Exploring Multi-Threaded Java Application Performance on Multicore Hardware

Jennifer B. Sartor and Lieven Eeckhout
Ghent University, Belgium

Abstract

While there have been many studies of how to schedule applications to take advantage of increasing numbers of cores in modern-day multicore processors, few have focused on multi-threaded managed language applications which are prevalent from the embedded to the server domain. Managed languages complicate performance studies because they have additional virtual machine threads that collect garbage and dynamically compile, closely interacting with application threads. Further complexity is introduced as modern multicore machines have multiple sockets and dynamic frequency scaling options, broadening opportunities to reduce both power and running time.

In this paper, we explore the performance of Java applications, studying how best to map application and virtual machine (JVM) threads to a multicore, multi-socket environment. We explore both the cost of separating JVM threads from application threads, and the opportunity to speed up or slow down the clock frequency of isolated threads. We perform experiments with the multi-threaded DaCapo benchmarks and pseudojbb2005 running on the Jikes Research Virtual Machine, on a dual-socket, 8-core Intel Nehalem machine to reveal several novel, and sometimes counter-intuitive, findings. We believe these insights are a first but important step towards understanding and optimizing managed language performance on modern hardware.

Categories and Subject Descriptors D3.4 [*Programming Languages*]: Processors—Memory management (garbage collection); Optimization; Run-time environments

General Terms Performance, Measurement

Keywords Performance Analysis, Multicore, Managed Languages, Java

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.

1. Introduction

Multicore systems are here to stay. Since the first multicore processor was released over a decade ago, today's processors implement up to a dozen cores, and each generation is expected to have an increasing number of cores because of Moore's law [19]. Multicore processors are abundant across all segments of the computer business, from hand-held devices such as smartphones and tablets, up to high-end systems. Today's servers even host multiple sockets, effectively creating systems with multiple tens of cores.

Given the ubiquity of multi-threaded managed language applications [22], now running on modern multicore hardware, their performance is critical to understand. While multicore performance has been extensively studied for multi-threaded applications written in traditional programming languages, understanding the performance of applications written in managed languages such as Java has received little attention. Part of the reason is the complexity of these managed workloads. Java virtual machine (JVM) service threads such as garbage collection, profiling, and compilation threads, interact with application threads in many complex, non-deterministic ways. Not only can these threads stop the application, e.g., in order to collect a full heap, they also interact with the application at the microarchitectural level, sharing hardware resources, such as cores, cache and off-chip bandwidth.

Looking forward, optimizing performance while taking into account power and energy will be even more important. Handheld devices have to minimize total energy consumption with given performance goals, e.g., soft real-time deadlines. For high-end servers, the goal is typically to optimize performance while not exceeding a given power budget. Several studies point out that with the end of Dennard scaling [7] — slowed supply voltage scaling — it will no longer be possible to power on the entire processor all the time, a problem referred to as dark silicon [9]. Intel's Turbo Boost technology [15], which enables increasing clock frequency for a limited number of cores and for a short duration of time, could be viewed as a form of dark silicon.

Understanding and optimizing managed application performance on multicore processors and systems is therefore a complicated endeavor. Some of the fundamental ques-

tions that arise are: How many application and JVM service threads yield optimum performance? Should one initiate as many application and garbage collection threads as there are cores on the chip? If so, does this hold true for multi-socket systems as well? Further, how should one distribute the application, garbage collection and compiler threads across the various cores and sockets in the system? In particular, in case of a multi-socket system, should one schedule JVM service threads on the same socket as the application threads, and what is the performance penalty, if any, from offloading JVM service threads to another socket? If optimizing performance, but taking power into consideration, should one speed up all threads or just the application threads? And how much does performance degrade by slowing down JVM service threads?

In this paper, we provide insight into Java multi-threaded application performance on multicore, multi-socket hardware. We perform a comprehensive set of experiments using a collection of multi-threaded DaCapo benchmarks and pseudojbb2005 on a dual-socket 8-core Intel Nehalem system with a current version of the Jikes Research Virtual Machine. We vary the number of cores and sockets, the number of application and garbage collection threads, clock frequency, thread-to-core mapping and pinning, and heap size.

We are the first to extensively explore the space of multi-threaded Java applications on multicore multi-socket systems, and we reveal a number of new, and sometimes surprising, conclusions:

1. Java application running time benefits significantly from increasing core frequency.
2. When isolating JVM collector threads onto a separate socket from application threads, there is a cost: less than 20% of performance. However, isolating the compilation thread is usually performance neutral, but for some benchmarks actually improves overall performance.
3. If power-constrained, lowering the frequency of JVM service threads costs a fraction of performance, but 3-5 times less than when scaling application threads.
4. Many benchmarks achieve good performance when all threads run on a single socket at the highest frequency, and the second socket is kept idle. However, all but one benchmark have optimal performance when isolating the compilation thread and lowering its frequency.
5. For our benchmarks, the best performance running on one socket is usually obtained by keeping the number of application threads equal to the number of collection threads, also equal to the number of cores. With two sockets, it is better to pair application and collector threads together (unless there is a lot of inter-thread communication), and almost all benchmarks benefit from increasing the number of application threads to be equal to the number of cores while setting the number of collection threads to half of that.
6. Pinning application and collector threads to cores does not always improve performance, and some benchmarks benefit from letting threads migrate.
7. During startup time, the cost of isolating JVM service threads to another socket is less than at steady-state time, while lowering their frequency still deteriorates performance several times less than that of lowering the application threads.

Analyzing Java applications on modern multicore, multi-socket hardware reveals that it is difficult to follow a set of rules that will lead to optimal performance for all applications. However, our results reveal insights that will assist in identifying the right number and scheduling of application and JVM service threads that will preserve performance while saving the critical resource of power in future systems.

2. Motivation

As managed languages have become ubiquitous from the embedded to the server market and in between, it is important to analyze their performance on multicore hardware. Because Java applications run on top of a virtual runtime environment, which includes its own management threads and introduces non-determinism for applications, choosing the correct parameters and configurations to balance performance and energy is complex. Modern machines also introduce extra runtime parameters to study, offering multiple sockets and frequency (and voltage) scaling options. Researchers have been studying the performance and power consumption for managed languages as part of prior work [4–6, 10, 11, 13], demonstrating that it is an important problem. However, studies in prior work were limited to single-socket systems, single-threaded Java applications, and/or isolated JVM service performance, not end-to-end performance. (See Section 5 for a more detailed description of prior work.)

In this work we expand the exploration space considerably. We consider multi-threaded applications written in a managed programming language, Java. We focus on multicore, multi-socket hardware. We also aim at understanding how design choices in terms of thread-mapping, frequency scaling, thread-pairing, pinning, and more affect end-to-end application performance. More specifically, there are a number of fundamental questions related to Java application performance on multicore multi-socket systems that merit further investigation.

Number of application and JVM service threads. First, we would like to explore the number of application and collection threads given a particular number of cores. Default methodology sets the number of application and collection threads equal to the number of cores; however, this does not necessarily take into account the sharing of hardware resources between these and other JVM threads, nor does it take into account memory access ramifications of commu-

nicating across sockets. Although current practice is to have thread-local allocation, collection threads are not tied to any particular application thread or core (in our Java virtual machine), and can touch many areas of memory and incur inter-thread synchronization in order to reclaim space. Other JVM threads such as those that perform dynamic compilation and on-stack replacement, also interact with application threads and share hardware resources. It is unclear whether hardware resource sharing between JVM and application threads actually helps or hurts performance, because of either better data locality or resource contention.

Thread-to-core/socket mapping. Inherent in the choice of the number of threads is where to place these threads on a multicore multi-socket system. Current systems largely leave thread scheduling up to the operating system which can preempt and context switch threads when necessary. Intuition suggests we should use all cores and maximize parallelism. Further, benchmarks with large working sets could benefit from the aggregate last-level cache capacity across sockets. However, as previously mentioned, moving data between sockets increases inter-thread communication, leading to more memory accesses and coherence traffic and potentially higher synchronization costs.

Frequency scaling and power implications. Because modern machines include the ability to dynamically scale the core frequency, another axis to explore is the ramifications of scaling on total application performance. Because power is a first-order concern, and will become even more constrained in future systems, it might be worth paying the price of somewhat reduced performance for power savings. Hence, we need to analyze both the cost of moving threads to another core or socket and then additionally, the cost of lowering the clock speed. Because JVM service threads do not run constantly, we surmise that they should be amenable to running at scaled-down frequencies without hurting application performance drastically. However, both compilation and collection can be on the application critical path if they need to stop the workload to perform on-stack replacement or collect garbage when the heap is full. Garbage collection threads have to trace all live heap pointers to identify dead data to reclaim, and could thus suffer from higher memory communication if moved to a separate socket.

Analyzing this challenging experimental space will shed light on the ramifications of the many configuration and scheduling decisions on overall goals, whether time or power. We explore the performance of Java workloads, while keeping power in mind, along these many orthogonal axes on modern multicore, multi-socket hardware to provide new, sometimes surprising, findings and insights.

3. Experimental Setup

Before presenting the key results obtained from this study, we describe our experimental setup of running Java workloads on multicore, multi-socket hardware.

