Dynamic Compilation: The Benefits of Early Investing

Prasad Kulkarni
Florida State University
IBM T.J. Watson Research Center
kulkarni@cs.fsu.edu

Matthew Arnold Michael Hind
IBM T.J. Watson Research Center
{marnold,hindm}@us.ibm.com

Abstract

Dynamic compilation is typically performed in a separate thread, asynchronously with the remaining application threads. This compilation thread is often scheduled for execution in a simple round-robin fashion either by the operating system or by the virtual machine itself. Despite the popularity of this approach in production virtual machines, it has a number of shortcomings that can lead to suboptimal performance.

This paper explores a number of issues surrounding asynchronous dynamic compilation in a virtual machine. We begin by describing the shortcomings of current approaches and demonstrate their potential to perform poorly under certain conditions. We describe the importance of enforcing a minimum level of utilization for the compilation thread, and evaluate the performance implications of varying the utilization that is enforced. We observed surprisingly large speedups by increasing the priority of the compilation thread, averaging 18.2% improvement over a large benchmark suite. Finally, we discuss options for implementing these techniques in a VM and address relevant issues when moving from a single-processor to a multiprocessor machine.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Optimization, Run-time environments, Compilers

General Terms Languages, Performance

Keywords Virtual machines, Dynamic compilation, Java

1. Introduction

Because of the widespread use of the Java programming language, language-level virtual machines (VMs) are widely deployed. Most high-performing modern virtual machines employ an adaptive optimization system [3], whose task is to monitor the running application to determine which portions should be further optimized to improve performance [21, 2, 26, 9, 20, 5]. In such systems, methods are initially interpreted or compiled by a quick compiler that produces unoptimized code, and then compiled, if appropriate, one or more times with an optimizing compiler. Most VMs perform this compilation by executing an optimizing compiler in a separate thread [2, 26, 9, 20, 5] because it offers two distinct advantages. First, it provides isolation between the compiler's and application's runtime data, such as providing separate runtime stacks. Second, and more important, it allows the compiler to execute asynchronously with the application threads. On a uniprocessor, this allows the thread scheduler to interleave the compilation with the application's execution, ensuring that a long compilation will not delay the application's progress. On a multiprocessor, the compiler can execute in parallel with the application. The common perception is that asynchronous compilation is highly effective, and this perception is supported by its use in many production-quality VMs [2, 26, 9, 20, 5].

The compilation thread is typically implemented in the same manner as application threads. On most VMs these are operating systems threads [26, 9, 20, 5], but in some cases they are threads that the virtual machine multiplexes on top of operating systems threads [2]. In many cases, the compilation thread is scheduled in a round-robin manner with other VM threads; however, since the compilation thread is an optional virtual machine-created thread, it is not obvious that a round-robin approach is appropriate. Arguments can be made that it should receive less utilization than the application threads [19] (it is optional, so stealing cycles from the application is not desirable) or more utilization than the application threads (because it will improve the application's performance, it should happen as soon as possible).

This paper argues that traditional asynchronous compilation implementations often do not achieve the highest performance. Even for the most simple case of a single-processor machine, a single-threaded application and a VM with one compilation thread, the performance of the round robin scheduler may be suboptimal. We observed an 18.2% *average* performance improvement over a large benchmark suite by simply increasing the priority of the compilation thread.

Even worse, the performance when using a round-robin scheduler is not robust and can degrade even further for certain types of applications. For example, as the number of threads in the application increases, the resources given to the compilation thread are reduced and the performance approaches that of a system without an optimizing compiler.

This paper describes the above mentioned problems in detail, and proposes alternate solutions. The contributions of this work are

- we show the importance of ensuring a level of utilization for the compilation thread and empirically validate this in a production virtual machine on a large benchmark suite;
- we evaluate a range of compilation thread utilizations and quantify the effect on both performance and pause times;
- we discuss issues when deploying a system on a multiprocessor and provide empirical data that illustrates these issues; and
- we discuss how the findings above would impact a VM with multiple compilation threads.

The results of this work will be applicable to the significant number of virtual machines that use a compilation thread.

Suite	Benchmark	Methods Executed	Multi-Threaded
Dacapo 2006-10 [6]	antlr	2409	no
	bloat	3473	no
	eclipse	9104	yes
	fop	4105	no
	hsqldb	2816	yes
	jython	4509	no
	luindex	2302	no
	lusearch	1731	yes
	pmd	3410	no
	xalan	1108	no
SPEC jvm98 [25]	compress	770	no
	db	782	no
	jack	746	no
	javac	1467	no
	jess	1140	no
	mpegaudio	866	no
	mtrt	853	yes
SPEC jbb2000 [24]	psuedojbb	1197	yes
Others	daikon [12]	2108	no
	kawa [18]	1855	no
	ipsixql [11]	828	no
	soot [23]	2061	no
	xerces [27]	521	no

Table 1. Benchmarks used in our experiments

The rest of this paper is organized as follows. Section 2 describes the empirical methodology we use for this work. Section 3 discusses problems with relying on a round-robin thread scheduler for asynchronous compilation on a single processor. Section 4 addresses these problems by ensuring the proper level of utilization for asynchronous compilation. Section 5 explores issues that arise when moving to a multiprocessor machine. Section 6 discusses how these ideas could be altered for use with a VM that supports multiple compilation threads. Section 7 discusses related work and Section 8 draws conclusions.

2. Methodology

The implementation and results presented in this paper were performed using a development version of IBM's J9 VM and its high-performance optimizing JIT compiler [14, 20]. The VM initially interprets methods and uses counters and sampling to promote methods to higher levels of optimization. Compilation is performed asynchronously on a separate thread. On operating systems that provide support for user-level thread priorities (such as AIX and Windows) J9 uses thread priorities to increase the priority of the compilation thread. As discussed in Section 4.5, Linux does not provide the desired thread priority support.

Table 1 lists our benchmark suite, which includes the complete suites from SPECjvm98 [25] and DaCapo [6], a variant of the SPECjbb2000 benchmark [24], and five other benchmarks for a total of 23 benchmarks. Each benchmark has a small and large input, resulting in 46 benchmark/input pairs. For SPECjvm98 the small and large inputs are the s10 and s100 inputs, respectively. For DaCapo, the "small" and "large" inputs provided by the benchmark suite were used. `pseudojbb` is a variant of SPECjbb2000 that runs for a fixed number of transactions, in contrast to SPECjbb2000, which runs for a fixed amount of time. The small and large inputs for pseudojbb consist of executing 12,000 and 200,000 transactions, respectively. The other five benchmarks do not come with defined inputs, so we created an appropriate small and large input for each.

