# JIT Compilation Policy on Single-Core and Multi-core Machines

Prasad A. Kulkarni and Jay Fuller
Department of Electrical Engineering and Computer Science
University of Kansas, Lawrence, KS
{prasadk, jfuller8}@ku.edu

## Abstract

*Dynamic or Just-in-Time (JIT) compilation is crucial to achieve acceptable performance for applications written in traditionally interpreted languages, such as Java and C#. Such languages enable the generation of* portable *applications that are written and compiled once, and can be executed by a* virtual machine *on any supported architecture. However, by virtue of occurring at runtime, dynamic compilation adds to the overall execution time of the application, and can potentially slow down program execution if performed injudiciously. Selective compilation is a technique that was developed for single-core architectures to manage the compilation overhead by dynamically determining and compiling only the most critical program regions.*

*Current processors have evolved from single-core machines to those supporting multiple tightly-integrated cores. Consequently, research is needed to explore the best JIT compilation policy on multi-core machines with several concurrent compiler threads. In this paper, we present novel experimental configurations to understand the benefits of dynamic compilation, and find the best JIT compilation policies on single and multi-core machines. Our results validate earlier claims that compiling a small percentage of methods has an inordinately large impact on performance. We show that spawning a greater number of simultaneous compiler threads can achieve better program performance on single-core machines. Our results also reveal that more than an increase in compiler aggressiveness, a small increase in the number of compiler threads achieves the best application performance on multi-core machines.*

## 1. Introduction

*Managed* languages such as Java [8] and C# [22] support the 'compile-once, run-anywhere' model for code generation and distribution. This model allows the generation of programs that can be portably distributed and executed on any device equipped with the corresponding virtual machine (VM). The portability constraint limits the format of the distributed program to a form that is independent of any specific processor architecture. Since the program binary format does not match the native architecture, VMs have to employ either interpretation or dynamic compilation before executing the program. Additionally, interpreted execution is inherently slow, which makes dynamic or Just-in-Time (JIT) compilation essential to achieve efficient runtime performance for such applications.

However, by operating at runtime, JIT compilation contributes to the overall execution time of the application and, if performed injudiciously, may result in further worsening the execution or *response time* of the program. *Selective compilation* is a technique that was invented by researchers to address this issue with dynamic compilation [13, 24, 19, 3]. 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 [15, 5, 3]. Selective compilation uses online profiling to detect this subset of *hot* methods to compile at program startup, and thus limits the overhead of JIT compilation while still deriving the most performance benefit at runtime.

Thus, dynamic JIT compilation attempts to improve program efficiency, while minimizing application pause times (interference). Most of the theory substantiating the best JIT compilation policies was developed for VMs with a single compiler thread running on single-processor machines. Dynamic compilation policies in such environments are necessarily conservative. However, modern hardware architectures now integrate multiple cores on a single processor die. Moreover, hardware researchers and processor manufacturers expect to continuously scale the number of cores available in future processor generations [1]. Thus, modern architectures allow the possibility of running the compiler thread on a separate core to minimize interference with the application thread. At the same time, recent state-of-the-art virtual machines also provide support for multiple compiler

threads to be active at the same time. Such evolution in the hardware and VM contexts may demand different compilation strategies to be most effective on both multi-core, as well as single-core machines.

The objective of this research is to develop, investigate, and compare new dynamic compilation strategies to find the most effective policy in the presence of multiple compiler threads on single-core and multi-core machines. In this paper, along with the experimental results evaluating competing JIT compilation policies, we also explain the novel configurations we developed to conduct our tests. Moreover, we not only explore the most effective dynamic compilation policies for existing machines with a single or small number of processor cores, but only study how such policies need to evolve for future machines with increasing number of available cores. Thus, the major contributions of this research work are the following:

1. We describe novel experimental configurations to study the benefits of JIT compilation and determine the most effective compilation policies on future multi-core and many-core machines.
2. We illustrate how *steady-state* program performance improves with increasing compiler aggressiveness.
3. We present experimental results that show the impact of multiple compiler threads on single-core machines.
4. We explain how different JIT compilation strategies interact on multi-core machines, with small and large number of free available cores.