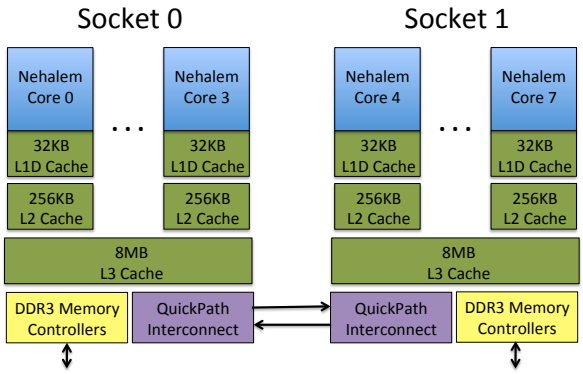

Figure 1. Diagram of our two-socket, eight-core Intel Nehalem experimental machine, with memory hierarchy.

3.1 Hardware Platform

The hardware platform considered in this study is an Intel Nehalem based system, more specifically, an IBM x3650 M2 with two Intel Xeon X5570 processors that are 45-nm Westmere chips with 4 cores each, see Figure 1. The two sockets are connected to each other through QuickPath Interconnect Technology; and feature a 1333 MHz front-side bus. Each core has a private L1 with 32 KB for data and 32 KB for instructions. The unified 256 KB L2 cache is private, and the 8 MB L3 cache is shared across all four cores on each socket. The machine has a total main memory capacity of 14 GB. We can use dynamic frequency scaling (DFS) on the Nehalem to vary core frequencies between 1.596 GHz and 3.059 GHz. However, on this machine, it is only possible to change the frequency at a socket-level. Therefore, all four cores on each socket run at the same frequency, and we are unable to evaluate more than two different frequencies simultaneously. For all of our experiments, we set the frequency only to the lowest and highest extremes to test the limits of performance. We turn hyperthreading off in all of our experiments.

3.2 Benchmarks and JVM Methodology

We perform experiments with the Jikes Research Virtual Machine (RVM), having obtained the source from the Mercurial repository in December, 2011[1]. We updated from a stable release of Jikes because of a revamping of their experimental methodology code. We modified Jikes slightly to identify and control JVM service thread placement for the purpose of frequency scaling. By default, we pin application and garbage collection threads to a particular core, and other JVM service threads are placed on the socket with application threads, but not pinned to a specific core. We also perform experiments without pinning application and JVM threads for comparison, and these results will be discussed in Section 4.5. In addition to threads that perform garbage col-

[1] changeset 10414: 5c59ac91ff06

Benchmark	Suite	Heap Size MB(min)	Total Allocation MB
avrora	DaCapo Bach	50	54
lusearch	DaCapo Bach	34	8152
lusearch-fix	DaCapo Bach	34	1071
pmd	DaCapo Bach	49	385
sunflow	DaCapo Bach	54	1832
xalan	DaCapo Bach	54	1104
pjbb2005	SPECjbb2005	200	1930

Table 1. Benchmark characteristics.

lection and compilation, other JVM service threads include a thread to perform finalization and do timing, many threads to coordinate on-stack-replacement (called organizer threads) of dynamically recompiled method code, and a main thread that is the first to execute the application's main method. Jikes does not perform compilation in parallel, so there is only one compilation thread.

We perform experiments on the multi-threaded Java applications in the DaCapo benchmark suite [4] version 9.12-bach that successfully run on our version of Jikes RVM. These include avrora, lusearch, pmd, sunflow, and xalan. We also perform experiments on pseudojbb2005 [3, 4], a variant of SPECjbb2005 with a fixed number of warehouses that allows for measuring the execution time. These benchmarks are real-world open-source applications. Table 1 details which suite our benchmarks came from, their minimum heap size in MB and their total allocation in MB [25]. We control the number of application and collection threads with command-line flags (which are by default both set to four). Our experiments use the best-performing, default generational Immix heap configuration [2]. Although there are garbage collection threads running in parallel, the collector is "stop-the-world" meaning that the application is stopped during both nursery and whole-heap collections. We performed all experiments at 1.5, 2, and 3 times the minimum heap size for each benchmark. For conciseness, we present some graphs only with 2 times the minimum heap size. We perform 10 invocations of each benchmark, and results are presented as the average of these 10 runs, along with 90% confidence intervals. For steady-state execution, which is the default reported throughout results, we measure the 15th iteration. Although there is non-determinism inherent in using the just-in-time adaptive compilation system that detects hot methods and dynamically re-compiles them to higher optimization levels, measuring the 15th iteration gives time for the benchmark to reach more steady-state behavior. For comparison, we present two separate sections of results at startup time, which are gathered during the first iteration of each benchmark. The compilation and on-stack-replacement threads are most active during startup time, and thus we include results for complete evaluation.

For targeted further analysis, we perform experiments with a version of the lusearch benchmark updated with a bug fix. The original lusearch derives from the 2.4.1 stable release of Apache Lucene, and has a very high allocation rate. This version included a bug that when fixed, only conditionally allocates a large data structure and cuts allocation by a factor of eight [25]. For additional experiments, we use a version updated with revision r803664 of Lucene that contains the bug fix [20], and we call this "lusearch-fix". We include results in all graphs for "lusearch-fix"; however, we still include the original lusearch because it is the official benchmark in the current release of the DaCapo suite.

3.3 Methodological Design

Recent hardware trends point to a need for power savings and capping while delivering high levels of performance. Already, recent research [6] has explored the effect of scaling the frequency of service threads on power and energy, explicitly targeting a heterogeneous multicore processor with big and small cores. While this is an important step to inform future hardware design, it is also important to focus on evaluating how to obtain the best performance from current multi-socket systems. Therefore, we measure end-to-end performance for multi-threaded workloads, and study the effect of both isolating and scaling JVM and application threads.

Because we are unable to perform per-core DFS in our hardware setup, we employ a two-step process. We first migrate certain threads to a separate socket and evaluate the cost of isolating threads on running time. Subsequently, we quantify and report *relative* performance differences when comparing frequency settings after isolating threads. By doing so, we separate the effect of scaling frequency from isolating threads onto another socket, and hence provide insight into how scaling affects end-to-end performance in multi-core systems.

The percentage of time spent in JVM threads is less than in the application threads. For this reason, we expect scaling down the frequency of JVM threads to have a smaller impact on end-to-end performance than scaling application threads. However, there is close interaction between application and JVM threads which makes this an interest space to explore. In particular, because we use a stop-the-world collector, application threads are stopped during garbage collection, hence slowing down collection only by a small fraction may have significant impact on overall application performance. Likewise, just-in-time compilation threads generate optimized code during runtime, and slowing these down may cause the application to run unoptimized code for a longer period of time, which may also have significant impact on overall performance. Furthermore, the interaction between JVM and application threads because of updating code and managing data leads to unpredictable interference in the memory subsystem. The interactions between appli-

cation and JVM threads are diverse and complex, and hence, we perform detailed experiments to evaluate the impact of both pairing and scaling the frequency of application and JVM threads.

4. Results

We now present our experimental results running multi-threaded Java applications on modern multicore, multi-socket hardware, in which we vary the number of cores and sockets, the number of application and garbage collection threads, clock frequency, thread-to-core mapping and pinning, and heap size. Because of the complexity of the space, we break up the analysis in a number of comprehensive steps. We first discuss the general effect frequency scaling has on application execution time. We then analyze the cost of isolating some JVM threads to a separate socket. After isolating JVM threads, we analyze the cost of scaling the frequency of isolated JVM service threads from the highest to the lowest value on end-to-end performance. Because we see a significant cost from isolating garbage collection threads, we then perform experiments that pair application and collection threads, but put some pairs on each socket. Finally, we analyze the effect that thread pinning has on performance, and the effect of varying both the number of application and collector threads while running on one socket.

4.1 The Effect of Scaling Frequency

Although a detailed power study is beyond the scope of this paper, we first wanted to explore the effect core frequency has on application performance. Figure 2 presents the speedup as a percentage of execution time of boosting core frequency from the lowest to the highest, when all threads, including four collector threads, run on only one socket. Results are for all six benchmarks for our range of three heap sizes.

Finding 1. *Java workloads benefit significantly from scaling up the clock frequency.*

Doubling clock frequency leads to between 27% and 50% performance improvement. Results are not sensitive to heap size. We see that doubling the clock speed does not lead to doubling the performance improvement, which would be 100%. Our benchmarks fall short of perfect scaling, most likely because of inter-thread synchronization and memory intensity. However, as we will see in all of our results, core frequency is one of the most significant factors in determining, and improving, application time. Below, we will show that scaling down JVM thread frequency does not affect performance as drastically as for application threads.

