JIT Compilation Policy for Modern Machines

Prasad A. Kulkarni
Department of Electrical Engineering and Computer Science, University of Kansas
prasadk@ku.edu

Abstract

Dynamic or Just-in-Time (JIT) compilation is crucial to achieve acceptable performance for applications (written in managed languages, such as Java and C#) distributed as intermediate language binary codes for a *virtual machine* (VM) architecture. Since it occurs at runtime, JIT compilation needs to carefully tune its compilation policy to make effective decisions regarding *if* and *when* to compile different program regions to achieve the best overall program performance. Past research has extensively tuned JIT compilation policies, but mainly for VMs with a single compiler thread and for execution on single-processor machines.

This work is driven by the need to explore the most effective JIT compilation strategies in their modern operational environment, where (a) processors have evolved from single to multi/many cores, and (b) VMs provide support for multiple concurrent compiler threads. Our results confirm that changing *if* and *when* methods are compiled have significant performance impacts. We construct several novel configurations in the HotSpot JVM to facilitate this study. The new configurations are necessitated by modern Java benchmarks that impede traditional static whole-program discovery, analysis and annotation, and are required for simulating future many-core hardware that is not yet widely available. We study the effects on performance of increasing compiler aggressiveness for VMs with multiple compiler threads running on existing single/multi-core and future many-core machines. Our results indicate that although more aggressive JIT compilation policies show no benefits on single-core machines, these can often improve program performance for multi/many-core machines. However, accurately prioritizing JIT method compilations is crucial to realize such benefits.

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

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

General Terms Languages, Performance

Keywords virtual machines, dynamic compilation, multi-core, Java

1. Introduction

Managed languages such as Java [12] and C# [28] support the 'compile-once, run-anywhere' model for code generation and distribution. This model allows the generation of programs that can be portably distributed and executed on any device equipped with the corresponding virtual machine (VM). The portability constraint limits the format of the distributed program to a form that is independent of any specific processor architecture. Since the program binary format does not match the native architecture, VMs have to employ either interpretation or dynamic compilation before executing the program. However, interpreted execution is inherently slow, which makes dynamic or Just-in-Time (JIT) compilation essential to achieve efficient runtime performance for such applications.

By operating at runtime, JIT compilation contributes to the overall execution time of the application and, if performed injudiciously, may result in further worsening the execution or *response time* of the program. Therefore, JIT compilation policies need to carefully tune *if* and *when* different program regions are compiled to achieve the best program performance. In addition to if and when, *how* to compile program regions is also an important component of any compilation policy. However, in contrast to the previous two components, the issue of how to compile program regions is not unique to dynamic compilation, as can be attested by the presence of multiple optimization levels in GCC, and the wide body of research in profile-driven compilation [9, 13] and optimization phase ordering/selection [16, 35] for static compilers. Also, the default OpenJDK HotSpot VM used in our experiments only supports a single compilation level. Consequently, we do not consider the issue of *how to compile* any further in this work.

The technique of *selective compilation* was invented by researchers to address the issues of *if* and *when* to com-

pile program methods during dynamic compilation [4, 19, 25, 30]. However, research on JIT compilation policies employing the above theories have primarily been conducted on single-processor machines and for VMs with a single compiler thread. As a result, existing JIT compilation policies that attempt to improve program efficiency while minimizing application pause times and interference are typically quite conservative.

Recent years have witnessed a major paradigm shift in microprocessor design from high-clock frequency single-core machines to processors that now integrate multiple cores on a single chip. Moreover, hardware researchers and processor manufacturers expect to continuously scale the number of cores available in future processor generations [1]. Thus, modern architectures allow the possibility of running the compiler thread(s) on a separate core(s) to minimize interference with the application thread. Virtual machine developers are also responding to this change in their hardware environment by making the compiler thread-safe, and allowing the user to simultaneously initiate multiple concurrent compiler threads. Such evolution in the hardware and VM contexts may demand radically different JIT compilation policies to achieve the most effective overall program performance.

Consequently, the objective of this research is to investigate and develop new JIT compilation strategies to realize the best performance on existing single/multi-core processors and future many-core machines for VMs with multiple compiler threads. Unfortunately, experimental constraints make it difficult to readily achieve this objective. For example, one constraint is imposed by modern Java benchmarks that impede static whole program discovery and analysis. Also, the commonly heralded many-core machines are not widely available just yet. We overcome such constraints by designing and constructing novel VM experimental configurations to conduct this work. We induce continuous progressive increases in the aggressiveness of JIT compilation strategies, as well as in the number of concurrent compiler threads and analyze their effect on average program performance. Thus, the major contributions of this research work are the following:

1. We present the novel VM configurations we develop to overcome the constraints imposed by modern Java benchmarks and unavailable many-core hardware during our exploration of effective JIT compilation policies.
2. We quantify the impact of altering 'if' and 'when' methods are compiled on application performance.
3. We demonstrate the effect of multiple compiler threads on average program performance for single-core machines.
4. We explain the impact of different JIT compilation strategies on available multi-core and future many-core machines.
5. We identify and show the benefit of prioritizing method compiles on program performance with different JIT compilation policies for modern hardware.

The rest of the paper is organized as follows. In the next section, we present background information and related work regarding existing JIT compilation policies. We describe our benchmark suite and general experimental setup in Section 3. In Section 4, we validate the impact of varying 'if' and 'when' methods are compiled on program performance. Our experiments exploring different JIT compilation strategies for VMs with multiple compiler threads on single-core machines are described in Section 5. In Section 6, we present results that explore the most effective JIT policies for multi-core machines. We describe the results of our novel experimental configuration to study compilation policies for future many-core machines in Section 7. We explain the impact of prioritizing method compiles in Section 8. Finally, we describe avenues for future work and present our conclusions from this study in Sections 9 and 10 respectively.

2. Background and Related Work

Several researchers have explored the effects of conducting compilation at runtime on overall program performance and application pause times. The ParcPlace Smalltalk VM [10] followed by the Self-93 VM [19] pioneered many of the adaptive optimization techniques employed in current virtual machines, including selective compilation with multiple compiler threads on single-core machines. For such machines, the total program run-time includes the application run-time as well as the compilation time. Therefore, aggressive compilations have the potential of degrading program performance by increasing the compilation time. The technique of selective compilation was invented by researchers to address this issue with dynamic compilation [4, 19, 25, 30]. This technique is based on the observation that most applications spend a large majority of their execution time in a small portion of the code [4, 8, 21]. Selective compilation uses online profiling to detect this subset of *hot* methods to compile at program startup, and thus limits the overhead of JIT compilation while still deriving the most performance benefit at runtime. Most current VMs employ selective compilation with a *staged* emulation model [17]. With this model, each method is initially interpreted or compiled with a fast non-optimizing compiler at program start to improve application response time. Later, the virtual machine attempts to determine the subset of hot methods to selectively compile, and then compiles them at higher levels of optimization to achieve better program performance.

Unfortunately, selecting the hot methods to compile requires *future* program execution information, which is hard to accurately predict [29]. In the absence of any better strategy, most existing JIT compilers employ a simple prediction model that estimates that frequently executed *current* hot methods will also remain hot in the future [2, 14, 22].

Online profiling is used to detect these current hot methods. The most popular online profiling approaches are based on instrumentation *counters* [17, 19, 22], interrupt-timer-based *sampling* [2], or a combination of the two methods [14]. Profiling using counters requires the virtual machine to count the number of invocations and loop back-edges for each method. Sampling is used to periodically interrupt the application execution and update a counter for the method(s) on top of the stack. The method/loop is sent for compilation if the respective method counters exceed a fixed threshold.

Finding the correct threshold value for each compilation stage is crucial to achieve good startup performance for applications running in a virtual machine. Setting a higher than ideal compilation threshold may cause the virtual machine to be too conservative in sending methods for compilation, reducing program performance by denying hot methods a chance for optimization. In contrast, a compiler with a very low compilation threshold may compile too many methods, increasing compilation overhead. High compilation overhead may negatively impact overall program performance on single-core machines. Therefore, most performance-aware JIT compilers experiment with many different threshold values for each compiler stage to determine the one that achieves the best performance over a large benchmark suite.