Two benchmarks, *jython* and *xalan*, with their large input did not execute correctly with our development version of the VM, so these inputs were excluded from the suite.

In addition, we have included results for the IBM Trade Performance Benchmark V6.1 (Trade 6.1) [17]. Trade is a large multithreaded server benchmark that runs on a J2EE Application server. Trade stresses most components of the virtual machine by executing more than 40,000 methods. At the same time, the distribution of runtime within the methods is relatively flat, so no single method dominates execution. As a result, over 6,500 methods are compiled by the JIT during a typical run.

All our experiments were conducted on Intel Xeon 2.8GHz processors in three different machine configurations: (a) single-processor, (b) single-processor with hyperthreading, and (c) two-processor with hyperthreading. A pure single processor configuration was achieved by using the BIOS settings to disable hyperthreading. We used Red Hat Linux kernel 2.6.9 as our operating system.

3. Limitations of Round-Robin Scheduling

Many Java VMs have a compilation thread that runs asynchronously and is scheduled together with the application threads in a round-robin fashion. This section describes the limitations of this approach and illustrates their impact on two benchmarks.

3.1 The Problem

Although round-robin scheduling appears to be a simple and fair scheme for scheduling the compilation thread, the actual CPU access given to the compilation thread can vary greatly, depending on the application. In the most extreme cases, the CPU resources given to either the compilation thread or the application thread(s) can approach zero.

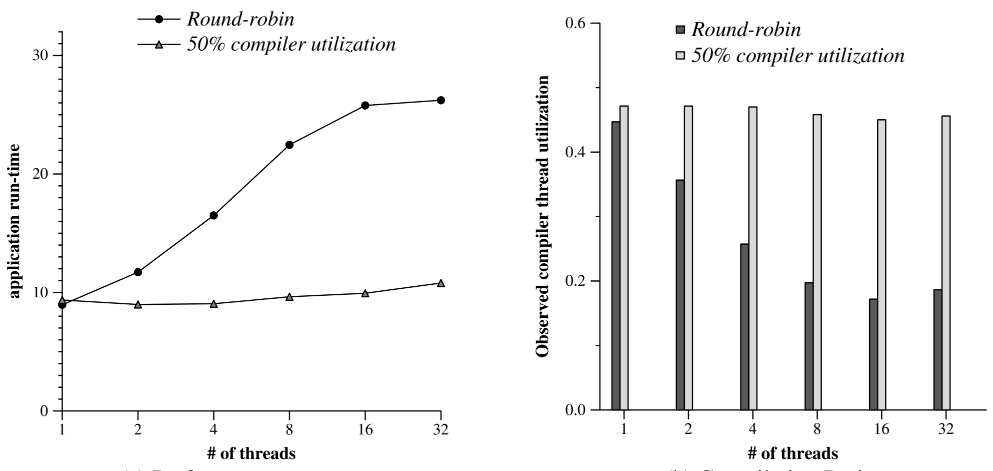
Reduced resources due to multithreading With round-robin scheduling, when an application has N threads, the resources given to the compilation thread are $1/(N+1)$. Thus, resources given to the compilation thread are reduced as the number of threads in the application increases. This effect can produce poor startup performance for multithreaded programs, because the compilation thread is not given enough time to execute. As the number of threads becomes large, performance approaches that of a system without an optimizing compiler.

Reduced resources due to yielding There are also more subtle scenarios where the relative resources given to the compilation thread can be reduced, or even increased, based on the application and compilation behavior.

The thread scheduler in the OS has a minimum time quantum at which context switches occur. If an executing thread yields on its own before the end of the time slice, then it gives up the rest of its slot and another thread begins executing. If the OS scheduling time slice is large, and some of the threads are yielding frequently (either the compilation thread or the application threads), then the resources given to the compilation thread can vary significantly. The version of Linux we are using has a thread scheduling time quantum of 100 milliseconds, which is a significant amount of execution time on a 4 GHz machine (400 million cycles per time slice).

The application could yield frequently for a number of reasons, such as I/O, or threads that are frequently passing work to each other. After the application threads all yield once, the compilation thread will be given a chance to execute for an entire time slot (or until there is nothing to compile).

Similarly, the compilation thread can also see reduced utilization if it yields for any reason. For example, printing or logging



(a) Performance (b) Compilation Ratio

Figure 1. Effect of increasing the number of threads for `mtrt`

the name of the method being compiled can cause the compilation thread to lose its time slot after each method name is printed, or when the I/O buffer is flushed.

Additionally, the compilation thread yields when the compilation queue becomes empty. It seems natural for the compilation thread to sleep when it has nothing to do. However, if compilation activity arrives in short bursts and the compilation thread forfeits its time slot quickly, then it will wait an entire time quantum to regain the CPU again. This is not a utilization issue, but one of latency; however, our results in Section 4.1 show that this latency can have a measurable impact on performance.

3.2 Minimum Compilation Thread Utilization

To evaluate the impact of compilation thread utilization in multithreaded applications we implemented a scheduler within a development version of the J9 VM that enforces a resource utilization ratio between the application and compilation threads. For a given compilation thread utilization of $X\%$, the scheduler will ensure the compilation thread receives at least $X\%$ of the total CPU resources.

We prototyped such a scheduler by using thread priorities in Linux. We defined our own VM-level time slice quantum (we chose 10ms). At the end of each time quantum, a high priority thread (which we call the *control thread*) wakes up to adjust the priorities of the other threads. The control thread lowers and raises the priority of the compilation thread to achieve an overall CPU utilization of $X\%$. For example, for 80% compilation thread utilization, the compilation thread is run at high priority for 4 of 5 time slices, and at low priority for 1 time slice.

Thread priorities are an effective implementation mechanism for two reasons. First, it scales well to a multiprocessor machine because the OS can schedule low-priority threads on processors that are not busy running high-priority threads. Second, even on a single processor machine, using priorities simplifies the implementation because it avoids deadlocks or accidentally leaving a processor idle; if a high-priority thread yields for any reason, such as there is nothing to compile or it is waiting for a lock that is held by a low-priority thread, the low-priority threads will automatically begin execution.

3.3 Experimental Evaluation

This section presents two experiments that demonstrate the shortcomings of round-robin scheduling. These two experiments are hand-picked examples to show the kind of issues that can occur; a more thorough evaluation using all of the benchmarks in our suite is contained in Section 4.

mtrt The first benchmark we consider is `mtrt` from the SPECjvm98 benchmark suite. We run with a modified driver that allowed us to vary the number of application threads. The total work performed remains fixed regardless of the number of threads; when there are N threads, each thread performs $1/N^{th}$ of the workload. If the VM were perfectly scalable in regard to threading, the program would complete in the same amount of time regardless of the number of threads.