4.2 The Cost of Isolation

We now analyze the performance penalty inherent to isolating JVM service threads to another socket. We investigate this cost for four collector threads, for the compilation thread, for all JVM service threads *except* collector threads,

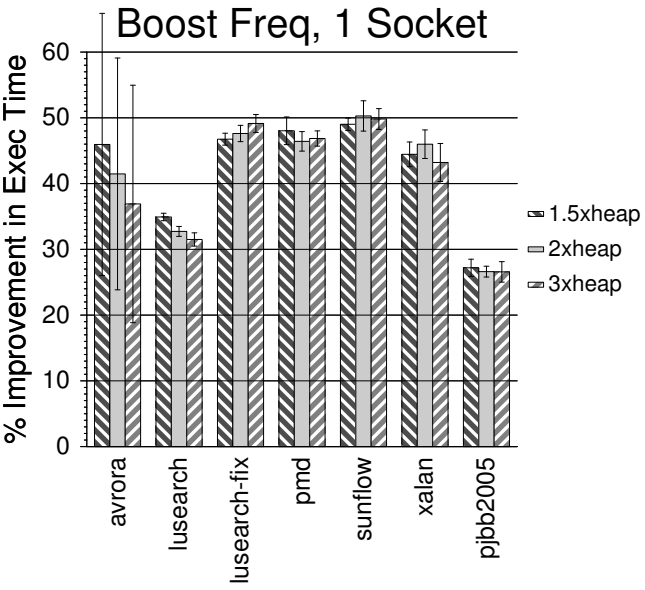

Figure 2. Percent execution time improvement when boosting frequency from lowest to highest on one socket.

and then all JVM threads together. The motivation for this experiment is twofold. First, in order to investigate the feasibility of scaling down the frequency of only JVM service threads, we must first isolate threads onto a separate socket, because we are only able to scale frequency at the socket-level. Second, we want to study how isolating JVM service threads to another socket hurts (through reduced data locality) or helps performance (by getting off the application's critical path).

Isolating garbage collection threads. Figures 3 through 6 show the cost of isolating some JVM threads to a second socket, as compared with the execution times when running all threads on one socket. The graphs present steady-state performance differences for our three heap sizes running at both the highest and lowest core frequencies. While some benchmarks' performance varies with heap size, we see larger performance differences between high and low core frequencies.

Finding 2. *Isolating garbage collection threads to a separate socket leads to a small performance degradation (no more than 17%) for most benchmarks because of increased latency between sockets; however, one benchmark substantially benefits (up to 66%) from increased cache capacity.*

Figure 3 shows that all but one application suffer from isolating four collection threads, due to more data communication between sockets. For all but lusearch, the degradation is less than 5% for the lowest frequency, and less than 17% for the highest. Lusearch is an outlier that particularly suffers from both increasing the number of collector threads (as shown in Figure 24) and from isolating those threads to another socket, with over 40% degradation in performance.

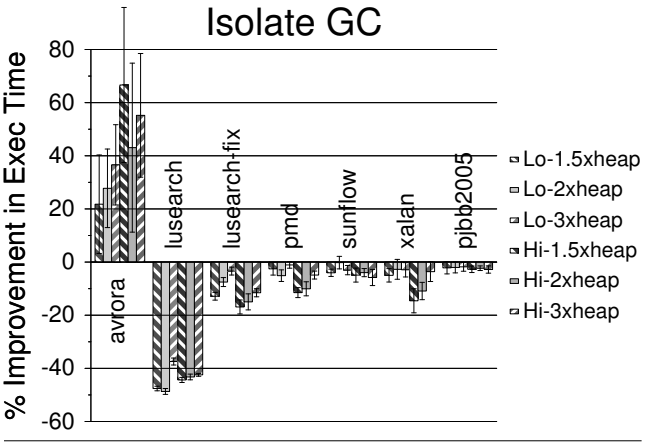

Figure 3. Percent execution time improvement of isolating four collector threads to a second socket.

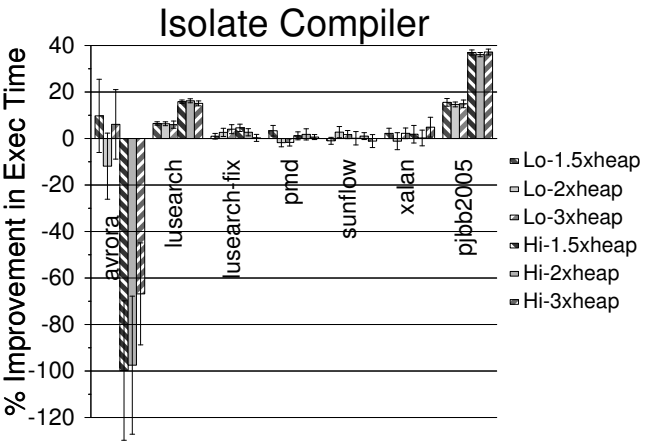

Figure 4. Percent execution time improvement of isolating the compiler thread to a second socket.

Figure 3 shows that when lusearch does not suffer from huge amounts of allocation, lusearch-fix's cost of moving collector threads to another socket lowers to be in line with other benchmark trends.

Interestingly, avrora benefits from isolating the collector threads to another socket. Performance improves by 36% and 66% at the lowest and highest frequencies, respectively. However, it should be noted that the confidence intervals for avrora are large, and thus performance greatly varies from run to run. For avrora, running all application and collector threads on one socket makes the threads contend more for the cache, and avrora benefits from the increased cache capacity of two sockets. Analyzing hardware performance counters revealed fewer L3 misses when the collector threads were isolated. We will see later that avrora is particularly sensitive to application-thread to core mapping because application threads do not have uniform behavior, and share data (see analysis in Sections 4.4 and 4.5).

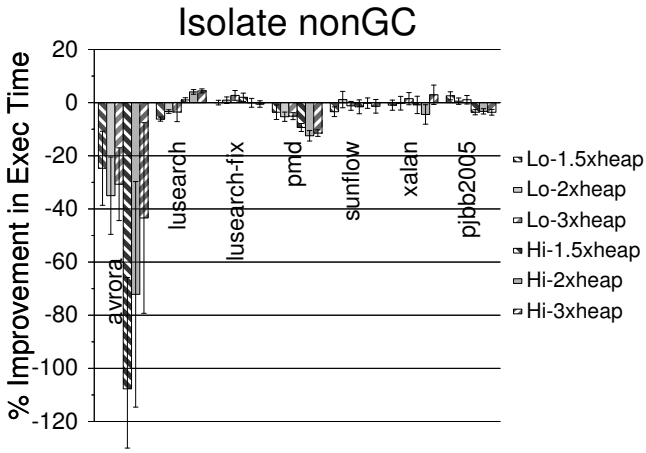

Figure 5. Percent execution time improvement of isolating all JVM non-collector threads to a second socket.

Isolating the JIT compilation thread. Figure 4 shows the effect of isolating just the compilation thread, with four collector and four application threads that are pinned to the first socket.

Finding 3. *Isolating the compilation thread to a separate socket leads to a either a performance boost, or is performance-neutral. Only avrora's performance at high frequencies suffers because of increased latency between sockets.*

Four benchmarks have very little change to performance when the compiler thread is isolated to a separate socket during steady-state. Lusearch and pjbb2005 see a performance win, especially at the higher frequency, by isolating the compiler away from other application and collector threads. Only avrora sees a performance hit, up to 100% with the highest frequency (although confidence intervals are large). It is possible that when the application is sped up, it is more sensitive to the compiler being separated from other JVM threads such as those that perform on-stack-replacement. When analyzing startup time, indeed avrora is the only benchmark for which performance degrades when isolating the compiler or on-stack-replacement threads (see further analysis in this section regarding Figures 8 and 9).

Isolating all JVM service threads. Figure 5 shows the cost of isolating all JVM service threads *except* for the four collection threads (or nonGC) which remain on the first socket with application threads. Figure 6 then shows the impact to performance when *all* JVM threads are isolated onto another socket.

Finding 4. *Isolating all JVM service threads to a separate socket leads to larger performance degradation for a few benchmarks (only one suffers more than 22% degradation, and only at the highest frequency), while others are only slightly negatively affected by offloading computation and memory to another socket.*

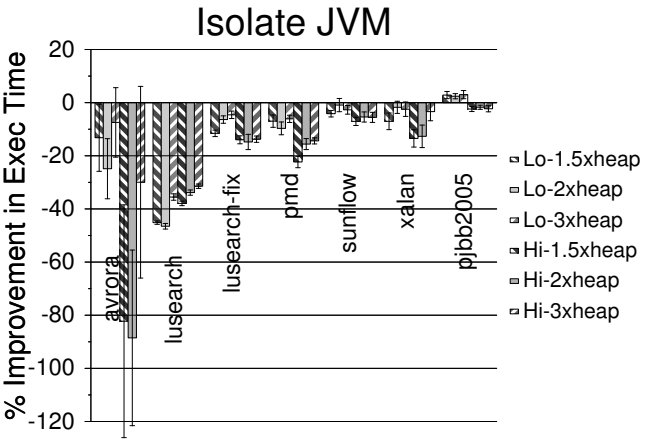

Figure 6. Percent execution time improvement of isolating all JVM threads to a second socket.

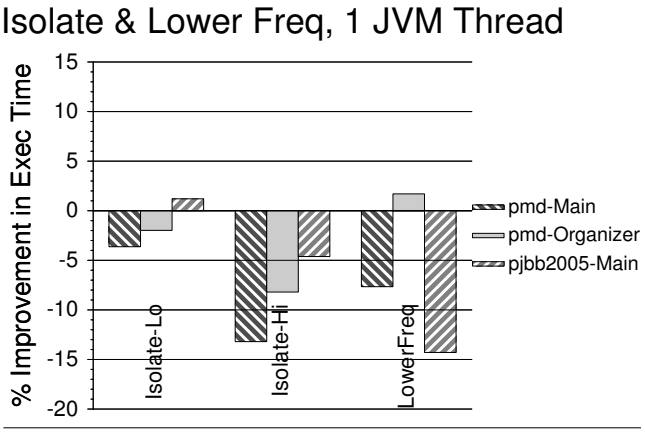

Figure 7. Percent execution time improvement of isolating and scaling the frequency of certain JVM threads for pmd and pjbb2005, at 1.5 times the minimum heap size.