The theoretical basis for tuning compiler thresholds is provided by the *ski-renting* principle [11, 20], which states that to minimize the worst-case damage of online compilation, a method should only be compiled after it has been interpreted a sufficient number of times so as to already offset the compilation overhead [29]. By this principle, a (slower) compiler with more/better optimization phases will require a higher compilation threshold to achieve the best overall program performance in a virtual machine.

Resource constraints force existing JIT compilation policies to make several tradeoffs regarding which methods are compiled/optimized at what stage of program execution. Thus, selective compilation is employed to limit the total time spent by the *compiler thread* at the cost of potentially lower *application thread* performance. Additionally, online profiling (used to select hot methods to compile) causes delays in making the compilation decisions at program startup. The first component of this delay is caused by the VM waiting for the method counters to reach the compilation *threshold* before deeming the method as hot and *queuing* it for compilation. The second factor contributing to the compilation delay occurs as each compilation request waits in the compiler queue to be serviced by a free compiler thread. Restricting method compiles and the delay in optimizing hot methods results in poor application startup performance as the program spends more time executing in unoptimized code [15, 23, 26].

Researchers have suggested strategies to address the first delay component for online profiling. Krintz and Calder employ offline profiling and classfile annotation to send hot methods to compile early [23, 24]. However, such mechanisms require an additional profiling pass, and are therefore not generally applicable. Namjoshi and Kulkarni propose a technique that can dynamically determine loop iteration bounds to *predict* future hot methods and send them to compile earlier [29]. Their suggested implementation requires additional computational resources to run their more expensive profiling stage. Gu and Verbruggee use online phase detection to more accurately estimate recompilation levels for different hot methods to save redundant compilation overheads and produce better code faster [15].

Researchers have also explored techniques to address the second component of the compilation delay that happens due to the backup and wait time in the method compilation queue. IBM's J9 virtual machine uses thread priorities to increase the priority of the compiler thread on operating systems, such as AIX and Windows, that provide support for user-level thread priorities [33]. Another technique attempts to increase the CPU utilization for the compiler thread to provide faster service to the queued compilation requests [18, 26]. However, the proposed thread-priority based implementations for these approaches can be difficult to provide in all existing operating systems. Jikes RVM provides a priority-queue implementation to reduce the delay for the *hotter* methods, but this study only evaluates their one strategy on single-core machines [3].

Most of the studies described above have been targeted for single-core machines. There exist few explorations of JIT compilation issues for multi-core machines. Krintz et al. investigated the impact of background compilation in a separate thread to reduce the overhead of dynamic compilation [25]. This technique uses a single compiler thread and employs offline profiling to determine and prioritize hot methods to compile. Kulkarni et al. briefly discuss performing parallel JIT compilation with multiple compiler threads on multi-core machines, but do not provide any experimental results [26]. Existing JVMs, such as Sun's HotSpot server VM [30] and the Azul VM (derived from HotSpot), support multiple compiler threads, but do not present any discussions on ideal compilation strategies for multi-core machines. Simultaneous work by Böhm et al. explores the issue of parallel JIT compilation with a priority queue based dynamic work scheduling strategy in the context of their dynamic binary translator [7]. Our earlier workshop publication explores the impact of varying the aggressiveness of dynamic compilation on modern machines for JVMs with multiple compiler threads [27]. This paper extends our earlier work by providing more comprehensive results, investigating the impact of aggressive JIT compilation schemes for existing multi-core machines, which also validates the accuracy of the simulation environment we develop for future many-core machines, and exploring the issues and impact of different priority-based compiler queue strategies for aggressive JIT compilation on modern many-core machines.

SPECjvm98		SPECjvm2008		DaCapo-9.12-bach	
Name	#Methods	Name	#Methods	Name	#Methods
_201_compress_100	517	compiler.compiler	3195	avrora_default	1849
_201_compress_10	514	compiler.sunflow	3082	avrora_small	1844
_202_jess_100	778	compress	960	batik_default	4366
_202_jess_10	759	crypto.aes	1186	batik_small	3747
_205_raytrace_100	657	crypto.rsa	960	eclipse_default	11145
_205_raytrace_10	639	crypto.signverify	1042	eclipse_small	5461
_209_db_100	512	mpegaudio	959	fop_default	4245
_209_db_10	515	scimark.fft.small	859	fop_small	4601
_213_javac_100	1239	scimark.lu.small	735	h2_default	2154
_213_javac_10	1211	scimark.monte_carlo	707	h2_small	2142
_222_mpegaudio_100	659	scimark.sor.small	715	jython_default	3547
_222_mpegaudio_10	674	scimark.sparse.small	717	jython_small	2070
_227_mtrt_100	658	serial	1121	luindex_default	1689
_227_mtrt_10	666	sunflow	2015	luindex_small	1425
_228_jack_100	736	xml.transform	2592	lusearch_default	1192
_228_jack_10	734	xml.validation	1794	lusearch_small	1303
				pmd_default	3881
				pmd_small	3058
				sunflow_default	1874
				sunflow_small	1826
				tomcat_default	9286
				tomcat_small	9189
				xalan_default	2296
				xalan_small	2277

Table 1. Benchmarks used in our experiments

3. Experimental Framework

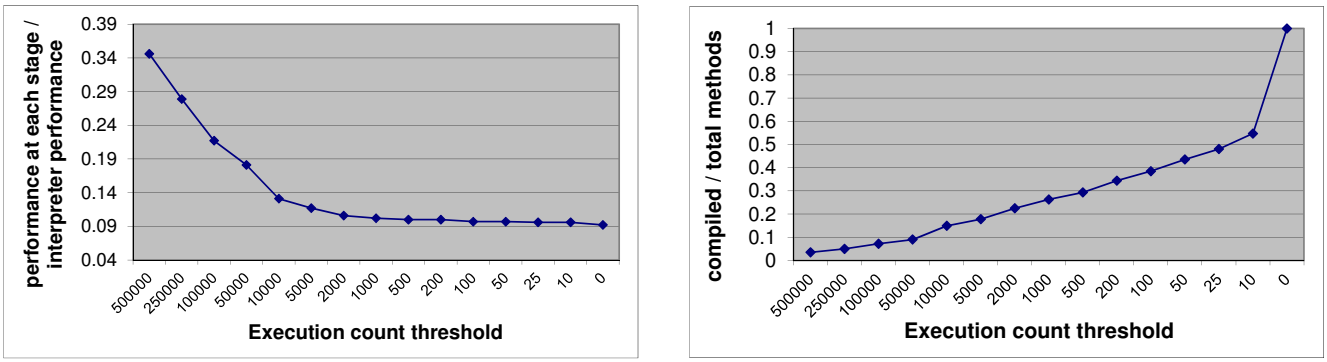
The research presented in this paper is performed using the server version of the Sun/Oracle's HotSpot java virtual machines (build 1.7.0-ea-b24) [30]. The latest development code for the HotSpot VM is available through Sun's OpenJDK initiative. The HotSpot VM uses interpretation at the start of program execution. It then employs a counter-based profiling mechanism, and uses the sum of a method's *invocation* and loop *back-edge* counters to detect and promote hot methods for compilation. We call the sum of these counters as the *execution count* of the method. Methods/loops are determined to be hot if the corresponding method execution count exceeds a fixed threshold. The tasks of detecting hot methods and dispatching them for compilation are performed at every method call (for whole-method compiles) and loop iteration (for on-stack-replacement compiles). The HotSpot server VM allows the creation of an arbitrary number of compiler threads, as specified on the command-line.

