Using HPM-Sampling to Drive Dynamic Compilation

Dries Buytaert[†] Andy Georges[†] Michael Hind[*] Matthew Arnold[*] Lieven Eeckhout[†]
Koen De Bosschere[†]

[†] Department of Electronics and Information Systems, Ghent University, Belgium
[*]IBM T.J. Watson Research Center, New York, NY, USA
{dbuytaer,ageorges}@elis.ugent.be, {hindm,marnold}@us.ibm.com, {leeckhou, kdb}@elis.ugent.be

Abstract

All high-performance production JVMs employ an adaptive strategy for program execution. Methods are first executed unoptimized and then an online profiling mechanism is used to find a subset of methods that should be optimized during the same execution. This paper empirically evaluates the design space of several profilers for initiating dynamic compilation and shows that existing online profiling schemes suffer from several limitations. They provide an insufficient number of samples, are untimely, and have limited accuracy at determining the frequently executed methods. We describe and comprehensively evaluate HPM-sampling, a simple but effective profiling scheme for finding optimization candidates using hardware performance monitors (HPMs) that addresses the aforementioned limitations. We show that HPM-sampling is more accurate; has low overhead; and improves performance by 5.7% on average and up to 18.3% when compared to the default system in Jikes RVM, without changing the compiler.

Categories and Subject Descriptors D.3.4 [*Programming languages*]: Processors—Compilers; Optimization; Runtime environments

General Terms Measurement, Performance

Keywords Hardware Performance Monitors, Java, Just-in-time compilation, Profiling

1. Introduction

Many of today's commercial applications are written in dynamic, type-safe, object-oriented languages, such as Java, because of the increased productivity and robustness these languages provide. The dynamic semantics of such a language require a dynamic execution environment called a virtual machine (VM). To achieve high performance, production Java virtual machines contain at least two modes of execution: 1) *unoptimized* execution, using interpretation [21, 28, 18] or a simple dynamic compiler [16, 6, 10, 8] that produces code quickly, and 2) *optimized* execution using an optimizing dynamic compiler. Methods are first executed using the unoptimized execution strategy. An online profiling mechanism is used to find a subset of methods to optimize during the same execution. Many systems enhance this scheme to provide multiple levels of optimized execution [6, 18, 28], with increasing compilation cost and benefits at each level. A crucial component to this strategy is the ability to find the important methods for optimization in a low-overhead and accurate manner.

Two approaches that are commonly used to find optimization candidates are method invocation *counters* [10, 18, 21, 28] and timer-based *sampling* [6, 8, 18, 28, 30]. The counters approach counts the number of method invocations and, optionally, loop iterations. Timer-based sampling records the currently executing method at regular intervals using an operating system timer.

Although invocations counters can be used for profiling unoptimized code, their overhead makes them a poor choice for use in optimized code. As a result, VMs that use multiple levels of optimization rely exclusively on sampling for identifying optimized methods that need to be promoted to higher levels. Having an accurate sampler is critical to ensure that methods do not get stuck at their first level of optimization, or in unoptimized code if a sample-only approach is employed [6, 8].

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

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

tive to the program execution) continues to decrease as hardware performance improves.

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

This paper makes the following contributions:

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

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

The rest of this paper is organized as follows. Section 2 provides further background information for this work. Section 3 details the HPM-sampling approach we propose. After detailing our experimental setup in Section 4, Section 5 presents a detailed evaluation of the HPM-sampling approach compared to existing techniques; the evaluation includes overall performance, overhead, accuracy, and robustness. Section 6 compares our contribution to related work and Section 7 concludes and discusses future directions.

2. Background

This section describes background for this work. Specifically, it describes the design space of sampling techniques for finding optimization candidates and discusses the shortcomings of these techniques; gives relevant details of Jikes RVM; and summarizes hardware performance monitor facilities, focusing on the particular features we employ in this work.

2.1 Sampling Design Space

Two important factors in implementing any method sampling approach are 1) the trigger mechanism and 2) the sampling mechanism. Figure 1 summarizes this 2-dimensional design space for sampling-based profilers. The horizontal axis shows the trigger mechanism choices and the vertical axis shows the sampling mechanism choices. The bullets in Figure 1 represent viable design points in the design space of sampling-based profilers – we will discuss the nonviable design points later. Section 5 compares the performance of these viable design points and shows that HPM-immediate is the best performing sampling-based profiling approach.

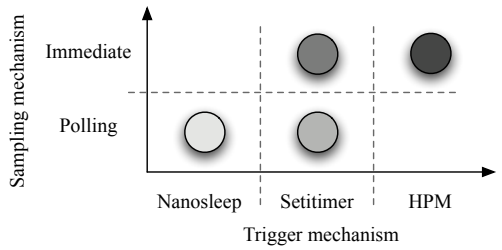

Figure 1. The design space for sampling profilers: sampling versus trigger mechanism.

2.1.1 Trigger Mechanisms

The trigger mechanism is the technique used to periodically initiate the sample-taking process. All production JVMs that we are aware of that use sampling to find optimization candidates use an operating system feature, such as nanosleep or setitimer, as the trigger mechanism for finding methods to optimize [6, 8, 30, 28, 18]. The *nanosleep* approach uses a native thread running concurrently with the VM. When the timer interrupt goes off, this native thread is scheduled to execute and then set a bit in the VM. When the VM next executes and sees the bit is set, a sample is taken. The *setitimer* approach does not use a native thread and thus has one less level of indirection. Instead, the interrupt handler for the VM sets the bit in the VM when the timer interrupt goes off. The third option, HPM, will be described in Section 3.

Timer-based sampling (using nanosleep or setitimer) is a low overhead profiling technique that can be used on all forms of code, both unoptimized and optimized code. It allows for the reoptimization of optimized code to higher levels, as well as for performing online profile-directed optimizations, when the profile changes, such as adaptive inlining [6].

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

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

kernel, which may not be feasible in all environments and will not be possible when source code is not available.