Interestingly, while avrora benefited from collector thread isolation, performance severely degrades when isolating all nonGC threads. At the higher frequency, performance degradation goes down to 107%! Comparing this to isolating all JVM service threads, we see that avrora still suffers, but less, with a maximum of only 88% degradation. Avrora is more sensitive to frequency changes in combination with isolation than other benchmarks, and is more sensitive to nonGC threads being isolated, probably because of memory interaction. The benefit avrora obtained from offloading the collector memory activity to another socket is shown in the difference between the nonGC and JVM isolation graphs.

Besides avrora, other benchmarks show smaller degradations to performance when isolating nonGC, or all JVM service threads. For nonGC thread isolation, pmd has up to 12% performance loss, but other benchmarks either slightly improve or slightly degrade performance. When isolating all JVM service threads, lusearch can suffer over 40%, but the fixed version of lusearch lowers this cost to less than 20%. In general, isolating all JVM threads seems to have a performance impact that is the sum of isolating collection threads and isolating nonGC threads (Figures 3 and 5), which is overall slightly degraded. Pmd can suffer as much as 22% when isolating JVM threads, which we investigate further by isolating certain JVM service threads. We isolate only the main thread, which calls the application's main method, and then only the organizer thread, which performs on-stack replacement. Results are shown in Figure 7 on the left-most bars, and we see that isolating each of these two JVM threads makes pmd's performance suffer, more so at the higher frequency.

Overall, we see that benchmarks respond differently to the isolation of particular JVM service threads. While avrora benefits from separating collection threads, all other benchmarks suffer up to 17%. However, other benchmarks benefit slightly or are neutral to isolating the compilation thread. All benchmarks but avrora do not suffer much from isolating nonGC threads. While the cost to isolate JVM service threads is not insignificant, it is not unreasonable if the need for power savings is paramount.

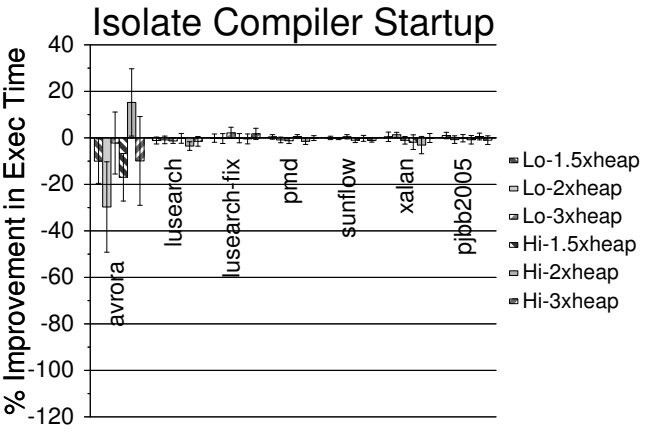

Figure 8. Percent execution time improvement of isolating the compiler thread to a second socket at startup time.

Isolation during startup. Although application performance matters most during steady-state execution, the compilation thread in particular is most active during JVM startup time. Thus, for completeness, we analyze the cost of isolating the compiler thread also at startup time.

Finding 5. *During startup time, isolating only the compilation thread has little impact on performance, but isolating all nonGC threads actually improves performance for all benchmarks but avrora. In general, the performance of isolating JVM threads to another socket is better, with some performance improvements, at startup time than at steady-state time.*

Figure 8 presents the cost of isolating the compiler thread at startup time (relative to a baseline run also at startup time), during the first iteration of a benchmark when the compiler

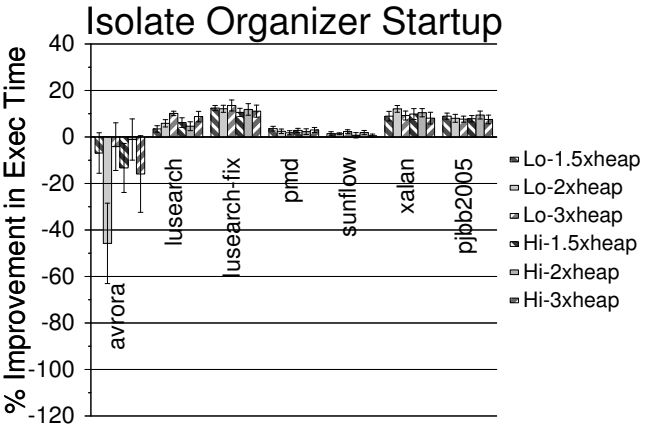

Figure 9. Percent execution time improvement of isolating the organizer (on-stack replacement) thread to a second socket at startup time.

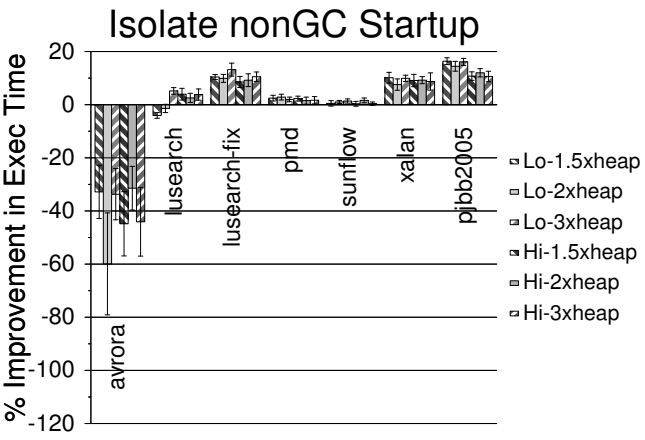

Figure 10. Percent execution time improvement of isolating all JVM non-collector threads to a second socket at startup time.

is most active. We see all benchmarks but avrora have negligible impact on performance when the compiler thread is placed onto another socket. Avrora, which has high variation between different runs, sometimes sees degradation of performance and sometimes improvement. Although during steady-state, in Figure 4, lusearch and pjbb2005, see an improvement in running time due to separating the compilation thread, at startup, there is not a big impact on performance. There is more communication and memory sharing between the JVM and application threads during actual compilation activity that does not exist during steady-state, and thus dampens the benefit of using extra CPU and memory resources during startup time. However, with avrora during startup, we do not see the large hit to performance at high frequencies that we did during steady-state, therefore concluding that avrora is less sensitive to frequency changes at startup time.

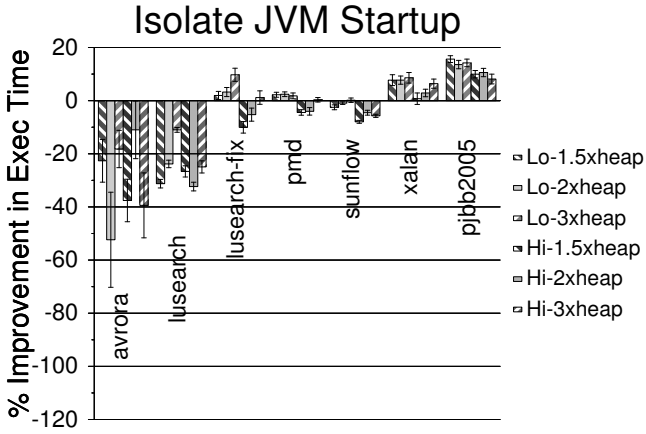

Figure 11. Percent execution time improvement of isolating all JVM threads to a second socket at startup time.

In Figures 10 and 11, we redo the experiments isolating nonGC and JVM threads at startup time. At startup time, isolating all JVM threads except the collector threads leads to performance improvements for all but avrora, including up to 16% for pjbb2005. This contrasts with the neutral or slightly negative (especially for avrora and pmd) results we saw at steady-state time. Apparently during startup, the nonGC threads benefit from the extra resources of another socket, and do not suffer from extra communication between threads via memory. We see similar trends for all JVM threads, which, at startup time, suffer less from isolation than at steady-state time, even leading to performance improvements for xalan and pjbb2005.

Because there is a noticable difference between the startup performance of isolating just the compiler thread, and all nonGC JVM threads, we perform experiments also isolating only the organizer threads that perform on-stack-replacement. Figure 9 shows that the organizer threads play the main part in the performance of nonGC threads. When organizer threads are isolated during startup time, all benchmarks but avrora see a performance benefit, following very similar trends to Figure 10, with pjbb2005 improving only slightly less. Although organizer threads interact with application memory, they benefit from being isolated to another socket with extra resources. In fact, the organizer threads have a larger impact on performance than the singular compilation thread itself. The difference between both avrora and pjbb2005's performance when isolating only the organizer thread and isolating all nonGC threads is due to the main thread (see pjbb2005 in Figure 7), and other JVM service thread interaction.