The experiments in this paper were conducted using all the benchmarks from three different benchmark suites, SPEC jvm98 [32], SPEC jvm2008 (startup) [31] and DaCapo-9.12-bach [5]. We employ two inputs (10 and 100) for benchmarks in the SPECjvm98 suite, two inputs (small and default) for the DaCapo benchmarks, and a single input (startup) for benchmarks in the SPECjvm2008 suite, resulting in 56 benchmark/input pairs. Two benchmarks from the DaCapo benchmark suite, *tradebeans* and *tradesoap*, did not always run correctly with the *default* version of the HotSpot VM, so these benchmarks were excluded from our set. Table 1 lists the name and the number of invoked methods for each benchmark in our suite.

All our experiments were performed on a cluster of 8-core Intel Xeon 2.833GHz processors. All machines use Fedora Linux as the operating system. We disable seven of the eight available cores (including hyperthreading) to run our single-core experiments. Our multi-core experiments utilize all available cores (without hyperthreading). More specific variations made to the hardware configuration are explained in the respective sections. Each benchmark is run in isolation to prevent interference from other user programs. Finally, to account for inherent timing variations during the benchmark runs, all the performance results in this paper report the average over 10 runs for each benchmark-configuration pair.

4. Tradeoffs of If and When to Compile

Existing JIT compilation policies tuned for single-core machines limit the number of methods compiled to reduce the time spent doing compilations, while achieving the best overall program performance. Additionally, the process of finding the set of hot methods to compile and the time spent by such methods in the compiler queue due to possible queue back-ups further delays compilations. Thus, although com-



(a) (b)

Figure 1. Understanding the effect of (*if* to perform) JIT compilation on program performance.

piling/optimizing all program methods at their earliest opportunity can likely allow more efficient *application thread* execution, existing JIT compilation policies bound by resource constraints cannot achieve this ideal. The recent and future availability of more abundant computing resources can enable more aggressive JIT compilation policies and improve application performance. However, before exploring new policies, in this section we first attempt to quantify the potential benefit of compiling *more* program methods *early*.

We develop a unique VM framework to conduct our experiments in this section. This framework provides two complementary capabilities. From the VM's point of view, our framework enables the VM to efficiently detect important program points as they are reached during execution. At the same time, from the executing program's point of view, it allows the program running within the virtual machine to call-on the enclosing VM to perform certain tasks at specific points during execution. Our framework employs Soot [34] to annotate specific user-defined program points statically. Currently, we only allow annotations at the level of individual methods. Our updated VM is able to detect these annotations on every such method invocation and perform the desired actions. Of course, we also need to implement the support to perform these actions in our modified VM. We found this capability to be extremely useful in several studies throughout this work.

4.1 Benefit of Dynamic Compilation (*If* to Compile?)

Executing native code (produced by compilation/optimization in a JVM) has been observed in earlier works to be much more efficient than interpreted execution. Therefore, increasing the fraction of compiled code is likely to result in more efficient program execution. However, not all methods contribute equally to performance improvement. Selective compilation in existing VMs exploits this observation to compile the most frequently executed program sections that are expected to produce the most performance benefit. Thus, any additional compilation may only produce diminishing returns. In this section we attempt to quantify the benefit of more aggressive compilation that may be enabled by modern machines. We accomplish this goal by studying the effect of varying the selective compilation threshold on *steady-state* program performance. The VM compiles all methods with execution counts that exceed the selected threshold. The *harness* of all our benchmark suites allows each benchmark to be *iterated* multiple times in the same VM run. We disable background compilation to force all hot methods to be compiled in the first iteration itself. The execution time of the fifth benchmark iteration is measured as its steady-state time. Thus, the steady-state configuration ignores the compilation overhead allowing us to explore the best-case scenario where the presence of abundant hardware resources causes compilations to be *effectively* free. We are not aware of any previous study to investigate the goals as stated here.

Our experimental configuration conducts an initial *offline* run to collect method execution counts for every benchmark in our set. For each benchmark, the following measurement run executes the program over several *stages*, with a few program *iterations* per stage, in a single VM run. Each successive stage in our modified HotSpot VM lowers the compilation threshold and compiles additional methods (over the previous stage) with offline execution counts that exceed the new lower threshold. We employ our new VM capability described earlier to enable the VM to detect the start of every benchmark iteration during the same VM run. The VM uses this detection to determine the end of the current stage, collect steady-state performance numbers for that stage, and then release the additional program methods for compilation to start the next stage. Thus, the first stage compiles no methods, and all methods are compiled by the final stage in a single VM run. Each intermediate stage compiles successively more program methods. Each stage consists of sufficient number of program iterations to compile all released methods. The final iteration in each stage performs no compilations and provides a measure of the benchmark performance at the end of that stage.

Figure 1(a) shows the average improvement in program performance compared to interpreted execution averaged

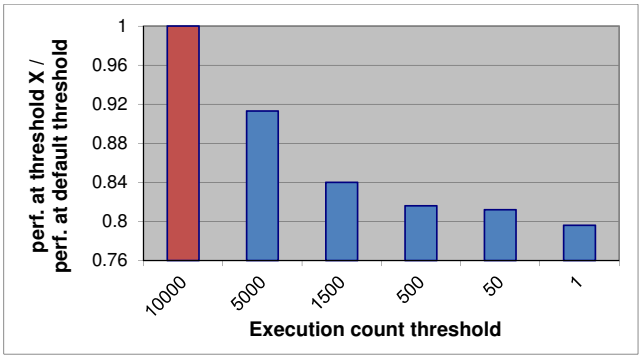


Figure 2. Performance benefit of compiling the *same* set of hot methods early. The default threshold is 10,000.

over all our benchmarks. The X-axis indicates the compile threshold used at each stage. At every stage, methods that have an offline execution count greater than the stage compile threshold are sent for compilation. Figure 1(b) shows the percentage of methods compiled at each stage, averaged over all 56 benchmark-input pairs in our set. Thus, we can see that JIT compilation of *all* program methods achieves a dramatic performance improvement over interpretation, achieving program execution in about 9% of the interpretation time, on average. As expected, most of the performance gain is obtained by compiling a very small fraction of the total executed methods. Indeed, the default HotSpot VM selective compilation threshold of 10,000 results in compiling about 15% of the methods and reduces execution time to around 13% of interpretation time. Moreover, it is important to note that although the added performance improvement of compiling *all* program methods does not seem significant compared to interpretation time, it results in over 30% more efficient program execution compared to the default VM configuration (with threshold 10,000) as the baseline.

4.2 Benefit of Early Compilation (*When* to Compile?)

Several researchers have made the observation that compiling methods early improves program efficiency since the program execution spends less time in interpreted or unoptimized code [15, 29]. In this section we report the benefit in average application performance for our benchmark programs by compiling the *hot* methods early. Our results in this section confirm the observations made by other researchers regarding the benefits of early compilation (with different/fewer benchmarks), and are presented here for completeness.

For these experiments we employ an offline profiling run to first determine the set of *hot* methods that are compiled by the default VM configuration (threshold 10,000). Our setup compiles the *same* methods early by initiating their compilations at lower method execution counts. Past studies to measure early compilation benefit used static method-level annotations to indicate the hot methods to the VM for compilation [23, 29]. However, the issues of reflection and runtime generation of classes make it difficult to statically discover and annotate all hot methods in newer Java benchmarks [6]. Therefore, we again employ our new VM capability for these experiments. Each VM run invokes two benchmark iterations. The first iteration does not perform any compilations, but is only used to discover and load all program methods. The VM detects the end of the first iteration and marks the set of hot methods at this point before initiating the next iteration. Measuring the application time of the second iteration provides the actual program performance.

Figure 2 illustrates the benefits of early compilation on program performance, averaged over all our 56 benchmark-input combinations. We find that early compilation of hot methods can improve performance by over 20% for our set of benchmarks. Thus, our results in this section show that the proper selection of 'if' and 'when' to compile program methods can have a significant influence on performance.

