Microarchitectural Characterization of Production JVMs and Java Workloads

Jungwoo Ha
Dept. of Computer Sciences
The University of Texas at Austin
habals@cs.utexas.edu

Magnus Gustafsson
Dept. of Information Technology
Uppsala Universitet
Magnus.Gustafsson.5755@student.uu.se

Stephen M. Blackburn
Dept. of Computer Science
The Australian National University
Steve.Blackburn@anu.edu.au

Kathryn S. McKinley
Dept. of Computer Sciences
The University of Texas at Austin
mckinley@cs.utexas.edu

Abstract

Understanding and comparing Java Virtual Machine (JVM) performance at a microarchitectural level can identify JVM performance anomalies and potential opportunities for optimization. The two primary tools for microarchitectural performance analysis are hardware performance counters and cycle accurate simulators. Unfortunately, the nondeterminism, complexity, and size of modern JVMs make these tools difficult to apply and therefore the microarchitectural performance of JVMs remains under-studied. We propose and use new methodologies for measuring unmodified production JVMs using both performance counters and a cycle accurate simulator. Our experimental design controls nondeterminism within a single measurement by using multiple iterations after a steady state is reached. We also use call-backs provided by the production JVMs to isolate application performance from the garbage collector, and where supported, the JIT. Finally, we use conventional statistical approaches to understand the effect of the remaining sources of measurement noise such as nondeterministic JIT optimization plans.

This paper describes these methodologies and then reports on work in progress using these methodologies to compare IBM J9, BEA JRockit, and Sun HotSpot JVM performance with hardware performance counters and simulation. We examine one benchmark in detail to give a flavor of the depth and type of analyses possible with this methodology.

1. Introduction

Despite the growing use of Java, performance methodologies for understanding its performance have not kept pace. Java performance is difficult to measure and understand because it includes dynamic class loading, just-in-time (JIT) compilation, adaptive optimization, dynamic performance monitoring, and garbage collection. These features introduce runtime overheads that are not directly attributable to the application code. These runtime overheads are nondeterministic and furthermore, nondeterministic optimization plans lead to nondeterministic application performance. These issues make it difficult to reproduce results in simulation and to obtain meaningful results from performance counters. However, these measurements are key to understanding the performance of modern Java applications on modern hardware and their implications for optimizations and microarchitecture design.

A number of researchers have proposed methodologies for dealing with some of these issues. Most of these methodologies dictate an experimental design that more fully exposes important design features and/or controls nondeterminism. For example, experimental design methodologies include recommendations on which iteration of a benchmark to measure [7], forcing the adaptive optimizer to behave deterministically with replay compilation support [9], and measuring multiple heap sizes proportional to the live size of the application to expose the space-time tradeoffs inherent in garbage collection [5, 2]. Although experimental design can eliminate some sources of nondeterminism, Georges et al.[8] propose statistically rigorous analysis to allow differences to be teased out when significant sources of nondeterminism remain.

In this paper, we focus on experimental designs for using performance counters and simulation to measure and compare JVMs. We use three approaches: i) statistical analysis [8], ii) separation of JVM overheads such as the JIT and GC from the application [7, 2], and iii) a new experimental design for the special case of reducing nondeterminism for performance counter measurements on stock JVMs. In separating the GC and JIT, we build on prior work for comparing JVMs [7] and measuring garbage collectors [2]. Eeckhout et al. show that measurements of the first iteration of a Java application inside a JVM tend to be dominated by the JVM overheads instead of by application behavior [7]. They therefore recommend invoking the JVM, running the application multiple times within a single invocation, and then measuring a later iteration, or a steady state. Although many others had already been reporting steady state measurements, e.g., Arnold et al. [1], Eeckhout et al. were the first to show that first iteration measurements did not represent the application itself. Unfortunately, this methodology does not solve all our problems because JVMs do not reach the *same* steady state when invoked multiple times on the same benchmark.

This inter-invocation nondeterminism is problematic because only a limited number of performance counters can be measured at once, yet analyses such as comparing cache behavior between JVMs with performance counters or configurations in simulation, often require correlated measurements of more than one execution, e.g., the number of loads, stores, and misses needs to be the same to compare cache performance counter measurements or cache simulation configurations. We therefore introduce the following methodology to eliminate the JIT as a source of nondeterminism. Our experimental design runs the same benchmark N times

(we use 10 in our experiments) to produce a realistic mix of optimized and unoptimized code, we turn off the JIT after iteration N, and run the benchmark one more time to drain any work in the compilation queues. We then measure K iterations (N+2 to N+K+1), where K is the required number of performance counter measurements or simulation configurations we need. We also perform a full heap garbage collection between each iteration to attain more determinism from the garbage collector. All of these measurements are now guaranteed to use the exact same application code, although some intra-invocation nondeterminism from profiling and garbage collection may still occur.