Lack of robustness among hardware implementations: Timer-based sampling is not robust among machines with different clock speeds and different microarchitectures because a faster machine will execute more instructions between timer-interrupts. Given a fixed operating system granularity, a timer-based profiler will collect fewer data points as microprocessors attain higher performance. Section 5 demonstrates this point empirically.

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

As we will see in Section 3, the HPM approach addresses all of these shortcomings.

2.1.2 Sampling Mechanisms

Once the sample-taking process has been initiated by some trigger mechanism, a sample can be taken. Two options exist for taking the sample (the vertical axis in Figure 1): *immediate* and *polling*. An immediate approach inspects the executing thread(s) to determine which method(s) they are executing. The polling approach sets a bit in the VM that the executing threads check at well-defined points in the code, called *polling points*. When a polling point is executed with the bit set, a sample is taken.

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

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

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

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

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

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

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

To avoid these limitations we advocate an immediate approach as described in Section 3.

2.2 Jikes RVM

Jikes RVM is an open source Java VM that is written in Java and has been widely used for research on virtual machines [2, 3]. Methods running on Jikes RVM are initially compiled by a baseline compiler, which produces unoptimized code quickly. An optimizing compiler is used to recompile important methods with one of its three optimization levels: 0, 1, and 2 [6, 7].

Jikes RVM uses timer-based sampling with polling for finding methods to be considered for optimization. Specifically, a timer-interrupt is used to set a bit in the virtual machine. On our platform, Linux/IA32, the default Jikes RVM system does this every 20 ms, which was the smallest level of granularity on Linux when that code was written several years ago. However, current Linux 2.6 kernels allow for a finer granularity of 4 ms by default.[1] Therefore, we also compare our work to an improved default system, where the timer-interrupt is smaller than 20 ms. Section 5 discusses the performance improvements obtained by reducing this inter-

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

rupt value. This illustrates an important shortcoming of an OS timer-based approach: as new versions of the OS are used, the sampling code in the VM may need to be adjusted.

Jikes RVM provides two implementation choices for timer-based sampling: 1) nanosleep-polling, and 2) setitimer-polling. The first strategy, which is the default, spawns a auxiliary, native thread at VM startup. This thread uses the nanosleep system call, and sets a bit in the VM when awoken before looping to the next nanosleep call. The polling mechanism checks this bit. The second strategy, setitimer, initiates the timer interrupt handler at VM startup time to set a bit in the VM when the timer goes off. Setitimer does not require an auxiliary thread. In both cases, the timer resolution is limited by the operating system timer resolution.

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

The controller thread uses a cost/benefit analysis to determine if a sampled method should be recompiled [6, 7]. It computes the expected future execution time for the method at each candidate optimization level. This time includes the cost for performing the compilation and the time for executing the method in the future at that level. The compilation cost is estimated using the expected compilation rate as a function of the size of the method for each optimization level. The expected future execution time for a sampled method is assumed to be the amount of time the method has executed so far, scaled by the expected speedup of the candidate optimization level.[2] For example, the model assumes that a method that has executed for N seconds will execute for N more seconds divided by the speedup of the level compared to the current level. The system uses the profile data to determine the amount of time that has been spent in a method.

After considering all optimization levels greater than the current level for a method, the model compares the sum of the compilation cost and expected future execution time with the expected future execution time at the current level, i.e., the future time for the method at a level versus performing no recompilation. The action associated with the minimum future execution time is chosen. Recompilations are performed by a separate compilation thread.

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

2.3 Hardware Performance Monitors

Modern processors are usually equipped with a set of performance event counter registers also known as hardware performance monitors (HPMs). These registers can be used by the microprocessor to count events that occur while a program is executing. The HPM hardware can be configured to count elapsed cycles, retired instructions, cache misses, etc. Besides simple counting, the hardware performance counter architecture can also be configured to generate an interrupt when a counter overflows. This interrupt can be converted to a signal that is delivered immediately to the process using the HPMs. This technique is known as event-based sampling. The HPM-sampling approach proposed in this paper uses event-based sampling using the elapsed cycle count as its interrupt triggering event.

3. HPM-Immediate Sampling

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

3.1 Benefits of Immediate Sampling

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

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

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

There are no restrictions on yield-point placement when using an immediate sampling mechanism. All epilogue yield-points can be removed, and the prologue and backedge yield-point placement can be optimized appropriately as long as it maintains correctness for the runtime system. The experimental results in Section 5 include a breakdown to

```
input     : CPU context
output    : void
begin
    registers ← GetRegisters (CPU context);
    processor ← GetJikesProcessorAddress ();
    if isJavaFrame (processor, registers) then
        stackFrame ← GetFrame (processor);
        methodID ← GetMethodID (stackFrame);
        sampleCount ← sampleCount + 1;
        samplesArray [sampleCount ] ← methodID;
    end
    HPMInterruptResumeResetCounter ()
end
```

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

show how much performance is gained by removing epilogue yield-points.

3.2 HPM-based Sampling

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

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

3.3 How HPM-immediate Fits in the Design Space of Sampling-Based Profiling

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

Referring back to Figure 1, there are two design points in sampling-based profiling that we do not explore because they are not desirable: nanosleep-immediate and HPM-polling. The nanosleep-immediate approach does not offer any advantage over setitimer-immediate: to get a sample, nanosleep incurs an even larger delay compared to setitimer, as explained previously. The HPM-polling approach is not desirable because it would combine a timely trigger mechanism (HPM-sampling) with a non-timely sampling mechanism (polling).

3.4 HPM Platform Dependencies

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

Standardizing the HPM interfaces is desirable because it can enable better synergy between the hardware and the virtual machine. In many ways it is a chicken-and-egg problem; without concrete examples of HPMs being used to improve performance, there is little motivation for software and hardware vendors to standardize their implementations. However, we hope this and other recent work [1, 23, 27] will show the potential benefit of using HPMs to improve virtual machine performance.

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

4. Experimental Setup

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

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

4.1 Virtual machine

As mentioned before, we use Jikes RVM, in particular the CVS version of Jikes RVM from April 10th, 2006. To ensure a fair comparison between HPM-immediate sampling with the other sampling-based profilers described in Section 2, we replace only the sampling-based profiler in Jikes RVM. The cost/benefit model for determining when to optimize to a specific optimization level and the compilers itself remain unchanged across all sampling-based profilers. In addition, we ensure that Jikes RVM's thread scheduling quantum remains unchanged at 20ms across different sampling-based profilers with different sampling rates.