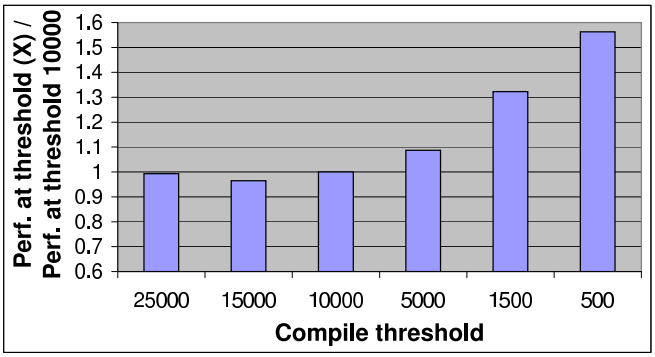
5. JIT compilation on Single-Core Machines

In this section we first explore the selective compilation threshold that achieves the best average performance with our set of benchmarks for a VM with one compiler thread executing on a single-core machine. This threshold serves as the baseline for the remaining experiments in this paper. Additionally, several modern VMs are now equipped with the ability to spawn multiple simultaneous compiler threads. Our second study in this section evaluates the impact of multiple compiler threads on program performance for machines with a single processor.

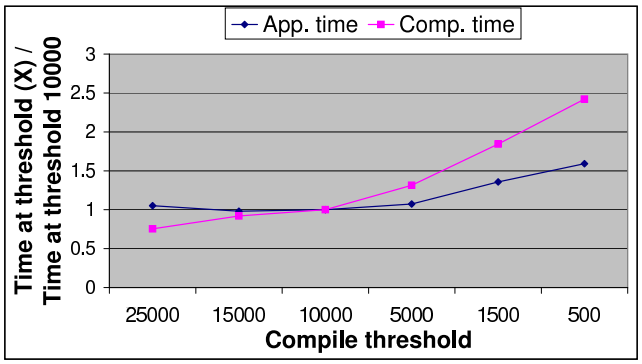
5.1 Compilation Threshold with Single Compiler Thread

By virtue of sharing the same computation resources, the application and compiler threads share a complex relationship for a VM running on a single-core machine. Thus, a high selective compile threshold may achieve poor overall program performance by spending too much time executing in non-optimized code resulting in poor application thread time. By contrast, a lower than ideal compile threshold may also produce poor performance by spending too long in the compiler thread. Therefore, the compiler thresholds need to be carefully tuned to achieve the most efficient average program execution on single-core machines over several benchmarks.

VM developers often experiment with several different compile threshold values to find the one that achieves the best overall program performance for their set of benchmark programs. We perform a similar experiment to determine the ideal compilation threshold with a *single* compiler thread on our set of benchmarks. These results are presented in Figure 3(a), which plots the ratio of the average overall program performance at different compile thresholds compared to the average program performance at the threshold of 10,000, which is the default compilation threshold for the HotSpot server VM. Not surprisingly, we can see this default



(a)



(b)

Figure 3. Effect of different compilation thresholds on average benchmark performance on single-core processors

threshold performs very well on our set of benchmarks, but a slightly higher compile threshold of 15,000 achieves the best overall performance for our benchmark set.

It is also interesting to note that performance worsens at both high and low compile thresholds. In order to better interpret these results, we plot the graph in Figure 3(b) that shows the break-down of the overall program execution time in terms of the ratios of the application and compiler thread times at different thresholds to their respective times at the compile threshold of 10,000, averaged over all benchmark programs. Thus, we can see that high thresholds ($>$ 15,000) compile less and degrade performance by not providing an opportunity to the VM to compile several important program methods. In contrast, the compiler thread times increase with lower compilation thresholds ($<$ 15,000) as more methods are sent for compilation. We expected this increased compilation to improve application thread performance. However, the behavior of the application thread times at low compile thresholds is less intuitive. On further analysis we found that although JIT compilation policies with lower thresholds send more methods to compile, this increase also grows the length of the compiler queue. The flood of less important program methods delays the compilation of the most critical methods, resulting in the non-intuitive degradation in application thread performance observed in Figure 3(b) for the lower thresholds. Due to its superior performance, we select the compile threshold of 15,000 as the baseline for our remaining experiments in this paper.

5.2 Effect of Multiple Compiler Threads on Single-Core Machines

To the best of our knowledge, the effect of multiple compiler threads on overall program performance on a single-core machine has never been previously discussed. In this section we conduct such a study and present our observations.

For each compiler threshold, a separate plot in Figure 4(a) compares the average overall program performance with multiple compiler threads to the average performance with a single compiler thread at that same threshold. Intuitively, a greater number of compiler threads should be able to reduce the method compilation queue delay, which is the time spent between sending a method to compile and generating optimized code. Indeed, we notice program performance improvements for small number of compiler threads (2–4), but the benefits do not seem to hold with increasing number of such threads ($>$4). We further analyzed the performance degradations with more compiler threads and noticed an increase in the overall *compiler thread* times in these cases. This increase suggests that several methods that were queued for compilation, but never got compiled before program termination with a single compiler thread are now compiled as we provide more resources to the VM compiler component. Unfortunately, many of these methods contribute little to improving program performance. At the same time, the increased compiler activity increases compilation overhead. Consequently, the potential improvement in application performance achieved by more compilations seems unable to recover the additional time spent by the compiler thread, resulting in a net loss in overall program performance.

Figure 4(b) compares the average overall program performance in each case to the average performance of a baseline configuration with a single compiler thread at a threshold of 15,000. Remember, that the baseline configuration used is the one that achieves the best average performance with a single compiler thread. These results reveal the best compiler policy on single-core machines with multiple compiler threads. Thus, we can see that the more aggressive thresholds perform quite poorly in relation to our selected baseline (with any number of compiler threads). Our analysis finds higher compiler aggressiveness to send more program methods for compilation, which includes methods that may not make substantial contributions to performance improvement (*cold* methods). Additionally, the default HotSpot VM uses a simple FIFO (first-in first-out) compilation queue, and compiles methods in the same order in which they are sent. Consequently, the cold methods delay the compilation of the really important hot methods relative to the application thread, producing the resultant loss in performance. Thus,

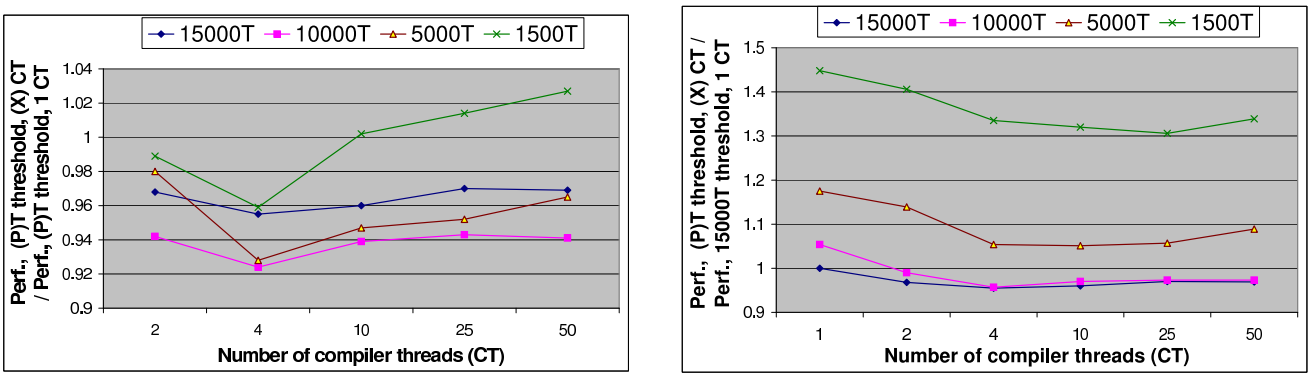


Figure 4. Effect of multiple compiler threads on single-core program performance. The discrete measured thread points are plotted equi-distantly on the x-axis.

this observation suggests that implementing a priority-queue to order compilations may enable more aggressive compilation thresholds to achieve better performances. We explore the effect of prioritized method compiles on program performance in further detail in Section 8. In the absence of a strategy to appropriately prioritize method compiles, our results indicate that there may be no need to change compiler thresholds with more compiler threads on single-core machines. However, a small increase in the number of compiler threads generally improves performance by reducing the compilation queue delay.