Figure 1 shows the performance of our modified `mtrt` with the number of threads varying from 1 to 32. The graph on the left shows the absolute execution time of the program, so higher is worse. One line represents performance when using a round-robin scheduler for the compilation thread, while the other shows performance when enforcing a compilation thread utilization of 50%, as described in Section 3.2. With the round-robin scheduler, the program execution time increases as the number of threads increases, taking 2.5 times as long to complete when the threads are increased from 1 to 16. With 50% utilization, the performance stays relatively constant regardless of the number of application threads.

The graph on the right in Figure 1 confirms the expected compilation activity for both systems. Compilation time was measured using the PAPI library [7] to measure the cycles executed by each thread. For the round-robin scheduler, the percentage of CPU cycles given to the compilation thread degrades as the number of application threads increases. With our scheduler, the compilation thread utilization remains constant.

Trade 6.1 As described in Section 2, Trade 6.1 is a J2EE server benchmark. Trade is inherently a multithreaded application so there was no easy way to control the number of threads that execute.[1] Instead, we varied the machine configuration, comparing two variations of the same machine. One is a multiprocessor machine with two hyper-threaded processors. The second is a single processor machine; it has one processor, and hyperthreading is disabled. Other than the processor configuration, the machines are identical.

[1] It is easy to control the load generation, which indirectly affects the number of threads that execute simultaneously in the application server. However, there are additional internal threads within the application server, so the total number of threads cannot easily be controlled.

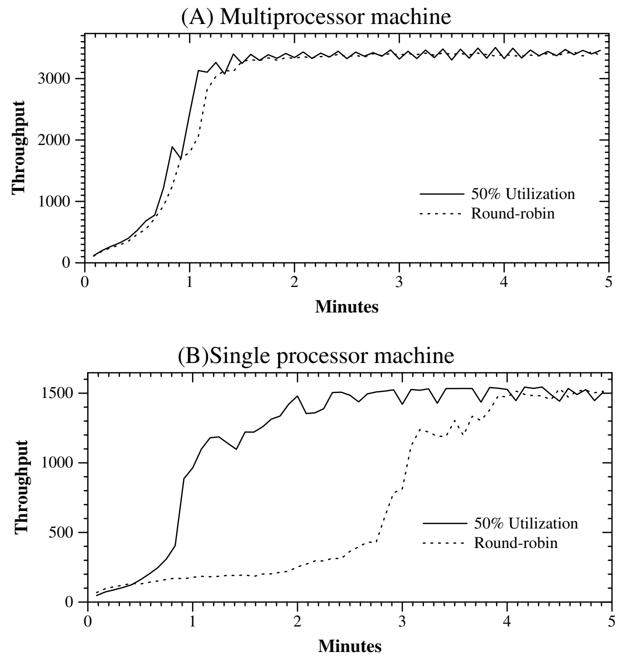


Figure 2. Websphere J2EE Application Server running Trade 6.1, performance of a round-robin scheduler compared to 50% utilization. The graphs show throughput over time, where higher is better. Rampup time of the round-robin scheduler is significantly degraded when CPU resources are constrained.

This machine configuration enables a performance evaluation as CPU contention is increased. On the single processor machine, threads are forced to contend for CPU resources in a round-robin fashion. On the multiprocessor machine, the compilation thread can run in parallel with the application threads and can potentially run uninterrupted during any periods that the processor would otherwise be idle.

Figure 2 compares the performance of a round-robin scheduler to the 50% compilation thread utilization. Performance is reported as throughput over time, where higher throughput is better. Both schedulers eventually converge on the same peak throughput; thus, the only varying factor is how long it takes to reach peak throughput, which is referred to as *rampup time*. As described in Section 2, Trade is a large application, so rampup time can be significant.

On the multiprocessor the round-robin scheduler and 50% utilization are roughly equal, with a rampup time of about 1.5 minutes. However, as the CPU resources are constrained, the rampup time increases to around 4 minutes: an increase of more than a factor of 2.5. With an enforced utilization of 50%, the rampup time stays mostly constant across the processor configurations (approximately 1.5 minutes).

4. Selecting a Compilation Thread Utilization

The previous section motivated that some level of utilization is required to ensure reasonable performance for multithreaded applications. This section addresses what level of utilization should be used. We argue that if the controller (the component of the adaptive optimization system that decides what should be compiled) is making good compilation decisions, then 100% compilation thread utilization will, on average, outperform lower utilizations.

Figure 3 helps illustrate this point by comparing the behavior of 50% and 100% utilization. For simplicity of explanation, assume that the application consists of only a single method; generalizing to multiple methods is straightforward. The horizontal lines represent the application thread running over time. This line becomes thick when the optimized code is executing. The gray box(es) represent a method being compiled by the compilation thread. With 50% utilization, the compilation is interleaved with the application. The vertical dotted lines align points in absolute time where the application has made equal forward progress, assuming optimized code runs faster than unoptimized code.

If the program runs long enough (past point B in the figure), then 100% utilization produces faster overall performance. The total amount of time spent compiling is the same in both cases, but 100% utilization finishes compiling sooner. Thus, more of the application execution occurs in optimized code.

The only scenario in which 50% utilization is faster is if the program completes at some point between time A and time B in the figure. In this case, 50% outperforms 100% because the program can complete without waiting for the whole compilation to finish.

However, having the program complete between A and B is precisely what the adaptive optimization system tries to avoid; that is, wasting time compiling methods right before the program is about to complete. The goal of the controller component in an adaptive optimization system is to predict when a method is likely to continue running long enough to justify the cost of optimizing it. Thus, if a controller is tuned to achieve good performance, the number of occurrences where methods stop executing shortly after being compiled will be small, and as a result, 100% utilization is likely to outperform lower utilizations.

4.1 Experimental Evaluation

This section evaluates the impact of various compilation thread utilizations on our benchmark suite, using our utilization-based scheduler described previously in Section 3.2. The recompilation strategy of the VM partly determines which utilization performs best, so we evaluate our scheduler using two different controllers, which we refer to as *aggressive* and *conservative*.

Both controllers use the same basic recompilation framework; all methods begin executing in the interpreter, and a combination of method invocation counts and timer-based sampling is used to determine when methods are recompiled. The only difference between the two controllers is how methods progress through the optimization levels.

