Exploring Single and Multilevel JIT Compilation Policy for Modern Machines[1]

MICHAEL R. JANTZ and PRASAD A. KULKARNI, University of Kansas, Lawrence, Kansas

Dynamic or Just-in-Time (JIT) compilation is essential to achieve high-performance emulation for programs written in *managed* languages, such as Java and C#. It has been observed that a conservative JIT compilation policy is most effective to obtain good runtime performance without impeding application progress on single-core machines. At the same time, it is often suggested that a more aggressive dynamic compilation strategy may perform best on modern machines that provide abundant computing resources, especially with virtual machines (VMs) that are also capable of spawning multiple concurrent compiler threads. However, comprehensive research on the best JIT compilation policy for such modern processors and VMs is currently lacking. The goal of this work is to explore the properties of single-tier and multitier JIT compilation policies that can enable existing and future VMs to realize the best program performance on modern machines.

In this work, we design novel experiments and implement new VM configurations to effectively control the compiler aggressiveness and optimization levels (*if* and *when* methods are compiled) in the industry-standard Oracle HotSpot Java VM to achieve this goal. We find that the best JIT compilation policy is determined by the nature of the application and the speed and effectiveness of the dynamic compilers. We extend earlier results showing the suitability of conservative JIT compilation on single-core machines for VMs with multiple concurrent compiler threads. We show that employing the free compilation resources (compiler threads and hardware cores) to aggressively compile *more* program methods quickly reaches a point of diminishing returns. At the same time, we also find that using the free resources to reduce compiler queue backup (compile selected hot methods *early*) significantly benefits program performance, especially for slower (highly optimizing) JIT compilers. For such compilers, we observe that accurately prioritizing JIT method compiles is crucial to realize the most performance benefit with the smallest hardware budget. Finally, we show that a tiered compilation policy, although complex to implement, greatly alleviates the impact of more and early JIT compilation of programs on modern machines.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Optimizations, runtime environments, compilers*

General Terms: Languages, Performance

Additional Key Words and Phrases: Virtual machines, dynamic compilation, multicore, Java

ACM Reference Format:
Jantz, M. R. and Kulkarni, P. A. 2013. Exploring single and multilevel JIT compilation policy for modern machines. ACM Trans. Architec. Code Optim. 10, 4, Article 22 (December 2013), 29 pages.
DOI: http://dx.doi.org/10.1145/2541228.2541229

[1]**Extension of Conference Paper.** This work extends our conference submission, titled *JIT Compilation Policy for Modern Machines*, published in the ACM International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA) [Kulkarni 2011]. We extend this earlier work by (a) reimplementing *all* experiments in the latest HotSpot JVM that provides a new state-of-the-art multitier

This work is supported by the National Science Foundation, under NSF CAREER award CNS-0953268.
Authors' addresses: M. R. Jantz and P. A. Kulkarni, Department of Electrical Engineering and Computer Science, University of Kansas, Lawrence, KS 66045.

DOI: http://dx.doi.org/10.1145/2541228.2541229

1. INTRODUCTION

To achieve application portability, programs written in *managed* programming languages, such as Java [Gosling et al. 2005] and C# [Microsoft 2001], are distributed as machine-independent intermediate language binary codes for a *virtual machine* (VM) architecture. Since the program binary format does not match the native architecture, VMs have to employ either interpretation or dynamic compilation for executing the program. Additionally, the overheads inherent during program interpretation make dynamic or Just-in-Time (JIT) compilation essential to achieve high-performance emulation of such programs in a VM [Smith and Nair 2005].

Since it occurs at runtime, JIT compilation contributes to the overall execution time of the application and can potentially impede application progress and further degrade its *response* time, if performed injudiciously. Therefore, JIT compilation policies need to carefully tune *if*, *when* and *how* to compile different program regions to achieve the best overall performance. Researchers invented the technique of *selective compilation* to address the issues of *if* and *when* to compile program methods during dynamic compilation [Hölzle and Ungar 1996; Paleczny et al. 2001; Krintz et al. 2000; Arnold et al. 2005]. Additionally, several modern VMs provide multiple optimization levels along with decision logic to control and decide *how* to compile each method. While a *single-tier* compilation strategy always applies the same set of optimizations to each method, a *multitier* policy may compile the same method multiple times at distinct optimization levels during the same program run. The control logic in the VM determines each method's *hotness* level (or how much of the execution time is spent in a method) to decide its compilation level.

Motivation: Due to recent changes and emerging trends in hardware and VM architectures, there is an urgent need for a fresh evaluation of JIT compilation strategies on modern machines. Research on JIT compilation policies has primarily been conducted on single-processor machines and for VMs with a single compiler thread. As a result, existing 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. These modern architectures allow the possibility of running the compiler thread(s) on a separate core(s) to minimize interference with the application thread. VM developers are also responding to this change in their hardware environment by allowing the VM to simultaneously initiate multiple concurrent compiler threads. Such evolution in the hardware and VM contexts may require radically different JIT compilation policies to achieve the most effective overall program performance.

Objective: The objective of this research is to investigate and recommend JIT compilation strategies to enable the VM to realize the best program performance on existing single-/multicore processors and future many-core machines. We vary the *compilation threshold*, the number of initiated compiler threads, and single and multitier compilation strategies to control *if*, *when*, and *how* to detect and compile important program methods. The compilation threshold is a heuristic value that indicates the *hotness* of each method in the program. Thus, more aggressive policies employ a smaller compilation threshold so that more methods become *hot* sooner. We induce progressive increases in the aggressiveness of JIT compilation strategies and the number of

compiler and supports improved optimizations in the server compiler; (b) for the first time, investigating the effects of aggressive compilation and multiple compiler threads on *multi-tiered* JIT compilation strategies; (c) providing more comprehensive results, with differentiation on benchmark features; (d) reanalyzing our observations and conclusions; and (e) exploring a different set of heuristic priority schemes.

concurrent compiler threads and analyze their effect on program performance. While a single-tier compilation strategy uses a single compiler (and fixed optimization set) for each hot method, a multitier compiler policy typically compiles a hot method with progressively *advanced* (that apply more and better optimizations to potentially produce higher-quality code), but slower, JIT compilers. Our experiments change the different multitier hotness thresholds in lock-step to also *partially* control how (optimization level) each method is compiled.[2] Additionally, we design and construct a novel VM configuration to conduct experiments for many-core machines that are not commonly available yet.

Findings and Contributions: This is the first work to thoroughly explore and evaluate these various compilation parameters and strategies (a) on multicore and many-core machines and (b) together. We find that the most effective JIT compilation strategy depends on several factors, including the availability of free computing resources and program features (particularly the ratio of hot program methods) and the compiling speed, quality of generated code, and method prioritization algorithm used by the compiler(s) employed. In sum, the major contributions of this research are:

(1) We design original experiments and VM configurations to investigate the most effective JIT compilation policies for modern processors and VMs with single- and multilevel JIT compilation.
(2) We quantify the impact of altering "if," "when," and one aspect to "how" methods are compiled on application performance. Our experiments evaluate JVM performance with various settings for compiler aggressiveness and the number of compilation threads, as well as different techniques for prioritizing method compiles, with both single- and multilevel JIT compilers.
(3) We explain the impact of different JIT compilation strategies on available single-/multicore and future many-core machines.

The rest of the article is organized as follows. In the next section, we present background information and related work on existing JIT compilation policies. We describe our general experimental setup in Section 3. Our experiments exploring different JIT compilation strategies for VMs with multiple compiler threads on single-core machines are described in Section 4. In Section 5, we present results that explore the most effective JIT compilation policies for multicore machines. We describe the results of our novel experimental configuration to study compilation policies for future many-core machines in Section 6. We explain the impact of prioritizing method compiles and the effect of multiple application threads in Sections 7 and 8. Finally, we present our conclusions and describe avenues for future work 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 [Deutsch and Schiffman 1984] followed by the Self-93 VM [Hölzle and Ungar 1996] pioneered many of the adaptive optimization techniques employed in current VMs, including selective compilation with multiple compiler threads on single-core

[2]In contrast to the two components of "if" and "when" to compile, the issue of how to compile program regions is much broader and 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 [Graham et al. 1982; Chang et al. 1991; Arnold et al. 2002; Hazelwood and Grove 2003] and optimization phase ordering/selection [Whitfield and Soffa 1997; Haneda et al. 2005; Cavazos and O'Boyle 2006; Sanchez et al. 2011; Jantz and Kulkarni 2013] for static and dynamic compilers. Consequently, we only explore one aspect of "how" to compile methods in this work.

machines. Aggressive compilation on such machines has the potential of degrading program performance by increasing the compilation time. The technique of selective compilation was invented to address this issue with dynamic compilation [Hölzle and Ungar 1996; Paleczny et al. 2001; Krintz et al. 2000; Arnold et al. 2005]. 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 [Knuth 1971; Bruening and Duesterwald 2000; Arnold et al. 2005]. 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. Most current VMs employ selective compilation with a *staged* emulation model [Hansen 1974]. With this model, each method is interpreted or compiled with a fast nonoptimizing compiler at program start to improve application response time. Later, the VM determines and selectively compiles and optimizes only the subset of hot methods to achieve better program performance.

Unfortunately, selecting the hot methods to compile requires *future* program execution information, which is hard to accurately predict [Namjoshi and Kulkarni 2010]. 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 [Grcevski et al. 2004; Kotzmann et al. 2008; Arnold et al. 2000a]. Online profiling is used to detect these current hot methods. The most popular online profiling approaches are based on instrumentation *counters* [Hansen 1974; Hölzle and Ungar 1996; Kotzmann et al. 2008], interrupt-timer-based *sampling* [Arnold et al. 2000a], or a combination of the two methods [Grcevski et al. 2004]. The method/loop is sent for compilation if the respective method counters exceed a fixed threshold.

Finding the correct threshold value is crucial to achieve good program startup performance 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. 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.

Resource constraints force JIT compilation policies to make several tradeoffs. Thus, selective compilation limits the time spent by the compiler at the cost of potentially lower application performance. Additionally, the use of online profiling 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 *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. The 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 [Kulkarni et al. 2007; Krintz 2003; Gu and Verbrugge 2008].

Various strategies have been developed to address these delays in JIT compilation at program startup. Researchers have explored the potential of offline profiling and class-file annotation [Krintz and Calder 2001; Krintz 2003], early and accurate prediction of hot methods [Namjoshi and Kulkarni 2010], and online program phase detection [Gu and Verbrugge 2008] to alleviate the first delay component caused by online profiling. Likewise, researchers have also studied techniques to address the second component of the compilation delay caused by the backup and wait time in the method compilation queue. These techniques include increasing the priority [Sundaresan et al. 2006] and CPU utilization [Kulkarni et al. 2007; Harris 1998] of the compiler thread and

providing a priority queue implementation to reduce the delay for the *hotter* program methods [Arnold et al. 2000b].