We show that this methodology reduces nondeterminism sufficiently to make it possible to statistically compare the reasons for performance differences between 1.5 and 1.6 versions of the IBM J9, BEA JRockit, and Sun HotSpot JVMs using performance counters and simulation. We measure the SPECjvm98, SPECjbb2000, and DaCapo benchmarks [3, 12, 11]. In this work-in-progress report, we show the total performance results for all the benchmarks, and show the power of our methodology and analysis via some small case studies. We plan a future conference publication with a more detailed analysis of all the benchmarks on all the JVMs.

2. Methodology

Modern JVMs and modern architectures are complex and their interaction is even more complex. Thus, meaningfully measuring modern JVM performance at a microarchitectural level is a significant challenge in itself. In this section we describe our methodology for providing as much clarity and soundness in our measurements, including details of what we believe are some methodological innovations. We also describe the details of the JVMs we evaluate and the platforms on which we evaluate them.

2.1 Isolating Application Code

The execution of a JVM on a given applications consists of at least two distinct components: a) the compiled and/or interpreted user code, including its use of the Java class libraries, and b) the JVM itself. The JVM itself has multiple distinct components, including the JIT, the GC, profiling mechanisms, thread scheduling, and other aspects of the Java runtime. As Eeckhout et al. noted, these distinctions must be teased apart for correct performance analysis, but separately measuring these components is challenging [7]. For example, a compiler writer concerned with the quality of the code produced by the JIT needs to isolate the execution of application code from the JIT and GC. The JIT and GC are typically written in C or C++, and thus executing code generated by an entirely different compiler. Conversely, a JIT writer may be concerned with the performance of the execution of the JIT itself, and would therefore want to isolate and measure the JIT itself in a controlled experiment. Alternatively, a garbage collector developer needs to separately measure the performance of the collector and the application [2]. In addition to fraction of total time spent in the collector, these experiments must carefully consider the collector's influence on the application code quality. For example, the allocation policy influences the data locality of the application and the collector choice imposes different read and/or write barriers which influence application code quality [4].

In each case, it is highly desirable to tease apart the JVM performance into its different components. In prior work, we added direct support to a JVM to measure and control the JIT and GC [2], see Section 2.2 below. In comparing several production JVMs where we do not have source access, we are limited to using JVMTI [13] call-backs to identify garbage collection and application phases. We have also used the methodology espoused by Eeckhout et al.: timing steady state iterations where JIT activity is minimized, rather than early iterations where JIT activity may dominate.

Even with JVMTI call-backs, differences in JVM thread models and mapping of functions to threads complicate our measurements. Thus we developed a methodology based on the following observations. 1) Each JVM may have a different number of JVM helper threads including GC and JIT. For example, The HotSpot JVMs use more finalizer and signal dispatcher threads than other JVMs. 2) JVMTI call-backs may be asynchronous. Thus, we must carefully design our call-back methods to be race-free and non-blocking, otherwise they would cause significant perturbations. 3) The measurements must not make assumptions about the underlying thread model. For example, J9 has user-level threads mapped onto pthreads. Hence, GC and mutator threads may run on the same pthread. Since the our performance counter toolkit, PAPI [6], and the associated Linux kernel patch work at the Linux task (pthread) granularity, we must take care to observe performance counters when functionality changes within the same thread context.

We handle the above observations and isolate application code behavior relatively easily using JVMTI [13] and a suitable experimental design. In the case where a thread is application only, we simply start performance counters at thread creation and accumulate them when each thread is stopped. In the case where the GC or other JVM task starts on the same pthread that a user application was running, we store and account for intermediate counter values. To access the counter without blocking and asynchronously, we use perfect hashtable. Since the hash key is based on pthread id, it can be non-blocking and race free. JVMTI does not specify an interface for observing JIT behavior. We therefore turn off the JIT at the start of the last warm-up iteration. This step gives the JIT sufficient time to drain its work queues before the measurement iterations start.

The picture is further complicated when the JIT and/or GC operate asynchronously and/or concurrently with respect to the application. Although the above methodology isolates garbage collection and JIT activity that arise in separate threads, a concurrent garbage collector imposes a more direct burden on the mutator via read and/or write barriers. To minimize this effect, we choose non-concurrent GCs and take measurements on uniprocessors.