1. **Aggressive controller:** The lowest level of optimization (level O0) is not used. The first time a method is selected for optimization it is optimized at a moderate level of optimization (level O1). If the method remains hot, it may be recompiled again at higher levels. This strategy is optimized for reaching steady state quickly, but is not optimal for startup (i.e., first run) performance.
2. **Conservative controller** The lowest level of optimization (O0) *is* used for the first compilation of a method, and further optimization levels (O1 and higher) are used if the method remains hot. This strategy provides improved startup performance, but takes longer to converge on steady state performance (where compilation stabilizes and peak performance is reached).

Both controllers have their counter thresholds tuned on a large benchmark suite, so both can be considered “good” controllers, given the set of optimization levels that they utilize.

Aggressive controller results We begin by measuring the performance impact of varying the compiler utilizations when using the aggressive controller. We measure the performance of a single execution for all benchmark/input pairs described previously in Section 2 (each program has two inputs). All timings reported were col-

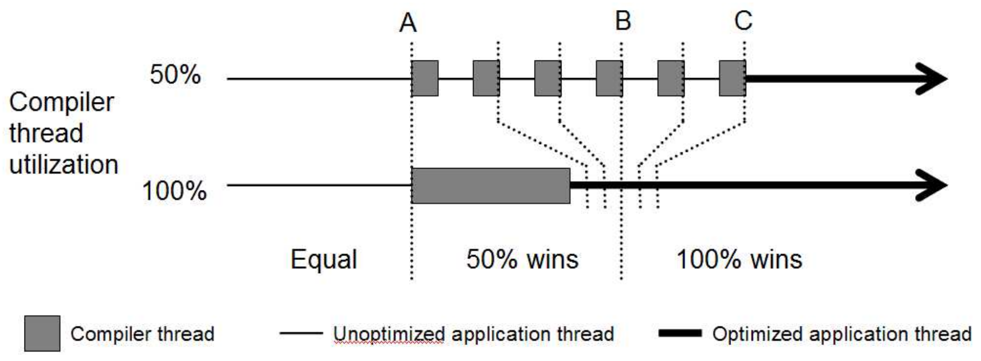

A
B
C
50%
Compiler thread utilization
100%
Equal
50% wins
100% wins
Compiler thread
Unoptimized application thread
Optimized application thread

Figure 3. Comparing compilation thread utilizations 100% and 50%

lected 10 times on an idle machine and averaged to reduce noise. Likewise, other metrics are computed as the average of these 10 runs.

Table 2 presents the result for the aggressive controller on our benchmark suite. Each compiler utilization from 10% to 100% is represented by the middle columns labeled "Compilation Thread Utilization" and the rightmost column labeled "Round Robin" represents the round-robin OS scheduler, i.e., no compilation thread utilization is enforced.

The rows represent the following four quantities:

1. **Performance improvement**: Performance improvement relative to the round-robin OS scheduler.
2. **Time in queue:** The average time that a method spends waiting in the compilation queue, i.e., waiting to be compiled by the compilation thread. The time is observed when the method is removed from the queue. The average time is computed first for each benchmark (while the benchmark runs), then averaged across benchmarks.
3. **Length of queue:** The average number of methods in the compilation queue, measured when a method is removed from the queue. Averaging is first performed within each benchmark, then across all benchmarks.
4. **Methods compiled:** The number of methods compiled during the application's execution.

The most important observation in the table is the significant impact that thread utilization has on performance. At 100% utilization, performance is improved over the round-robin scheduler by 18.2% on average. 50% utilization roughly matches the round-robin scheduler, yielding an average speedup of 2.2%. This is expected because the majority of the applications are single-threaded. Utilizations 10%–30% reduce performance because not enough resources are given to the compilation thread.

The rows "Time in queue" and "Length of queue" help explain why compilation thread utilization has such a large impact on performance. For the round-robin scheduler, the average queue length and time in queue are quite large (60 methods and 1465 milliseconds, respectively). This large queue delay is a result of the large amount of compilation performed by the aggressive controller. As the compilation thread priority increases, the average compilation queue length and the average time that a method spends in the queue both decrease. At 80% utilization, average queue length is 2 methods, and the average delay is 32ms, which are both reduced by a factor of 30 or more over the round-robin scheduler.

At first it was surprising to see that 100% utilization was so effective given the amount of compilation that is taking place. Conventional wisdom might suggest that if the queue of methods to compile is getting too long, it might be beneficial to use a lower utilization to avoid over-compiling. However, recall that the thresholds for the aggressive controller were tuned well (using the round-robin scheduler) so it is not the case that the controller is performing unnecessary compilations; these methods are important to be compiled for good performance. Higher utilizations allow these compilations to finish sooner, minimizing the time spent in the unoptimized versions.

Given the long compilation queue delays, we experimented with different sorting criteria for removing methods from the queue. The hypothesis was that maybe higher utilization would not be as beneficial if compilations in the queue were properly ordered. We implemented a "continuously sorted" compilation queue that moved a method forward in the queue if it became hot while in the queue. However, this approach did not yield any performance improvements over the original system.

Also surprising was that 100% utilization actually *reduced* the average number of methods complied. The controller logic for triggering compilations was not changed, and it was expected that more compilations would complete at higher utilization levels; however, the number of compilations actually decreased at high utilizations. We believe this was because the program's execution time was significantly shorter. The round-robin system runs longer and has more time for compiles to be triggered by the timer-based profiler.

Figure 4 shows a breakdown of the 100% utilization performance relative to the round-robin OS scheduler. Each benchmark is represented by its own bar, and higher represents better performance. Four of the benchmarks have speedups of more than 50%, meaning that the benchmark completed in less than half the time of the round-robin scheduler. 100% utillization degraded performance for 3 of 46 benchmark/input pairs. The largest slowdown was -24% for `pmd-small`. This benchmark has a large number of methods that run for a relatively short amount of time, so the decision to compile them did not pay off. As a result, 50% utilzation beats 100% for this benchmark. This is expected to occur for some benchmarks because the compilation heuristics are tuned for average performance; in fact, we were surprised that more benchmarks were not degraded with 100% utilization.

Conservative controller results We repeated the same experiment using the conservative controller; the results are presented in Table 3 in the same format as the previous table.