However, most of the studies described above have only been targeted for single-core machines. There exist few explorations of JIT compilation issues for multicore machines. Krintz et al. investigated the impact of background compilation in a separate thread to reduce the overhead of dynamic compilation [Krintz et al. 2000]. 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 multicore machines but do not provide any experimental results [Kulkarni et al. 2007]. Existing JVMs, such as Sun's HotSpot server VM [Paleczny et al. 2001] and the Azul VM (derived from HotSpot), support multiple compiler threads but do not present any discussions on ideal compilation strategies for multicore machines. Prior 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 [Böhm et al. 2011]. Esmaeilzadeh et al. study the scalability of various Java workloads and their power/performance tradeoffs across several different architectures [Esmaeilzadeh et al. 2011]. Our earlier publications explore some aspects of the impact of varying the aggressiveness of dynamic compilation on modern machines for JVMs with multiple compiler threads [Kulkarni and Fuller 2011; Kulkarni 2011]. This article extends our earlier works by (a) providing more comprehensive results, (b) reimplementing most of the experiments in the latest OpenJDK JVM that provides a state-of-the-art multitier compiler and supports improved optimizations, (c) differentiating the results and reanalyzing our observations based on benchmark characteristics, (d) exploring different heuristic priority schemes, and (e) investigating the effects of aggressive compilation and multiple compiler threads on the multitiered JIT compilation strategies. Several production-grade Java VMs, including the Oracle HotSpot and IBM J9, now adopt a multitier compilation strategy, making our results with the multitiered compiler highly interesting and important.

3. EXPERIMENTAL FRAMEWORK

The research presented in this article is performed using Oracle's OpenJDK/HotSpot Java virtual machine (build 1.6.0_25-b06) [Paleczny et al. 2001]. The HotSpot VM uses interpretation at program startup. 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 the *execution count* of the method. Methods/loops are determined to be hot if the corresponding method execution count exceeds a fixed threshold. The HotSpot VM allows the creation of an arbitrary number of compiler threads, as specified on the command line.

The HotSpot VM implements two distinct optimizing compilers to improve application performance beyond interpretation. The *client compiler* provides relatively fast compilation times with smaller program performance gains to reduce application startup time (especially on single-core machines). The *server compiler* applies an aggressive optimization strategy to maximize performance benefits for longer running applications. We conducted experiments to compare the overhead and effectiveness of HotSpot's client and server compiler configurations. We found that the client compiler is immensely fast and only *requires about 2% of the time, on average, taken by the server compiler* to compile the same set of hot methods. At the same time, the simple and fast *client compiler is able to obtain most (95%) of the performance gain (relative to interpreted code) realized by the server compiler*.

In addition to the single-level client and server compilers, HotSpot provides a *tiered compiler* configuration that utilizes and combines the benefits of the client and

Table I. Threshold Parameters in the Tiered Compiler

Parameter	Description	Client Default	Server Default
Invocation Threshold	Compile method if invocation count exceeds this threshold	200	5,000
Back-edge Threshold	OSR compile method if back-edge count exceeds this threshold	7,000	40,000
Compile Threshold	Compile method if invocation + back-edge count exceeds this threshold (and invocation count > Minimum Invocation Threshold)	2,000	15,000
Minimum Invocation Threshold	Minimum number of invocations required before method can be considered for compilation	100	600

server compilers. In the most common path in the tiered compiler, each hot method is first compiled with the client compiler (possibly with additional profiling code inserted), and later, if the method remains hot, is recompiled with the server compiler. Each compiler thread in the HotSpot tiered compiler is dedicated to either the client or server compiler, and *each compiler is allocated at least one thread*. To account for the longer compilation times needed by the server compiler, HotSpot automatically assigns the compiler threads at a 2:1 ratio in favor of the server compiler. The property of the client compiler to quickly produce high-quality optimized code greatly influences the behavior of the tiered compiler under varying compilation loads, as our later experiments in this article will reveal.

There is a single *compiler queue* designated to each (client and server) compiler in the tiered configuration. These queues employ a simple execution-count-based priority heuristic to ensure the most active methods are compiled earlier. This heuristic computes the execution count of each method in the appropriate queue since the last queue removal to find the most active method. As the load on the compiler threads increases, HotSpot dynamically increases its compilation thresholds to prevent either the client or server compiler queues from growing prohibitively long. In addition, the HotSpot tiered compiler has logic to automatically remove *stale* methods that have stayed in the queue for too long. For our present experiments, we disable the automatic throttling of compilation thresholds and removal of stale methods to appropriately model the behavior of a generic tiered compilation policy. The tiered compiler uses different thresholds that move in lockstep to tune the aggressiveness of its component client and server compilers. Table I describes these compilation thresholds and their default values for each compiler in the tiered configuration.

The experiments in this article were conducted using all the benchmarks from three different benchmark suites, SPECjvm98 [SPEC98 1998], SPECjvm2008 [SPEC2008 2008], and DaCapo-9.12-bach [Blackburn et al. 2006]. 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 57 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. In order to limit possible sources of variation in our experiments, we set the number of application threads to one whenever possible. Unfortunately, several of our benchmarks employ multiple application threads due to *internal multithreading* that cannot be controlled by the harness application. Table II lists the name, number of invoked methods (under the column labeled #M), and number of application threads (under the column labeled #AT) for each benchmark in our suite.

All our experiments were performed on a cluster of dual quad-core, 64-bit, x86 machines running Red Hat Enterprise Linux 5 as the operating system. The cluster

Table II. Benchmarks Used in Our Experiments

SPECjvm98			SPECjvm2008			DaCapo-9.12-bach		
Name	*#M*	*#AT*	Name	*#M*	*#AT*	Name	*#M*	*#AT*
_201_compress_100	517	1	compiler.compiler	3195	1	avrora_default	1849	6
_201_compress_10	514	1	compiler.sunflow	3082	1	avrora_small	1844	3
_202_jess_100	778	1	compress	960	1	batik_default	4366	1
_202_jess_10	759	1	crypto.aes	1186	1	batik_small	3747	1
_205_raytrace_100	657	1	crypto.rsa	960	1	eclipse_default	11145	5
_205_raytrace_10	639	1	crypto.signverify	1042	1	eclipse_small	5461	3
_209_db_100	512	1	derby	6579	1	fop_default	4245	1
_209_db_10	515	1	mpegaudio	959	1	fop_small	4601	2
_213_javac_100	1239	1	scimark.fft.small	859	1	h2_default	2154	3
_213_javac_10	1211	1	scimark.lu.small	735	1	h2_small	2142	3
_222_mpegaudio_100	659	1	scimark.monte_carlo	707	1	jython_default	3547	1
_222_mpegaudio_10	674	1	scimark.sor.small	715	1	jython_small	2070	2
_227_mtrt_100	658	2	scimark.sparse.small	717	1	luindex_default	1689	2
_227_mtrt_10	666	2	serial	1121	1	luindex_small	1425	1
_228_jack_100	736	1	sunflow	2015	5	lusearch_default	1192	1
_228_jack_10	734	1	xml.transform	2592	1	lusearch_small	1303	2
			xml.validation	1794	1	pmd_default	3881	8
						pmd_small	3058	3
						sunflow_default	1874	2
						sunflow_small	1826	2
						tomcat_default	9286	6
						tomcat_small	9189	6
						xalan_default	2296	1
						xalan_small	2277	1

includes three models of server machine: Dell M600 (two 2.83GHz Intel Xeon E5440 processors, 16 GB DDR2 SDRAM), Dell M605 (two 2.4GHz AMD Opteron 2378 processors, 16GB DDR2 SDRAM), and PowerEdge SC1435 (two 2.5GHz AMD Opteron 2380 processors, 8GB DDR2 SDRAM). We run all of our experiments on one of these three models, but experiments comparing runs of the same benchmark always use the same model. There are no hyperthreading or frequency scaling techniques of any kind enabled during our experiments.

We disable seven of the eight available cores to run our single-core experiments. Our multicore experiments utilize all available cores. 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. In order to account for inherent timing variations during the benchmark runs, all the performance results in this article report the average over 10 runs for each benchmark-configuration pair. All the experiments in this article measure *startup* performance. Thus, any compilation that occurs is performed concurrently with the running application.

Finally, we present a study to compare the program performance on single-core and multicore machines. Figure 1 shows the multicore performance of each benchmark relative to single-core performance for both the default server and tiered compiler configurations. To estimate the degree of variability in our runtime results, we compute 95% confidence intervals for the difference between the means [Georges et al. 2007] and plot these intervals as error bars. Not surprisingly, we observe that most benchmarks run much faster with the multicore configuration. Much of this difference is simply due to increased parallelism, but other microarchitectural effects (such as cache affinity and intercore communication) may also impact performance depending on the workload.

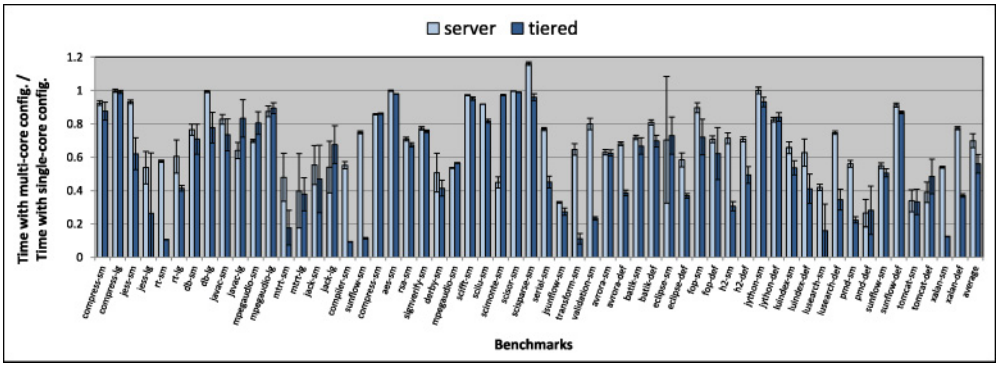


Fig. 1. Ratio of multicore performance to single-core performance for each compiler configuration.

Another significant factor, which we encounter in our experiments throughout this work, is that additional cores enable *earlier* compilation of hot methods. This effect accounts for the result that the tiered VM, with its much more aggressive compilation threshold, exhibits a more pronounced performance improvement, on average, than the server VM. The remainder of this article explores and explains the impact of different JIT compilation strategies on modern and future architectures using the HotSpot server and tiered compiler configurations.