The above combination of call-backs, command-line flags and suitable selection of GC and uniprocessor architectures allowed us to accurately measure application performance on the stock JVMs. We feel that better support from the JVMs would greatly assist the task of performance analysis, particular on multi-core processors. In particular, we would appreciate a) better exposed call-backs for the JIT (only J9 does this), b) controlled and/or exposed concurrent or asynchronous JVM activities.

2.2 Controlling Nondeterminism

Modern JVMs use adaptive *hotspot* optimization systems to focus the optimization efforts of the JIT on the most frequently executed code. Since these systems use noisy profiling mechanisms such as sampling to drive their heuristics, they introduce significant nondeterminism into the execution of a JVM. Furthermore, they generate nondeterministic optimization plans, and thus nondeterministic application performance. Such nondeterminism introduces two distinct problems. First, when comparing systems, one must understand whether observed differences are artifacts due to noise in different optimization plans system, or statistically significant differences in the JVMs under study [8]. Second, in cases where it is necessary to take multiple measurements of a system to perform the evaluation, we must know the basis of the measurements, i.e., is the system the same or, if not, how dissimilar they are. Hardware performance counters are especially sensitive to this second issue, because of limitations in their implementation. Typically only a limited number of counters can be measured at once, and thus it is necessary to take multiple distinct measurements to perform performance analysis of a system. For example, comparing the cache

JVM	Version	Options
J9 1.5	1.5.0 SR6b	`-server -Xgcpolicy:optthruput -Xcompactexplicitgc`
J9 1.6	1.6.0 GA	`-server -Xgcpolicy:optthruput -Xcompactexplicitgc`
JRockit 1.5	1.5.0_12-b04	`-server -Xgc:parallel`
JRockit 1.6	1.6.0_02-b05	`-server -Xgc:parallel`
HotSpot 1.5	1.5.0_14-b03	`-server -XX:UseParallelOldGC`
HotSpot 1.6	1.6.0_04-b12	`-server -XX:UseParallelOldGC`

Table 1. Production JVMs configurations. We chose options that have the highest code quality, including stop-the-world parallel GC that minimizes application overhead.

behavior of two JVMs requires the number of loads, stores, and misses. For a particular JVM, we need to take multiple measurements. Of course, if the system under measure changes between each measurement, meaningful analysis becomes difficult or impossible.

We attack the first problem by running a large number of trials where each is a distinct invocation of a JVM. We then perform statistical analysis to compute 95% confidence intervals across each set of trials in all our experiments. These confidence intervals allow us to determine whether observed performance differences among JVMs are statistically significant or not.

We attack the second problem by running multiple iterations of each benchmark *within a single JVM invocation*. These measurements come *after* the JIT has reached steady state. We first perform 10 unmeasured iterations of each benchmark, and turn the JIT off after the 10th iteration. We run the 11th iteration unmeasured to drain any JIT work queues. Then, we measure K iterations. On each iteration, we gather different performance counters of interest. Since the JIT has reached steady state and is turned off, the variation between the subsequent iterations should be low. Our results show that it is indeed low: the standard deviation in total number of cycles on each measured iteration is 2.07%, as described in detail in Section 3.

Previously, we used a different methodology when measuring Jikes RVM, an open source Java-in-Java JVM, with which we have extensive experience. To solve this problem with JVM support, our research group invented and implemented *replay compilation* [9]. Replay compilation collects profile data and a compilation plan from one or more training runs, and then replays this same optimization plan in subsequent, independent timing invocations. This methodology thus can deterministically JIT compile a program, but requires modifications to the JVM. It isolates the JIT activity, since replay aggressively compiles to the plan's final optimization level aggressively instead of based on hotspot recompilation triggers. It also removes most profiling overheads associated with the adaptive optimization system, which is turned off. This methodology is preferable to the one proposed here because it requires fewer benchmark executions and provides determinism from the JIT compiler on multiple benchmark invocations. However, as far as we are aware, production JVMs do not support replay compilation. We would appreciate such support for both better performance debugging [10] and better experimental methodology more generally [9].

2.3 Evaluation Environment

This section presents our evaluation details, including the JVM versions, hardware platforms, OS support, benchmarks, and heap sizes. We compare production JVMs, publicly available for evaluation. We use IBM J9, BEA JRockit, and Sun HotSpot. We use both the latest 1.6 versions and 1.5 versions of each to which we have access. Table 1 shows the version numbers and command-line options.