For the conservative controller, enforcing utilization again resulted in significant speedups, but with a number of differences from the aggressive controller results. 100% utilization yields an average speedup of 9.3%, which is a smaller improvement than was seen with the aggressive controller. The conservative controller compiles more methods than the aggressive controller, but since methods are compiled at a lower optimization level (O0 rather than O1) the compilations complete much faster so there is very little

	Compilation Thread Utilization										Round Robin
	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%	
Performance improvement (%)	-85.8	-39.2	-17.0	-4.3	2.2	8.3	11.7	14.8	16.9	18.2	0.0
Time in queue (ms)	6500	3885	2408	1412	956	509	254	96	32	21	1465
Length of queue (# methods)	157	112	90	61	46	27	14	5	2	1	60
Methods compiled (# methods)	399	472	516	550	572	585	584	569	551	523	567

Table 2. Effect of changing compilation thread utilization with the aggressive controller

	Compilation Thread Utilization										Round Robin
	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%	
Performance improvement (%)	-45.7	-10.7	0.2	7.0	9.7	10.2	10.1	9.9	9.5	9.3	0.0
Time in queue (ms)	862	301	142	54	30	16	9	3	1	1	130
Length of queue (# methods)	83	41	21	12	7	4	3	2	1	1	25
Methods compiled (# methods)	632	682	688	696	702	694	690	684	678	672	705

Table 3. Effect of changing compilation thread utilization with the conservative controller

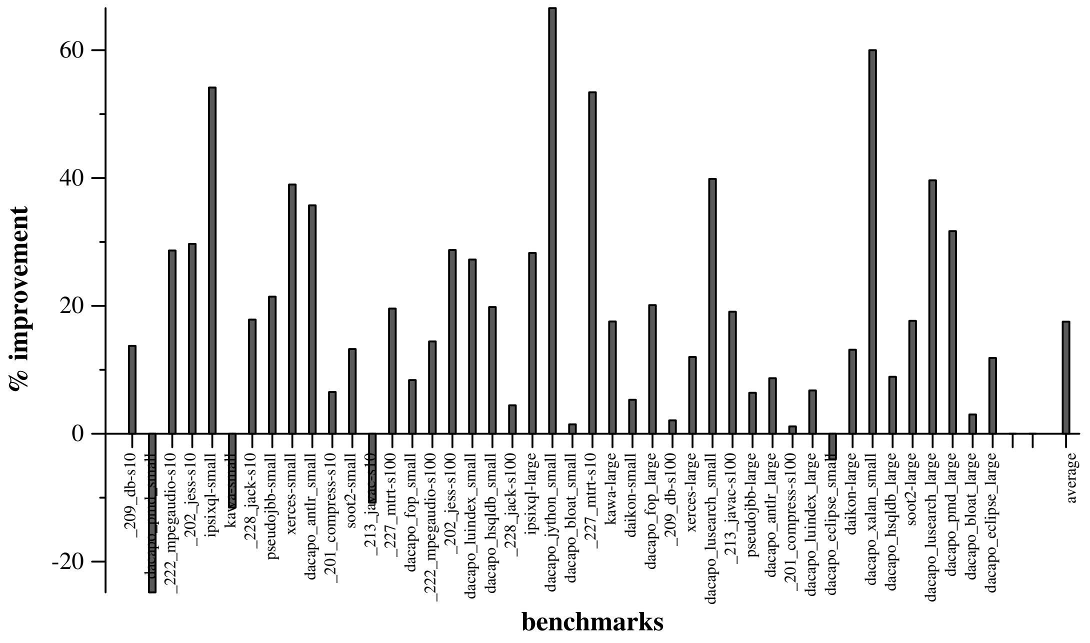


Figure 4. Performance improvement of 100% compilation thread utilization over the round-robin thread scheduler for the aggressive controller

backup of the compilation queue. If the compilation thread has enough time to finish compiling all methods in the queue, then it simply sleeps; there is no advantage to higher utilizations if there is nothing to compile.

More surprising is that all utilizations from 50% – 90% performed as well as, or slightly better than, 100% utilization, and all were significantly better than the round-robin scheduler. 18 of the 23 benchmarks are single threaded, so we were not expecting large improvements from 50% utilization. Our current hypothesis is that this is more of a latency issue than utilization. With the compiles completing quickly and having very little delay in compilation queue, the compilation thread yields on its own more frequently. Once it yields, methods that enter the queue will not be compiled until the compilation thread runs again in 100ms. The average time in queue supports this hypothesis, being reduced from 130ms in the round-robin system to 30ms at 50% utilization. The latency is reduced with 50% utilization because, during the periods of higher priority, the compilation thread will wake and immediately preempt the executing application threads when methods arrive in the queue.

4.2 Impact on Pause Times

This section discusses the impact higher compilation utilization can have on application pause times. For many applications, such as batch applications, an increase in application pause times may not be significant, particularly if it results in a corresponding increase in performance, such as those described in the previous section.

However, minimizing application pauses is important for applications with hard or soft real-time constraints, as well as for interactive applications. While measuring pause times, it is important to cluster nearby pauses together because they can appear as a single pause to a user [16]. A more general metric used by the real time garbage collection community is *minimum mutator*[2] *utilization* (MMU) [8], which attempts to quantify the minimum amount

[2] Mutator is another term for application threads.

Window (ms)	Specified Compilation Thread Utilization										Round Robin
	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%	
100	77	56	51	45	46	31	24	14	6	0	1
200	82	67	57	53	48	35	27	18	8	1	24
300	84	72	62	55	49	37	28	19	10	1	29
400	85	75	64	57	49	38	28	19	12	7	36
500	86	76	66	57	49	39	29	21	14	10	37
600	87	77	66	58	49	39	30	22	16	13	40
700	88	78	67	59	49	40	31	24	18	15	41
800	88	78	67	59	50	40	32	24	19	16	43
900	88	78	68	59	50	40	32	24	20	17	44
1000	88	79	68	59	50	41	33	26	22	20	45

Table 4. Observed MMU (Minimum Mutator Utilization) for a range of time windows, for the aggressive controller

of utilization given to the application during some parameterized time window. For example, MMU(100 ms) is the minimum mutator utilization that occurs during *any* 100ms window of execution. A single pause of 100ms results in an MMU(100ms) of zero.

Table 4 reports the MMU for time windows ranging from 100ms to 1000ms (1 second). The overall MMU is calculated by taking the minimum MMU across the benchmark suite. Thus, the benchmark with the worst MMU determines the value reported.[3]

As expected, using a high compilation thread utilization lowers the MMU for short windows (100–300). Clearly, a VM with support for 100% compilation thread utilization cannot provide strong MMU guarantees. If pause times are relevant, then a lower utilization should be considered. In this work, we were mainly concerned with illustrating the performance impact of higher compilation thread utilization.

For soft real-time applications, choosing a lower utilization provides performance guarantees that cannot be provided by the round-robin scheduler. For example, enforcing a compilation thread utilization of 70% (or lower) achieves a better MMU than the round-robin scheduler for windows sizes 100–300. Thus, this technology could be applicable to a real-time virtual machine [4].