4. JIT COMPILATION ON SINGLE-CORE MACHINES

In this section, we report the results of our experiments conducted on single-core processors to understand the impact of aggressive JIT compilation and more compiler threads in a VM on program performance. Our experimental setup controls the aggressiveness of distinct JIT compilation policies by varying the selective compilation threshold. Changing the compilation threshold can affect program performance in two ways: (a) by compiling a lesser or greater percentage of the program code (*if* a method is compiled) and (b) by sending methods to compile early or late (*when* is each method compiled). We first employ the HotSpot server VM with a single compiler thread to find the selective compilation threshold that achieves the best average performance with our set of benchmark programs.[3] Next, we evaluate the impact of multiple compiler threads on program performance for machines with a single processor with both the server and tiered compilers in the HotSpot JVM.

4.1. Compilation Threshold with Single Compiler Thread

By virtue of sharing the same computation resources, the application and compiler threads share a complex relationship in a VM running on a single-core machine. A highly selective compile threshold may achieve poor overall program performance by spending too much time executing in nonoptimized code resulting in poor overall program runtime. 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.

We perform an experiment to determine the ideal compilation threshold for the HotSpot server VM with a *single* compiler thread on our set of benchmarks. These results are presented in Figure 2(a). The figure compares the average overall program

[3]The tiered compiler spawns a minimum of two compiler threads and is therefore not used in this single compiler thread configuration.

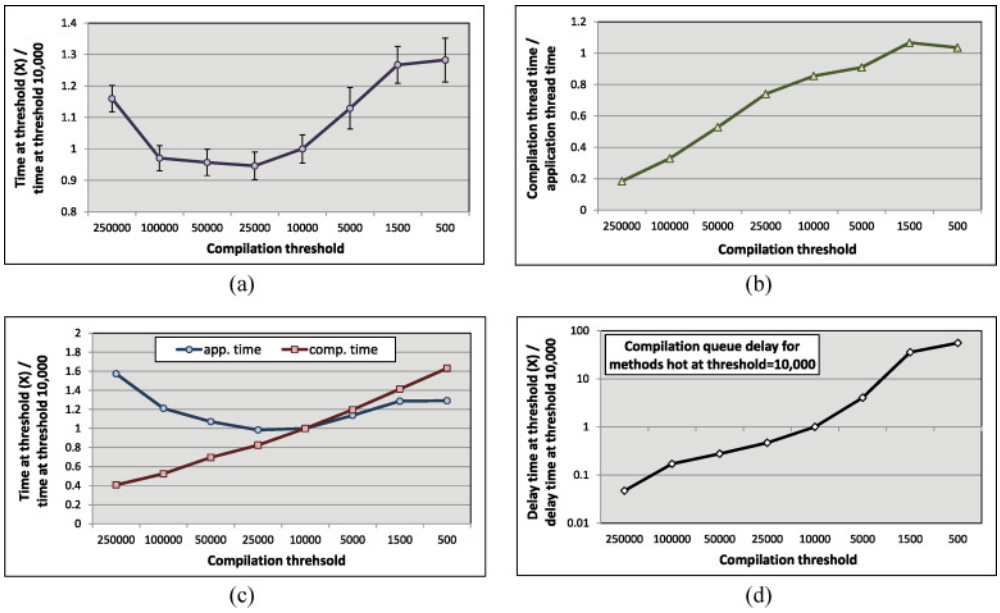


Fig. 2. Effect of different compilation thresholds on average benchmark performance on single-core processors.

performance at different compile thresholds to the average program performance at the threshold of 10,000, which is the default compilation threshold for the HotSpot server compiler. We find that a few of the less aggressive thresholds are slightly faster, on average, than the default for our set of benchmark programs (although the difference is within the margin of error). The default HotSpot server VM employs two compiler threads and may have been tuned with applications that run longer than our benchmarks, which may explain this result. The average benchmark performance worsens at both high and low compile thresholds.

To better interpret these results, we collect individual thread times during each experiment to estimate the amount of time spent doing compilation compared to the amount of time spent executing the application. Figure 2(b) shows the ratio of compilation to application thread times at each threshold averaged over all the benchmarks. Thus, compilation thresholds that achieve good performance spend a significant portion of their overall runtime doing compilation. We can also see that reducing the compilation threshold increases the relative amount of time spent doing compilation. However, it is not clear how much of this trend is due to longer compilation thread times (from compiling more methods) or reduced application thread times (from executing more native code).

Therefore, we also consider the effect of compilation aggressiveness on each component separately. Figure 2(c) shows the breakdown of the overall program execution in terms 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. We observe that high thresholds (>10,000) 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 (<10,000) 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.

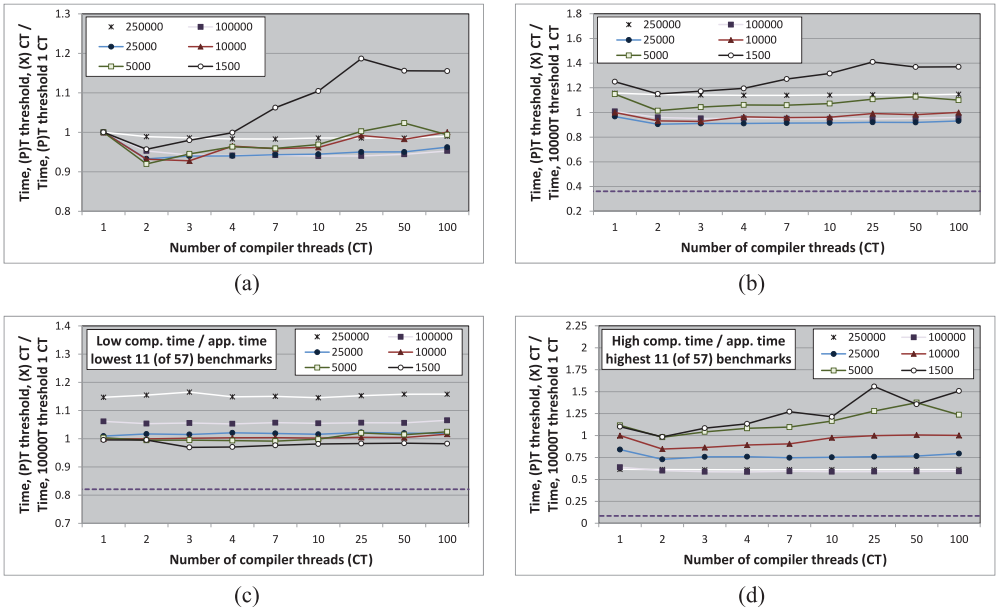


Fig. 3. Effect of multiple compiler threads on single-core program performance in the HotSpot VM with server compiler. The discrete measured thread points are plotted equidistantly on the x-axis.

On further analysis we found that JIT compilation policies with lower thresholds send more methods to compile and contribute to compiler queue backup. We hypothesize that the flood of less important program methods delays the compilation of the most critical methods, resulting in the nonintuitive degradation in application performance at lower thresholds. To verify this hypothesis, we conduct a separate set of experiments that *measure the average compilation queue delay (time spent waiting in the compile queue) of hot methods in our benchmarks.* These experiments compute the mean average compilation queue delay only for methods that are hot at the default threshold of 10,000 for each benchmark/compile threshold combination.

Figure 2(d) plots the average compilation queue delay at each compile threshold relative to the average compilation queue delay at the default threshold of 10,000 averaged over the benchmarks.[4] As we can see, the average compilation queue delay for hot methods increases dramatically as the compilation threshold is reduced. Thus, we conclude that increasing compiler aggressiveness is not likely to improve VM performance running with a single compiler thread on single-core machines.

4.2. Effect of Multiple Compiler Threads on Single-Core Machines

In this section, we analyze the effect of multiple compiler threads on program performance on a single-core machine with the server and tiered compiler configurations of the HotSpot VM.

4.2.1. Single-Core Compilation Policy with the HotSpot Server Compiler. For each compilation threshold, a separate plot in Figure 3(a) compares the average overall program

[4]We cannot compute a meaningful ratio for benchmarks with zero or very close to zero average compilation queue delay at the baseline threshold. Thus, these results do not include 14 (of 57) benchmarks with an average compilation queue delay less than 1msec (the precision of our timer) at the default threshold.

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. Indeed, we notice program performance improvements for one or two extra compiler threads, but the benefits do not hold with increasing number of such threads (>3). We further analyzed the performance degradation 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 VM compiler resources. While the increased compiler activity increases compilation overhead, many of these methods contribute little to improving program performance. Consequently, the potential improvement in application performance achieved by more compilations seems unable to recover the additional compiler overhead, resulting in a net loss in overall program performance.

Figure 3(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 10,000. These results reveal the best compiler policy on single-core machines with multiple compiler threads. Thus, we can see that, on average, the more aggressive thresholds perform quite poorly, while moderately conservative thresholds fare the best (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 server compiler in HotSpot 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.

To further evaluate the configurations with varying compilation resources and aggressiveness (in these and later experiments), we design an *optimal* scenario that measures the performance of each benchmark *with all of its methods precompiled*. Thus, the "optimal" configuration reveals the best-case benefit of JIT compilation. The dashed line in Figure 3(b) shows the optimal runtime on the single-core machine configuration relative to the same baseline startup performance (single benchmark iteration with one compiler thread and a threshold of 10,000), averaged over all the benchmarks. Thus, the "optimal" steady-state configuration achieves much better performance compared to the "startup" runs that compile methods concurrently with the running application on single-core machines. On average, the optimal performance is about 64% faster than the baseline configuration and about 54% faster than the fastest compilation thread/compile threshold configuration (with two compilation threads and a compile threshold of 25,000).