6. JIT Compilation on Multi-Core Machines

Dynamic JIT compilation on single-processor machines has to be conservative to manage the compilation overhead at runtime. Modern multi-core machines provide the opportunity to spawn multiple compiler threads and run them concurrently on separate (free) processor cores, while not interrupting the application thread(s). As such, it is a common perception that a more aggressive compilation policy is likely to achieve better application thread and overall program performance on multi-core machines for VMs with multiple compiler threads. Aggressiveness, in this context, can imply compiling early or compiling more methods by lowering the compile threshold. In this section, we report the impact of varying JIT compilation aggressiveness on program performance for multi-core machines.

Our experimental setup controls the aggressiveness of distinct JIT compilation policies by varying the selective compilation threshold. Lowering the compilation threshold can benefit program performance in two ways: (a) by compiling a greater percentage of the program code, and (b) by sending methods to compile early. Thus controlling the compile threshold enables us to simultaneously control both our variables ('if' and 'when') for exploring the impact of compilation policy aggressiveness on program performance. We explore the effect of several compilation thresholds, from the selected baseline threshold of 15,000 to a very aggressive threshold of 50. At the same time, we also alter the number of spawned concurrent compiler threads. More compiler threads will typically have the effect of compiling methods early relative to the application thread. We vary the number of simultaneously active compiler threads in our experiment from 1 to 100. The experiments are conducted on a cluster of identical 8-core machines with hyperthreading disabled.

Figure 5 illustrates the results of our experiments. For each indicated compile threshold, a corresponding line-plot in Figure 5(a) shows the ratio of the program performance with different number of compiler threads to the program performance with a single compiler thread at that same threshold, averaged over our 56 benchmark-input pairs. Thus, we can see that increasing the number of compiler threads improves application performance at all compile thresholds. Additionally, configurations with more aggressive compilation thresholds derive a greater benefit in program performance from more compiler threads. At the same time, the relative gain in program performance does seem to taper off with each additional compiler thread. Moreover, higher compilation thresholds need fewer compiler threads to reach their maximum achievable program performance. These results are expected since a greater number of compiler threads only benefit performance as long as there is work to do for those additional threads. More aggressive thresholds send a greater number of methods to compile and therefore continue deriving performance benefits with more compiler threads. It is also interesting to note that there is almost no further performance improvement for any level of compiler aggressiveness after about seven compiler threads. We believe that this result is a consequence of our hardware setup that uses processors with eight distinct cores. We explore this result further in Section 7.

Figure 5(b) compares all the program performances (with different thresholds and different number of compiler threads) to a single baseline program performance. The selected baseline is the program performance with a single

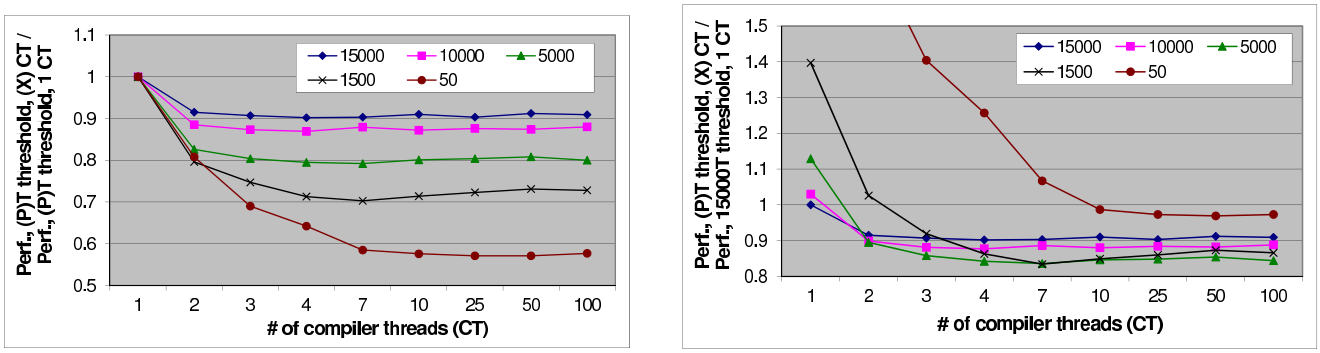


Figure 5. Effect of multiple compiler threads on multi-core application performance

compiler thread at the threshold of 15,000. We can see that in the best case (configuration with threshold 5,000 and 7 compiler threads) the combination of increased compiler aggressiveness with more compiler threads improves performance by about 17%, on average, over our baseline. However, about 10% of that improvement is obtained by simply reducing the *compilation queue delay* that is realized by increasing the number of compiler threads at the baseline (15,000) threshold. Thus, the higher compiler aggressiveness achieved by lowering the selective compilation threshold seems to offer relatively modest benefits over the baseline compilation threshold employed by the default compiler policy on single-core machines.

Another interesting observation that can be made from the plots in Figure 5(b) is that aggressive compilation policies require more compiler threads (implying greater computational resources) to achieve *good* program performance. Indeed, our most aggressive compiler threshold of 50 performs extremely poorly in relation to the baseline threshold, and never improves upon the conservative baseline threshold even with a large number of compiler threads. This result seems to correspond with our observations from the last section regarding the effect of *cold* program methods flooding the queue at aggressive compile thresholds and delaying the compilation of hotter methods. We study different priority queue implementations to alleviate this issue in Section 8.

7. JIT Compilation on Many-Core Machines

Our observations regarding aggressive JIT compilation policies on modern multi-core machines in the last section were limited by our existing 8-core processor based hardware. In future years, architects and chip developers are expecting and planning a continuously increasing number of cores in modern microprocessors. It is possible that our conclusions regarding JIT compilation policies may change with the availability of more abundant hardware resources. However, processors with a large number of cores (or *many-cores*) are not easily available just yet. Therefore, in this section, we construct a unique experimental configuration to conduct experiments that investigate JIT compilation strategies for such future many-core machines.

Our novel experimental setup simulates many-core VM behavior using a single processor/core. To construct this setup, we first update our HotSpot VM to report the *category* of each operating system thread that it creates (such as, application, compiler, garbage-collector, etc.), and to also report the creation or deletion of any VM/program thread at runtime. Next, we modify the *harness* of all our benchmark suites to not only report the overall program execution time, but to also provide a break-down of the time consumed by each individual VM thread. We use the `/proc` file-system interface provided by the Linux operating system to obtain individual thread times, and employ the JNI interface to access this platform-specific OS feature from within a Java program. Finally, we also use the thread-processor-*affinity* interface methods provided by the Linux OS to enable our VM to choose the set of processor cores that are eligible to run each VM thread. Thus, on each new thread creation, the VM is now able to assign the processor affinity of the new VM thread (based on its category) to the set of processors specified by the user on the command-line. We use this facility to constrain all application and compiler threads in a VM to run on a single processor core.

Our experimental setup to evaluate the behavior of many-core (with unlimited cores) application execution on a single-core machine is illustrated in Figure 6. Figure 6(a) shows a snapshot of one possible VM execution order with multiple compiler threads, with each thread running on a distinct core of a many-core machine. Our experimental setup employs the OS thread affinity interface to force all application and compiler threads to run on a single core, and relies on the OS round-robin thread scheduling to achieve a corresponding thread execution order that is shown in Figure 6(b). It is important to note that JIT compilations in our simulation of many-core VM execution (on single-core machine) occur at about the same time relative to the application thread as on a physical many-core machine. We also constrain most of our benchmark programs to spawn a single application