4.3 Trade

Figure 2 from Section 3 reported rampup performance of the Trade benchmark for 50% utilization and the round-robin scheduler. Figure 5 repeats this experiment with four additional utilization values: 10%, 20%, 70%, 100% on a single processor machine. These four utilizations best convey the performance trend while allowing the lines to be discernible; the missing utilizations fit within the trend and can be interpolated.

The results are as expected. With 10% compilation thread utilization, rampup time (the time to get to peak throughput) is poor, and is slightly worse than the round-robin scheduler, which was previously shown in Figure 2. Increasing the utilization slightly to 20% makes a surprisingly large improvement in rampup time. At the other extreme, 100% utilization maximized rampup time, but minimized throughput for the first 30 seconds of execution. A utilization of 70% provided near optimal rampup time, while also providing reasonable performance during the first 30 seconds.

[3] Four benchmarks (`pmd`, `antlr`, `luindex`, and `eclipse`) are excluded from the data presented in Table 4. These benchmarks incur application thread pauses independent of the compilation thread-scheduling mechanism, i.e., they are present with the round-robin scheduler, as well as any level of compilation thread utilization. These pauses could be caused by garbage collection, application I/O, or other system activity unrelated to the compilation thread. Because we report MMU by taking the minimum over all benchmarks, including these benchmarks causes several rows in the table to become zero, or near zero. A detailed pause time analysis of the VM is beyond the scope of this paper.

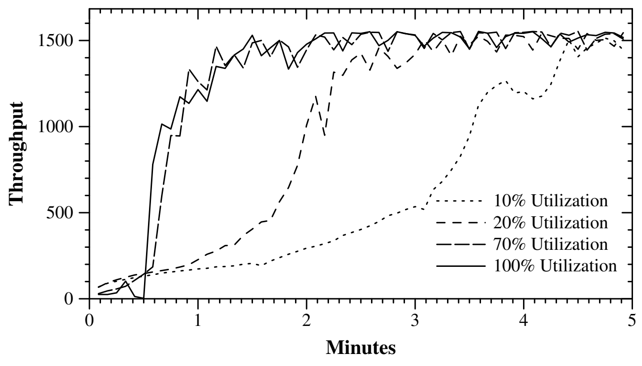


Figure 5. Trade for various utilizations on a single processor machine.

4.4 Hybrid Utilization Approaches

It is not required that the same utilization be used for all compilations. In fact, higher optimization levels are more likely to benefit from reduced utilization (50%, etc.) due to the higher cost and diminishing returns of higher opt-levels. For example, if a method is hot, it is important to perform the initial compilation quickly; there is a large performance difference between the interpreter and the level O0, so getting delayed in the interpreter is costly. In addition, compilations at O0 are quick, so it is more difficult to hurt performance by over-compiling. As a result, for low compilation levels, high utilization offers big rewards with low risk.

For high levels of optimization, the opposite is true: high utilization has higher risks and lower rewards. Compiler optimizations generally provide diminishing returns at higher optimization levels, so the penalty of delaying a compilation is smaller. They also have high compilation costs, so the penalty of a bad compilation decision is much more severe.

However, a simple scheme that assigns utilizations based on optimization level may still not be assessing risk properly. Consider a simple program that spends all of its time in a single, small method. Using a low utilization for higher levels (O1 and O2) will likely degrade performance for this program. Compiling the small method has relatively low cost, even at high optimization levels, so optimizing it will have a relatively large impact because all of the program execution occurs in this method.

A successful hybrid scheme needs to estimate the ratio of the following two quantities: 1) the cost of delaying the compilation, and 2) the cost of performing the compilation. If the first quantity is large relative to the second, then having a high compilation thread utilization is critical. If the reverse is true, high utilization is still likely to improve performance on average (assuming the controller is making good compilation decisions, as discussed in

Section 4.1), but the risk of high utilization increases, and using lower utilizations may improve performance, in practice.

An important concern when using multiple utilizations is a scenario where a low-utilization compilation in progress may delay future high-utilization compilations from starting. To avoid this problem, the system needs to either have multiple compilation threads (one for each priority) or employ some sort of preemption mechanism.

4.5 Implementing Utilization Without OS Support

As discussed in Section 3, thread priorities can provide a key building block for implementing a utilization-based scheduler. For a VM to rely on thread priorities, the priorities need to be accessible without special permissions, such as root access, and using the priorities in one process should not adversely impact other executing processes. Unfortunately, some operating systems, such as many versions of Linux, do not provide this functionality. As a result, relying on a priority-based scheduler to obtain an appropriate level of utilization is not a viable solution for these systems.

An alternate approach is for the VM to monitor how much CPU time each thread is receiving, and periodically yield any thread that is receiving more than its share. The only operating system requirement for this approach is to monitor CPU cycles consumed on a per-thread basis. Unfortunately, operating systems do not always support this functionality. For Linux there are kernel patches available [7] that provide access to thread-specific execution timings, but the functionality is not available in a standard version.

If attaining a precise utilization is not required, it is possible to boost the priority of the compilation thread by using heuristics to occasionally yield the application threads if the compilation queue is not empty (which implies that the compilation thread is either running, or is waiting to run). We implemented such an approach and observed speedups very similar to those measured with the priority-based scheduler (Tables 2 and 3). This heuristic approach cannot provide the strict utilization guarantees that can be achieved by using either priorities, or accurate thread-local timings, but it may be effective in practice for non-realtime applications.

5. Issues for Multiprocessor Machines

The preceding sections explained the necessity of enforcing a certain compiler utilization to ensure consistent performance for multithreaded applications, and maximum performance for single-threaded applications. Those experiments were conducted on a single-processor machine to best assess the performance tradeoffs caused by varying compiler utilizations. On a machine with more processors than application threads, all application threads and the compilation thread can run uninterrupted on different processors. The problems caused by reduced compiler utilization are no longer present because the threads are not preventing each other from executing. However, such multiprocessor environments raise a new issue of searching for the best compilation strategy to take advantage of the additional cycles that are available to the compiler to improve performance. The proliferation of multicore processors, and the inability of current applications to make use of all available computing resources, will only compound this problem in the future. This section explores this problem by evaluating the performance of controller policies on varying machine configurations, and show if (and how) the current policy should be changed to exploit free cycles.