The experiments are run on a Linux 2.6 kernel. We use the `perfctr` tool version 2.6.19 [22] for accessing the HPMs.

4.2 Benchmarks

Table 1 gives our benchmark suite. We use the SPECjvm98 benchmark suite [26] (first seven rows), the DaCapo benchmark suite [9] (next six rows), and the pseudojbb benchmark [25] (last row). SPECjvm98 is a client-side Java benchmark suite. We run all SPECjvm98 benchmarks with the largest input set (`-s100`). The DaCapo benchmark is a recently introduced open-source benchmark suite; we use release version 2006-10. We use only the benchmarks that execute properly on our baseline system, the April 10th, 2006 CVS version of Jikes RVM. SPECjbb2000 emulates the middle-tier of a three-tier system; we use pseudojbb, which runs for a fixed amount of work, i.e., for a fixed number of transactions, in contrast to SPECjbb2000, which runs for a fixed amount of time. We consider 35K transactions as the input to pseudojbb. The second column in Table 1 shows the number of application threads. The third column gives the number of application methods executed at least once; this does not include VM methods or library methods used by the VM. The fourth column gives the running time on our main hardware platform, the Athlon XP 3000+, using the default Jikes RVM configuration.

We consider two ways of evaluating performance, namely *one-run* performance and *steady-state* performance, and use the statistical rigorous performance evaluation methodology as described by Georges et al. [15]. For one-run performance, we run each application 11 times each in a new VM invocation, exclude the measurement of the first run, and then report average performance across the remaining 10 runs. We use a Student t-test with a 95% confidence interval to verify that performance differences are statistically meaningful. For SPECjbb2000 we use a single warehouse for measuring one-run performance.

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

Benchmark	Application threads	No. methods executed	Running time (s)
compress	1	189	6.1
jess	1	590	2.9
db	1	184	12.1
javac	1	913	6.5
mpegaudio	1	359	5.8
mtrt	3	314	3.9
jack	1	423	3.3
antlr	1	1419	5.6
bloat	1	1891	14.2
fop	1	2472	6.1
hsqldb	13	1277	7.3
jython	1	3093	21.2
pmd	1	2117	14.8
pseudojbb	1	812	6.6

Table 1. Benchmark characteristics for the default Jikes RVM configuration on the Athlon XP 3000+ hardware platform.

Processor	Frequency	L2	RAM	Bus
1500+	1.33GHz	256KB	1GB	133MHz
3000+	2.1GHz	512KB	2GB	166MHz

Table 2. Hardware platforms

4.3 Hardware platforms

We consider two hardware platforms, both are AMD Athlon XP microprocessor-based computer systems. The important difference between both platforms is that they run at different clock frequencies and have different memory hierarchies, see Table 2. The reason for using two hardware platforms with different clock frequencies is to demonstrate that the performance of HPM-sampling is more robust across hardware platforms with different clock frequencies than other sampling-based profilers.

5. Evaluation

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

5.1 Performance Evaluation

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

5.1.1 One-run Performance

Impact of sampling rate. Before presenting per-benchmark performance results, we first quantify the impact of the sample rate on average performance. Figure 2 shows the percentage average performance improvement on the Athlon XP 3000+ machine across all benchmarks as a function of the sampling interval compared to the default Jikes RVM, which uses a sampling interval of 20ms. The horizontal axis varies the sampling interval from 0.1ms to 40ms for the nanosleep- and setitimer-sampling approaches. For the HPM-sampling approach, the sampling interval varies from 100K cycles up to 90M cycles. On the Athlon XP 3000+, this is equivalent to a sampling interval varying from 0.047ms to 42.85ms. Curves are shown for all four sampling-based profilers; for the immediate-sampling methods, we also show a version including and excluding epilogue yield-points to help quantify the reason for performance improvement. Because an immediate approach does not require any polling points, the preferred configuration for the immediate sampling approach is with no yield-points.

We make several observations from Figure 2. First, comparing the setitimer-immediate versus the setitimer-polling curves clearly shows that an immediate sampling approach outperforms a polling-based sampling mechanism on our benchmark suite. The setitimer-immediate curve achieves a higher speedup than setitimer-polling over the entire sample rate range. Second, HPM-based sampling outperforms OS-triggered sampling — the HPM-immediate curve achieves higher speedups than setitimer-immediate. Third, removing epilogue yield-points yields a slight performance improvement for both the HPM-based and OS-triggered immediate sampling approaches. So, in summary the overall performance improvement for the HPM-sampling approach that we advocate in this paper comes from three sources: (i) HPM-based triggering instead of OS-triggered sampling, (ii) immediate sampling instead of polling-based sampling, and (iii) the removal of epilogue yield-points.

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

Per-benchmark results. Figure 3 shows the per-benchmark percentage performance improvements of all sampling profiler approaches (using each profiler's best sample rate) relative to the default Jikes RVM's nanosleep-polling sampling approach, which uses a sampling interval of 20ms. The graph on the left in Figure 3 is for the best sample rates on the Athlon XP 1500+ machine. The graph on the right is for the best sample rates on the Athlon XP 3000+.

The results in Figure 3 clearly show that HPM-immediate sampling significantly outperforms the other sampling profiler approaches. In particular, HPM-immediate results in an average 5.7% performance speedup compared to the default Jikes RVM nanosleep-polling approach on the Athlon XP 3000+ machine and 3.9% on the Athlon XP 1500+ machine. HPM-immediate sampling results in a greater than 5% performance speedup for many benchmarks (on the Athlon XP 3000+): antlr (5.9%), mpegaudio (6.5%), jack (6.6%), javac (6.6%), hsqldb (7.8%), jess (9.6%) and mtrt (18.3%).