4.3 Analyzing Frequency Scaling

After exploring the cost of isolating JVM threads, we now analyze the cost of scaling each socket's clock frequency from the highest to the lowest in regards to execution time. In Figure 12 to 15, we compare against a baseline of separat-

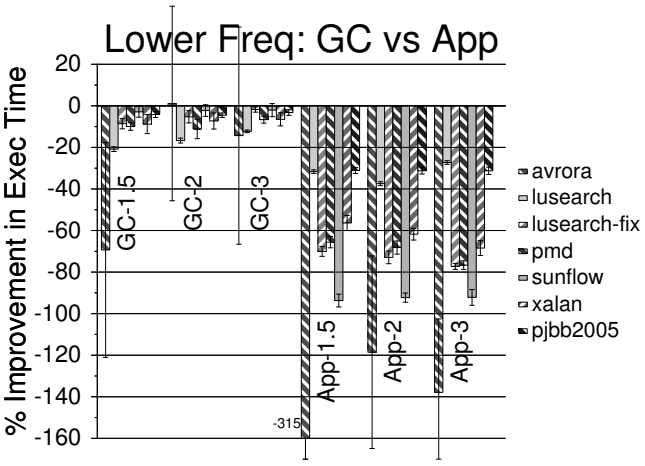

Figure 12. Percent execution time improvement when lowering frequency from highest to lowest, either four collector threads, or application (plus the rest) socket.

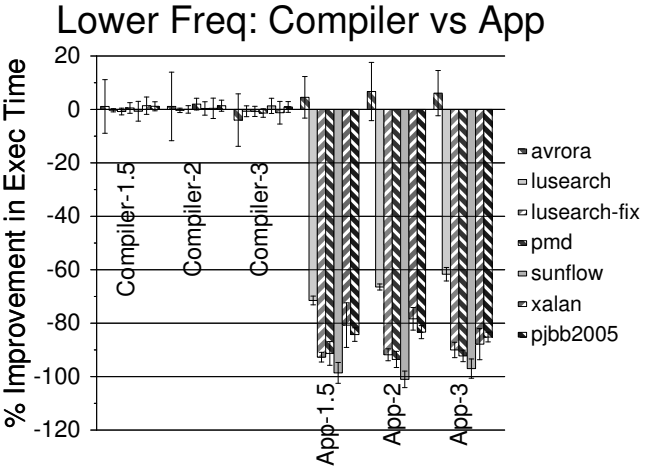

Figure 13. Percent execution time improvement when lowering frequency from highest to lowest, either compiler thread, or application (plus the rest) socket.

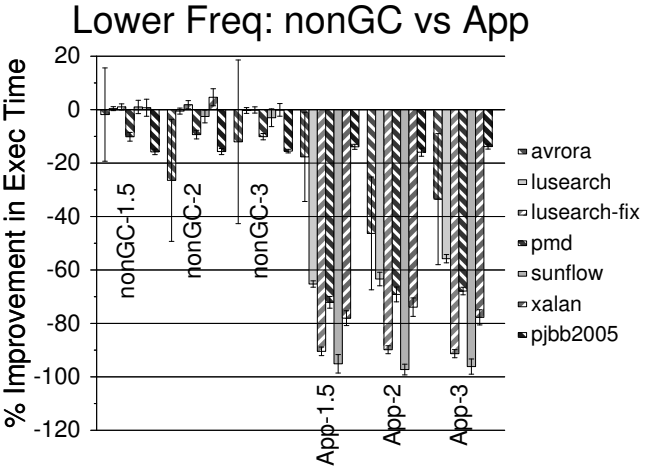

Figure 14. Percent execution time improvement when lowering frequency from highest to lowest, either JVM non-collector threads, or application (plus the rest) socket.

ing some JVM threads to a second socket, with both sockets at the highest frequency. The left groupings of bars compare a configuration lowering only the isolated JVM threads' frequency, while the right groupings compare lowering only the application and non-isolated threads' frequency. We present all results at all three heap sizes for all benchmarks, but note that there is little heap-size variation.

Finding 6. *On average, lowering the frequency of collector threads does degrade performance (usually less than 20%), but degrades about five times less than lowering application thread frequency.*

Figure 12 compares scaling down the frequency of four collector threads in the left three bar groupings versus application threads on the right. Although collector threads can be on the application critical path because they force the application to pause during collection, collector threads do not run all the time, and thus they are amenable to being scaled down for more power-conscience environments. Maximally, avrora performance degrades 69% for scaling collector threads (with large confidence intervals), with the next highest benchmark degradation at 20%. After scaling application threads, we see avrora's performance can degrade up to 315%!

Finding 7. *Lowering the core frequency for the isolated compiler thread affects performance very little, while application performance suffers greatly.*

Figure 13 shows that lowering the core frequency for only the compiler thread does not affect steady-state performance on average. In Figure 16, we show that at startup time, the compilation thread is also unaffected by lowering frequency. In comparison, benchmark application threads can degrade by as much as 100% when we scale down frequency at steady-state time. Interestingly, avrora is the only application that does not see a performance degradation when the application and other threads are scaled down together.

Finding 8. *If worrying about a power budget, scaling down the frequency of JVM threads, while costing some performance (usually less than 20%), has a much more reasonable effect on overall execution time as compared to scaling application threads, which can take twice the running time.*

Looking at the cost of scaling all but collector threads in Figure 14, and scaling all JVM service threads in Figure 15, we see that sunflow's application threads suffer the most, more than 90%, from running at a lower clock speed, while JVM threads' performance degrades less than 30% for all benchmarks. It is interesting to note the change in performance for scaling pjbb2005's application threads down, between the configuration where the compilation thread is

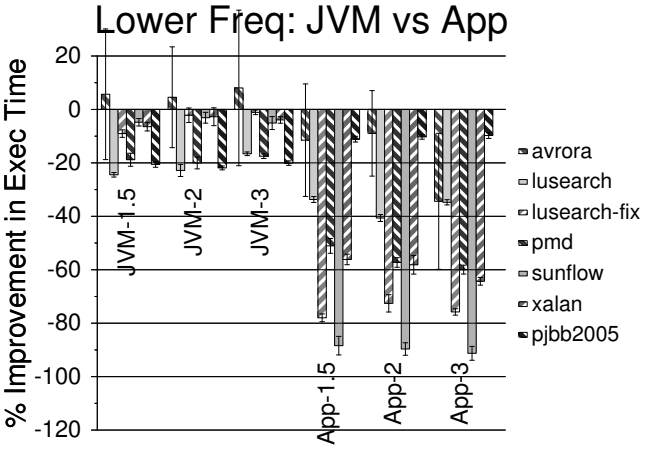

Figure 15. Percent execution time improvement when lowering frequency from highest to lowest, either JVM threads, or application socket.

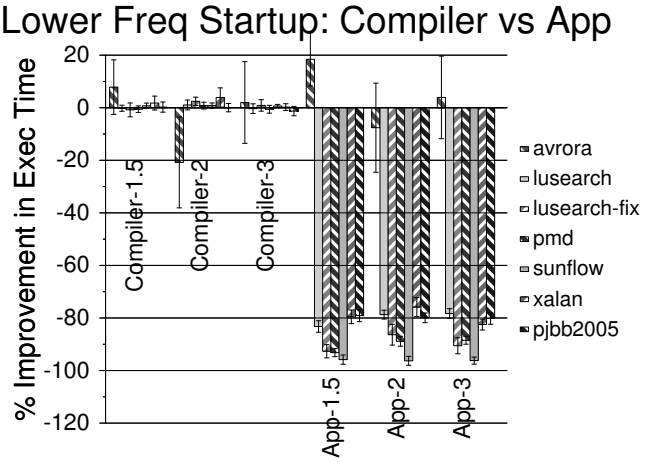

Figure 16. Percent execution time improvement when lowering frequency from highest to lowest, either compiler thread, or application (plus the rest) socket, at startup time.

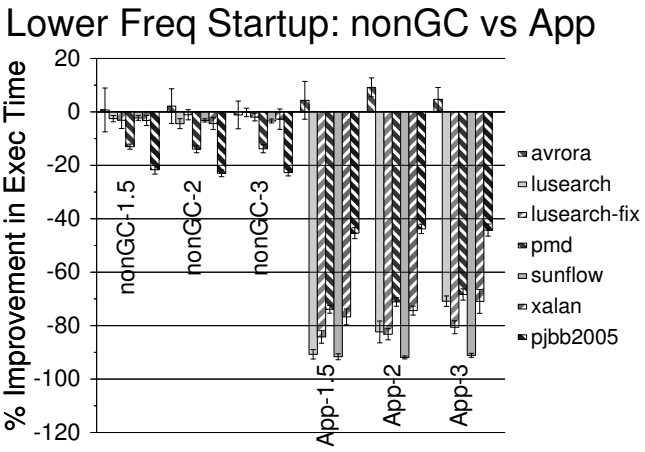

Figure 17. Percent execution time improvement when lowering frequency from highest to lowest, either JVM non-collector threads, or application (plus the rest) socket, at startup time.

isolated (above 80% in Figure 13), and when nonGC threads are isolated (less than 20%). We see similar, but less drastic, trends for pmd. Both benchmarks see more of a performance hit when isolated nonGC threads' frequency is scaled down on the left in Figure 14. The difference in performance is due to the JVM main service thread, as analyzed with performance counters, which is grouped with the application when the compiler is isolated and with nonGC threads when they are isolated. Figure 7 shows that the cost of lowering the frequency of the isolated main JVM thread is quite significant for pmd and pjbb2005.