## 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 [6] followed by the Self-93 VM [13] pioneered many of the adaptive optimization techniques employed in current virtual machines, including selective compilation with multiple compiler threads. Most current VMs employ selective compilation with a *staged* emulation model [11]. With this model, each method is initially interpreted or compiled with a fast non-optimizing compiler at program start to improve application response time. Later, the virtual machine attempts to determine the subset of hot methods to selectively compile, and then compiles them at higher levels of optimization to achieve better program performance.

Unfortunately, selecting the hot methods to compile requires *future* program execution information, which is hard to accurately predict [23]. 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 [9, 16, 2]. Online profiling is used to detect these current hot methods. The most popular online profiling approaches are based on instrumentation *counters* [11, 13, 16], interrupt-timer-based *sampling* [2], or a combination of the two methods [9]. Profiling using counters requires the virtual machine to count the number of invocations and loop back-edges for each method. Sampling is used to periodically interrupt the application execution and update a counter for the method(s) on top of the stack. The method/loop is sent for compilation if the respective method counters exceed a fixed threshold.

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

The theoretical basis for tuning compiler thresholds is provided by the *ski-renting* principle [7, 14], which states that to minimize the worst-case damage of online compilation, a method should only be compiled after it has been interpreted a sufficient number of times so as to already offset the compilation overhead [23]. By this principle, a (slower) compiler with more/better optimization phases will require a higher compilation threshold to achieve the best overall program performance in a virtual machine.

Current implementations of selective compilation suffer from several drawbacks. One major issue is the delay in making the compilation decisions at program startup. One component of this delay is caused by the VM waiting for the method counters to reach the compilation *threshold* before deeming the method as hot and *queuing* it for compilation. Another factor contributing to the compilation delay occurs as each compilation request waits in the compilation queue to be serviced by a free compiler thread. This delay in compiling/optimizing hot methods results in poor application startup performance as the program spends more time executing in unoptimized code [21, 17, 10].

Researchers have suggested strategies to address the first delay component for online compilation. Krintz and Calder explored mechanisms that employ offline profiling and classfile annotation to send hot methods to compile early [18, 17]. However, such mechanisms require an additional profiling pass, and are therefore not generally applicable. Namjoshi and Kulkarni propose a technique that

can dynamically determine loop iteration bounds to *predict* future hot methods and send them to compile earlier [23]. Their suggested implementation requires additional computational resources to run their more expensive profiling stage. Gu and Verbruggee use online phase detection to more accurately estimate recompilation levels for different hot methods to save redundant compilation overheads and produce better code faster [10].

Researchers have also explored techniques to address the second component of the compilation delay that happens due to the backup and wait time in the method compilation queue. IBM's J9 virtual machine uses thread priorities to increase the priority of the compiler thread on operating systems, such as AIX and Windows, that provide support for user-level thread priorities [27]. Another technique attempts to increase the CPU utilization for the compiler thread to provide faster service to the queued compilation requests [21, 12]. However, the proposed thread-priority based implementations for these approaches can be difficult to provide in all existing operating systems.

Most importantly, all the studies described above have mostly been performed on single-core machines. There exist very few studies that explore JIT compilation issues for multi-core machines. Krintz et al. investigated the impact of background compilation in a separate thread to reduce the overhead of dynamic compilation [20]. However, this technique used a single compiler thread and also employed offline profiling to determine and prioritize hot methods to compile. Kulkarni et al. briefly discussed performing parallel compilation on multiple compiler threads to exploit the additional processing resources available on multi-core machines, but did not provide any experimental results [21]. A few existing virtual machines, such as Sun's HotSpot server VM [24] and the Azul VM (derived from HotSpot), support multiple compiler threads, but have not presented any discussions on ideal compilation strategies for multi-core machines. Consequently, research is sorely lacking in understanding dynamic compilation issues and evaluating potential JIT compilation strategies in the presence of multiple compiler threads on current and future multi-core and many-core machines. In this paper, we investigate issues for dynamic compilation on modern machines and compare potential strategies with existing techniques.