As mentioned before, this overall performance improvement comes from three sources. First, immediate sampling yields an average 3.0% speedup over polling-based sampling. Second, HPM-sampling yields an additional average 2.1% speedup over OS-triggered sampling. Third, eliminating the epilogue yield-points contributes an additional 0.6% speedup on average. For some benchmarks, removing the epilogue yield-points results in significant performance speedups, for example jack (4.1%) and bloat (3.4%) on the Athlon XP 3000+ machine.

Statistical significance. Furthermore, these performance improvements are statistically significant. We use a one-sided Student t-test with a 95% confidence level following the methodology proposed by Georges et al. [15] to verify that HPM-immediate-no-yieldpoints does result in a significant performance increase over the non-HPM techniques. For each comparison, we require one test where the null hypothesis is that both compared techniques result in the same execution time on average, the alternative hypothesis is that HPM-immediate-no-yieldpoints has a smaller execution time. The null hypothesis is rejected for 10 out of 14 benchmarks when we compare to nanosleep-polling; it is rejected for 8 out of 14 benchmarks when we compare to setitimer-immediate-no-yieldpoints; and it is rejected for even 5 out of 14 benchmarks when we compare to HPM-immediate. This means that HPM-immediate-no-yieldpoints outperforms the best execution times compared to the other techniques with 95% certainty.

Robust performance across machines. The two graphs in Figure 3 also show that the HPM-immediate sampling profiler achieves higher speedups on the Athlon XP 3000+ machine than on the 1500+ machine. This observation supports our claim that HPM-sampling is more robust across hard-

[4] As Figure 2 shows, we explored a wide range of values between 2M and 9M for the HPM-immediate approach. An ANOVA and a Tukey HSD post hoc [15, 20] test with a confidence level of 95% reveal that in only 1.5% of the cases (7 out of 468 comparisons), the execution times differ significantly. This means one can use any of the given rates in [2M; 9M] without suffering a significant performance penalty. Therefore, we chose 9M as the best value for HPM-immediate.

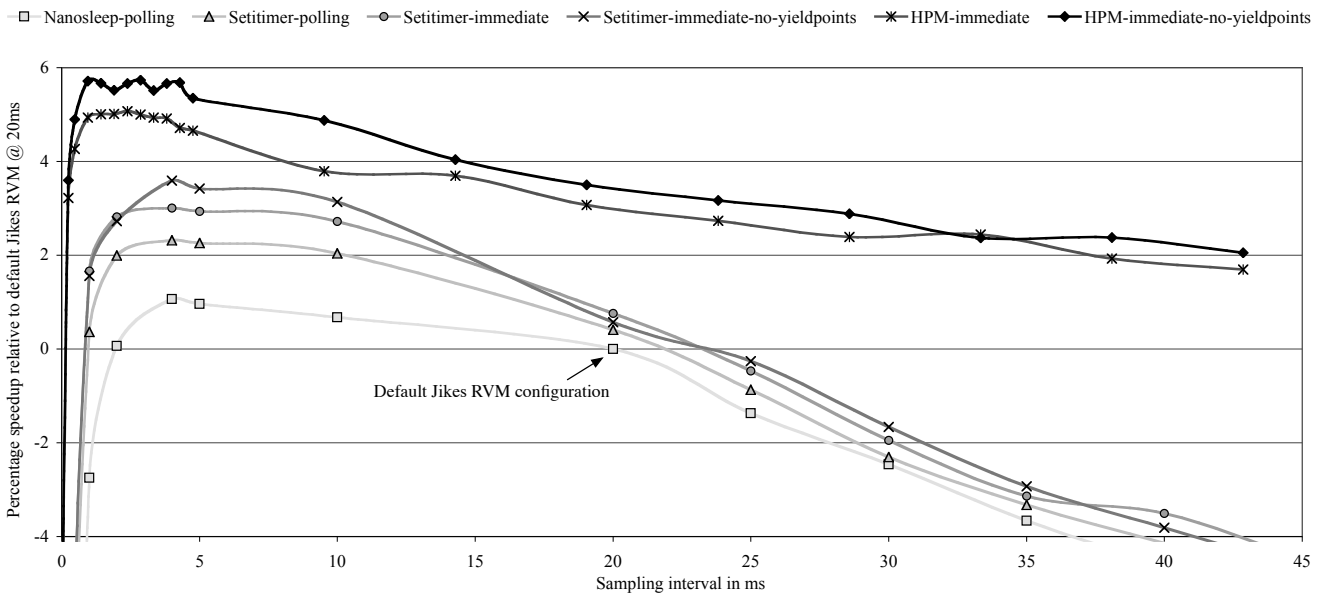

Figure 2. Percentage average performance speedup on the Athlon XP 3000+ machine for the various sampling profilers as a function of sample rate, relative to default Jikes RVM, which uses a sampling interval of 20ms.