Hardware and Operating System. We conducted our performance counter experiments on a single 2GHz Pentium-M processor which has 32KB 8-way separate instruction and data L1 cache, and 2MB of 4-way L2 cache. It supports 2 hardware counters, and 18 counters if multiplexed. We found multiplexing the performance counters led to variations that were too high to obtain statically significant results. To increase statistical precision, we therefore measured 2 hardware counter metrics on each iteration. We performed the experiment on 32bit Linux 2.6.20 kernel with Mikael Pettersson's perfctr patch, and used PAPI 3.5.0 library to interface to the hardware performance counters.

Benchmark	Size (MB)
compress	20
jess	16
raytrace	16
db	40
javac	40
mpegaudio	16
mtrt	40
jack	16
antlr	16
bloat	16
eclipse	80
fop	40
hsqldb	400
jython	16
lusearch	16
luindex	16
pmd	80
xalan	40
pjbb2000	400

Table 2. Heap sizes for each benchmark. We selected heap sizes that are the maximum of 16MB (minimum size required by JRockit) or 4 times minimum heap size.

Benchmarks. We run the 8 SPEC JVM 98 benchmarks, 11 of 12 DaCapo benchmarks, and pseudojbb a fixed workload variant of SPECjbb2000 which uses eight warehouses and executes 12500 transactions on each [3, 12, 11]. We omit eclipse from DaCapo because it causes J9 to crash due to the use of an out-of-date class library which contains an error that is fixed in the most recent version of Apache Harmony's class libraries.

Heap Sizes. We configure the heap size as a function of four times of minimum heap size requirement. We determine the minimum heap size experimentally on these JVM, picking the smallest size among the three JVMs. However, JRockit does not permit heap sizes smaller than 16MB, so we used 16MB if the 4 times of minimum heap size is smaller than 16MB. Fixing the heap size controls for variations within the same JVM between executions of the same benchmark due to heap size adaptations of the garbage collector based on load. Across JVMs, fixing the heap size makes comparing the collectors easier since they all have the same amount of memory to work. Variations are thus due to choice of algorithm

and its space efficiency, rather than the JVM making the heap to handle the allocation load.

3. Results

We now present a preliminary analysis of results from three performance studies. The first study uses hardware performance counters to provide in-depth insight into JVM performance. We first show bottom line performance numbers for each benchmark. To show the usefulness of this approach, we examine the performance of one program, the DaCapo jython benchmark, in detail (Section 3.1 and Figures 2 through 6). Our second study uses a cycle accurate simulator to reveal information not available with performance counters. We show micro-instruction issue and retire rates, micro-instruction mix, and store-to-load forwarding behavior. The third study uses the simulator to perform a simple limit study which is not possible with real hardware: we measure JVM performance with a 'perfect' cache to determine how much performance is lost due to poor cache memory performance.

3.1 Performance Counters for Performance Debugging

We use 40 distinct performance counter metrics to analyze the performance of the six JVMs on a Pentium-M platform. Figure 1 shows the normalized total execution time for each JVM on each of the 19 benchmarks. These graphs report the normalized number of executed cycles for the N+2th (11th) iteration of each benchmark, averaged over 10 invocations/trials. In each case, the result is normalized to the JVM with the slowest average time. Error bars indicate 95% confidence intervals, indicating the level of inter-invocation noise.

In many cases, the JVMs perform comparably and when there are small differences, these differences are within the error bars, which indicates they are not significant. In the cases where there are large differences between JVMs, e.g., jython, pmd, javac, jbb2000, and hsqldb, they are, for the most part, statistically significant with 95% confidence because the error bars do not overlap. JRockit has the most variation of the JVMs, see antlr as an example. This variation can account for the differences between it and the other two JVMs. Additional measurements may be needed on JRockit to provide higher confidence, or we may need additional methodologies that controling the unresolved source of nondeterminism in JRockit. We leave to future work more in-depth analysis of all the benchmarks, but perform an in-depth analysis of jython as a case study. We chose jython because the purpose of these tools is performance debugging, and jython presents a particularly striking performance problem where HotSpot performs much better than J9.

