effect of different GC algorithms on data locality.

7.3 HPM feedback-guided optimizations

Given current and future trends in software and hardware architecture feedback-guided optimizations using HPM data will become more important in modern compilers and runtime systems.

With processors getting more and more complex it becomes more difficult for a compiler or runtime system to accurately predict the outcome of an optimization. Compilers use many heuristics and usually have a large number of parameters that need to be tuned for a specific platform. On VLIW platforms like the IPF, the compiler is much more important for efficient program execution. These platforms can benefit even more from hardware feedback.

Managed runtime environments as used for Java or C# programs are becoming more wide-spread. Such an environment is a good target for online optimizations: With online performance monitoring it has the possibility to adapt optimizations not only for the target hardware platform, but also specifically for each application or even for different inputs of a program. HPM data allow the compiler to replace static heuristics with dynamic feedback. In this thesis we show two optimizations that benefit from hardware feedback: Object co-allocation and loop unrolling.

7.3.1 Object co-allocation

We implemented a system that uses the results of hardware performance monitoring for online data locality optimization. Object co-allocation is an optimization that shows how

a garbage collector with knowledge about frequently missed objects and references can improve data locality and can detect at run-time if a data-locality optimization has a positive or a negative impact on performance.

We implemented co-allocation in a Java VM to demonstrate that it is feasible to perform fully automatic online optimization with current hardware. With the co-allocation technique for matured objects that is discussed here, L1 cache misses are reduced by up to 28%. The resulting application speedup is up to 16% (average 3%), A more refined model of the micro-architecture in the compiler may be able to better exploit the performance data.

Object co-allocation is only effective for some programs. Other optimizations like prefetching work well for other programs. One interesting open question is whether HPM data can be used to determine which optimizations should be applied to an application. In our analysis we find that only programs with a large portion of cache misses in the mature object space benefit from object co-allocation. By counting cache misses in the different regions of the Java heap the system could predict whether co-allocation will be successful.

The infrastructure is flexible to allow compiler and GC implementers to include such information into their system as an additional source of runtime feedback. In our system the VM can actually "observe" the effect of data locality optimization. This is especially important since modeling the behavior of complex modern hardware architecture is very hard, and it is often a challenge to predict the effect of an optimization prior to performing the transformation. Feedback from the lower layers of the execution platform can be valuable information to guide such optimizations.

For a future production runtime system it would be a good strategy to combine different cache optimizations (like co-allocation, class splitting, prefetching) to get better overall results across different applications than when using one optimization in isolation.

7.3.2 Loop unrolling

We show that HPM feedback can be used to improve heuristics for loop unrolling. Existing compilers often do not apply loop unrolling aggressively enough. In an offline setting we can improve overall performance of C programs by up to 38% (average 6%) over the fasted default compiler configuration (-O3) by using an application-specific set of parameters for loop unrolling.

By using the debug information generated by the compiler we are able to assess the performance of individual loops with different unrolling parameters. When applying loop-specific unrolling parameters we usually match the performance of the best single per-application heuristic. The average speedup across our benchmarks is 5% here (max. 39%). In some cases we see a speedup of 1-2% over the globally tuned per-application heuristic. However, there are also programs where the local per-loop adaptation is slightly slower (2% in the worst case). Here, future research should not only focus on loop unrolling alone, but also investigate in detail how different loop transformations interact to

obtain a better model for improving optimization heuristics with feedback.

Using HPM data makes it possible to translate the offline approach into an online optimization. With current hardware it may be difficult to implement such a system since it would require to be able to sample more than one HPM event at the same time. Our offline approach shows the potential improvement. It is still an open question if adaptive online loop unrolling can approach this performance.

Bibliography

[1] Open64, The Open Research Compiler, version 4.2. http://www.open64.net.

[2] The GNU Compiler for the Java Programming Language. http://gcc.gnu.org/java.

[3] XED2 X86 Encoder Decoder. http://rogue.colorado.edu/pin/.

[4] Hardmeter - a memory profiling tool. Available at http://sourceforge.jp/projects/hardmeter, 2003.

[5] Intel Itanium2 Processor Reference Manual For Software Development and Optimization. 2004.

[6] IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide. 2005.

[7] Dual-Core Update to the Intel Itanium2 Processor Reference Manual For Software Development and Optimization. 2006.

[8] Intel Itanium Architecture Software Developer's Manual, Volume 1: Application Architecture. January 2006.