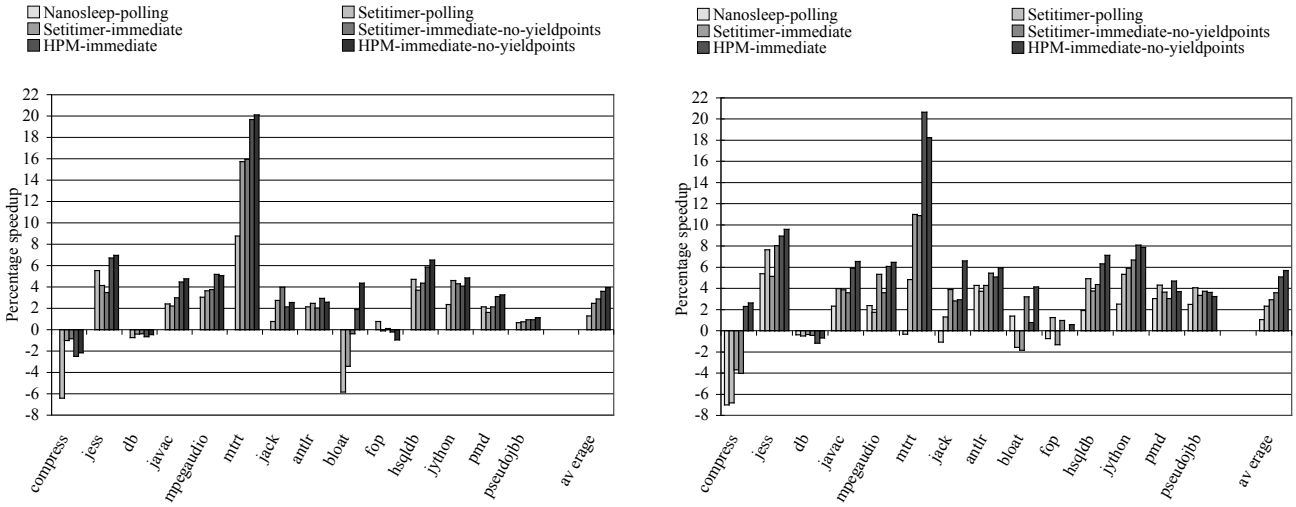

Figure 3. These figures show the percentage performance improvement relative to the default Jikes RVM configuration (nanosleep-polling with a 20ms sampling interval). Each configuration uses the single best sampling interval as determined from the data in Figure 2 for all benchmarks. The left and right graphs give improvement on the Athlon XP 1500+ with a 20M sample interval and the Athlon XP 3000+ with a 9M sample interval, respectively. On the former, the nanosleep-polling bars show no improvement, because the default configuration rate performed best for that particular technique; as a result, the left graph has one bar less.

ware platforms, with potentially different clock frequencies. The reason is that OS-triggered profilers collect relatively fewer samples as clock frequency increases (assuming that the OS time quantum remains unchanged). As such, the profile collected by an OS-triggered profiler becomes relatively less accurate compared to HPM-sampling when clock frequency increases.

5.1.2 Steady-state Performance

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

Figure 4 shows the average execution time per benchmark iteration across all VM invocations and all benchmarks. Two observations can be made: (i) it takes several benchmark iterations before we reach steady-state behavior, i.e., the execution time per benchmark iteration slowly decreases with the number of iterations, and (ii) while HPM-immediate is initially faster, the other sampling mechanisms perform equally well for steady-state behavior. This suggests that HPM-immediate is faster at identifying and compiling hot methods, but that if the hot methods of an application are stable and the application runs long enough, the other mechanisms also succeed at identifying these hot methods. Once all important methods have been fully optimized, no mechanism has a significant competitive advantage over the other, and they exhibit similar behavior. In summary, HPM-sampling yields faster one-run times and does not hurt steady-state performance. Nevertheless, in case a long-running application would experience a phase change at some point during the execution, we believe HPM-immediate will be more responsive to this phase change by triggering adaptive recompilations more quickly.

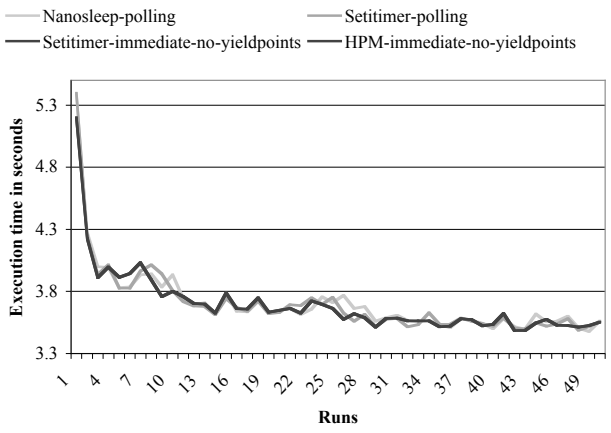

Figure 4. Quantifying steady-state performance of HPM-immediate-no-yieldpoints sampling: average execution time per run for 50 consecutive runs.

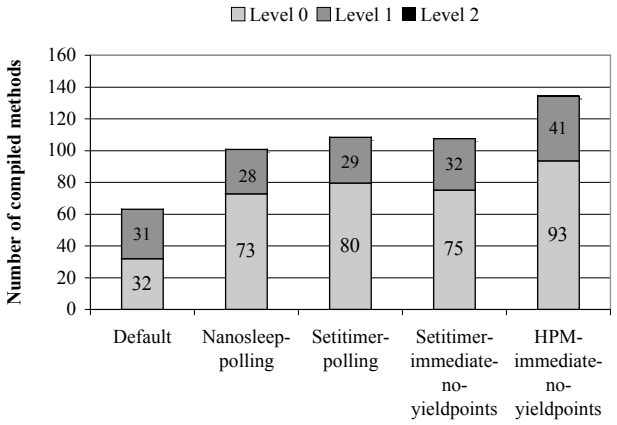

Figure 5. The average number of method recompilations by optimization level across all benchmarks on the Athlon XP 3000+.

5.2 Analysis

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

5.2.1 Recompilation Activity

Table 3 shows a detailed analysis of the number of methods sampled, the number of methods presented to the analytical cost/benefit model, and the number of method recompilations for the default system and the four sampling-based profilers. The difference between default and nanosleep-polling is that the default system uses 20ms as the sleep interval, whereas nanosleep-polling uses the best sleep interval for our benchmark suite, which is 4ms. Table 3 shows that HPM-sampling collects more samples (2066 versus 344 to 1249, on average) and also presents more samples to the analytical cost/benefit model (594 versus 153 to 408, on average) than OS-triggered sampling. This also results in more method recompilations (134 versus 63 to 109, on average).

Figure 5 provides more details by showing the number of method recompilations by optimization level. These are average numbers across all benchmarks on the Athlon XP 3000+ as reported by Jikes RVM's logging system. This figure shows that HPM-immediate sampling results in more recompilations, and that more methods are compiled to higher levels of optimization, i.e., compared to nanosleep polling, HPM-immediate compiles 27% more methods to optimization level 0 and 46% more methods to optimization level 1.

Figure 3 showed that HPM-immediate sampling resulted in an 18.3% performance speedup for `mtrt`. To better understand this speedup we performed additional runs of the base-

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

Table 3. Detailed sample profile analysis for one-run performance on the Athlon XP 3000+: 'S' stands for the number of method samples taken, 'P' stands for the number of methods that have been proposed to the analytical cost/benefit model and 'C' stands for the number of method recompilations.