Figures 2 through 6 show detailed performance counter data for the jython benchmark. We present 40 metrics, starting with total performance, shown in Figure 2(a). Unless otherwise noted, all data are normalized to the JVM with the highest (worst) result. Figure 2(b) shows time spent in the application ('mutator'), and Figure 2(c) presents the fraction of total time in GC solely as a function of the JVM. We also show the number of retired and reissued instructions and fraction of stalled cycles. For the remainder of the results (Figures 3 through 6), we isolate the GC, so the results are for the *mutator only*. Figure 3 presents metrics relating to the mutator data cache. Figure 4 shows metrics for the mutator instruction cache and I-TLB. Figure 5 shows mutator cache coherency and hardware interrupt metrics. Finally, Figure 6 shows mutator branch behavior for jython. Together, these data reveal a rich story behind the poor performance of jython on J9.

Garbage Collection. Figure 2(a) shows that the J9-1.6 performance lags HotSpot-1.6 by around a factor of two. The first point to note is that a large part of this problem appears to be due to J9's garbage collector. Figure 2(c) shows that while HotSpot spends around 7% of the time in GC, J9-1.6 spends nearly 38% of the time in GC on jython. Recall that the benchmarks are executed in fairly generous heap sizes (at least 4 $\times$ the minimum).

Instruction Count. However, GC is not the only problem for J9 on jython. Figure 2(b) reveals that J9-1.6 also spends around 35% more time in the mutator than HotSpot-1.6. Figure 2(e) shows us that about half of this (about 16%), is simply due to J9-1.6 retiring more instructions. Thus about half of the mutator slowdown on jython is due to high level differences, such as different class library implementations or high level optimizations, each of which may lead to a difference in the number of retired instructions on a fixed Java workload. Understanding these high level inefficiencies better may be helped by studying the bytecode execution patterns for each JVM or via more careful measurement of the libraries. Our focus here, however is on low-level inefficiencies, which we can explore via more detailed performance metrics.

Data Cache. Figure 3 examines the data cache performance for each JVM on jython. We see in Figure 3(c) that J9 performs about 40% more memory accesses than HotSpot. Even accounting for the 16% higher total instruction count, it is clear that J9 is accessing memory more often. There are numerous possibilities for this increase, including more register spills. Although J9 misses the L1 data cache about 10% less *frequently* than HotSpot (Figure 3(b)), its *total* L1 misses are still nearly 20% worse (Figure 3(a)). It is interesting to note (Figure 3(e) and (f)) that J9 suffers fewer misses on loads than the other JVMs, but much more misses on stores. Figures 3(g) and (h) show that J9 misses the data L2 nearly five times as often as HotSpot and about ten times as often as JRockit. This data suggests that in addition to increased data accesses, J9 suffers some locality problems on jython, when compared to the other JVMs.

Instruction Cache. Figure 4 examines the instruction cache performance for each JVM on jython. The error bars indicate that these metrics are particularly sensitive to inter-invocation nondeterminism. This is unsurprising since the primary source of nondeterminism is in the optimization plan which will directly impact instruction access patterns. Nonetheless, it is interesting to note that J9 hits the L1 more often than the other JVMs, but misses the L2 significantly more often. The variation in I-TLB is so great that the error bars extend well beyond the edges of the graph, suggesting that the I-TLB is particularly sensitive to inter-invocation nondeterminism.

Cache Coherency. Figure 5 shows that J9 issues far more requests for clean cache lines and for cache line intervention. However, it is not clear that these are the source of any of the slowdown on the uniprocessor Pentium-M we are measuring in this study.

Branch Behavior. Finally, Figure 6 shows the branch behavior of each of the JVMs on jython. Here we see that HotSpot issues more than 30% fewer branch instructions than J9. Even though about half of those could be due to HotSpot's 16% lower instruction count, this difference still indicates significantly more branches in the J9 code (Figure 6(b)). Both J9 and HotSpot take branches at about the same rate (Figure 6(e)), and have roughly the same total number of branch mispredictions (Figure 6(f)). However, J9 has a lower number of branches per cycle and a lower misprediction ratio due to its higher cycle count and branch count.

Together the above analysis reveals numerous sources of J9's relatively poor performance on jython and areas to explore to improve performance, painting a rich picture of JVM performance.