[9] Intel Itanium Architecture Software Developer's Manual, Volume 2: System Architecture. January 2006.

[10] Intel Itanium Architecture Software Developer's Manual, Volume 3: Instruction Set Reference. January 2006.

[11] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture. September 2008.

[12] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M. September 2008.

[13] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z. September 2008.

[14] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide. September 2008.

[15] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide. September 2008.

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

[17] Ali-Reza Adl-Tabatabai, Richard L. Hudson, Mauricio J. Serrano, and Sreenivas Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *Proc. of the ACM Conf. on Programming Language Design and Implementation (PLDI 2004)*, pages 267–276, New York, NY, USA, 2004. ACM Press.

[18] Advanced Micro Devices, Inc. *Software Optimization Guide for AMD Family 10h Processors (Quad-Core)*, 2008.

[19] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[20] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine Project: Building an Open-source Research Community. *IBM Syst. J.*, 44(2):399–417, 2005.

[21] Bowen Alpern, C. Richard Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark F. Mergen, Janice C. Shepherd, and Stephen Smith. Implementing Jalapeno in Java. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPLSA 1999)*, pages 314–324, 1999.

[22] Bowen Alpern, Dick Attanasio, John Barton, Michael Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Ton Ngo, Mark Mergen, Vivek Sarkar, Mauricio Serrano, Janice Shepherd, Stephen Smith, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeno virtual machine. *IBM Systems Journal, Java Performance Issue*, 39(1), 2000.

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

[24] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeno JVM. In *Proc. of the Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2000)*, pages 47–65, New York, 2000. ACM Press.

[25] Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online feedback-directed optimization of Java. In *Proc. of the Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002)*, pages 111–129, New York, USA, 2002. ACM Press.

[26] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. of the Conf. on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2006)*, New York, October 2006. ACM Press.

[27] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: the performance impact of garbage collection. In *SIGMETRICS 2004/PERFORMANCE 2004: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 25–36, New York, NY, USA, 2004. ACM Press.

[28] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? high performance garbage collection in Java with mmtk. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 137–146. IEEE Computer Society, 2004.

[29] Stephen M. Blackburn and Kathryn S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 22–32, New York, NY, USA, 2008. ACM.

[30] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. Wake up and smell the coffee: evaluation methodology for the 21st century. *Commun. ACM*, 51(8):83–89, 2008.

[31] S. Browne, J. Dongarra, N. Garner, G. Ho, and P Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[32] Bryan R. Buck and Jeffrey K. Hollingsworth. Using Hardware Performance Monitors to Isolate Memory Bottlenecks. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 40, Washington, DC, USA, 2000. IEEE Computer Society.

[33] Bryan R. Buck and Jeffrey K. Hollingsworth. Data centric cache measurement on the Intel ltanium 2 processor. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 58, Washington, DC, USA, 2004. IEEE Computer Society.

[34] Dries Buytaert, Andy Georges, Lieven Eeckhout, and Koen De Bosschere. Bottleneck analysis in java applications using hardware performance monitors. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 172–173, New York, NY, USA, 2004. ACM.

[35] Dries Buytaert, Andy Georges, Michael Hind, Matthew Arnold, Lieven Eeckhout, and Koen De Bosschere. Using hpm-sampling to drive dynamic compilation. *SIGPLAN Not.*, 42(10):553–568, 2007.

[36] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21(12):1301–1321, Dec 1991.

[37] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *Proc. of the ACM SIGPLAN'99 Conf. on Programming Language Design and Implementation (PLDI 1999)*, pages 13–24, New York, NY, USA, 1999. ACM Press.

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

[39] Michal Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing judo: Java under dynamic optimizations. In *Proc. of the ACM Conf. on Programming Language Design and Implementation (PLDI 2000)*, pages 13–26, New York, NY, USA, 2000. ACM Press.

[40] D. W. Clark, P. J. Bannon, and J. B. Keller. Measuring VAX 8800 performance with a histogram hardware monitor. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 176–185, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[41] Intel Corp. VTune: Visual Tuning Environment.

[42] Sun Microsystems Corp. The Java Hotspot™ Virtual Machine. http://java.sun.com/products/hotspot.

[43] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[44] Jack W. Davidson and Sanjay Jinturkar. Aggressive loop unrolling in a retargetable optimizing compiler. In *CC '96: Proceedings of the 6th International Conference on Compiler Construction*, pages 59–73, London, UK, 1996. Springer-Verlag.