On average, scaling JVM threads degrades performance by around 11% in comparison with 50% for application threads. As we see in Figure 15, lusearch-fix, pmd, sunflow, and xalan suffer the most from lowering application frequency, which are the same benchmarks that benefitted the most from boosting frequency in Figure 2. On the other hand, lusearch, pmd, and pjbb2005 suffer the most for scaling down JVM threads. Avrora, while having large variation in results, does not suffer as much degradation for either the application or the JVM threads when their frequency is scaled down.

Frequency scaling during startup. Here, we study the effects of frequency scaling during startup time, and juxtapose that with the previous results for steady-state performance.

Finding 9. *Scaling the frequency of JVM threads at startup time follows similar trends as scaling at steady-state time. During startup, lowering the frequency of JVM threads degrades performance three times less than when scaling down application thread frequency.*

Figure 16 shows the change in execution time when we scale down the frequency, from highest to lowest, of the isolated compilation thread versus all other threads scaled down on the other socket at startup time. This graph is very similar with Figure 13, showing that although avrora experiences performance variation, all other benchmarks are unaffected by scaling compilation thread frequency, and suffer heavily (around 80% or more) when the other socket's frequency is scaled down.

When scaling the frequency of nonGC and all JVM threads, Figures 17 and 18 show that startup time has very similar effects as during steady-state execution. When placing the application and collector threads at a lower frequency, lusearch and pjbb2005 see a slightly worse degradation for startup time as compared with steady-state time in Figures 14 and Figure 15. These benchmarks might be more sensitive to changes at startup time because of higher levels of dynamic compilation. Avrora alone seems to be largely unaffected by frequency scaling at startup time. Overall, lowering the frequency of all JVM threads at startup time

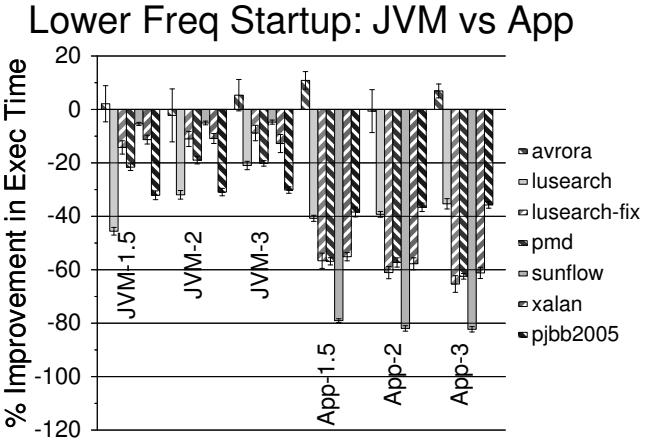

Figure 18. Percent execution time improvement when lowering frequency from highest to lowest, either JVM threads, or application socket, at startup time.

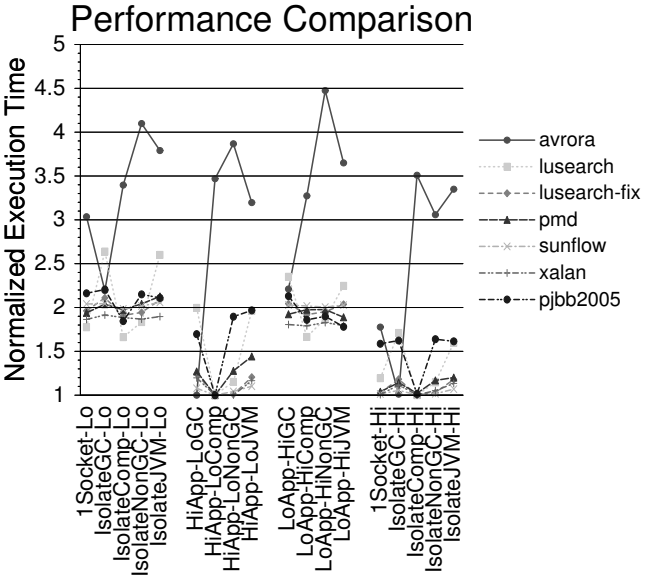

Figure 19. Execution time comparison normalized to minimum with different thread mappings and scaling frequencies, at twice the minimum heap size.

degrades performance on average by 16%, and with application threads 47%. While these degradations are slightly less than steady-state time's degradations of 10% and 50%, respectively, we observe the same trends.

Overall best performance. Figure 4.3 compares overall execution times of all benchmarks as we move from lower to higher frequencies (left to right) and explore isolating various JVM threads, and scaling the frequency of either the isolated or application and other threads. These results present runs for two times the minimum heap size. Whereas previous graphs gave a percentage improvement in execution time over a baseline run, we present here running times normalized to the minimum time over this set of experiments (so lower is better) in order to analyze trends.

Finding 10. *When power-constrained in a multi-socket environment, it is better to either keep application and JVM service threads on one socket, and power down the other socket(s), or to isolate the compilation thread onto the second socket and lower its frequency.*

Figure 4.3's left grouping shows performance when all sockets are run at the low frequency. Moving right to the second grouping, we boost the frequency of the application and non-isolated threads. Performance generally improves (closer to one on the graph), but not always for avrora. Almost universally, going from the second to the third grouping, now keeping the first socket at the lowest frequency and boosting only the isolated JVM threads on the second socket, running time increases. Avrora shows similar trends, but unlike other benchmarks, achieves the lowest performance when *collection* threads are separated from application threads and application threads are boosted. We surmise avrora has a lot of inter-application communication and collection threads interfere with cache and bandwidth resources. Finally, the last grouping shows overall improvements when we boost the frequency of both sockets. For all benchmarks but avrora, the fastest run times come from either the configuration where the compilation thread is isolated and the other socket is boosted (HiApp-LoComp), or the configuration also with the compilation thread isolated, but both sockets boosted (IsolateComp-Hi). For benchmarks except for avrora, lusearch, and pjbb2005, the configuration with all threads running together on one socket at the highest frequency (1Socket-Hi) performs almost optimally as well.

4.4 Pairing Application and Collector Threads

Because our benchmarks have up to 17% performance degradation from isolating collector threads to another socket, here we explore the effect of splitting work between sockets without separating all application from collection threads. In this section, we pair an application and a collection thread and place half of the pairs on each socket.

Figure 20 presents results for running two application and two collection threads on each socket, with all application and collection threads pinned to cores. The graph shows both running all threads on one socket, and isolating collection threads to a second socket versus this paired-and-divided configuration. We present results at high and low frequencies for two times the minimum heap size.

Finding 11. *Other than the anomalous avrora benchmark which likely has high levels of inter-thread communication, applications benefit more from pairing collector threads together with application threads while running on multiple sockets, with performance comparable to running on only one socket.*

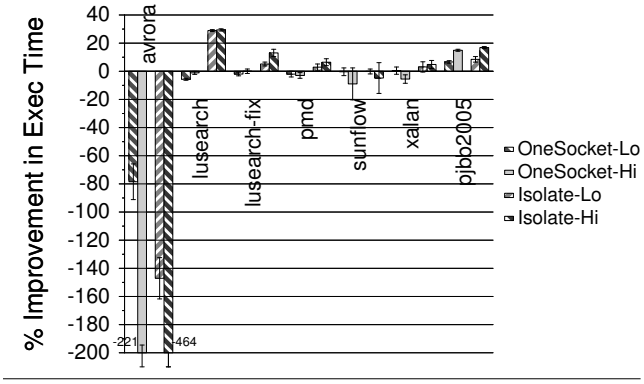

Figure 20. Percent execution time improvement going from all threads on one socket and from isolating four collector threads on a second socket to pairing application and collector threads and placing half each socket, at twice the minimum heap size.

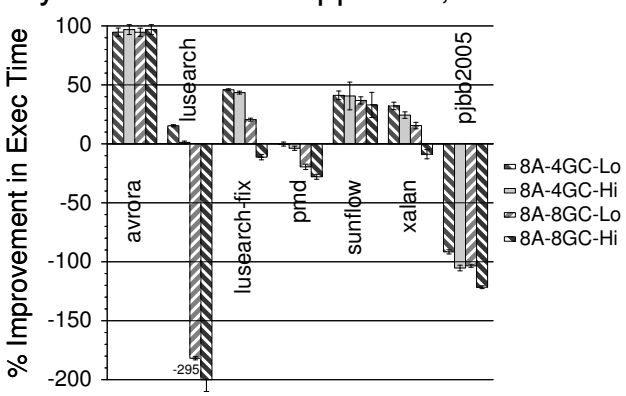

Figure 21. Percent execution time improvement when pairing application and collector threads and placing half on each socket, varying the number of application and collector threads, at twice the minimum heap size.

In comparison with all threads on one socket (left bars), lusearch, lusearch-fix, pmd, sunflow, and xalan have almost the same running time when pairing and dividing application and collection threads. At the highest frequency, sunflow degrades performance by 9% because of more communication through memory, while pjbb2005 has improved performance by 15% by using twice the last-level cache as with one socket. Unfortunately, although avrora benefitted from separating collector threads, dividing application threads costs up to 220% of performance. Upon further investigation, unhalted cycle and L3 miss performance counters on the Nehalem reveal that avrora's application threads have non-uniform behavior. Some run many more cycles and incur more last-level cache misses than others. It is also possible avrora's application threads have significant data sharing, because the application threads suffer many more L3 misses when divided between sockets.