5.1 Effect of Changing Compiler Aggressiveness on Performance

To the best of our knowledge, there has not been any formal study evaluating the changes needed in a controller policy tuned on

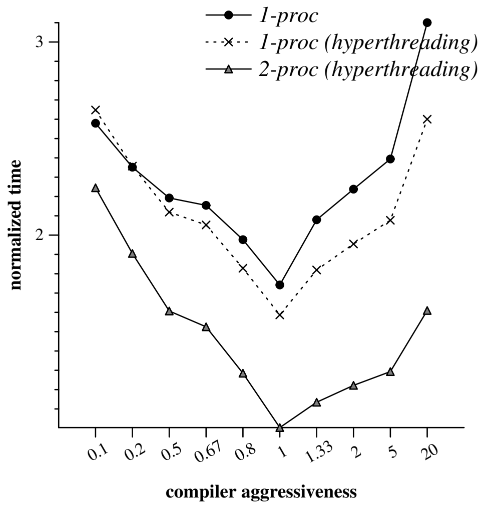


Figure 6. Effect of varying compiler aggressiveness on performance for different architectures.

single-processor machines to optimize performance on multiprocessor platforms. At the same time, it is a common perception that the controller could (and should) make more aggressive optimization decisions to make use of the available free cycles. Aggressiveness, in this context, can imply compiling early, or compiling at higher optimization levels.

To test this intuition, we evaluated the impact of controller aggressiveness on performance for three different processor configurations: single processor, single processor with hyperthreading, and two processors with hyperthreading. We modified the recompilation thresholds (i.e., the value that must be met before recompilation occurs) for the aggressive controller strategy described in Section 4.1 to make it more or less aggressive. Lower thresholds make the controller more aggressive because compilations occur earlier during the execution; higher thresholds make the controller less aggressive. Prevailing intuition suggests that different levels of controller aggressiveness will be optimal on different machine configurations because multiprocessor machines will benefit from more aggressive controllers.

Figure 6 shows the result of varying the controller aggressiveness for the three machines described above. The x-axis plots different controller aggressiveness factors, relative to the default controller; higher values indicate higher controller aggressiveness. Average performance over our benchmark suite is plotted on the y-axis. All performance numbers are normalized relative to the best average performance reached over all controller configurations. The figure shows that performance is improved slightly by enabling hyperthreading (compare 1 processor to 1 processor with hyperthreading), and performance is improved significantly by moving to a multiprocessor because compilation and application execution can now execute in parallel. However, all three plots show the same exact performance trend: the optimal recompilation threshold remaining constant regardless of the processor configuration. Thus, contrary to common intuition, our results suggest that a controller policy tuned for single processor architectures is likely to work unchanged on a multiprocessor.

This unintuitive result can be explained by recalling the discussion from Section 4 about the length of the compilation queue, and the delays that can occur. When a program starts executing there is often a steady stream of compilation activity. During this time period, the compiler cannot be treated as a "free" resource because it is busy compiling important methods. We call these compilations *primary compilations* because they would have been performed by a controller tuned for a single processor. Scheduling additional compilations during this period only increases the length of the compilation queue, and creates additional delay for the primary compilations, thus reducing performance.[4]

While the compilation thread has primary compilations to perform, running on a 2-processor machine is not much different than doubling the clock rate of the 1-processor machine. On the 2-processor machine the application and compilation thread are running on different processors, but they are still both making equal progress relative to each other. Doubling the performance of the 1-processor machine and performing round-robin scheduling would achieve the same result. There is no need to modify the compilation thresholds in either case.

However, when there are no primary compilations being performed, and there are idle processor cycles, it is feasible that performing additional compilations may improve performance, as is discussed in the next section.

5.2 Exploiting Free Cycles

We have seen that the controller policy regarding the primary compilations should not be changed when migrating from a single processor to a multiprocessor machine. However, there may be free cycles on the multiprocessor machines when there are no primary compilations scheduled. It is an interesting research problem to determine how these free cycles can be exploited to improve application performance. For this section, we assume the VM has only a single compilation thread; multiple compilation threads are discussed in Section 6.

We propose modifying the controller to perform *secondary* compilations, which are not scheduled unless a) there are no primary compilations to perform, and b) there are idle processor cycles available. A controller could implement secondary compilation by employing a second compilation queue, which we will refer to as the *secondary queue.* There can be several ways that methods could be selected to be placed in the secondary queue, but we have experimented only with one approach for this paper. In our approach, we put all methods removed from the primary queue by the compilation thread into the secondary queue to be compiled at higher optimization levels. Methods from the secondary queue are sorted based on profile information, and scheduled for compilation only if the primary queue is empty. If a primary compile is scheduled midway between a currently executing secondary compile, then we provide an option to preempt the secondary compile, so as to not affect the normal progress of primary compilations.

Surprisingly, on our system, secondary compiles were not able to achieve any significant performance benefit on average for both the aggressive and conservative controllers described in Section 4. We observed that the important methods that are needed for high performance are detected quickly by the profiling mechanisms and compiled at a higher optimization level, even without performing secondary compilations. Using the secondary queue causes a larger number of (less important) methods to be upgraded, but has a small overall impact on performance.

Thus, for multiprocessor machines, there seems to be little incentive to change the aggressiveness of the controller if the VM has only one compilation thread.

[4] To avoid these delays, we tried using more sophisticated sorting heuristics for the compilation queue. This slightly reduced the degradation that occurs as the controller becomes more aggressive, but did not improve the performance at, or near, the optimal tuning point.

5.3 Identifying Free Cycles

As explained earlier, we propose to use secondary compilations only if there are *free* cycles available, and there are no primary methods to compile. However, detecting the availability of free cycles is a nontrivial task. The presence of a free processor may seem obvious if the number of processors exceeds the number of application threads, but on many hyperthreaded architectures the OS reports multiple *logical* processors as distinct CPUs. Using a hyperthreaded core to execute secondary compiles may in fact degrade overall performance by impeding the normal progress of the primary application and compilation threads.

An instruction, such as the *cpuid* instruction on x86 processors [1], can be used in certain cases to distinguish between logical and actual multiprocessors. This type of support from the hardware and OS is valuable in certain instances for helping the VM make informed decisions, but even with this support, partial sharing of resources makes it difficult to speculate whether one processor can be used without reducing performance of the other processor. Even with multicore processors there is generally some sharing of resources between cores, such as the sharing of higher levels of memory hierarchy. Such resource sharing is likely to increase in the future, making the problem of identifying free cycles even more difficult.

6. Multiple Compilation Threads

Previous sections considered virtual machines with support for only a single compilation thread, which is common in most JVMs available today. Some VMs, such as the Azul VM, implement multiple compilation threads to allow compilation to be performed in parallel. The Azul VM is targeted for highly parallel architectures and spawns on the order of 50 compilation threads during the rampup period of large programs, such as application servers [10].

Using multiple compilation threads is clearly an effective solution, and is the only way to exploit the additional resources available on multicore architectures. If the number of processors is large relative to the number of application threads, then no thread is ever waiting on other threads, so the utilization issues discussed earlier are no longer relevant.