Figure 3(c) shows the same plots as in Figure 3(b) but only for the 11 (20%) benchmarks with the lowest compilation-to-application time ratio. Thus, for applications that spend relatively little time compiling, only the very aggressive compilation thresholds cause some compilation queue delay and may produce small performance improvements in some cases. For such benchmarks, all the hot methods are always compiled before program termination. Consequently, the small performance improvements with the more aggressive thresholds are due to compiling hot methods earlier (reduced queue delay). Furthermore, there is only a small performance difference between the startup and optimal runs. By contrast, Figure 3(d) only includes the 11 (20%) benchmarks with a relatively high compilation-to-application time ratio. For programs with such high compilation activity, the effect of compilation queue delay is more pronounced. We find that the less aggressive compiler policies produce better

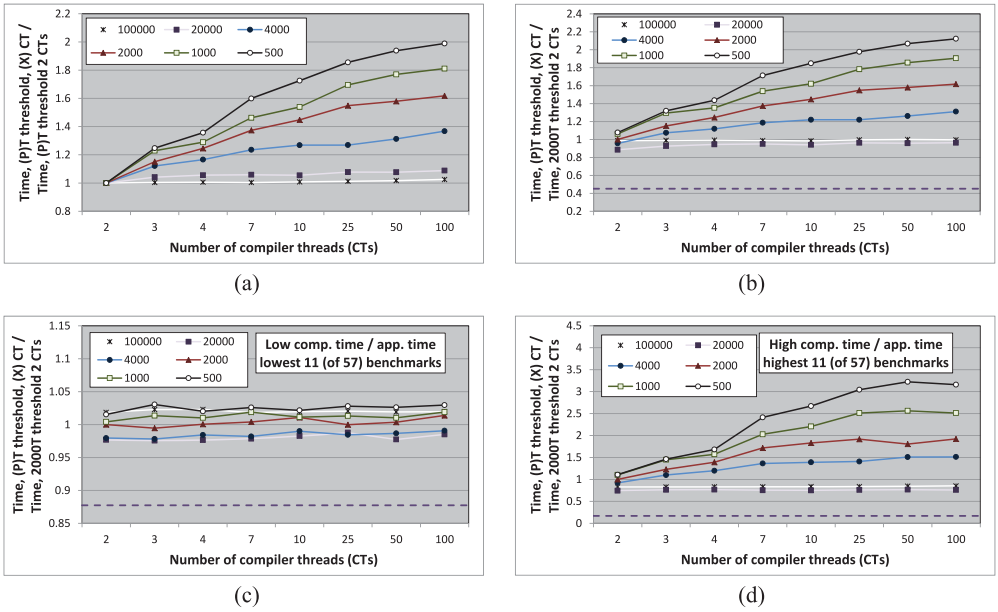


Fig. 4. Effect of multiple compiler threads on single-core program performance in the HotSpot VM with tiered compiler. The discrete measured thread points are plotted equidistantly on the x-axis.

efficiency gains for these programs, but there is still much room for improvement as evidenced by optimal performance results.

These observations suggest that a VM that can adapt its compilation threshold based on the compiler load may achieve the best performance for all programs on single-core machines. Additionally, implementing a priority queue to order compilations may also enable the more aggressive compilation thresholds to achieve better performance. We explore the effect of prioritized method compiles on program performance in further detail in Section 7. Finally, a small increase in the number of compiler threads can also improve performance by reducing the compilation queue delay.

4.2.2. Single-Core Compilation Policy with the HotSpot Tiered Compiler. In this section, we explore the effect on program performance of changing the compiler aggressiveness and the number of compiler threads with a tiered compiler configuration on single-core machines. For our experiments with the tiered compiler, we vary the client and server compiler thresholds in lock-step to adjust the aggressiveness of the tiered compiler and use the corresponding first-level (client) compiler threshold in the graph legends.

Each line-plot in Figure 4(a) compares the average overall program performance in the tiered compiler with multiple compiler threads to the average performance with only one client and one server compiler thread at that same threshold. In contrast to the server compiler configuration, increasing the number of compiler threads does not yield any performance benefit and, for larger increases, significantly degrades performance at every threshold. This effect is likely due to the combination of two factors: (a) a very fast first-level compiler that prevents significant backup in its compiler queue even with a single compiler thread while achieving most of the performance benefits of later recompilations, and (b) the priority heuristic used by the tiered compiler that may be able to find and compile the most important methods first. Thus, any additional compilations performed by more compiler threads only increase the compilation overhead without commensurately contributing to program performance. In Section 7.2,

we compare the default tiered compiler to one that employs FIFO (first-in first-out) compilation queues to evaluate the effect of prioritized compilation queues on program performance.

Figure 4(b) compares the average program performances in each case to the average performance of the baseline tiered configuration with one client and one server compiler thread and the default threshold parameters (with a client compiler threshold of 2,000). The default tiered compiler employs significantly more aggressive compilation thresholds compared to the default stand-alone server compiler and, on average, queues up more than three times as many methods for compilation. Consequently, relatively conservative compile thresholds achieve the best performance on single-core machines. The dashed line in Figure 4(b) plots the runtime of the optimal configuration (measured as described in the previous section) relative to the runtime of the baseline tiered configuration. Thus, with the tiered VM on single-core machines, the optimal runtime is still much faster than any other start-up configuration. However, due to the fast client compiler and effective priority heuristic, the performance of the tiered VM is significantly closer (10% in the best case) to the optimal runtime than the server VM configurations presented in the previous section.

We again observe that applications with extreme (very low or very high) compilation activity show different performance trends than the average over the complete set of benchmarks. Figure 4(c) plots the average performance of the HotSpot tiered compiler VM at different threshold and compiler thread configurations for the 20% benchmarks with the lowest compilation-to-application time ratio. As expected, compile threshold aggressiveness and the amount of compilation resources have much less of a performance impact on these applications. Additionally, the performance achieved is much closer to the optimal runtime for this set of benchmarks. However, in contrast to the server compiler results in Figure 3(c), some less aggressive thresholds are *marginally* more effective in the tiered compiler, which again indicates that the compilation queue delay is much less of a factor in the presence of a fast compiler and a good heuristic for prioritizing method compiles. Alternatively, in Figure 4(d), we study benchmarks with relatively high compilation activity and find that less aggressive compile thresholds yield very significant performance gains, due to a very aggressive default threshold used by the tiered compiler.

In summary, for single-core machines it is crucial to select the compiler threshold such that only the most dominant program methods are sent to compilation. With such an ideal compiler threshold, only two compiler threads (one client and one server) are able to service all compilation requests prior to program termination. An ideal threshold combined with a very fast client compiler and a good priority heuristic negates any benefit of additional compiler threads reducing the queue delay. A less aggressive threshold lowers program performance by not allowing the tiered VM to compile all hot methods. In contrast, with more aggressive thresholds, a minimum number of compiler threads are not able to service all queued methods, producing performance degradations due to the overhead of increased compiler activity with more compiler threads.

5. JIT COMPILATION ON MULTICORE MACHINES

Dynamic JIT compilation on single-processor machines has to be conservative to manage the compilation overhead at runtime. Modern multicore 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 multicore 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

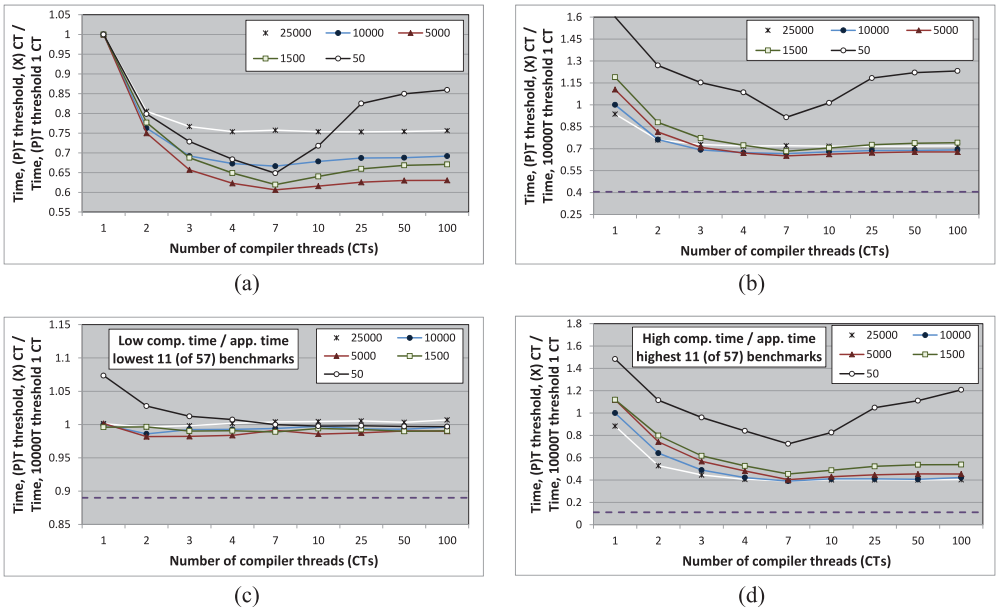


Fig. 5. Effect of multiple compiler threads on multicore application performance with the HotSpot Server VM.

report the impact of varying JIT compilation aggressiveness and the number of compiler threads on the performance of the server and tiered VM on multicore machines.

5.1. Multicore Compilation Policy with the HotSpot Server Compiler

Figure 5 illustrates the results of our experiments with the HotSpot server compiler on multicore machines. 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 57 benchmark-input pairs. Thus, we can see that increasing the number of compiler threads up to seven threads improves application performance at all compile thresholds. However, larger increases in the number of compiler threads (>7) derive no performance benefits and actually degrade performance with the more aggressive compilation thresholds.

As mentioned earlier, additional compiler threads can improve performance by reducing compilation queue delay, allowing the important program methods to be compiled earlier. Early compilation allows a greater fraction of the program execution to occur in optimized native code (rather than being interpreted), which produces significant gains in program performance. The additional compiler threads impose minimal impediment to the application threads as long as that computation can be off-loaded onto free (separate) cores. Our existing hardware setup only provides eight distinct processing cores. Consequently, larger increases in the number of compiler threads cause application and compilation threads to compete for machine resources. Moreover, configurations with aggressive compilation thresholds frequently compile methods that derive little performance benefit. This additional (but incommensurate) compilation overhead can only be sustained as long as compilation is free and results in significant performance losses in the absence of free computational resources.

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 compiler thread at the default HotSpot server compiler threshold of 10,000. We can see that while the optimal performance (again indicated by the dashed line) is much faster than the performance of the baseline configuration, increasing compilation activity on otherwise free compute cores enables the server VM to make up much of this difference. In the best case (configuration with threshold 5,000 and seven compiler threads), the combination of increased compiler aggressiveness with more compiler threads improves performance by 34.6%, on average, over the baseline. However, most of that improvement (roughly 33%) is obtained by simply reducing the compilation queue delay that is realized by increasing the number of compiler threads at the default HotSpot (10,000) threshold. Thus, the higher compiler aggressiveness achieved by lowering the selective compilation threshold seems to offer relatively small benefits over the more conservative compiler policies.

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 threshold of 50 performs extremely poorly compared to the default threshold with only one compiler thread (over 60% worse) and requires seven compiler threads to surpass the baseline performance.