[45] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. Profileme: Hardware support for instruction-level profiling on out-of-order processors. In *Proc. of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 292–302, Washington, DC, USA, 1997. IEEE Computer Society.

[46] Julian Dolby. Automatic inline allocation of objects. In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI'97)*, volume 32(5) of *ACM SIGPLAN Notices*, pages 7–17. ACM Press, June 1997.

[47] Paul J. Drongowski. Instruction-based sampling (ibs): A new performance analysis technique for amd family 10h processors. http://developer.amd.com/assets/AMD_IBS_paper_EN.pdf.

[48] John H. Edmondson, Paul I. Rubinfeld, Peter J. Bannon, Bradley J. Benschneider, Debra Bernstein, Ruben W. Castelino, Elizabeth M. Cooper, Daniel E. Dever, Dale R. Donchin, Timothy C. Fischer, Anil K. Jain, Shekhar Mehta,

Jeanne E. Meyer, Ronald P. Preston, Vidya Rajagopalan, Chandrasekhara Somanathan, Scott A. Taylor, and Gilbert M. Wolrich. Internal organization of the alpha 21164, a 300-mhz 64-bit quad-issue cmos risc microprocessor. *Digital Tech. J.*, 7(1):119–135, 1995.

[49] Stéphane Eranian. What can performance counters do for memory subsystem analysis? In *MSPC '08: Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness*, pages 26–30, New York, NY, USA, 2008. ACM.

[50] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically Rigorous Java Performance Evaluation. *SIGPLAN Not.*, 42(10):57–76, 2007.

[51] Andy Georges, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere. Method-level phase behavior in Java workloads. In *Proc. of the ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, pages 270–287, New York, NY, USA, 2004. ACM Press.

[52] Matthias Hauswirth, Amer Diwan, Peter F. Sweeney, and Michael C. Mozer. Automating vertical profiling. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 281–296, New York, NY, USA, 2005. ACM.

[53] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. Vertical profiling: understanding the behavior of object-priented applications. In *Proc. of Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, pages 251–269, New York, NY, USA, 2004. ACM Press.

[54] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 143–153, New York, NY, USA, 2005. ACM.

[55] Hewlett-Packard Development Company, L.P. Libpfm. Available at http://www.hpl.hp.com/research/linux/perfmon/.

[56] Hewlett-Packard Development Company, L.P. Perfmon. Available at http://www.hpl.hp.com/research/linux/perfmon/.

[57] Martin Hirzel. Data layouts for object-oriented programs. In *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 265–276, New York, NY, USA, 2007. ACM.

[58] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: improving program locality. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, pages 69–80, New York, NY, USA, 2004. ACM Press.

[59] Xianglong Huang, Brian T Lewis, and Kathryn S McKinley. Dynamic code management: Improving whole program code locality in managed runtimes. In *VEE '06:*

Proc. of the second international Conf. on Virtual Execution Environments, pages 133–143, New York, USA, 2006. ACM Press.

[60] WhiteTimberwolf (SVG version) Imperator3733 (original). Timeline of intel processor codenames including released, future and canceled processors. WikiCommons, http://en.wikipedia.org/wiki/Image:IntelProcessorRoadmap.svg, 2008.

[61] Thomas Kistler and Michael Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Trans. Program. Lang. Syst.*, 22(3):490–505, 2000.

[62] Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. Online performance auditing: Using hot optimizations without getting burned. In *Proc. Conf. on Programming Language Design and Implementation (PLDI 2006)*, pages 239–251, New York, USA, 2006. ACM Press.

[63] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. A compiler framework for speculative analysis and optimizations. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 289–299, New York, NY, USA, 2003. ACM.

[64] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. Register promotion by sparse partial redundancy elimination of loads and stores. *ACM SIGPLAN Notices*, 33(5):26–37, 1998.

[65] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[66] Chi-Keung Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney. Ispike: A Post-link Optimizer for the Intel Itanium Architecture. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 15, Washington, DC, USA, 2004. IEEE Computer Society.

[67] Markus Mock, Ricardo Villamarin, and Jose Baiocchi. An empirical study of data speculation use on the intel itanium 2 processor. In *INTERACT '05: Proceedings of the 9th Annual Workshop on Interaction between Compilers and Computer Architectures*, pages 22–33, Washington, DC, USA, 2005. IEEE Computer Society.