## 3. Experimental Framework

The research presented in this paper is performed using the server version of the Sun/Oracle's HotSpot java virtual machines (build 1.7.0-ea-b24) [24]. The latest development code for the HotSpot VM is available through Sun's OpenJDK initiative. The HotSpot VM uses interpretation at the start of program execution. It then employs a counter-based profiling mechanism, and uses the sum of a method's *invocation* and loop *back-edge* counters to detect and promote hot methods for compilation. We call the sum of these counters as the *execution* count of the method. Methods/loops are determined to be hot if the corresponding method execution count exceeds a fixed threshold. The tasks of detecting hot methods and dispatching them for compilation are performed at every method call (for whole-method compiles) and loop iteration (for on-stack-replacement compiles). The HotSpot server VM allows the creation of an arbitrary number of compiler threads, as specified on the command-line.

The experiments in this paper were conducted using all the benchmarks from three different benchmark suites, SPEC jvm98 [26], SPEC jvm2008 (startup) [25] and DaCapo-9.12-bach [4]. We employ two inputs (10 and 100) for benchmarks in the SPECjvm98 suite, two inputs (small and default) for the DaCapo benchmarks, and a single input (startup) for benchmarks in the SPECjvm2008 suite, resulting in 56 benchmark/input pairs. Two benchmarks from the DaCapo benchmark suite, *tradebeans* and *tradesoap*, did not always run correctly with our version of the HotSpot VM, so these benchmarks were excluded from our set.

All our experiments were performed on a cluster of 8-core Intel Xeon 2.833GHz processors. All machines use Fedora Linux as the operating system. We disable seven of the eight available cores (including hyperthreading) to run our single-core experiments. Our multi-core experiments utilize all available cores. More specific variations made to the hardware configuration are explained in the respective sections. Each benchmark is run in isolation to prevent interference from other user programs. Finally, to account for inherent timing variations during the benchmark runs, all the performance results in this paper report the average over 10 runs for each benchmark-configuration pair.

## 4. Measuring Benefit of Dynamic Compilation

Dynamic just-in-time compilation and optimization is known to generate code that results in significantly improved program performance over VM interpretation or execution in unoptimized code. The potential of JIT compilation to improve application performance is directly proportional to the fraction of total program methods that are compiled. Thus, JIT compilation of all program methods should potentially achieve the most efficient application performance. However, this efficiency in program performance will likely come at a prohibitively high compilation cost in most cases. At the same time, based on their execution frequencies, compilation of different methods will contribute differently to performance improvement. As explained earlier, selective compilation uses this principle to only compile methods that are likely to contribute most to improving program efficiency. In this section we study the potential of different selective compilation thresholds in

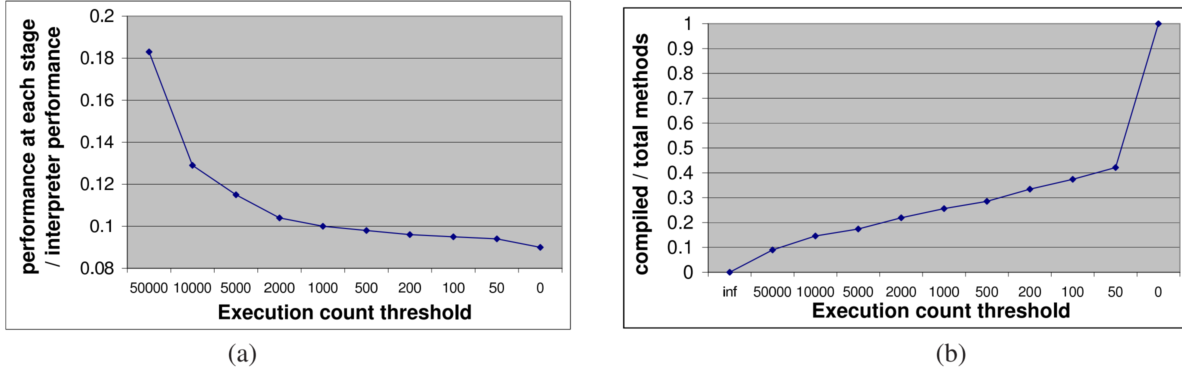



**Figure 1. Understanding the effect of JIT compilation on program performance**

improving *steady-state* program performance. Steady-state performance is measured after all methods with execution counts above the selected threshold have finished compilation, thus discounting the compilation overhead. This information may enable dynamic compilers to make a more informed decision regarding the fraction of total methods to compile based on available computational resources.