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

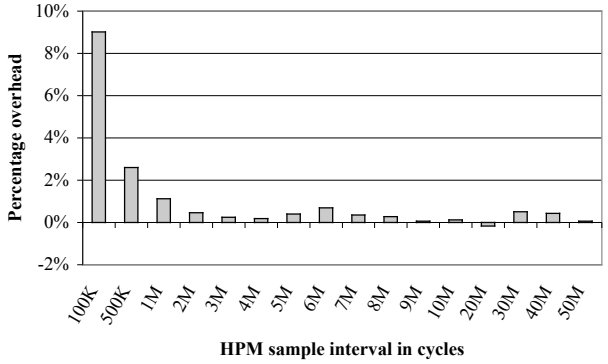

Figure 6. The average overhead through HPM-immediate-no-yieldpoints sampling for collecting samples at various sample rates.

5.2.2 Overhead

Collecting samples and recompiling methods obviously incurs overhead. To amortize this cost, dynamic compilation systems need to balance code quality with the time spent sampling methods and compiling them. To evaluate the overhead this imposes on the system, we investigate its two components: (i) the overhead of collecting the samples and (ii) the overhead of consuming these samples by the adaptive optimization system. As explained in Section 2.2, the latter is composed of three parts implemented as separate threads in Jikes RVM: (a) the organizer, (b) the controller, and (c) the compiler.

To identify the overhead of only collecting samples, we modified Jikes RVM to discard samples after they have been captured in both an HPM-immediate configuration and the baseline nanosleep-polling configuration. In both configurations no samples are analyzed and no methods are recompiled by the optimizing compiler. By comparing the execution times from the HPM-immediate configuration with those of the default Jikes RVM configuration that uses nanosleep-polling with the 20ms sample interval, we can study the overhead of collecting HPM samples. Figure 6 shows this overhead for a range of HPM sample rates averaged across all benchmarks; at the sampling interval of 9M cycles, the overhead added by HPM-immediate sampling is limited to 0.2%.

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

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

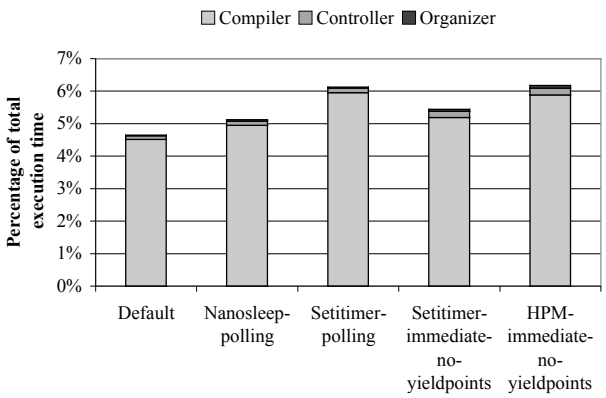

Figure 7. The average overhead of consuming the samples across all benchmark. The default systems uses 20ms as the sample interval, where as the other systems use their best sample intervals for our benchmark suite.

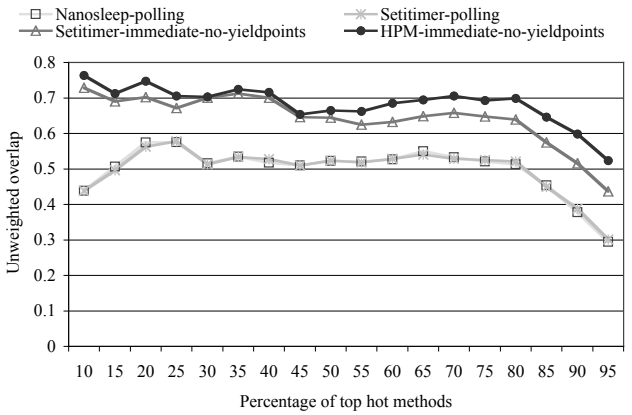

Figure 8. This graph quantifies sampling accuracy using the unweighted accuracy metric: the average overlap with the detailed profile is shown on the vertical axis for the top N percent of hot methods on the horizontal axis.

thread, we use Jikes RVM's logging system to capture each thread's execution time. Figure 7 shows the fraction of the time spent in the organizer, controller, and the compiler for all sampling profiler approaches averaged across all benchmarks. Based on Figure 7, we conclude that although HPM-immediate collects many more samples, the overhead of the adaptive optimization system increases by only roughly 1% (mainly due to the compiler). This illustrates that Jikes RVM's cost/benefit model successfully balances code quality with time spent collecting samples and recompiling methods — even when presented with significantly more samples as discussed in the previous section. We believe this property is crucial for the applicability of HPM-immediate sampling.

5.2.3 Accuracy

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

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

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

This metric attempts to measure how many methods in the detailed profile also appear in the profiles for the sampling approach of interest. Because the metric ignores how often a method is sampled in the detailed profile, it seems appropriate to consider only the top N percent of most frequently executed methods in the detailed profile.

Figure 8 shows the average unweighted metric for the various sampling-based profilers for the top N percent of hot methods. The horizontal axis shows the value of N. The vertical axis shows the unweighted metric score, so higher is better for this metric. For example, the accuracy of the profilers finding the top 20% of methods found in the detailed profile is 57.4%, 56.3%, 70.2%, and 74.5% for nanosleep-polling, setitimer-polling, setitimer-immediate-no-yieldpoints, and HPM-immediate-no-yieldpoints, respectively. Immediate sampling techniques are clearly superior to polling-based techniques on our benchmark suite.