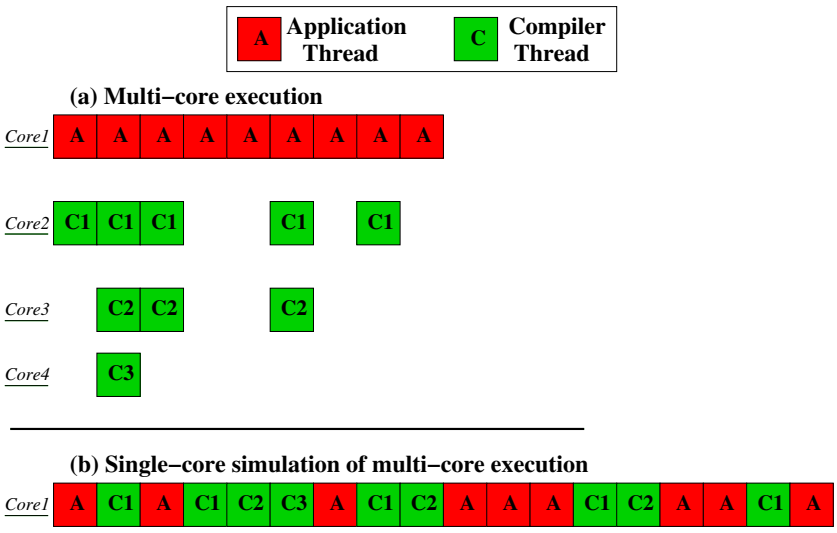


Figure 6. Simulation of multi-core VM execution on single-core processor

thread. Now, on a many-core machine, where each compiler thread runs on its own distinct core concurrently with the application thread, the total program run-time is equal to the application thread run-time alone, as understood from Figure 6(a). Therefore, our ability to precisely measure individual application thread times in our single-core simulation enables us to realistically emulate an environment where each thread has access to its own core. This framework allows us to study the behavior of different JIT compilation strategies with any number of compiler threads running on separate cores on future many-core hardware.

We now employ our new experimental setup to perform the same experiments as done in the last section. Figures 7(a) and (b) show the results of these experiments and plot the application thread times with varying number of compiler threads and compiler aggressiveness. These plots correspond with the graphs illustrated in Figures 5(a) and (b) respectively. We can see that the trends in these results are mostly consistent with our observations from the last section. This similarity confirms the accuracy of our simple simulation model to study JIT compilation policies on many-core machines, in spite of the potential differences between inter-core communication, cache models and other low-level microarchitectural effects. The primary distinction between the two sets of results occurs for larger number of compiler threads. More precisely, unlike the plots in Figure 5(a), Figure 7(a) shows that application thread performance for aggressive compiler thresholds continues gaining improvements beyond a small number of compiler threads. Thus the lack of performance benefits beyond about 7-10 compiler threads in the last section is, in fact, caused due to the limitations of the underlying 8-core hardware. This result shows the utility of our novel setup to investigate VM properties for future many-core machines. These results also show that even 100 active compiler threads are unable to substantially improve program performance for aggressive compiler thresholds beyond the performance obtained by the conservative single-core JIT compilation threshold.

8. Effect of Priority-Based Compiler Queue

The existing HotSpot VM employs a FIFO compiler queue as the communication interface between the application and compiler threads. Thus, methods sent for compilation by the application thread are placed in the compiler queue (and serviced by the compiler thread) in their order of arrival. Our results in the earlier sections suggest that the relatively poor performance achieved by the aggressive JIT compilation policies may be an artifact of the FIFO compiler queue that cannot adequately prioritize the compilations by *actual* hotness levels of application methods. Therefore, in this section, we explore and measure the potential of different priority queue implementations to improve the performance obtained by different JIT compilation strategies.

8.1 Ideal Priority-Based Compiler Queue

First, we attempt to understand the performance impact of an *ideal* strategy for ordering method compilations. An ideal compilation strategy should be able to precisely determine the actual hotness level of all methods send to compile, and always compile them in that order. Unfortunately, such an ideal strategy requires knowledge of future program behavior, which is very difficult to determine or obtain. In lieu of such information, we devise a compilation strategy that prioritizes method compiles based on their total execution counts over an earlier profile run. With this strategy, the compiler thread always selects and compiles the method with the highest profiled execution counts from the available candidates in the compiler queue. We call this our *ideal* compilation strategy. We do note that even our ideal profile-driven strategy may not achieve the *actual* best results because the

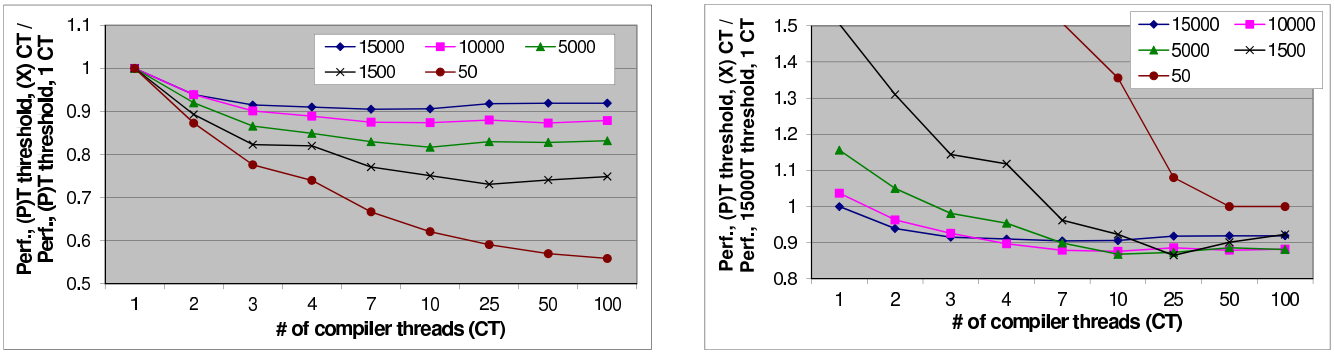


Figure 7. Effect of multiple compiler threads on many-core application performance

candidate method with the highest hotness level may still not be the best method to compile at *that* point during program execution. To the best of our knowledge, this is the first attempt at studying a potentially ideal priority queue scheme in the context of JIT compilation.

Thus, our ideal priority-queue strategy requires a profile-run of every benchmark to determine its method hotness counts. For maximum efficiency, offline profiling necessitates using method-level annotations to indicate the measured hotness levels to the VM. However, as mentioned earlier, issues with reflection and runtime method creation makes accurate and complete method annotations very difficult, especially for newer Java benchmarks. Therefore, we again employ the novel VM capability we developed to allow the virtual machine to detect specific points of interest during a program run. Our experimental setup runs two iterations for every benchmark program. The first iteration performs no compilations, but records the execution counts of each method. As a side-effect this initial iteration also discovers and loads all necessary program methods. The VM detects the end of the first iteration and uses this indicator to mark all hot methods with a rank based on their execution counts collected by the VM during the first iteration. The VM then re-enables compilations for the next iteration. The second (and final) iteration employs a priority queue to sort methods sent for compilation by the application thread in descending order of their attached ranks. For maximum efficiency, we implement our sorted priority queue as a binary tree.

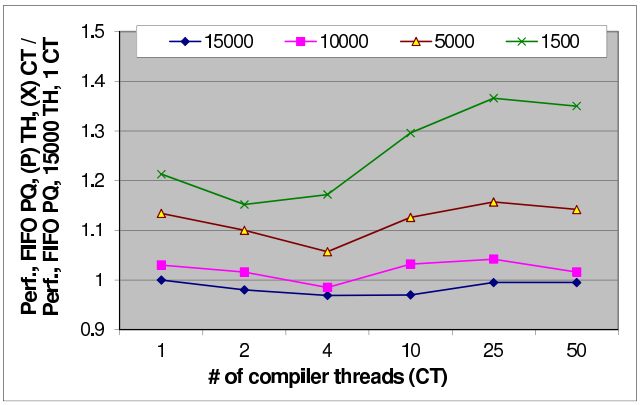
8.1.1 Single-Core Machine Configuration

Figure 8 compares the performance results of our ideal compiler priority queue and the default FIFO priority queue implementations for single-core machines. Figure 8(a) plots the average performance ratio with the FIFO priority queue for different VM compile thresholds and different number of compiler threads with the baseline single compiler thread, 15,000 threshold VM performance, averaged over our 56 benchmark-input pairs. These measurements vary slightly from earlier results shown for the similar configuration in Figure 4(b). This variation is due to the different experimental configuration adopted in this section that causes some methods (mainly from the benchmark harness and the Java library) that are hot only in the first benchmark iteration of every run to not be compiled. It is more relevant to note that the results with our new configuration still show the same performance trends as discussed in Section 5.2.