We constructed a novel experimental setup to collect these performance measurements. Our experimental configuration conducts an initial offline run to collect the individual method execution counts for every benchmark in our set. The methods in every benchmark are then sorted based on their execution counts. Each of our three benchmark suite provides the ability to execute several *iterations* of any given benchmark program in a single VM run. We exploit this ability to run each benchmark over several *stages*, with a few benchmark iterations per stage. Each successive stage in our modified HotSpot VM lowers the compilation threshold and compiles additional methods (over the previous stage) with progressively lower execution counts. Thus, the first stage compiles no methods, and all methods are compiled by the final stage. Each intermediate stage compiles successively more program methods. We disable *background compilation*, allowing the first iteration of every stage to immediately finish all sent compilations. The final iteration in each stage provides a measure of the benchmark performance at the end of that stage.

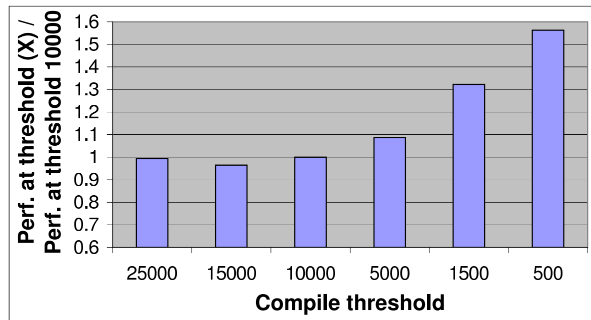
Figure 1(a) shows the average improvement in program performance over interpreted execution across all benchmarks at each configuration stage. The X-axis indicates the compile threshold used at each configuration stage. At each stage, methods that have an execution count greater than the stage compile threshold (during the offline run) are sent for compilation. Figure 1(b) shows the percentage of methods compiled at each stage, averaged over all benchmarks in our set. Thus, we can see that JIT compilation of all program methods achieves a dramatic performance improvement over interpretation, achieving program execution in less than 10% of the interpretation time, on average. At the same time, it is interesting to see that most of the performance gain is obtained by compiling a very small fraction of the total executed methods. This observation is important because JIT compilation not only consumes computational resources, but also generates native code that increases memory pressure and garbage collection overheads, and may also increase non-determinism due to the pause times associated with garbage collections. Thus, these results show that, even with practically unlimited computational resources, it might still be beneficial to limit the number of methods compiled at runtime.

## 5. JIT compilation on Single-Core Machines

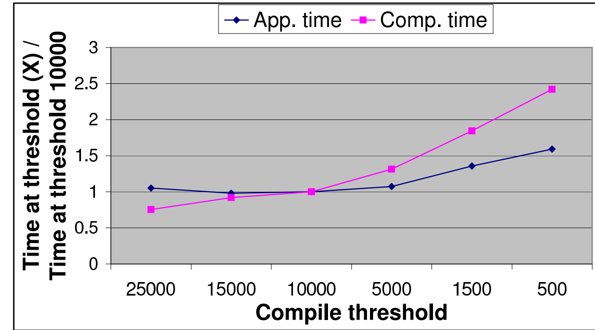
The execution time for managed language applications on single-core machines is the sum of the run-times of each individual VM component. The application and compiler threads are the most prominent VM components that contribute to the overall program execution time. These two components share a complex relationship, in the sense that just decreasing the compilation time (by reducing the number of methods compiled) may not produce a comparable effect on the overall execution time. Instead, compiling fewer methods may produce a bigger increase in the application thread time due to the resultant program executing in poorer quality code. Therefore, the compiler thresholds need to be carefully tuned to achieve the most efficient average program execution on single-core machines over several benchmarks.