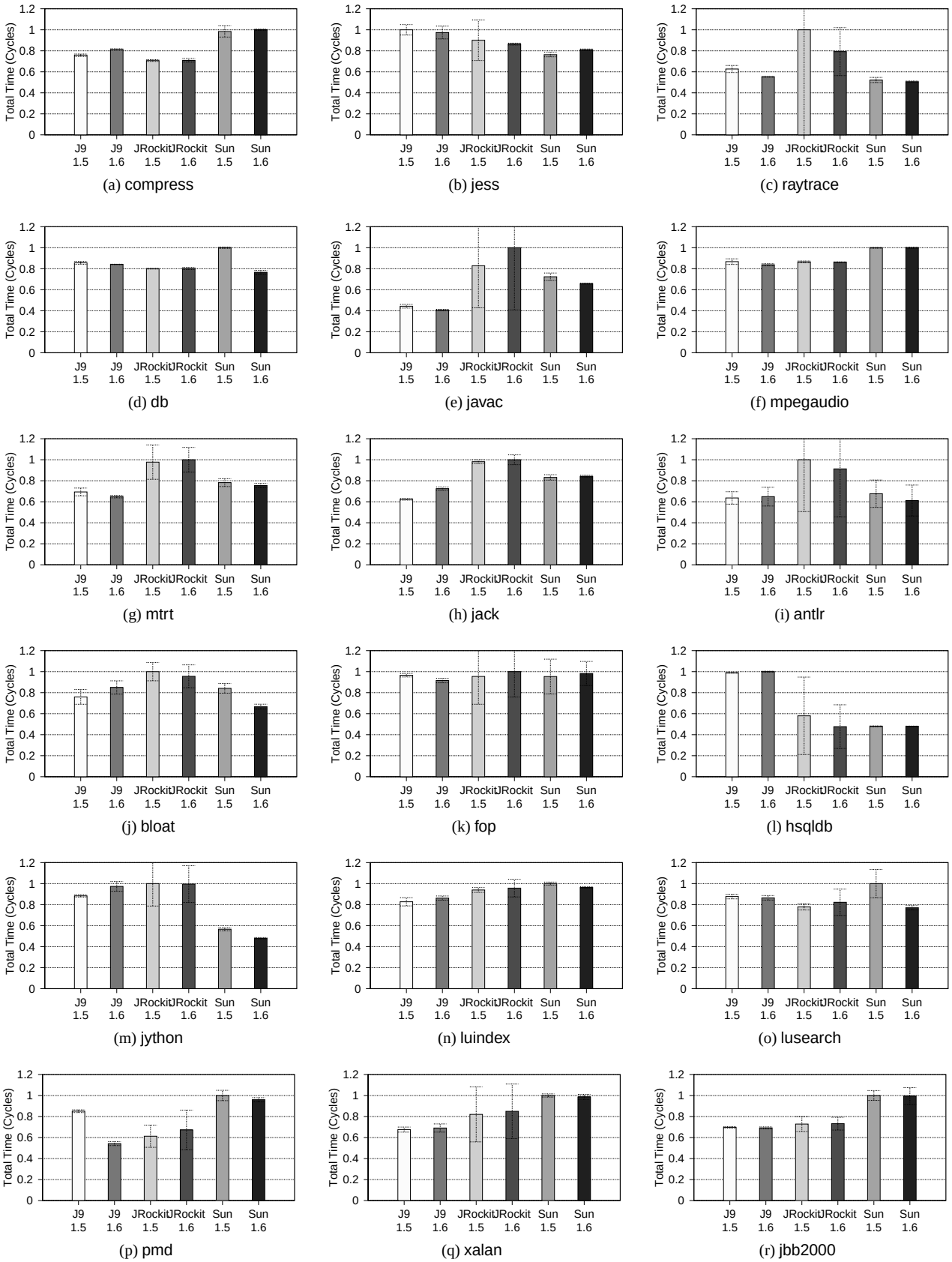
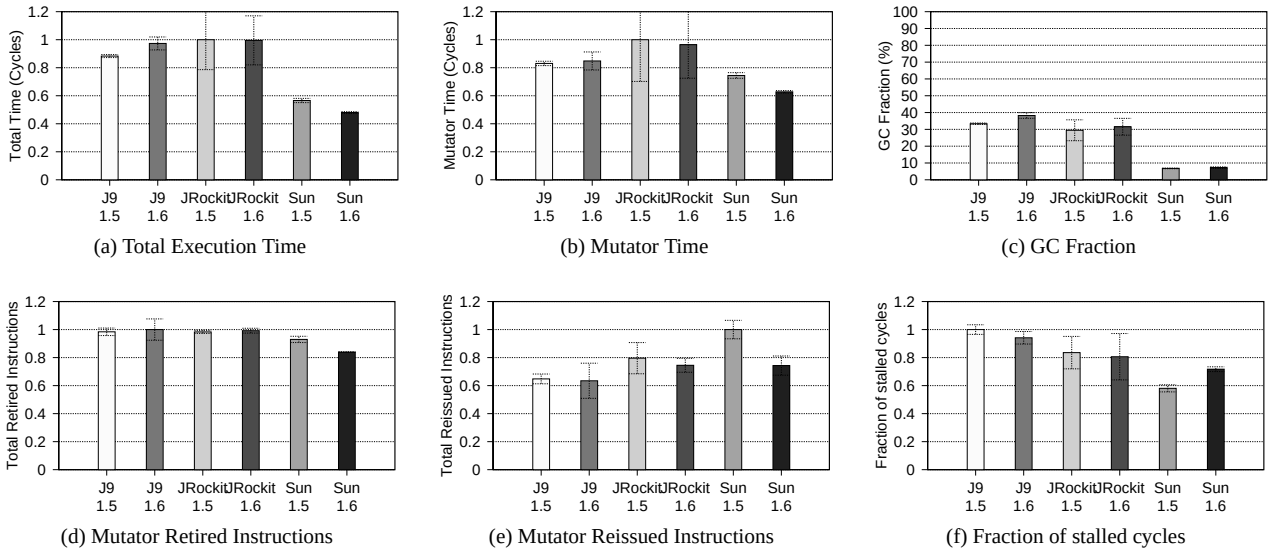