As the compiler thresholds get more aggressive, we witness (from Figure 5(a)) successively larger performance losses with increasing the number of compiler threads beyond seven. These losses are due to increasing application interference caused by compiler activity at aggressive thresholds and are a result of the computational limitations in the available hardware. In Section 6, we construct a simulation configuration to study the behavior of aggressive compilation policies with large number of compiler threads on (many-core) machines with virtually unlimited computation resources.

Similar to our single-core configurations, we find that these results change dramatically depending on the compilation characteristics of individual benchmark applications. Figures 5(c) and 5(d) plot the average performance at each multicore compilation threshold and compiler thread configuration for 20% of the benchmarks with the lowest and highest compilation-to-application time ratios in our baseline configuration, respectively. Varying compilation thresholds and resources has much less of a performance effect on benchmarks that spend relatively little time doing compilation. The best configuration for these benchmarks (with a compilation threshold of 5,000 and two compiler threads) yields less than a 2% improvement over the baseline configuration. Also, for these benchmarks, the baseline configuration achieves performance that is much closer to optimal (again, indicated by the dashed line) compared to the overall average in Figure 5(b). Alternatively, for benchmarks that spend relatively more time doing compilation, as shown in Figure 5(d), there is even more room for improvement compared to the average over all benchmarks. As expected, exploiting the free processor resources to spawn additional compiler threads results in a more substantial performance benefit (an average efficiency gain of over 60%) for these 11 benchmarks.

5.2. Multicore Compilation Policy with the HotSpot Tiered Compiler

In this section, we evaluate the effect of varying compiler aggressiveness on the overall program performance delivered by the VM with its tiered compilation policy. The compilation thresholds for the two compilers in our tiered experimental configurations are varied in lock-step so that they always maintain the same ratio. For each compilation threshold setting with the tiered compiler, a separate plot in Figure 6(a) compares the average overall program performance with multiple compiler threads to the average performance with two (one client and one server) compiler threads at that same compilation threshold. In stark contrast to our results in Section 5.1 with

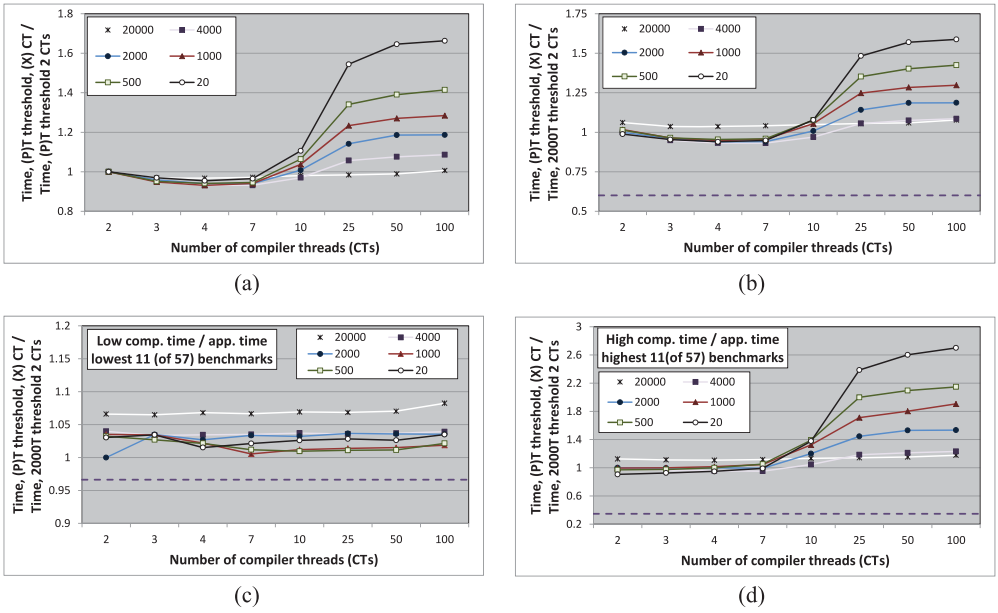
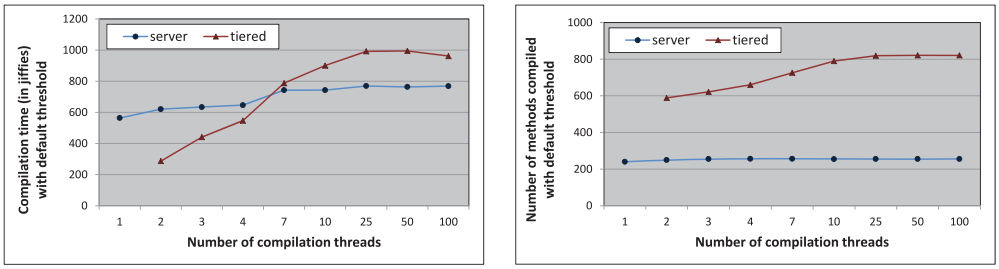


Fig. 6. Effect of multiple compiler threads on multicore application performance with the HotSpot tiered VM.

the server compiler, increasing the number of compiler threads for the tiered compiler only marginally improves performance at any compile threshold. This result is due to the speed and effectiveness of the HotSpot client compiler. As mentioned earlier in Section 3, the HotSpot client compiler imposes a very small compilation delay and yet generates code of only a slightly lower quality as compared to the much slower server compiler. Consequently, although the hot methods promoted to level 2 (server compiler) compilation may face delays in the compiler queue, its performance impact is much reduced since the program can still execute in level 1 (client compiler) optimized code. The small improvement in program performance with more compiler threads (up to seven) is again the result of reduction in the server compiler queue delay. However, overall we found that the compiler queue delay is much less of a factor with the tiered compilation policy.

Our results also show large and progressively more severe performance losses with increasing compilation threshold aggressiveness as we increase the number of compiler threads past the number of (free) available processing cores. In order to explain these performance losses, we extend our framework to report additional measurements of compilation activity. Figures 7(a) and 7(b) respectively show compilation thread times and the number of methods compiled with the server and tiered VMs with their respective default compile thresholds and with increasing numbers of compiler threads, averaged over all the benchmarks. We find that, due to its extremely fast client compiler, the tiered VM employs a much more aggressive compile threshold, which enables it to compile more methods more quickly, and often finish the benchmark run faster, than the server VM. However, with its multilevel compilation strategy, this often results in a situation with many methods remaining in the level 2 (server) compilation queues at the end of the program run. Increasing the number of compilation threads enables more of these methods to be (re-)compiled during the application run. This additional compilation (with lower program speed benefit returns) obstruct application progress as the number of threads is raised beyond the limits of available hardware.



(a) (b)

Fig. 7. Effect of multiple compiler threads on multicore compilation activity with the server and tiered VM.

Therefore, we conclude that *the number of compiler threads in the tiered VM should be set within the limits of available hardware in order to prevent sharp performance losses*.

Figure 6(b) presents the ratio of average program performance delivered by the VM with varying tiered compilation thresholds and compiler threads when compared to a single *baseline* performance (client compiler threshold of 2,000) with two (one client and one server) compiler threads. Thus, employing a small number of compilation threads ($<$10) typically achieves the best performance. This performance is much closer to optimal than the baseline performance with the server VM, although the server VM shows greater improvements as the number of compiler threads is increased.

Other trends in the graph in Figure 6(b) are similar to those presented in Figure 6(a) with the additional recommendation against employing overly conservative compiler policies on multi-core machines. Although conservative policies do a good job of reducing compilation overhead for single-core machines (Figure 4(b)), they can lower performance for multicore machines due to a combination of two factors: (a) not compiling all the important program methods and (b) causing a large delay in compiling the important methods. However, we also find that a wide range of compiler policies (from the client compiler thresholds of 4,000 to 20) achieve almost identical performance as long as compilation is free. This observation indicates that (a) not compiling the important methods (rather than the compiler queue delay) seems to be the dominant factor that can limit performance with the tiered compiler, and (b) compiling the less important program methods does not substantially benefit performance.

The plots in Figures 6(c) and 6(d) only employ the results of benchmarks that show very low or very high compilation activity, respectively, and are constructed similar to the graph in Figure 6(b) in all other aspects. These graphs reiterate the earlier observation that program characteristics greatly influence its performance at different compiler aggressiveness. For benchmarks with only a few very hot methods (Figure 6(c)), varying compiler thresholds has little to no effect on overall performance. And Figure 6(d) shows that the average trends noticed across all of our benchmarks in Figure 6(b) are exaggerated when considered only over the programs displaying high compilation activity. This graph again indicates that the delay in compiling the hot methods does not seem to be a major factor affecting program runtime when using tiered compilation with a fast and good level 1 compiler.

6. JIT COMPILATION ON MANY-CORE MACHINES

Our observations regarding aggressive JIT compilation policies on modern multicore machines in the last section were limited by our existing eight-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

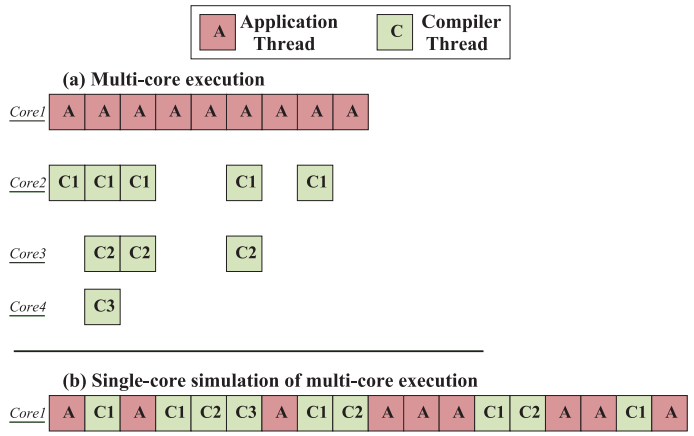


Fig. 8. Simulation of multicore VM execution on single-core processor.