### 5.1. Compilation Threshold with Single Compiler Thread

VM developers select the compile threshold by experimenting with several different threshold values on their set of benchmark programs to find the one that achieves the best overall performance. We performed a similar experiment




(a)




(b)

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

to determine the ideal compilation threshold with a *single* compiler thread on our set of benchmarks. These results are presented in Figure 2(a), which plots the ratio of the average overall program performance at different compile thresholds compared to the average program performance at the threshold of 10,000. The threshold value of 10,000 is selected as our baseline for comparison because that is the default compilation threshold used by the HotSpot server VM. Not surprisingly, we can see this default threshold (10,000) performs very well on our set of benchmarks, but a slightly higher compile threshold of 15,000 achieves the best overall performance for our benchmark set.

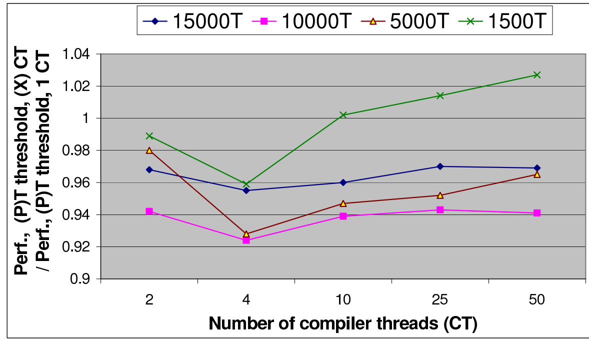
Figure 2(b) shows the break-down of the overall program execution time in terms of the ratios of the application and compiler thread times at different thresholds to their respective times at the compile threshold of 10,000, averaged over all benchmark programs. As expected, the compiler thread times increase with lower compilation thresholds as more methods are sent for compilation. However, the behavior of the application thread times is less intuitive. A compilation policy with high thresholds (25,000) compiles fewer methods and results in poorer-quality code, causing higher application thread times. By contrast, policies with lower thresholds send more methods to compile. However, this increase also grows the length of the compilation queue and delays the compilation of the most important methods, resulting in the non-intuitive degradation in application thread performance observed in Figure 2(b). Due to its superior performance, we select the compile threshold of 15,000 as the baseline for our remaining experiments in this paper.

### 5.2. Effect of Multiple Compiler Threads

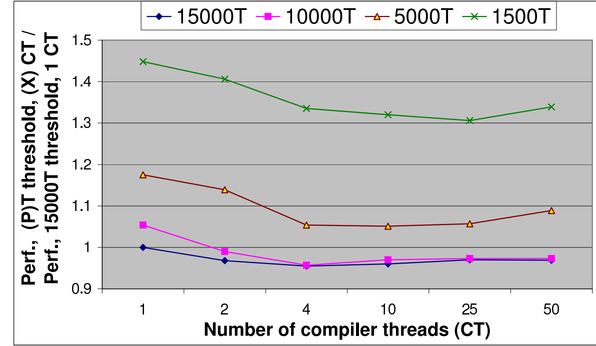
Several virtual machines now provide the capability of spawning multiple compiler threads. However, to the best of our knowledge, the effect of multiple compiler threads on overall program performance on a single-core machine has never been previously discussed. In this section we conduct such a study and present our observations.

For each compilation threshold, a separate plot in Figure 3(a) compares the average overall program performance with multiple compiler threads to the average performance with a single compiler thread at that same threshold. Intuitively, a greater number of compiler threads should be able to reduce the method compilation queue delay, which is the time spent between sending a method to compile and generating optimized code. Indeed, we notice program performance improvements for small number of compiler threads (2–4), but the benefits do not seem to hold with increasing number of such threads (>4). For the larger compile thresholds (15,000–10,000), the performance degradations with more compiler threads are very small, and may be due to added application interference or noise. For the smaller compile thresholds, we notice an increase in the overall *compiler thread* times with more compiler threads. This increase suggests that several methods that were queued for compilation, but never got compiled before program termination with a single compiler thread are now compiled as we provide more resources to the VM compiler component. Unfortunately, many of these methods contribute little to improving application performance. At the same time, the increased compiler activity increases compilation overhead and results in lowering overall program performance.

Figure 3(b) compares the average overall program performance in each case to our baseline average performance with a single compiler thread at a threshold of 15,000. These results reveal the best compiler policy on single-core machines with multiple compiler threads. The results indicate that there may be no need to change compiler thresholds with more compiler threads. However, a small increase in the number of compiler threads generally improves performance by reducing the compilation queue delay.