Figure 1. Total Running Time (Normalized to Slowest)



(a) Total Execution Time (b) Mutator Time (c) GC Fraction

(d) Mutator Retired Instructions (e) Mutator Reissued Instructions (f) Fraction of stalled cycles

Figure 2. Base jython Performance

3.2 Simulation Results

This section takes a deeper look at the performance of jython through the use of a cycle accurate simulator. We use PTLsim, an open source, cycle-level full system x86-64 simulator, which has been validated on the AMD K8 [14]. PTLsim provides native speed fast forwarding when executing on the same architecture as the one being simulated. This functionality is critical for our methodology because PTLsim can perform all the warm-up iterations at native speed, and only pay the full price of cycle-level full system simulation for the measured iterations. In these experiments, we configure PTLsim to model a single core AMD Athlon 3500+ processor (although PTLsim is capable of multi-core simulation). Unfortunately, we found that PTLsim could not reliably simulate JRockit, although it had no problems with J9 and HotSpot. Because of the time-intensive nature of cycle accurate simulation and JRockit problems, we present results for the 1.6 JVMs: J9-1.6 and HotSpot-1.6 in Table 1.

The amount and variety of information available from simulation is enormous. Here we explore two metrics which are not available with performance counters: a) the mix of executed micro-operations, and b) the amount of store-to-load forwarding that occurs within the load-store queues. Figure 7 shows the micro-operation mix for the two JVMs running jython. Each bar reflects the dynamic count of a particular micro-operation class as a percentage of all micro-operations. Since x86 processors decode x86 instructions into micro-operations before issue, the micro-operation mix is a key element of performance. However, micro-operations are not exposed in the ISA and are therefore not visible through the performance counters. The data reveals a number of interesting patterns. One pattern is that J9 issues significantly more loads and stores than HotSpot, which is consistent with the performance counter data in Figure 3. It is also clear that HotSpot makes much more use of the `addshift` micro-operation, whereas J9 tends to use `addsub` more. HotSpot also performs significantly more logical operations.

Figure 8 shows that J9 sees many more of its loads serviced by store forwarding than does HotSpot which may partly be a by-product of J9's greater store frequency, or it may indicate a pattern