However, as long as there are more application threads than processors, all of the same basic concepts from the previous section still apply. Achieving peak performance requires devoting the proper amount of resources to compilation threads. Utilization can now be controlled by creating (or destroying) compilation threads, but only to a certain degree. For example, consider a program with 16 application threads running on a 4-processor machine. Even if 4 compilation threads are created, only 20% of the total CPU resources are dedicated to compilation (executing round-robin with 4 other application threads on a processor). It is unclear whether a sufficient compiler utilization can be attained in practice by creating additional compilation threads, or whether there will remain a need to enforce a utilization for each processor.

7. Related Work

The Self-93 VM [16] pioneered many of the adaptive optimization techniques employed in today's virtual machines. Self focused particularly on keeping application pause times to a minimum. It initially compiled each method with a fast nonoptimizing compiler, and identified hot methods using method invocation counts, which decayed over time. Compilation appears to have occurred in the application's thread, so none of the issues examined in this paper were explored.

Harris [15] describes a technique for controlling the worst-case pauses of a Java runtime compiler on the Nemesis operation system [22]. Compilation occurs on a separate compilation thread, and its CPU allocation can be varied by using features of the Nemesis operating system. Harris shows the results running the CaffeineMark benchmark with compiler utilization levels of 5, 30 and 50%, where the JVM as a whole is given 70% of the CPU. Our work differs from this work in that we provide a more comprehensive study of utilization in a production JVM over a large benchmark suite on a standard operating system.

Krintz et al. [19] explore background compilation on a separate Java thread in the Jalapeño VM, an earlier name for Jikes RVM, with the goal of reducing the overhead of dynamic compilation. The compilation thread is provided with a list of methods to compile from a previous run of the application. The work is motivated by using an extra processor for this compilation. They do not explore issues of utilization for the compilation thread nor issues of single processor vs. multiple processors.

The rest of this section summarizes other popular VMs. No publications describing these systems have discussed the issue of compilation thread utilization.

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 whether it performs recompilation on a separate thread or simply suspends the application thread to perform a recompilation. However, most of its design was inspired by the Self-93 VM [16].

BEA's JRockit [5] VM uses a compile-only approach to program execution. In such systems, the code produced by a fast nonoptimizing compiler is used for the method's initial executions. This compilation occurs in the application's thread. Recompilation is driven by a sampler thread that finds methods to optimize. This thread suspends the applications threads and takes a sample at regular intervals. Although full details are not publicly available, recompilation appears to be performed on a separate thread from the application [13].

The IBM DK for Java [26] initially interprets methods and uses method entry and back edge counters to find interpreted methods to optimize. A sampling-based profiler is used to potentially compile methods at higher levels of optimizations. All compiled occurs in a separate compilation thread.

IBM's J9 VM [20] is similar to the IBM DK for Java in that it is also interpreter-based with multiple optimization levels. It uses counters and a sampling thread to periodically sample the application. Compilation is performed on a separate thread. As described in Section 2, J9 uses thread priorities on some platforms to increase the priority of the compilation thread.

Jikes RVM [2] uses a compile-only approach with multiple recompilation levels. It uses a cost/benefit model to determine which methods should be recompiled and at what optimization level. Initial compilation is performed on the application's thread. Three separate Java threads are used for profiling, decision-making, and recompilation.

Intel's ORP VM [9] employs a compile-only strategy with one level of recompilation. A method is recompiled when its counter passes a threshold or when a separate thread finds a method with a counter value that suggests compiling it in the background.

The Azul VM is derived from the HotSpot VM to run on Azul's multicore hardware. It uses multiple compilation threads for compilation [10].

8. Conclusion

Most modern virtual machines perform compilation on separate threads with the goal of not interfering with the application thread's progress. This paper demonstrated that the conventional wisdom of treating the compilation as a background, low-priority activity can lead to suboptimal performance. We showed that guaranteeing a certain level of utilization for the compilation thread is necessary for robust performance of multithreaded applications, and evaluated the performance over a range of utilizations. Higher utilizations resulted in average speedups of 9% and 18% for two different compilation strategies over the default thread scheduler. In addition, we showed that no real changes are necessary to the compilation strategies when moving to a multiprocessor machine.

This work has demonstrated that significant performance gains can be achieved on a production virtual machine without adding any optimizations to the compiler. This illustrates that in a dynamic compilation setting, making good decisions about when and what to compile (decisions the controller makes) can be as important as how the code is compiled (decisions the dynamic compiler makes).

Acknowledgments

The authors thank the J9 JIT team for their assistance with this work, Martin Hirzel for his help with PAPI, and Laureen Treacy for proofreading earlier drafts of this work.

References

[1] J. Andrews. Detecting hyper-threading technology and dual cores. http://www.developers.net/intelisnshowcase/view/579.

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

[3] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2), 2005. Special issue on Program Generation, Optimization, and Adaptation.

[4] D. F. Bacon, P. Cheng, D. Grove, M. Hind, V. T. Rajan, E. Yahav, M. Hauswirth, C. M. Kirsch, D. Spoonhower, and M. T. Vechev. High-level real-time programming in Java. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 68–78, New York, NY, USA, 2005. ACM Press.

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

[6] 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 *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming Languages, Systems, and Applications*, pages 169–190, New York, NY, USA, 2006. ACM Press.

[7] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, Dallas, TX, Nov. 2000.

[8] P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. *ACM SIGPLAN Notices*, 36(5):125–136, May 2001. In *Conference on Programming Language Design and Implementation (PLDI)*.

[9] 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.

[10] C. Click. The Azul VM, Azul systems. personal communication.

[11] http://www-plan.cs.colorado.edu/henkel/projects/colorado_bench.

[12] The Daikon dynamic invariant detector. http://pag.csail.mit.edu/daikon.

[13] 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.

[14] N. Grcevski, A. Kilstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java just-in-time compiler and virtual machine improvements for

server and middleware applications. In *3rd Virtual Machine Research and Technology Symposium (VM)*, May 2004.

[15] T. Harris. Controlling run-time compilation. In *IEEE Workshop on Programming Languages for Real-Time Industrial Applications*, pages 75–84, Dec. 1998.

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

[17] https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=trade6.

[18] Kawa, the Java-based Scheme system. http://www.gnu.org/software/kawa.

[19] C. J. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Software—Practice and Experience*, 31(8):717–738, July 2001.

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

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

[22] T. Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, University of Cambridge Computer Laboratory, April 1995. Available as Technical Report No. 376.

[23] http://www.sable.mcgill.ca/software/#soot.

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

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

[26] 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*, 27(4):732–785, July 2005.

[27] http://xml.apache.org/xerces2-j/index.html.