Weighted metric. The weighted metric considers the sample counts associated with each method and computes an overlap score. To determine the weighted overlap score, we first normalize each vector with respect to the total number of samples taken in that vector. This yields two vectors with relative sample counts. Taking the element-wise min-

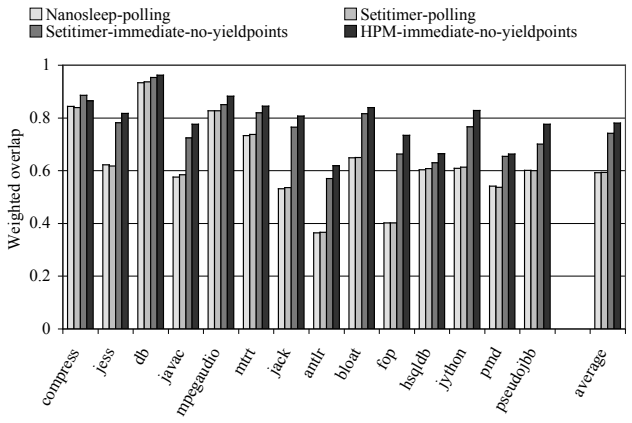

Figure 9. This figure shows the accuracy using the weighted metric of various sampling-based profilers compared to the detailed profile.

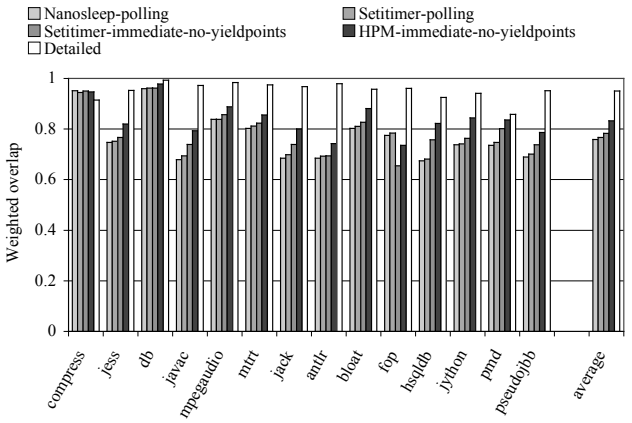

Figure 10. Quantifying sampling stability: higher is better.

imum of these vectors gives the weighted *presence* vector. The score then is the sum of the elements in the *presence* vector. For example, for x and y as defined above, the score is 0.71. Indeed, the only element that has samples in both vectors, a, scores $\frac{5}{5+0+2} = 0.71$ in x and $\frac{30}{30+4+0} = 0.88$ in y.

Figure 9 demonstrates the accuracy using the weighted metric for all benchmarks. This graph shows that polling-based sampling achieves the lowest accuracy on average (59.3%). The immediate techniques score much better, attaining 74.2% on average for setitimer. HPM improves this even further to a 78.0% accuracy on average compared to the detailed profile.

5.2.4 Stability

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

For example, given three vectors $x = ((a, 5), (b, 1), (c, 4))$, $y = ((a, 6), (b, 0), (c, 3))$ and $z = ((a, 5), (b, 2), (c, 3))$ that correspond to three benchmark runs of a particular configuration, the stability is computed as follows. First, we normalize the vectors and take the element-wise minimum of all the different combinations of vectors. Comparing x and y yields $((a, \frac{1}{2}), (b, 0), (c, \frac{1}{3}))$, comparing x and z yields $((a, \frac{1}{2}), (b, \frac{1}{10}), (c, \frac{3}{10}))$ and comparing y and z results in $((a, \frac{1}{2}), (b, 0), (c, \frac{3}{10}))$. Next, we compute the sum of the elements in each of the vectors and compute the final stability score as the mean of these values. That is, $\frac{0.83+0.9+0.8}{3} = 0.84$.

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

6. Related Work

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

6.1 Existing Virtual Machines

As mentioned earlier, Jikes RVM [6] uses a compile-only approach to program execution with multiple recompilation levels. All recompilation is driven by a polling-based sampler that is triggered by an operating system timer.

BEA's JRockit [8, 24] VM also uses a compile-only approach. A sampler thread is used to determine methods to optimize. This thread suspends the application threads and takes a sample at regular intervals. Although full details are not publicly available, the sampler seems to be triggered by an operating system mechanism. It is not clear if the samples are taken immediately or if a polling approach is used. Friberg's MS thesis [14] extends JRockit to explore the use of HPM events on an Itanium 2 to better perform specific compiler optimizations in a JIT. The thesis reports that using HPM events improves performance by an average of 4.7%. This work also reports that using HPM events to drive only recompilation (as we advocate in this work) does not improve performance, but does compile fewer methods. In

a workshop presentation, Eastman et al. [13] report similar work to Friberg's work. The slides claim to extend JRockit to use an HPM-immediate approach using Itanium 2 events to drive optimization. Unlike Friberg they do not report how the new approach compares to existing approach for finding optimization candidates, but instead focus on how HPM events improve performance when used by compiler optimizations.

Whaley [30] describes a sampling profiler that is implemented as a separate thread. The thread awakes periodically and samples the application threads for the purpose of building a dynamic call graph. He mentions that this sampling thread could be triggered by operating system or processor mechanisms, but used an operating system sleep function in his implementation. No performance results are reported comparing the various trigger approaches. The IBM DK for Java [28] interprets methods initially and uses method entry and back edge counters to find methods to optimize. Multiple optimization levels are used. A variant of the sampling technique described by Whaley (without building the dynamic call graph) is used to detect compiled methods that require further optimization.

IBM's J9 VM [18] is similar to the IBM DK for Java in that it is also interpreter-based with multiple optimization levels. It uses a sampling thread to periodically take samples. The implementation of the sampler is similar to Jikes RVM in that it uses a polling-based approach that is triggered by an operating system timer.

Intel's ORP VM [10] employs a compile-only strategy with one level of recompilation. Both the initial and the optimizing compiler use per-method counters to track important methods. A method is recompiled when the counter passes a threshold or when a separate thread finds a method with a counter value that suggests compiling it in the background. Optimized methods are not recompiled. No sampling is used.

In a technical report, Tam et al. [29] extend Intel's ORP to use HPMs to trigger dynamic code recompilation. They instrument method prologues and epilogues to read the HPM cycle counter. The cycle counter value is read upon invocation of a method and upon returning from the method, and used to compute the time spent in each method. These deltas are accumulated on a per method basis, and when a method's accumulated time exceeds a given threshold (and that method has been executed at least two times), the method is recompiled at the next optimization level. They report large overheads, and conclude that this technique cannot be used in practice. Our approach is different in that it does not use instrumentation and relies on sampling to find candidate methods.