In comparison with isolating all collection threads from application threads (right bars), benchmarks mostly see positive impacts to performance when pairing and dividing application and collection threads. Again, avrora suffers heavily from dividing up application threads — this time by maximally 460%, albeit with large confidence intervals. Other benchmarks either improve (lusearch by 29% and pjbb2005 by 17%) or maintain performance by preserving some locality between application and collector threads.

We explore increasing the number of application and collection threads using the same paired-and-divided methodology. Using four application and four collector threads as a baseline, Figure 21 presents performance improvements for two times the heap size. We first increase the number of application threads to eight while keeping four collector threads, and then increase both to eight threads.

Finding 12. *When running on two sockets, surprisingly, it is not always recommended to set the number of threads equal to the number of cores. All but one benchmark benefit from setting the number of application thread to the number of cores, but only two of our benchmarks benefit from boosting the number of collection threads above four.*

The graph shows that almost all benchmarks improve performance by having as many application threads as cores (eight), but few benefit from increasing to eight collection threads. Specifically, avrora and sunflow benefit from using more application threads, and are less sensitive to the number of collector threads, improving performance by 95% and 34-40%, respectively. Other benchmarks degrade performance when going from four to eight collector threads. Lusearch's performance degrades significantly (up to 295%), but with the allocation bug-fix, lusearch-fix does not experience the large degradation when going to eight collection threads. Pseudojbb2005 alone has significant performance degradation with more application threads, around 100%. This degradation could be due to not enough work to keep application threads busy, and increases in thread-synchronization time (particularly with the main thread). In Section 4.6 we analyze the effect of varying the number of application and collection threads on one socket, but the optimal configuration highly depends on the benchmark.

4.5 The Effect of Pinning

Because all previous experiments were performed while pinning application and collection threads to cores, this section explores the effects of removing some thread-pinning on performance. The experiments presented in this section have all threads placed on one socket and are for two times the minimum heap size. Figure 22 compares pinning only the collector thread, only the application threads, and pinning no threads against a baseline of pinning all application and collector threads.

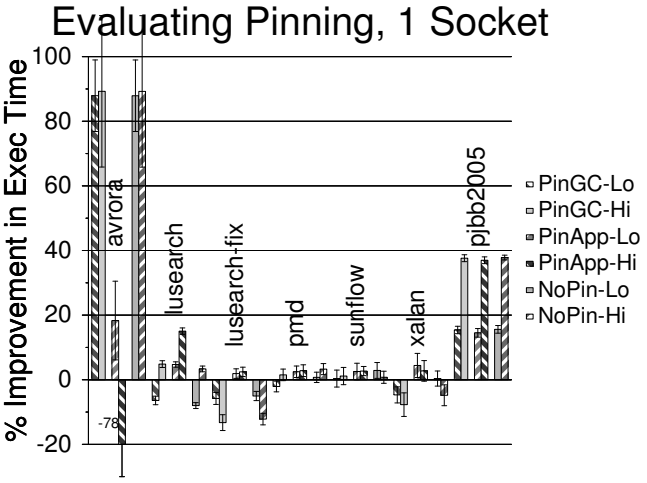

Figure 22. Percent execution time improvement when pinning only collection threads, only application threads, or with no pinning, as compared with pinning both, on one socket at twice the minimum heap size.

Finding 13. *Most benchmarks are neutral to application and collector thread pinning, but a few benefit from thread migration.*

Apart from avrora, Figure 22 shows that other benchmarks are mostly insensitive to pinning, with small exeuction time improvements from not pinning the collector (PinApp). Lusearch-fix and xalan perform slightly worse when not pinning the application (PinGC). Pjbb2005 alone clearly benefits from thread movement, and more at the higher frequency (38%) It is possible that, like avrora, pjbb2005 has more sharing between application threads, and because collection threads are not assigned work based on a particular application thread, they, too, benefit from moving to take advantage of sharing at higher levels of the cache.

Avrora is again an anomaly, having significant benefit, up to 89%, from not pinning application threads. We have surmised that avrora application threads could have a lot of data sharing, and have discussed that they have non-uniform behavior. We performed an experiment to test if avrora's large performance discrepancy between pinning and not pinning threads has to do with using only one of the two memory controllers on the socket. We re-executed the baseline experiment, pinning all application and collection threads on one socket, but this time starting pinning at core one instead of core zero (still subsequently using round-robin scheduling). With this small change, as compared with starting at core zero, avrora's execution time improves by 37 and 46% at the low and high frequencies, respectively. This surprising result reveals that avrora is particularly sensitive to how application threads are mapped to cores because perturbations radically change memory subsystem behavior. Other benchmarks did not suffer from the same artifact. Therefore, avrora is a particularly anomalous benchmark

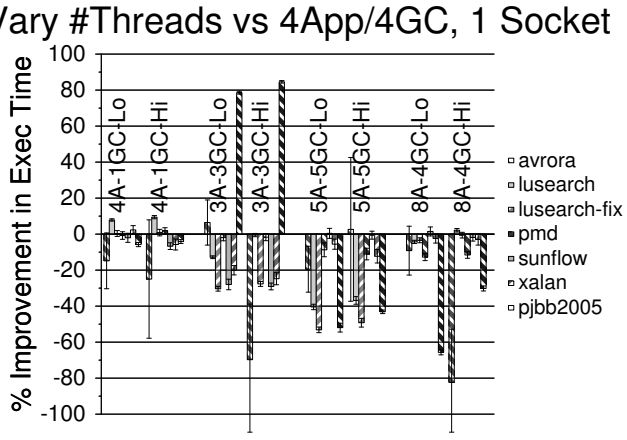

Figure 23. Percent execution time improvement when varying the number of application and collector threads, on one socket at twice the minimum heap size.

that prefers grouping application threads on the same socket and letting them migrate between cores, and separating only collection threads to another socket.

4.6 Changing the Number of Collector and Application Threads

Finally, we explore changing the number of application and collector threads, limiting experiments to one socket. Our baseline experiment is always with four application and four collection threads. We first analyze the effect of decreasing the number of collector threads while keeping the application threads at four. Then, keeping the same number of application and collector threads, we vary the number between three and five. And finally, we compare increasing the number of application threads from four to eight while holding the collector threads at four. All results are presented in Figure 23 at two times the minimum heap size.

Finding 14. *When running on only one socket, the performance sweet-spot is setting the number of application threads and the number of collection threads equal to the number of cores.*

Figure 23 shows that almost all benchmarks suffer slightly or remain neutral from lowering the number of collection threads from four to one, with four application threads. Avrora in particular has degraded performance. Only lusearch benefits, but this is due to its excessive allocation as lusearch-fix is performance-neutral. Because parallel collector threads steal work from a list of pointers to identify live data and are not accessing only core-local data, lusearch could be a pathological case where the collector threads are doing more coordination than useful work. We investigated the performance for lusearch and lusearch-fix going from one to four collector threads in Figure 24, with times normalized to lusearch with one collector thread (at each respective frequency), and lower numbers representing better

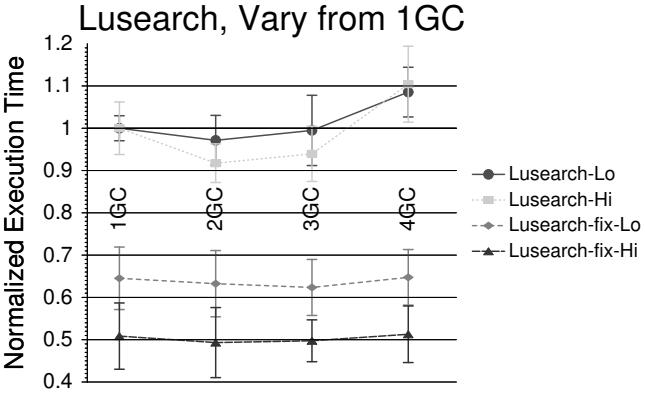

Figure 24. Execution time comparison for lusearch and lusearch-fix varying the number of collector threads normalized to lusearch with one collector thread, on one socket at twice the minimum heap size.

times. Although lusearch benefits for up to three collector threads, performance degrades significantly with four collector threads especially at a higher frequency. The same graph shows that lusearch-fix does not suffer in the same way and obtains better overall performance.

Figure 23 also compares running with three and five application and collector threads. Except for pjbb2005, all benchmarks degrade in performance when increasing or decreasing threads from the 4-4 configuration. Pseudojbb2005 surprisingly performs 85% better with only three application and collector threads, probably because these are sufficient to perform the computational work for the input set, and more threads just increase inter-thread communication. The right grouping in Figure 23 increases the number of application threads to eight, while keeping the collector threads at four. Because this experiment is performed on one socket, all benchmarks suffer because many threads are contending for limited resources. In general, setting the number of collector threads equal to application threads, and equal to the number of cores, seems to obtain best performance for our multi-threaded benchmarks running on one socket.

5. Related Work

We first discuss previous work that tried to understand the performance of managed language applications, and their ramifications on power. We then discuss research that is related to dynamic voltage frequency scaling.