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 experimental setup *estimates* 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 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 (unlimited cores) application execution on a single-core machine is illustrated in Figure 8. Figure 8(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 8(b). It is important to note that JIT compilations in our simulation of many-core VM execution (on a single-core machine) occur at about the same time relative to the application thread as on a physical many-core machine. Now, on a many-core machine, where each compiler thread runs on its own distinct core concurrently with the application thread, the total program runtime is equal to the application thread runtime alone, as understood from Figure 8(a). Therefore, our ability to precisely measure individual application thread times in our single-core simulation enables us to realistically emulate an environment

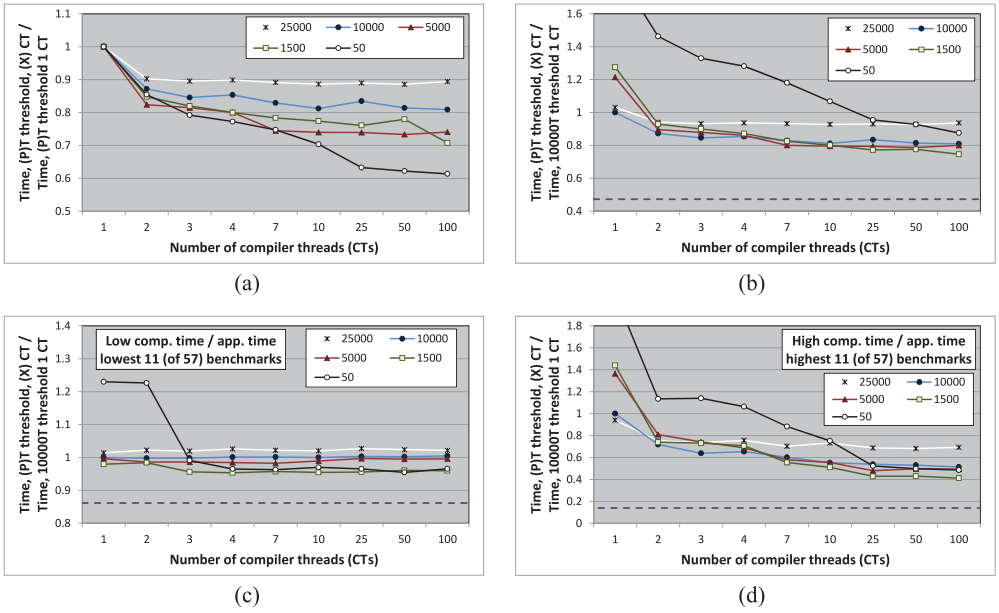


Fig. 9. Effect of multiple compiler threads on many-core application performance with the HotSpot server VM.

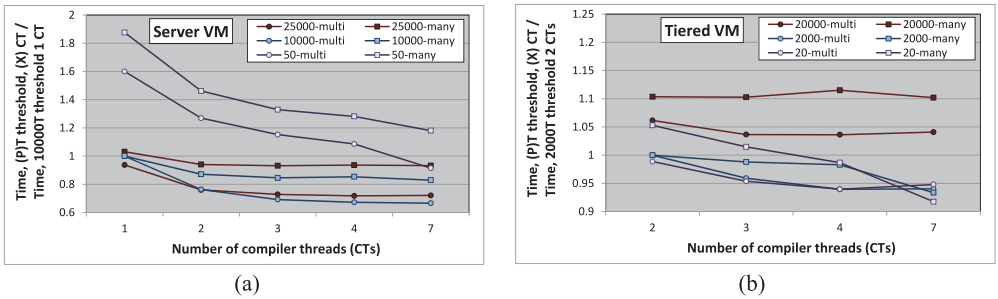


Fig. 10. Comparison of multi- and many-core performance results for the server and tiered VM.

where each thread has access to its own core. Note that, while this configuration is not by itself new, its application to measure "many-core" performance is novel. This framework, for the first time, 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.

6.1. Many-Core Compilation Policy with the HotSpot Server Compiler

We now employ our many-core experimental setup to conduct similar experiments to those done in Section 5.1. Figure 9 shows the results of these experiments and plots the average application thread times with varying number of compiler threads and compiler aggressiveness for all of our benchmark applications. These plots correspond with the graphs illustrated in Figure 5. In order to assess the accuracy of our simulation, we plot Figure 10(a), which shows a side-by-side comparison of a subset of the results for the multi- and many-core configurations with one to seven compiler threads. From these plots, we can see that the trends in these results are mostly consistent with our

observations from the last section for a small ($\leq$7) number of compiler threads. This similarity validates the ability of our simple simulation model to estimate the effect of JIT compilation policies on many-core machines, in spite of the potential differences between intercore communication, cache models, and other low-level microarchitectural effects.

Figure 9(a) shows that, unlike the multicore plots in Figure 5(a), given unlimited computing resources, application thread performance for aggressive compiler thresholds continues gaining improvements beyond a small number of compiler threads. Thus, the performance degradation for the more aggressive thresholds beyond about 7 to 10 compiler threads in the last section is, in fact, caused by the limitations of the underlying eight-core hardware. This result shows the utility of our novel setup to investigate VM properties for future many-core machines. From the results plotted in Figure 9(b), we observe that the more aggressive compilation policies eventually (with $>$10 compiler threads) yield performance gains over the baseline server compiler threshold of 10,000 with one compiler thread. Additionally, we note that the difference between the baseline configuration and the optimal performance (indicated by the dashed line in Figure 9(b)) with our many-core simulation is similar to our results with multicore machines.

We also find that isolating and plotting the performance of benchmarks with relatively small or relatively large amounts of compilation activity in our many-core configuration shows different trends than our complete set of benchmarks. As shown in Figure 9(c), increasing the number of compiler threads for benchmarks that spend relatively little time compiling does not have a significant impact on performance at any threshold. At the same time, early compilation of the (small number of) hot methods reduces the benchmark runtimes at aggressive compilation thresholds. Alternatively, as seen from Figure 9(d), benchmarks with more compilation activity tend to show even starker performance improvements with increasing number of compiler threads. This result makes intuitive sense, as applications that require more compilation yield better performance when we allocate additional compilation resources. Comparing the optimal performance over each set of benchmarks, we find that our many-core experiments show trends that are similar to our previous results—benchmarks with relatively little compilation activity achieve performance that is much closer to optimal, while benchmarks with relatively high compilation activity have more room for performance improvement.

6.2. Many-Core Compilation Policy with the HotSpot Tiered Compiler

In this section we analyze the results of experiments that employ our many-core framework to estimate the average runtime behavior of programs for the tiered compiler strategy with the availability of unlimited free compilation resources. The results in this section enable us to extend our observations from Section 5.2 to many-core machines. The results in Figures 11(a) and 11(b) again reveal that, in a wide range of compilation thresholds, changing compiler aggressiveness has a smaller effect on performance as compared to the single-level server compiler. As we expect, a side-by-side comparison of the multi- and many-core results in Figure 10(b) shows that the trends in these results are mostly consistent with the results in Section 5.2 for a small number of compiler threads. The most distinctive observation that can be made from our many-core experiments is that, given sufficient computing resources, there is no significant performance loss with larger numbers of compiler threads since all compilation activity is considered free. Unfortunately, as noticed with the multi-core results, increasing the number of compiler threads is likely to only produce a modest impact on program runtime with the tiered compiler. This observation again indicates that techniques to

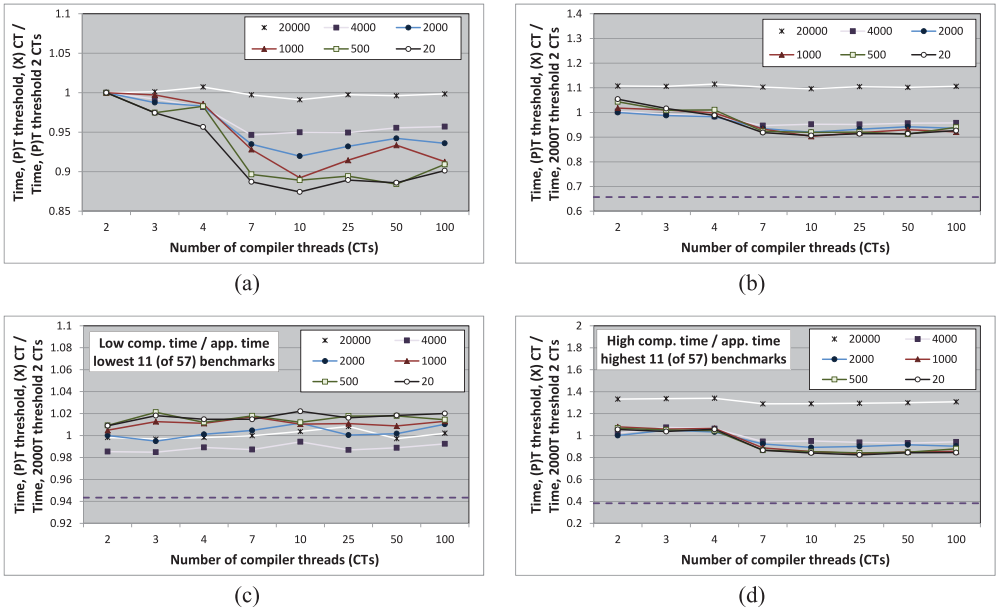


Fig. 11. Effect of multiple compiler threads on many-core application performance with the HotSpot tiered VM.

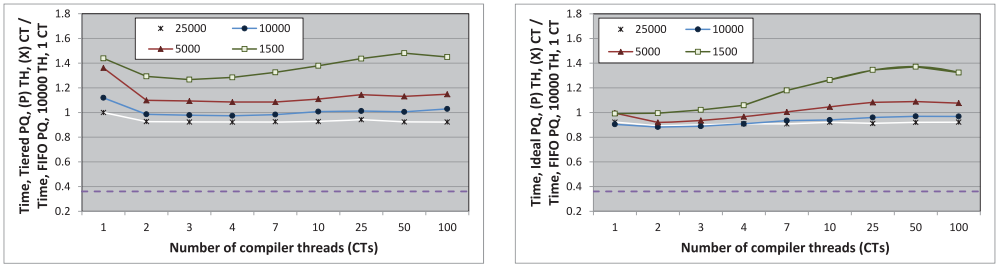
reduce the delay in compiling hot methods are not as effective to improve runtime performance with the tiered compiler.

Figures 11(c) and 11(d) plot the same program runtime ratio as Figure 11(b), but only for benchmarks that have a very low or very high compilation activity. As observed in all similar plots earlier, benchmarks with low compilation activity have only a small number of very active methods, and all compiler aggressiveness levels produce similar performances. Benchmarks with a high compilation load mostly exaggerate the trends noticed over all benchmarks (Figure 11(b)). Very conservative compiler thresholds cause large performance losses for such benchmarks. Additionally, with free compiler resources, higher compiler aggressiveness can produce marginally better results than the default threshold for such benchmarks by compiling and optimizing a larger portion of the program.

7. EFFECT OF PRIORITY-BASED COMPILER QUEUES

Aggressive compilation policies can send a lot of methods to compile, which may back up the compile queue. Poor method ordering in the compiler queue may result in further delaying the compilation of the most important methods, as the VM spends its time compiling the less critical program methods. Delaying the generation of optimized code for the hottest methods will likely degrade application performance. An algorithm to effectively prioritize methods compiles may be able to nullify the harmful effects of backup in the compiler queue. In this section, we study the effect of different compiler queue prioritization schemes on the server and tiered compiler configurations.