Sun's HotSpot VM [21] interprets methods initially and uses method entry and back edge counters to find methods to optimize. No published information is available on what, if any, mechanism exists for recompiling optimized methods and if sampling is used. However, HotSpot was greatly influenced by the Self-93 system [16], which used a compile-only approach triggered by decayed invocation counters. Optimized methods were not further optimized at higher levels.

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

6.2 Other Related Work

Lu et al. [17] describe the ADORE dynamic binary optimization system, which uses event-sampling of HPMs to construct a path profile that is used to select traces for optimization. At a high level this is a similar approach to what we advocate, HPM profiling for finding optimization candidates, but details of the optimization system (a binary optimizer versus a virtual machine) are quite different.

Adl-Tabatabai et al. [1] used HPMs as a profiling mechanism to guide the placement of prefetch instructions in the ORP virtual machine. We believe prefetching instructions are inserted by forcing methods to be recompiled. Although this work may compile a method more than once based on HPM information, it does not rely on HPMs as a general profiling mechanism to find optimization candidates. Su and Lipasti [27] describe a hybrid hardware-software system that relies on hardware support for detecting exceptions in specified regions of code. The virtual machine then performs speculative optimizations on these guarded region. Schneider et al. [23] show how hardware performance monitors can be used to drive locality-improving optimizations in a virtual machine. Although these works are positive examples of how hardware can be used to improve performance in a virtual machine environment, neither address the specific problem we address: finding candidates for recompilation.

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

Zhang et al. [31] describe the Morph Monitor component of an online binary optimizer that uses an operating system kernel extension to implement an immediate profiling approach. Duesterwald et al. [12] describe a low-overhead software profiling scheme for finding hot paths in a binary optimization systems. Although both works are tackling the problem of finding optimization candidates, neither use HPMs.

Merten et al. [19] propose hardware extensions to monitor branch behavior for runtime optimization at the basic block and trace stream level. Conte et al. [11] describe a hardware extension for dedicated profiling. Our work differs in that we use existing hardware mechanisms for finding hot methods.

7. Conclusions and Future Work

To our knowledge, this is the first work to empirically evaluate the design space of several sampling-based profilers for dynamic compilation. We describe a technique, HPM-sampling, that leverages hardware performance monitors (HPMs) to identify methods for optimization. In addition, we show that an immediate sampling approach is significantly more accurate in identifying hot methods than a polling-based approach. Furthermore, an immediate sampling approach allows for eliminating epilogue yield-points. We show that, when put together, HPM-immediate sampling with epilogue yield-point elimination outperforms all other sampling techniques. Compared to the default Jikes RVM configuration, we report a performance improvement of 5.7% on average and up to 18.3% without modifying the compiler. Moreover, we show that HPM-based sampling consistently improves the accuracy, robustness, and stability of the collected sample data.

We believe there are a number of potentially interesting directions for future research. First, this paper used the cycle counter event to sample methods. Other HPM events may also be useful for identifying methods for optimization, such as cache miss count events and branch misprediction count events. Second, it may be interesting to apply the improved accuracy of HPM-based sampling to other parts of the adaptive optimization system, such as method inlining.

This work has demonstrated that there is potential for better synergy between the hardware and virtual machines. Both try to exploit a program's execution behavior, but little synergy has been demonstrated to date. This is partially due to the narrow communication channel between hardware and software, as well as the lack of cross-subdiscipline innovation in these areas. We hope this work encourages others to explore this fruitful area of greater hardware-virtual machine synergy.

Acknowledgments

We thank Steve Fink, David Grove, and Martin Hirzel for useful discussions about this work and Laureen Treacy for proofreading earlier drafts of this work. We also thank the anonymous reviewers for their valuable feedback.

Dries Buytaert is supported by a fellowship from the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). Andy Georges is a Research Assistant at Ghent University. Lieven Eeckhout is a Postdoctoral Fellow of the Fund for Scientific Research—Flanders (Belgium) (F.W.O.—Vlaanderen). Ghent University is a member of the HiPEAC Network of Excellence.

References

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

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

[3] B. Alpern, S. Augart, S. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. McKinley, M. Mergen, J. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes Research Virtual Machine project: Building and open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.

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

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

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

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

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

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

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

[11] T. M. Conte, K. N. Menezes, and M. A. Hirsch. Accurate and practical profile-driven compilation using the profile buffer. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 36–45, Dec. 1996.

[12] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *Proceedings of the International Conference on Architectural Support for Programming*

Languages and Operating Systems (ASPLOS), 202–211, Nov. 2000.

[13] G. Eastman, S. Aundhe, R. Knight, and R. Kasten. Dynamic profile-guided optimization in the BEA JRockit JVM, In *3rd Workshop on Managed Runtime Environments (MRE) held in conjunction with the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Mar. 2005.

[14] S. Friberg. Dynamic profile guided optimization in a VEE on IA-64. Master's thesis, KTH – Royal Institute of Technology, 2004. IMIT/LECS-2004-69.

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

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

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

[18] D. Maier, P. Ramarao, M. Stoodley, and V. Sundaresan. Experiences with multithreading and dynamic class loading in a Java just-in-time compiler. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 87–97, Mar. 2006.

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

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

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

[22] perfctr. perfctr version 2.6.19. http://user.it.uu.se/ mikpe/linux/perfctr.

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

[24] K. Shiv, R. Iyer, C. Newburn, J. Dahlstedt, M. Lagergren, and O. Lindholm. Impact of JIT JVM optimizations on Java application performance. In *Proceedings of the 7th Annual Workshop on Interaction between Compilers and Computer Architecture (INTERACT) held in conjunction with the International Symposium on High-Performance Computer Architecture (HPCA)*, 5–13, Mar. 2003.

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

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

[27] L. Su and M. H. Lipasti. Speculative optimization using hardware-monitored guarded regions for Java virtual machines. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE)*, 22–32, June 2007.

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

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

[30] J. Whaley. A portable sampling-based profiler for Java virtual machines. In *Proceedings of the ACM 2000 Conference on Java Grande*, 78–87, June 2000.

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