5.1 Understanding JVM Services' Performance and Power

Hu and John [13] perform a simulation-based study and evaluate how processor core characteristics, such as issue queue size, reorder buffer size and cache size, affect JIT compiler and garbage collection performance. They conclude that JVM services yield different performance and power characteristics compared to the Java application itself.

Esmaeilzadeh et al. [10] evaluate the performance and power consumption across five generations of microprocessors using benchmarks implemented in both native and managed programming languages. The authors considered end-to-end Java workload performance, like our work, but assumed a single socket where we use multi-socket systems. Further, we explore how isolating and slowing/speeding up JVM service threads affects end-to-end performance.

Cao et al. [6] study how a Java application can potentially benefit from hardware heterogeneity. They tease apart the interpreter, JIT compiler and garbage collector, concluding that JVM services consume on average 20% of total energy, ranging from 10% to 55% across the set of applications considered in the study. They further study how clock frequency, cache size, hardware parallelism and gross microarchitecture design options (in-order versus out-of-order processor cores) affect the performance achieved per unit of energy for each of the JVM services. Through this analysis, they advocate for heterogeneous multicore hardware, in which JVM services are run on customized simple cores and Java application threads run on high-performance cores.

There are at least two key differences between this prior work [6] and ours. First, our experimental setup considers multi-socket systems, not individual processors. Second, we focus on end-to-end Java workload performance whereas Cao et al. consider the Java application and the various JVM services in isolation. These key differences enable us to evaluate how scaling down frequency for particular JVM services affects overall Java workload performance. This is done by separating out the JVM service of interest to another socket and scaling its frequency. A number of conclusions that we obtain are in line with Cao et al. In particular, we confirm that the Java application itself benefits significantly from increasing clock frequency, and garbage collection benefits much less. We also confirm a slight improvement to application performance if the compiler is isolated, regardless of whether the isolated compiler runs at the highest or lowest frequency. However, we also obtain a number of conclusions that are quite different from Cao et al. Whereas Cao et al. conclude that high clock frequency is energy-efficient for the JIT compiler, we find that it has limited impact on overall end-to-end performance. Also, although reducing clock frequency for the garbage collector may be energy-efficient according to Cao et al., we find that it negatively affects end-to-end benchmark performance.

5.2 DVFS

Dynamic Voltage and Frequency Scaling (DVFS) is a widely used power reduction technique: DVFS lowers supply voltage and clock frequency to reduce both dynamic and static power consumption. DVFS is being used in commercial processors across the entire computing range: from the embedded and mobile market up to the server market. Extensive research has been done towards how to take advantage of DVFS and reduce overall energy consumption while meet-

ing specific performance targets, or improving performance while not exceeding a given power budget, either through the operating system [17], managed runtime system [23], compiler [12, 24] or architecture [14, 16, 21].

Intel's TurboBoost technology [15] provides the ability to increase clock frequency for a short duration of time if the processor is operating below its power, current and temperature specification limits. The frequency at which the processor can operate during a TurboBoost period depends on the number of active cores, i.e., the maximum frequency is higher when fewer cores are active. TurboBoost thus has the potential to improve performance by increasing clock frequency during single-threaded execution phases of a multi-threaded workload, or when few programs are active in case of a multi-program workload environment.

DVFS is typically applied across the entire chip, i.e., in case of a multicore processor, all cores are scaled up or down. For example, our experimental setup using the Intel Nehalem machine only allows for setting the clock frequency at the socket-level. However, recent work has focused on per-core scaling, see for example [8, 16, 18]. Per-core DVFS would be a useful extension to the work presented in this paper, but we leave it for future work.

Barroso and Hölzle [1] coined the term energy-proportional computing to refer to the property that a computer system should consume energy proportional to the amount of work that it performs. Ideally, a computer system should not consume energy when idling, and should only consume energy proportional to its utilization level. Our findings suggest that many benchmarks obtain very good performance when all threads run on one socket, and hence also advocating minimizing idle power.

6. Conclusions

This paper is one of the first to explore the many axes of experimental setup of multi-threaded Java applications running on multicore, multi-socket hardware, drawing novel conclusions that will help optimize running time while minimizing power. While varying the number of threads, we have shown on a single socket, the number of application and collector threads should be equal to the number of cores. On two sockets, most benchmarks benefit from pairing application and collection threads, but achieve best performance with fewer collector threads than application threads. While we found a cost, usually less than 20%, to offloading JVM collection threads to another socket, we found that then lowering the core frequency of that socket provided reasonable performance degradations in comparison with lowering the clock speed of application threads, which is very detrimental to performance. However, isolating the compilation thread is performance-neutral or improves performance; and isolating threads at startup-time is less detrimental, also sometimes improving benchmark performance, than during steady-state time. Many benchmarks achieve good performance when keeping all threads on one socket, leaving the second socket idle to also save power. However, overall execution time is lowest for all but one benchmark when the compilation thread is isolated, and for the other benchmark when the collector thread is isolated (both regardless of the isolated thread's frequency). These interesting insights will help balance time and power goals of both managed language workloads and hardware resources.

This paper is a first, but important, step towards suggesting enhancements to both operating systems and hardware to facilitate balancing performance and energy on multi-socket systems. It would be useful for the operating system to work with the virtual machine to identify JVM threads separately from application threads and offer more fine-grained control over their movement and mapping to cores. If pinning threads is discovered to be detrimental to performance based on profiling or hardware performance counters, threads could be dynamically moved. Similarly, if hardware provides support for monitoring and scaling power dynamically, on a per-core basis for finer time quanta, we could scale individual JVM threads for compilation and on-stack-replacement, for example, just at startup time. We could potentially also avoid the cost of isolating collection threads if hardware could scale one core only during collection, and not during application running time. When the cost of isolation cannot be avoided, cache optimizations for prefetching or otherwise avoiding inter-socket traffic could improve memory system behavior. Our extensive analysis of end-to-end performance of multi-threaded managed applications running on multicore chips offers insights into how to design and optimize machines from application to virtual machine to operating system to memory system, down to hardware.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. We also thank Wim Heirman for his help and technical expertise on thread pinning and frequency scaling on the Intel Nehalem machine. The research leading to these results has received funding from the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295.

References

[1] L. A. Barroso and U. Hölzle. The case for energy-proportional systems. *IEEE Computer*, 40:33–37, Dec. 2007.

[2] S. M. Blackburn and K. S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator locality. In *Programming Language Design and Implementation (PLDI)*, pages 22–32, Tuscon, AZ, June 2008.

[3] S. M. Blackburn, M. Hirzel, R. Garner, and D. Stefanović. pjbb2005: The pseudojbb benchmark. URL `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, 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, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, Oct. 2006.

[5] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffman, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Communications of the ACM*, 51(8):83–89, Aug. 2008.

[6] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *The 39th International Symposium on Computer Architecture (ISCA)*, pages 225–236, June 2012.

[7] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, Oct 1974.

[8] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An integrated quad-core Opteron processor. In *Proceedings of the International Solid State Circuits Conference (ISSCC)*, pages 102–103, Feb. 2007.

[9] H. Esmaeilzadeh, E. R. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *38th International Symposium on Computer Architecture (ISCA)*, pages 365–376, June 2011.

[10] H. Esmaeilzadeh, T. Cao, Y. Xi, S. M. Blackburn, and K. S. McKinley. 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)*, pages 319–332, June 2011.

[11] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Languages, Applications and Systems (OOPSLA)*, pages 57–76, Oct. 2007.

[12] C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *Proceedings of the International Symposium on Programming Language Design and Implementation (PLDI)*, pages 38–48, June 2003.

[13] S. Hu and L. K. John. Impact of virtual execution environments on processor energy consumption and hardware adaptation. In *International Conference on Virtual Execution Environments (VEE)*, pages 100–110, June 2006.

[14] C. J. Hughes, J. Srinivasan, and S. V. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO)*, pages 250–261, Dec. 2001.

[15] Intel Coorporation. Intel turbo boost technology in Intel core microarchitecture (Nehalem) based processors, Nov 2008.

[16] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 347–358, Dec. 2006.

[17] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems and application to dynamic power management. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 359–370, Dec. 2006.

[18] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 123–134, Feb. 2008.

[19] G. E. Moore. Readings in computer architecture. chapter Cramming more components onto integrated circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[20] Y. Seeley. JIRA issue LUCENE-1800: QueryParser should use reusable token streams, 2009. URL `https://issues.apache.org/jira/browse/LUCENE-1800`.

[21] G. Semeraro, D. H. Albonesi, S. G. Dropsho, G. Magklis, S. Dwarkadas, and M. L. Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 356–367, Nov. 2002.

[22] TIOBE Software. TIOBE programming community index, 2011. http://tiobe.com/tpci.html.

[23] Q. Wu, V. J. Reddi, Y. Wu, J. Lee, D. Connors, D. Brooks, M. Martonosi, and D. W. Clark. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 271–282, Nov. 2005.

[24] F. Xie, M. Martonosi, and S. Malik. Compile-time dynamic voltage scaling settings: Opportunities and limits. In *Proceedings of the International Symposium on Programming Language Design and Implementation (PLDI)*, pages 49–62, June 2003.

[25] X. Yang, S. Blackburn, D. Frampton, J. Sartor, and K. McKinley. Why nothing matters: The impact of zeroing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 307–324, Oct 2011.