(a)




(b)

**Figure 3. Effect of multiple compiler threads on single-core program performance**

## 6. JIT Compilation on Multi-Core Machines

Dynamic JIT compilation on single-processor machines has to be conservative to manage the compilation overhead at runtime. Multi-core machines provide the opportunity to spawn multiple compiler threads and move these threads to different (free) cores so as to not interrupt the application threads. As such, it is generally believed that dynamic compilation on multi-core machines should be made more aggressive to achieve better application thread and overall program performance.

In this section we present our results that show the effect of increased compiler aggressiveness on application performance for multi-core machines with a small and large number of free available cores. Although architects and chip developers are planning a continuously increasing number of cores for future chips, processors with a large number of cores (*many-cores*) are not easily available just yet. Therefore, in order to investigate JIT compilation strategies on such future many-core machines, we have constructed a unique experimental configuration for our experiments in this section. Our setup achieves simulation of multi/many-core VM behavior using a single processor/core. To construct this setup, we first updated our HotSpot VM to report the *category* of each operating system thread that it creates (application, compiler, garbage-collector, etc.). We then modified the benchmark *harness* of all our benchmark suites to not only report the overall program execution time, but to also provide a break-down of the time consumed by each VM thread. Finally, we employed 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.

Our experimental setup to evaluate the behavior of multi-core application execution on a single-core machine is illustrated in Figure 4. Figure 4(a) shows a snapshot of one possible VM execution order with multiple compiler threads, with each thread running on a distinct core of a multi-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 4(b). Thus, JIT compilations in both the original multi-core and our simulated single-core execution orders occur at about the same time relative to the application thread. Later, we use our ability to precisely measure individual thread times to realistically simulate an environment where each thread has access to its own core. Thus, this framework allows us to study the behavior of different JIT compilation strategies with any number of compiler threads running on separate cores on future many-core hardware.

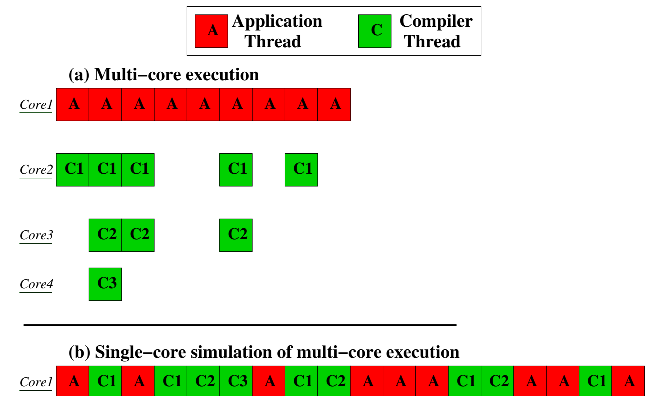
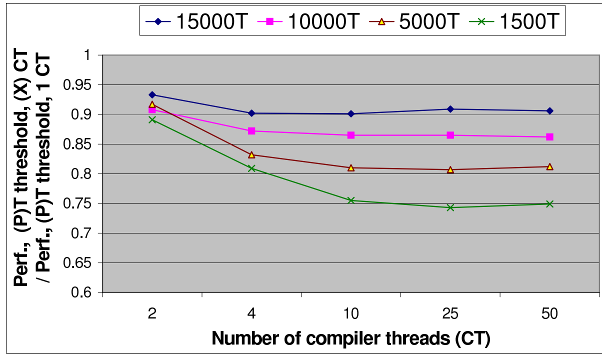


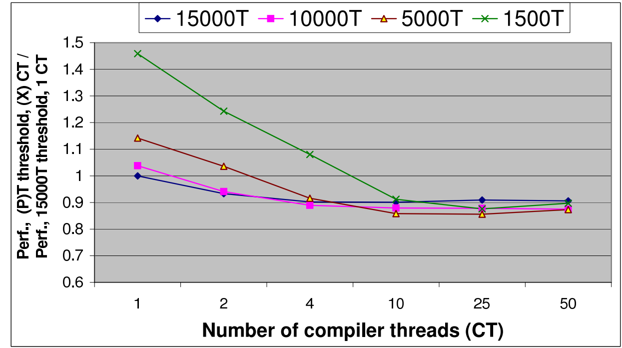

**Figure 4. Simulation of multi-core VM execution on single-core processor**

### 6.1. Effect of Aggressive Compiler Thresholds with a Single Compiler Thread

One approach of increasing compiler aggressiveness is to lower the compilation *threshold*, which is the minimum