We present results of experiments with three different priority queue implementations. The first-in first-out (FIFO) queue implementation is the default strategy employed by the HotSpot server compiler that compiles all methods in the order they are sent for compilation by the application threads. By default, the HotSpot *tiered* compiler uses a heuristic priority queue technique for ordering method compiles. When selecting a method for compilation with this heuristic, the tiered compiler computes



(a) (b)

Fig. 12. Performance of the tiered and ideal compiler priority algorithms over FIFO for HotSpot server compiler on single-core machines.

an *event rate* for every eligible method and selects the method with the maximum event rate. The event rate is simply the sum of invocation and back-edge counts per millisecond since the last dequeue operation. We modified the HotSpot VM to make the *FIFO* and *tiered* queue implementations available to the multistage tiered and single-stage server compilers, respectively.

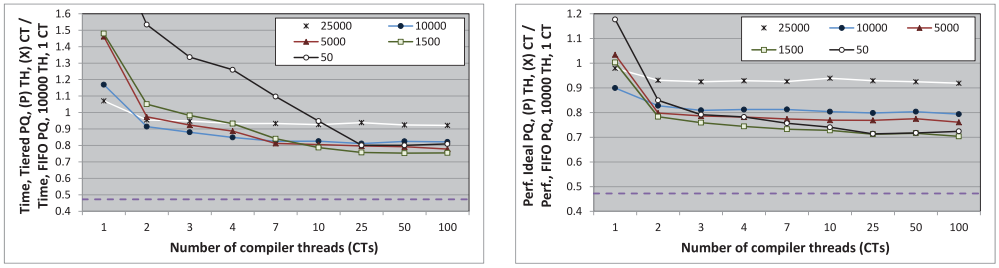
Both the FIFO and tiered techniques for ordering method compiles use a completely online strategy that only uses past program behavior to detect hot methods to compile to speed up the remaining program run. Additionally, the more aggressive JIT compilation policies make their hotness decisions earlier, giving any online strategy an even reduced opportunity to accurately assess method priorities. Therefore, to set a suitable goal for the online compiler priority queue implementations, we attempt to construct an *ideal* strategy for ordering method compilations. An ideal compilation strategy should be able to precisely determine the actual hotness level of all methods sent to compile, and always compile them in that order. Unfortunately, such an ideal strategy requires knowledge of future program behavior.

In lieu of future program 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. Thus, our ideal priority queue strategy requires a profile run of every benchmark to determine its method hotness counts. We collect these total method hotness counts during this previous program run and make them available to the ideal priority algorithm in the measured run. We do note that even our ideal profile-driven strategy may not achieve the *actual* best results because the candidate method with the highest hotness level may still not be the best method to compile at *that* point during program execution.

7.1. Priority-Based Compiler Queues in the HotSpot Server Compiler

Our results in the earlier sections suggest that the relatively poor performance achieved by aggressive JIT compilation policies in the server compiler 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.

7.1.1. Single-Core Machine Configuration. Figures 12(a) and 12(b) show the performance benefit of the tiered and ideal compiler priority queue implementations, respectively. Each line in these graphs is plotted relative to the default FIFO priority queue implementation with a single compiler thread at the default threshold of 10,000 on single-core machines. These graphs reveal that the online tiered prioritization technique is



(a) (b)

Fig. 13. Performance of the tiered and ideal compiler priority algorithms over FIFO for HotSpot server compiler on many-core machines.

not able to improve performance over the simple FIFO technique and actually results in a performance loss for a few benchmarks. In contrast, the VM performance with the ideal prioritization scheme shows that accurate assignment of method priorities is important and allows the smaller compile thresholds to also achieve relatively good average program performance for a small number of compiler threads.

We had also discovered (and reported in Section 4.2) that 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 overall performance loss. We find that 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 better performance than the best achieved with the FIFO queue.

7.1.2. Many-Core Machine Configuration. Figures 13(a) and 13(b) compare the performance results of using our tiered and ideal compiler priority queue, respectively, with a baseline VM that uses the simple FIFO-based compiler queue implementation with the compile threshold of 10,000 for many-core machines. The results with the ideal priority queue implementation show that 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.

We also find that 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 configuration, and by about 9% over the best performance achieved by the server VM's default compilation threshold of 10,000 with any number of compiler threads. Unfortunately, the online tiered

prioritization heuristic is again not able to match the performance of the ideal priority queue. Thus, more research may be needed to devise better online priority algorithms to achieve the most effective overall program performance on modern machines.

7.2. Priority-Based Compiler Queues in the HotSpot Tiered Compiler

As explained earlier, the HotSpot tiered configuration uses a simple and fast priority heuristic to order methods in the queue. In this section, we describe the impact of FIFO, tiered (default), and ideal prioritization algorithms for all the compiler queues in the tiered configuration.

We find that prioritizing method compiles has no significant effect on program performance at any compiler aggressiveness for the tiered compiler on all machine configurations. These results suggest that the program performance behavior with the tiered compilers is dominated by the very fast HotSpot client compiler that generates good code quality without causing a significant compiler queue backup. The very hot methods that are promoted to be further optimized by the server compiler do take time and cause queue backup. The larger server compiler overhead increases program runtime on single-core machines, but not on the many-core machines where compilation is free.

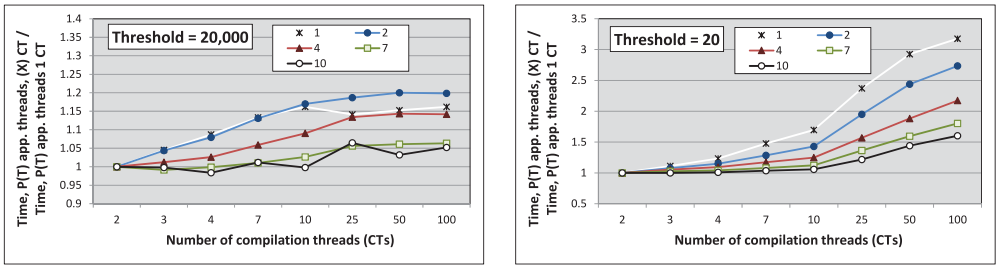
8. EFFECT OF MULTIPLE APPLICATION THREADS

All our earlier experiments were primarily conducted with single-threaded benchmarks. However, real-world applications widely vary in the number and workload of concurrent application threads. In this section, we explore the effect of compiler aggressiveness and resources on the performance of benchmarks with different numbers of application threads.

Experiments in this section were conducted using 16 SPECjvm2008 benchmarks (all except *derby*).[5] Our other benchmark suites do not allow an easy mechanism to uniformly and precisely control the number of spawned application threads. While SPECjvm98 only permits runs with a fixed number of application threads, many of the DaCapo programs spawn a variable number of *internal* threads that cannot be controlled by the harness (see Table II). For each VM configuration, we selected three compilation thresholds (least aggressive, most aggressive, and default) and ran the benchmarks with a different number of compiler and application threads to understand their interactions. Results with the server and tiered VMs show similar application–compiler thread interaction trends, and therefore, we only report results with the tiered VM in this section.

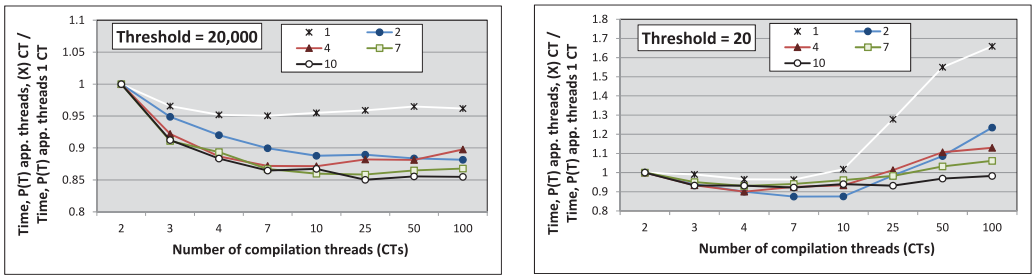
Figures 14(a) and 14(b) show the results of these experiments with the tiered VM on our single-core configuration for the least (CT = 20,000) and most aggressive (CT = 20) compile thresholds, respectively. Each line in the figures (plotted for a specific number of application threads as indicated in the legend) shows the ratio of program speed with a different number of compiler threads to the performance with a single compiler thread *and that same number of application threads*. A separate baseline for each application thread configuration is necessitated because SPECjvm2008 employs a different workload size for each such setting. As explained previously in Section 4.2.2 (Figure 4), Figure 14 again shows that a fast and high-quality level 1 compiler that doesn't cause much queue delay is responsible for the loss in program performance with one (or few) application threads as the number of compiler threads is increased. However, we now observe that the magnitude of this loss diminishes as the number of application threads is increased. We believe this effect is caused by the change in relative

[5]SPECjvm2008 automatically scales the benchmark workload in proportion to the number of application threads. This causes the *derby* benchmark with a greater number of application threads to often fail with an out-of-memory error.



(a) (b)

Fig. 14. Effect of different numbers of application threads on single-core performance with the HotSpot Tiered VM.



(a) (b)

Fig. 15. Effect of different numbers of application threads on multi-core performance with the HotSpot Tiered VM.

application thread interference that is due to the compiler threads servicing less important program methods with limited hardware resources. In other words, the likelihood of the OS scheduler selecting an application thread is smaller when there are fewer application threads in the run queue. Thus, adding application threads diminishes the relative loss in performance by increasing the likelihood that an application thread runs.

Another interesting trend we noticed in both the server and tiered VMs can be seen from the multicore experiments displayed in Figure 15. We find that (especially at less aggressive compile thresholds) configurations with larger numbers of application threads tend to show slightly larger improvements as the number of compiler threads is increased. In programs with multiple application threads, the output of a compilation may be used simultaneously in threads executing in parallel. Thus, early compilations may benefit more of the application's computation in comparison to single-threaded applications.

We found that our many-core experiments also demonstrate very similar trends. However, in the absence of interference to the application threads with unlimited computation resources, we do not find any significant performance losses with larger numbers of compiler threads. These results also support the observation that configurations with more application threads tend to show more pronounced performance improvements from early compilation.