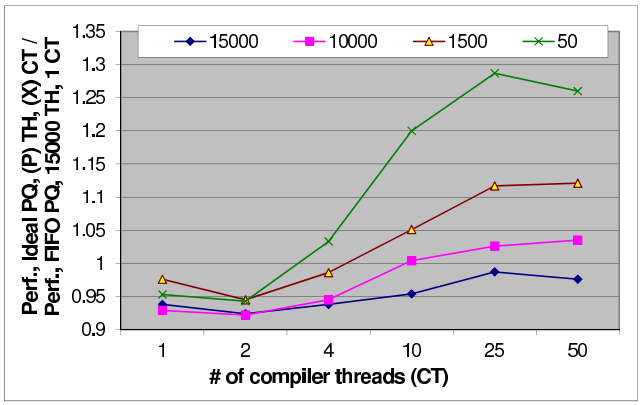
Figure 8(b) presents the results of VM runs with our ideal priority queue implementation compared to the same default FIFO priority queue implementation with a single compiler thread and a threshold of 15,000. The graph illustrates that accurate assignment of method priorities allows the higher compile thresholds to also achieve good average program performance for small number of compiler threads. As described in Section 5.2, initiating a greater number of compiler threads on single-core machines results in compiling methods that are otherwise left uncompiled (in the compiler queue) upon program termination with fewer compiler threads. The resulting increase in the compilation overhead is not sufficiently compensated by the improved application efficiency, resulting in a net loss in overall performance. This effect persists regardless of the method priority algorithm employed. We do see that accurately ordering the method compiles enables the VM with our ideal priority queue implementation to obtain slightly better performance than the best achieved with the FIFO queue.

8.1.2 Many-Core Machine Configuration

Figure 9 compares the performance results of using our ideal compiler priority queue with the baseline VM that uses the default FIFO-based compiler queue implementation for many-core machines. Figure 9(a) plots the average performance ratio with the default FIFO priority queue. Again, these measurements vary slightly from earlier results for the similar configuration in Figure 7(b) due to the different experimental configuration adopted in this section. We also note that results with this configuration still show the same performance trends as discussed in Section 6.

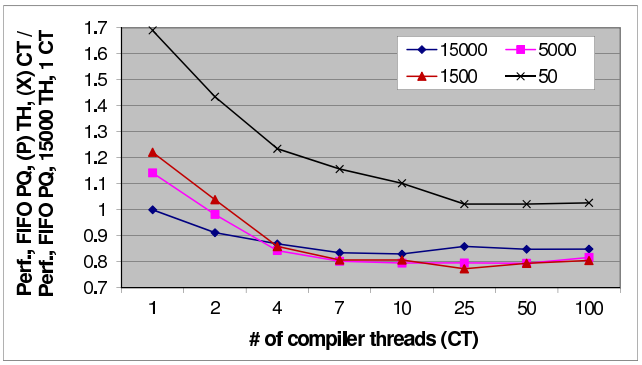


(a) FIFO-priority

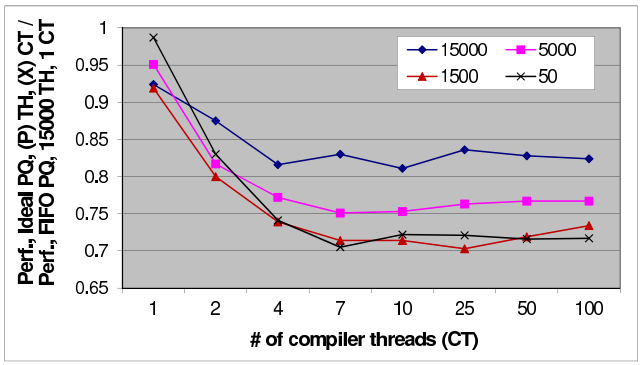


(b) Ideal Priority

Figure 8. Comparison of ideal compiler priority queue implementation with baseline FIFO compiler queue for single-core machine configuration



(a) FIFO-priority



(b) Ideal Priority

Figure 9. Comparison of ideal compiler priority queue implementation with baseline FIFO compiler queue for many-core machine configuration

Figure 9(b) displays the performance ratio of the VM runs with our ideal priority-based compiler queue implementation with the baseline VM performance (15,000 threshold, single compiler thread, FIFO compiler queue) that was also used in Figure 9(a). These results show several interesting trends. First, appropriately sorting method compiles significantly benefits program performance at all threshold levels. At the same time, the performance benefits are more prominent for aggressive compile thresholds. This behavior is logical since more aggressive thresholds are more likely to flood the queue with low-priority compiles that delay the compilation of the *hotter* methods with the FIFO queue. Second, the best average benchmark performance with our ideal priority queue for every threshold plot is achieved with a smaller number of compiler threads, especially for the more aggressive compiler thresholds. This result shows that our ideal priority queue does realize its goal of compiling the hotter methods before the cold methods. The later lower priority method compilations seem to not make a major impact on program performance. Finally, we can also conclude that using a good priority compiler queue allows more aggressive compilation policies (that compile a greater fraction of the program early) to improve performance over a less aggressive strategy on multi/many-core machines. Moreover, a small number of compiler threads is generally sufficient to achieve the best average application thread performance. Overall, the best aggressive compilation policy improves performance by about 30% over the baseline, and by about 11% over the best performance achieved by the default single-core compilation threshold of 15,000.

8.2 Heuristic Priority-Based Compiler Queue

Our ideal priority queue strategy requires *offline* profile information to correctly order method compilations and improve VM performance. However, collecting offline program profiles is often cumbersome and may be infeasible to obtain in many cases. Additionally, method compile priorities provided by offline profiling may be invalidated if the input/environment during the actual program run varies from that used during the profiling run. Therefore, it may be

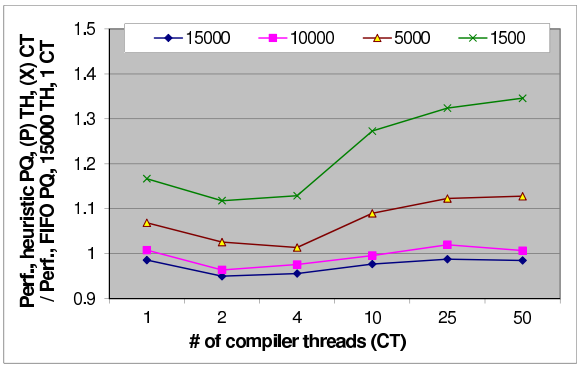


Figure 10. Comparison of our best heuristic dynamic compiler priority queue implementation with baseline FIFO compiler queue for single-core machine configuration

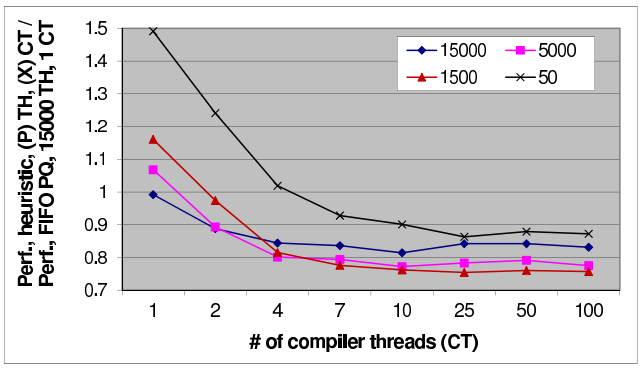


Figure 11. Comparison of our best heuristic dynamic compiler priority queue implementation with baseline FIFO compiler queue for many-core machine configuration

important to find techniques that can assess method priorities dynamically during every program run. However it may be difficult to obtain comparable performance results using a completely online strategy. An online strategy can only see past program behavior. Additionally, the more aggressive JIT compilation policies make their hotness decisions earlier, giving any online strategy an even reduced opportunity to accurately access method priorities. The default FIFO queue mechanism uses a heuristic that assigns method priorities based on their order of arrival. As part of this work, we experimented with a few other (and arguably more complex) schemes to assign dynamic method priorities.