[68] Brian R. Murphy, Vijay Menon, Florian T. Schneider, Tatiana Shpeisman, and Ali-Reza Adl-Tabatabai. Fault-safe code motion for type-safe languages. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 144–154, New York, NY, USA, 2008. ACM.

[69] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM.

[70] Mikael Petterson. The perfctr linux performance monitoring counters driver. Available at http://sourceforge.net/projects/perfctr/.

[71] K. Pettis and R. Hansen. Profile guided code positioning. In *Proc. ACM SIGPLAN'90 Conf. on Prog. Language Design and Implementation*, pages 16–27, White Plains, N.Y., June 1990. ACM.

[72] Shai Rubin, Rastislav Bodik, and Trishul Chilimbi. An efficient Profile-Analysis framework for data-layout optimizations. In *Proc. of the Symp. on Principles Of Programming Languages (POPL 2002)*, pages 140–153, New York, NY, USA, 2002. ACM Press.

[73] Vivek Sarkar. Optimized unrolling of nested loops. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 153–166, New York, NY, USA, 2000. ACM.

[74] Florian Schneider and Thomas Gross. Using platform-specific performance counters for dynamic compilation. In *Proc. of the International Workshop on Compilers for Parallel Computing (LCPC 2005)*, October 2005.

[75] Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel, and Jaswinder Pal Singh. Creating and preserving locality of Java applications at allocation and garbage collection times. In *Proc. of the Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002)*, pages 13–25, New York, 2002. ACM Press.

[76] David Siegwart and Martin Hirzel. Improving locality with parallel hierarchical copying gc. In *Proceedings of the 2006 International Symposium on Memory Management (ISMM 2006)*, pages 52–63, New York, USA, 2006. ACM Press.

[77] Brinkley Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):64–71, 2002.

[78] Brinkley Sprunt. Pentium 4 performance monitoring features. In *IEEE Micro*, pages 72–82, July–August 2002.

[79] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *Proc. of the ACM Conf. on Object Oriented Programming, Systems, Languages, and Applications (OOPLSA 2001)*, pages 180–195, New York, NY, USA, 2001. ACM Press.

[80] Sun Microsystems, Inc. JVM Tool Interface (JVM TI). http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/index.html.

[81] Peter F. Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. Using hardware performance monitors to understand the behavior of java applications. In *VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[82] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 47–58, New York, NY, USA, 2007. ACM.

[83] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98, 1996.

[84] The Standard Performance Evaluation Corporation. SPEC JBB2000 Benchmark. http://www.spec.org/jbb2000/, 2000.

[85] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proc. of the Software Engineering Symposium on Practical Software Development Environments (SDE 1)*, pages 157–167, New York, USA, 1984. ACM Press.

[86] Shlomo Weiss and James E. Smith. A study of scalar compilation techniques for pipelined supercomputers. *ACM Trans. Math. Softw.*, 16(3):223–245, 1990.

[87] Christian Wimmer and Hanspeter Mössenböck. Automatic feedback-directed object inlining in the Java Hotspot virtual machine. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 12–21, New York, NY, USA, 2007. ACM.

[88] Christian Wimmer and Hanspeter Mössenböck. Automatic array inlining in Java virtual machines. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 14–23, New York, NY, USA, 2008. ACM.

[89] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 274–286, Washington, DC, USA, 1996. IEEE Computer Society.

[90] Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz. Performance analysis using the MIPS R10000 performance counters. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 16, Washington, DC, USA, 1996. IEEE Computer Society.

[91] Qin Zhao, Rodric Rabbah, Saman Amarasinghe, Larry Rudolph, and Weng-Fai Wong. Ubiquitous Memory Introspection. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 299–311, Washington, DC, USA, 2007. IEEE Computer Society.

List of Figures

List of Tables

Curriculum Vitae

Florian T. Schneider

July 31, 1979	Born in Wels, Austria
1989 – 1997	Gynasium Kollegium Petrinum, Linz, Austria
1997 – 2002	Diploma studies in Computer Science, ETH Zurich, Switzerland
2002 – 2009	Research and teaching assistant
	Laboratory for Software Technology, ETH Zurich
2006	Internship at Intel Corp., Santa Clara, CA, USA
2007	Internship at Intel Corp., Santa Clara, CA, USA
since 2009	Post-doctoral researcher and lecturer at ETH Zurich