9. CONCLUSIONS

Many virtual machines now allow the concurrent execution of multiple compiler threads to exploit the abundant computing resources available in modern processors to improve overall program performance. It is expected that more aggressive JIT

compilation strategies may be able to lower program runtime by compiling and optimizing more program methods early. The goal of this work is to explore the potential performance benefit of more aggressive JIT compilation policies for modern multi-/many-core machines and VMs that support multiple simultaneous compiler threads. We explore the properties of two VM compiler configurations: a single-level highly optimizing (but slow) *server* compiler and a multilevel *tiered* compiler. The HotSpot tiered compiler uses a very fast (but lightly optimizing) *client* compiler at the first stage and a powerful SSA-based (*server*) compiler for recompiling the very hot methods. Due to its recompilations, the tiered compiler induces a higher compilation load compared to the single-level server compiler. Our experiments vary the hotness thresholds to control compiler aggressiveness and employ different numbers of concurrent compiler threads to exploit free computation resources. We also develop a novel experimental framework to evaluate our goal for future many-core processors.

Results from our experiments allow us to make several interesting observations:

(1) Properties of the tiered compiler are largely influenced by the fast client compiler that is able to obtain most of the performance benefits of the slower server compiler at a small fraction of the compilation cost.
(2) Program features, in particular the ratio of hot program methods, impact their performance behavior at different compiler aggressiveness levels. We found that programs with a low compilation-to-application time ratio are not significantly affected by varying compiler aggressiveness levels or by spawning additional compiler threads.
(3) On single-core machines, compilation can impede application progress. The best compilation policy for such machines seems to be an aggressiveness level that only sends as many methods to compile as can be completely serviced by the VM in one to two compiler threads for most benchmarks.
(4) For machines with multiple cores, methods compiled by a moderately aggressive compile threshold are typically sufficient to obtain the best possible performance. Compiling any more methods quickly results in diminishing returns and very minor performance gains.
(5) Reducing the compiler queue delay and compiling early is more important to program performance than compiling all program methods. Spawning more compiler threads (at a given compilation aggressiveness) is an effective technique to reduce the compiler queue delay.
(6) On single-core and multicore machines, an excessive amount of compiler activity and threads may impede program progress by denying the application threads time to run on the CPU. This effect is reduced as the ratio of application to compiler threads increase. We also saw evidence that benchmarks with more application threads tend to show slightly sharper improvements from early compilation.
(7) The tiered VM, with its extremely fast client compiler, is able to compile methods more quickly with fewer compilation resources and typically outperforms the server VM on single-core machines or when the number of compiler threads is kept at a minimum. Alternatively, when there are additional computing resources available, increasing the number of compiler threads can sharply improve performance with the server VM.
(8) Effective prioritization of method compiles is important to find and compile the most important methods early, especially for the slow, powerful, highly optimizing compilers. However, additional research is necessary to find good *online* prioritization algorithms.
(9) Initiating more compiler threads than available computing resources typically hurts performance.

Based on these observations, we make the following recommendations regarding a good compilation policy for modern machines: (a) JIT compilers should use an adaptive compiler aggressiveness based on availability of free computing resources. At the very least, VMs should employ two (sets of) compiler thresholds, a conservative (large) threshold when running on single-core processors and a moderately aggressive (small) threshold on multi-/many-core machines. (b) Spawn as many compiler threads as available free compute cores (and as constrained by the specific power budget). (c) Employ an effective priority queue implementation to reduce the compilation queue delay for the slower compilers. (d) The more complex tiered compilation strategies used in some of the existing state-of-the-art JVMs can achieve the most effective startup program performance with minimal compiler threads and little need for effective prioritization of method compiles. We believe that our comprehensive research will guide future VM developers in making informed decisions regarding how to design and implement the most effective JIT compilation policies to achieve the best application performance.

10. 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 will allow 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. More native code produced by aggressive JIT compilation can raise memory pressure and garbage collection overheads, which may then affect program nondeterminism due to the increased pause times associated with garbage collections. Second, in this article we explored some priority queue implementations that may be more suitable for aggressive compilation policies, especially with a slow, highly optimizing compiler. We plan to continue our search for better method prioritization schemes in the future. Third, this work shows that the optimal settings for compilation threshold and the number of compiling threads depend on factors, such as application characteristics, that cannot be determined statically. Thus, we plan to conduct experiments to study the performance potential of adaptively scaling these parameters at runtime. Fourth, this work primarily focuses on exploring *if* and *when* to compile program methods to maximize overall program performance for modern machines. We do consider some aspects of *how* to compile with the tiered HotSpot configuration. However, how to compile program regions is a much broader research topic that includes issues such as better *selection* and *ordering* of optimizations at different compilation levels. We have begun to conduct research to address some of these issues in the HotSpot VM [Jantz and Kulkarni 2013]. We plan to exploit the observations from this work to focus on optimizations (and methods) with the greatest impact on program performance and build new and more effective online models. Finally, we are currently also conducting similar experiments in other virtual machines (JikesRVM) to see if our conclusions from this work hold across different VMs. Later, we plan to also validate our results for different processor architectures.

REFERENCES

Arnold, M., Fink, S., Grove, D., Hind, M., and Sweeney, P. F. 2000a. Adaptive optimization in the Jalapeno JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 47–65.

Arnold, M., Fink, S., Grove, D., Hind, M., and Sweeney, P. F. 2000b. 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)*.

Arnold, M., Fink, S., Grove, D., Hind, M., and Sweeney, P. F. 2005. A survey of adaptive optimization in virtual machines. *Proc. IEEE 92*, 2, 449–466.

Arnold, M., Hind, M., and Ryder, B. G. 2002. Online feedback-directed optimization of Java. *SIGPLAN Not. 37*, 11, 111–129.

Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. 2006. 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)*. 169–190.

Böhm, I., von Koch, T. J. K. E., Kyle, S. C., Franke, B., and Topham, N. 2011. 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)*. 74–85.

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

Cavazos, J. and O'Boyle, M. F. P. 2006. Method-specific dynamic compilation using logistic regression. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*. ACM, New York, 229–240.

Chang, P. P., Mahlke, S. A., and Hwu, W. m W. 1991. Using profile information to assist classic code optimizations. *Software Prac. Experience 21,* 1301–1321.

Deutsch, L. P. and Schiffman, A. M. 1984. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'84)*. ACM, New York, 297–302.

Esmaeilzadeh, H., Cao, T., Xi, Y., Blackburn, S. M., and McKinley, K. S. 2011. Looking back on the language and hardware revolutions: Measured power, performance, and scaling. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, 319–332.

Georges, A., Buytaert, D., and Eeckhout, L. 2007. Statistically rigorous java performance evaluation. In *Proceedings of the Conference on Object-Oriented Programming Systems and Applications (OOPSLA'07)*. 57–76.

Gosling, J., Joy, B., Steele, G., and Bracha, G. 2005. *The Java(TM) Language Specification* 3rd Ed. http://dl.acm.org/citation.cfm?id=1036643.

Graham, S. L., Kessler, P. B., and Mckusick, M. K. 1982. Gprof: A call graph execution profiler. *SIGPLAN Not. 17*, 6, 120–126.

Grcevski, N., Kielstra, A., Stoodley, K., Stoodley, M., and Sundaresan, V. 2004. 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*. 12.

Gu, D. and Verbrugge, C. 2008. Phase-based adaptive recompilation in a JVM. In *Proceedings of the 6th IEEE/ACM Symposium on Code Generation and Optimization (CGO'08)*. 24–34.

Haneda, M., Knijnenburg, P. M. W., and Wijshoff, H. A. G. 2005. Generating new general compiler optimization settings. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS'05)*. 161–168.

Hansen, G. J. 1974. *Adaptive Systems for the Dynamic Run-time Optimization of Programs*. Ph.D. Dissertation. Carnegie-Mellon Univ., Pittsburgh, PA.

Harris, T. 1998. Controlling run-time compilation. In *Proceedings of the IEEE Workshop on Programming Languages for Real-Time Industrial Applications*. 75–84.

Hazelwood, K. and Grove, D. 2003. Adaptive online context-sensitive inlining. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*. IEEE Computer Society, Washington, DC, 253–264.

Hölzle, U. and Ungar, D. 1996. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst. 18*, 4, 355–400.

Jantz, M. R. and Kulkarni, P. A. 2013. Performance potential of optimization phase selection during dynamic JIT compilation. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEEE'13)*.

KNUTH, D. E. 1971. An empirical study of FORTRAN programs. *Software: Pract. Experience 1*, 2, 105–133.

KOTZMANN, T., WIMMER, C., MÖSSENBÖCK, H., RODRIGUEZ, T., RUSSELL, K., AND COX, D. 2008. Design of the Java HotSpot™ client compiler for Java 6. *ACM Trans. Archit. Code Optim. 5*, 1, 1–32.

KRINTZ, C. 2003. Coupling on-line and off-line profile information to improve program performance. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*. Washington, DC, 69–78.

KRINTZ, C. AND CALDER, B. 2001. Using annotations to reduce dynamic optimization time. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. 156–167.

KRINTZ, C., GROVE, D., SARKAR, V., AND CALDER, B. 2000. Reducing the overhead of dynamic compilation. *Software: Pract. Experience 31*, 8, 717–738.

KULKARNI, P., ARNOLD, M., AND HIND, M. 2007. Dynamic compilation: The benefits of early investing. In *VEE '07: Proceedings of the 3rd International Conference on Virtual Execution Environments*. 94–104.

KULKARNI, P. A. 2011. JIT compilation policy for modern machines. In *Proceedings of the 2011 ACM International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'11)*. 773–788.

KULKARNI, P. A. AND FULLER, J. 2011. JIT compilation policy on single-core and multi-core machines. In *Proceedings of the 15th Workshop on Interaction between Compilers and Computer Architectures (INTERACT '11)*. 54–62.

MICROSOFT. 2001. *Microsoft C# Language Specifications* 1st Ed. Microsoft Press.

NAMJOSHI, M. A. AND KULKARNI, P. A. 2010. Novel online profiling for virtual machines. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'10)*. 133–144.

PALECZNY, M., VICK, C., AND CLICK, C. 2001. The Java hotspot TM server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium (JVM'01)*. USENIX Association, Berkeley, CA, 1–12.

SANCHEZ, R. N., AMARAL, J. N., SZAFRON, D., PIRVU, M., AND STOODLEY, M. 2011. Using machines to learn method-specific compilation strategies. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'11)*. 257–266.

SMITH, J. AND NAIR, R. 2005. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

SPEC2008. 2008. SPECjvm2008 Benchmarks. http://www.spec.org/jvm2008/.

SPEC98. 1998. SPECjvm98 Benchmarks. http://www.spec.org/jvm98/.

SUNDARESAN, V., MAIER, D., RAMARAO, P., AND STOODLEY, M. 2006. 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)*. 87–97.

WHITFIELD, D. L. AND SOFFA, M. L. 1997. An approach for exploring code improving transformations. *ACM Trans. Program. Lang. Syst. 19*, 6, 1053–1084.

Received May 2012; revised February 2013; accepted August 2013