In this section we describe the priority scheme that achieved our best results. This scheme uses a new global counter in addition to the counter used to hold the execution counts for each method. This global counter accumulates the execution counts of all methods from the start of each run. The value of this global counter is recorded in a new field (say, X) in every method header on the first invocation of that method. A method is still sent to compile when its normal execution counter exceeds the specified compilation threshold. However, now before insertion into the compile queue, the method priority is calculated as:

$$Priority = \frac{method\ execution\ count}{current\ global\ count - X} \qquad (1)$$

Thus, by the scheme, methods that attain their hotness promptly after their first invocation and have become hot in the more recent past are likely to be assigned a higher priority for compilation. We use the same binary-tree based implementation as employed earlier.

Figures 10 and 11 display the application thread performance of our best dynamic priority queue implementation compared to the corresponding baseline performance (with FIFO-based compiler queue at the threshold of 15,000 with one compiler thread) for single-core and many-core machines respectively. Thus, for both these graphs, we can see that the performance achieved by our new dynamic priority scheme is better than that achieved by the FIFO compiler queue at most measured points, but does not match the benefit of the ideal priority scheme. We are actively exploring other heuristic techniques to more accurately assign method priorities dynamically at runtime.

9. Future Work

This work presents several interesting avenues for future research. First, this work shows that the availability of abundant computation resources in future machines enables the possibility of program performance improvement by early compilation of a greater fraction of the program. With the development of profile-driven optimization phases, future work will have to consider the effect of early compilation on the amount of collected profile information and resulting impact on generated code. Additionally, researchers may also need to explore the interaction of increased compiler activity with garbage collection. Increased native code produced by aggressive JIT compilation can raise memory pressure and garbage collection overheads, which may then affect program non-determinism due to the increased pause times associated with garbage collections. Second, in this paper we explored some priority queue implementations that may be more suitable with aggressive compilation policies. We plan to continue our search for better method prioritization schemes, as well as possibly allowing the method priorities to be re-evaluated even after enqueuing, which was not considered in this work. Third, this work restricts the JIT compilation policy exploration to *if* and *when* methods are compiled at the same optimization level, but did not allow controlling *how* to perform compilation in different cases. In the future, we plan to also study increasing compiler aggressiveness by optimizing at higher compilation levels in a VM that provides robust support for tiered compilation. Finally, we are currently conducting similar experiments in other virtual machines (JikesRVM) to see if our conclusions from this work hold across different VMs.

10. Conclusions

Modern processors are expected to continue integrating increasing number of cores in future chips. The goal of this work was to explore the potential performance benefit of more aggressive JIT compilation policies for current and future multi/many-core machines and with newer virtual machines that support multiple simultaneous compiler threads. We first discover that compiling more and/or compiling early can significantly improve performance compared to that achieved by the default HotSpot VM. However, these improvements fade in comparison to that already obtained by the default JIT based VM over pure interpretation. We believe that these results will allow VM developers and users to assess if the maximum additional benefit of compiling more program methods early is worth the increased computation and memory resources necessary to generate and maintain the corresponding native code.

An important contribution of this work is the development of several novel VM configurations to facilitate our experiments. On single-core machines, we find that the same compilation threshold achieves the best overall program performance with a single and multiple compiler threads, and regardless of the priority queue algorithm that is used for the compiler queue. Our results on multi-core and many-core machines find that although modern multi-core and many-core hardware can enable more aggressive JIT compilation policies, and more aggressive policies can produce benefits to performance, achieving such benefits requires accurate assignment of method priorities. Finding dynamic online algorithms to assign such accurate method priorities is still an open research question. Thus, as we enter the new era of multi/many-core machines with increasing number of cores with every processor generation, we expect this research to assist VM developers to make more informed decisions regarding how to design and implement the best possible JIT compilation policies to achieve the best application performance.

Acknowledgments

We thank the anonymous reviewers for their constructive comments and suggestions. This research was supported in part by NSF CAREER award CNS-0953268.

References

[1] International technology roadmap for semiconductors. accessed from http://www.itrs.net/Links/2009ITRS/Home2009.htm, 2008-09.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalapeno jvm. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–65, 2000.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeo JVM: The controller's analytical model. In Proceedings of the 3rd ACM Workshop on Feedback Directed and Dynamic Optimization (FDDO '00), December 2000.

[4] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 92(2):449–466, February 2005.

[5] 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. E. B. Moss, 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 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 169–190, 2006.

[6] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE '11: International Conference on Software Engineering*, May 2011.

[7] I. Böhm, T. J. Edler von Koch, S. C. Kyle, B. Franke, and N. Topham. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 74–85, 2011.

[8] D. Bruening and E. Duesterwald. Exploring optimal compilation unit shapes for an embedded just-in-time compiler. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 13–20, 2000.

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

[10] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302, New York, NY, USA, 1984. ACM. ISBN 0-89791-125-3.

[11] M. X. Goemans. Advanced algorithms. Technical Report MIT/LCS/RSS-27, 1994.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(TM) Language Specification (3rd Edition)*. Prentice Hall, third edition, June 14 2005.

[13] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Notices*, 17(6):120–126, 1982.

[14] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *Proceedings of the conference on Virtual Machine Research And Technology Symposium*, pages 12–12, 2004.

[15] D. Gu and C. Verbrugge. Phase-based adaptive recompilation in a jvm. In *Proceedings of the 6th IEEE/ACM symposium on Code generation and optimization*, CGO '08, pages 24–34, 2008.

[16] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Generating new general compiler optimization settings. In *ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing*, pages 161–168, 2005. ISBN 1-59593-167-8.

[17] G. J. Hansen. *Adaptive systems for the dynamic run-time optimization of programs*. PhD thesis, Carnegie-Mellon Univ., Pittsburgh, PA, 1974.

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

[19] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Language Systems*, 18(4):355–400, 1996. ISSN 0164-0925.

[20] R. M. Karp. On-line algorithms versus off-line algorithms: How much is it worth to know the future? In *Proceedings of the IFIP World Computer Congress on Algorithms, Software, Architecture - Information Processing, Vol 1*, pages 416–429, 1992.

[21] D. E. Knuth. An empirical study of fortran programs. *Software: Practice and Experience*, 1(2):105–133, 1971.

[22] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java hotspot™client compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1): 1–32, 2008.

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

[24] C. Krintz and B. Calder. Using annotations to reduce dynamic optimization time. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 156–167, 2001.

[25] C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8):717–738, December 2000.

[26] P. Kulkarni, M. Arnold, and M. Hind. Dynamic compilation: the benefits of early investing. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 94–104, 2007.

[27] P. A. Kulkarni and J. Fuller. JIT compilation policy on single-core and multi-core machines. In the 15th Workshop on *Interaction between Compilers and Computer Architectures (INTERACT)*, pages 54–62, February 2011.

[28] Microsoft. *Microsoft C# Language Specifications*. Microsoft Press, first edition, April 25 2001.

[29] M. A. Namjoshi and P. A. Kulkarni. Novel online profiling for virtual machines. In *VEE '10: Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 133–144, 2010. ISBN 978-1-60558-910-7.

[30] M. Paleczny, C. Vick, and C. Click. The Java hotspottm server compiler. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, pages 1–12, Berkeley, CA, USA, 2001. USENIX Association.

[31] SPEC2008. Specjvm2008 benchmarks. http://www.spec.org/jvm2008/, 2008.

[32] SPEC98. Specjvm98 benchmarks. http://www.spec.org/jvm98/, 1998.

[33] V. Sundaresan, D. Maier, P. Ramarao, and M. Stoodley. Experiences with multi-threading and dynamic class loading in a Java just-in-time compiler. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 87–97, 2006.

[34] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13, 1999.

[35] D. L. Whitfield and M. L. Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, 1997.