(a)




(b)

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

execution count needed to send a method to compilation. Lowering the compilation threshold can benefit program performance in two ways: (a) by compiling a greater percentage of the program code, and (b) by sending methods to compile early. Both these effects can potentially enable more efficient program execution by allowing the program to spend more time running in optimized code. However, as seen from Figure 2(b) and discussed in Section 5, simply lowering the compile threshold with a single compiler thread does not produce better application thread performance. Our analysis of this behavior indicates that the additional compiles caused by lowering the compile threshold delays compilation of the more important methods, degrading application thread (and overall program) performance.

### 6.2. Effect of More Compiler Threads

On multi-core machines it is more interesting to study the impact on application performance of multiple compiler threads running on distinct processor cores. Figures 5(a) and (b) show the results of our experiments exploring the effect of multiple compiler threads on application performance. For each indicated compile threshold, a plot in Figure 5(a) shows the ratio of the application thread performance with different number of compiler threads to the application thread performance with a single compiler thread. Thus, we can see that increasing the number of compiler threads improves application performance. Moreover, configurations with more aggressive compilation thresholds derive a greater benefit in application performance from more compiler threads. However, the relative gain in application performance does seem to taper off with each additional compiler thread. Also, higher compilation thresholds need fewer compiler threads to reach their maximum achievable application performance.

Figure 5(b) is more important to determine the best JIT compilation policy to adopt on multi/many-core machines. Here, we compare all the application thread performances (with different number of compiler threads) to a single baseline application thread performance. The selected baseline is the application thread performance with a single compiler thread at the threshold of 15,000. Remember, this is the compiler thread configuration that achieved the best efficiency on single-core machines (with one compiler thread). It is very interesting to note that although performance improves with increasing compiler threads, higher compiler aggressiveness seems to offer little additional benefit over the baseline compile threshold that is employed on a single-core machine. Thus, increasing the number of spawned compiler threads by a small amount is more beneficial to performance than increasing compiler aggressiveness by lowering compilation thresholds for modern machines.

## 7. Future Work

This work presents several interesting avenues for future research. First, we will conduct similar experiments in other virtual machines to see if our conclusions from this work hold across different classes of VMs. Second, the HotSpot VM only provides one compilation *level*, which restricted this study to only explore one direction of increasing compiler aggressiveness, that is reducing compilation threshold. In the future, we will explore the other aspect of increasing compiler aggressiveness by optimizing at higher compilation levels in a VM that provides more robust support for tiered compilation, such as JikesRVM or IBM's J9. Third, we will study the impact of different compiler aggressiveness on memory consumption and garbage collection overhead on devices with different memory configurations, from embedded devices to high-performance servers. Finally, our goal is to develop an adaptive cross-VM compilation policy that will automatically employ the ideal compilation strategy based on available processor and memory resources.

## 8. Conclusions

In this work we presented several novel experimental configurations that we constructed to explore the impact of different dynamic JIT compilation policies with different compiler aggressiveness and different number of compiler threads on single-core, existing multi-core, and future many-core machines. Our results validate and quantify previous claims that a small fraction of compiled methods can accomplish most of the program performance gains at runtime. On single-core machines, our experiments show that the same compilation threshold achieves the best overall program performance with a single and multiple compiler threads. On multi-core machines, we observed that more than increasing compiler aggressiveness, spawning multiple compiler threads is the best approach to derive greater program performance benefits. Thus, as we enter the new era of multi-core and many-core machines with increasing number of cores with every processor generation, we expect this research to assist VM developers to make more informed compilation policy decisions for their virtual machines to achieve the best application performance.

## 9. Acknowledgments

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

## References

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

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

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

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 169–190, 2006.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[25] SPEC2008. Specjvm2008 benchmarks. http://www.spec.org/jvm2008/.

[26] SPEC98. Specjvm98 benchmarks. http://www.spec.org/jvm98/.

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