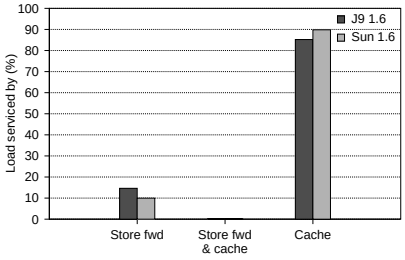


Figure 8. Store Forwarding for jython

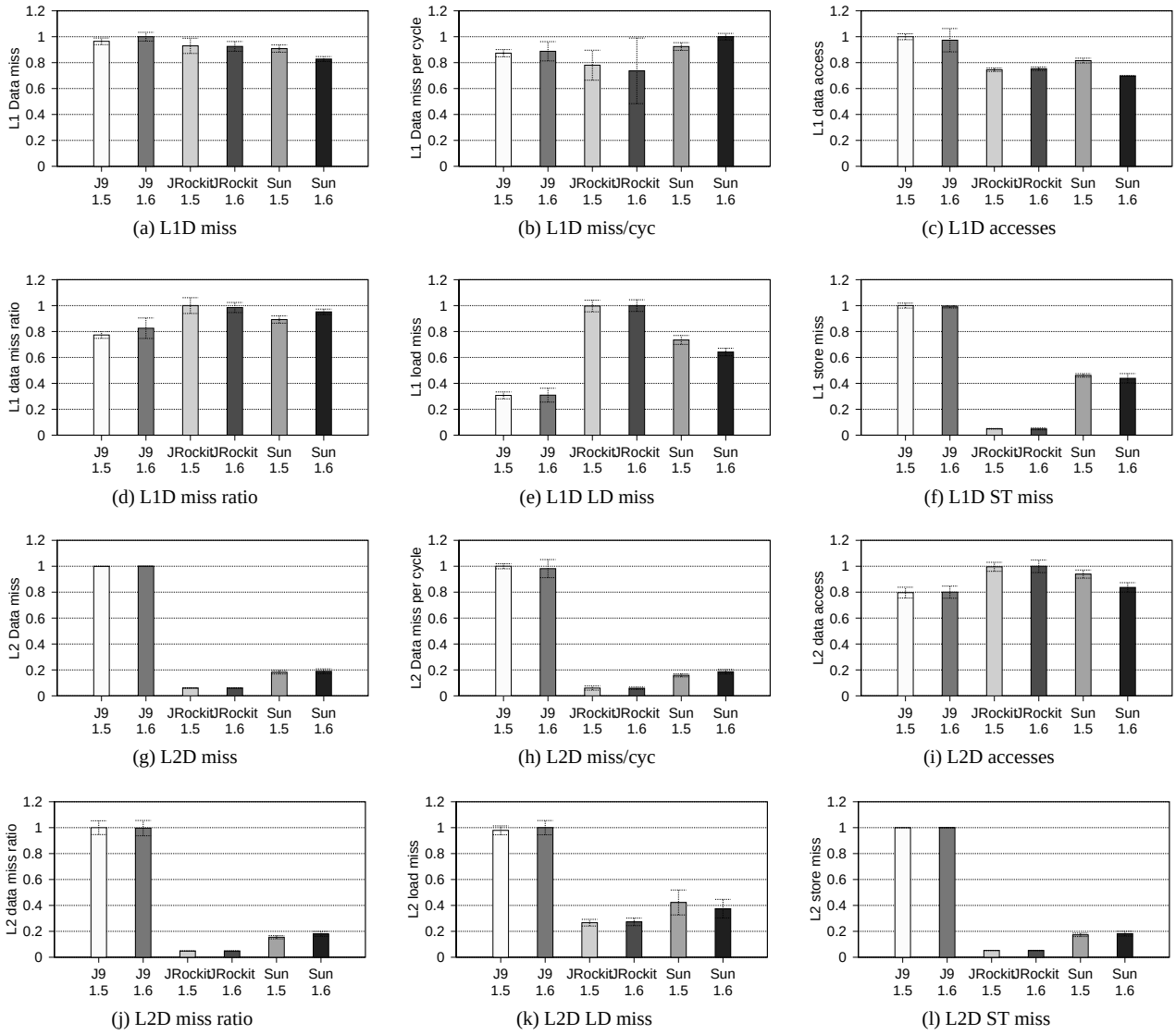
of tightly coupled stores and loads due to for example, register spills.

This data shows that the combination of performance counter results and cycle accurate simulation provide complementary and in depth performance analysis given an appropriate methodology that correlates their results.

3.3 Limit Studies with Simulation

Finally, we show an example of use of the simulator to examine the effects of radical hardware change. In this case, we explore the effect of a 'perfect' cache—one that never misses. These data were gathered before we developed our multi-invocation methodology for controlling nondeterminism (Section 2.2), so required us to run two distinct experiments, one with the regular AMD K8 3500+ cache, and one with the perfect cache. In the future, we will collect this data with the two cache configurations being explored in consecutive iterations of the same invocation to attain results that are more comparable. A related variation on this methodology is to take a machine state snapshot at the end of the warm-up iterations, and then start each of a set of measurement simulations with the same 'warmed up' snapshot. We have not yet used this methodology, but for some simulators, it will be an easier choice.

The data in Figure 9 shows the impact on IPC (instructions per cycle) of a perfect cache, and thus indirectly indicates the IPC overhead due to real cache latencies. Here we just show results for J9-



(a) L1D miss (b) L1D miss/cyc (c) L1D accesses

(d) L1D miss ratio (e) L1D LD miss (f) L1D ST miss

(g) L2D miss (h) L2D miss/cyc (i) L2D accesses

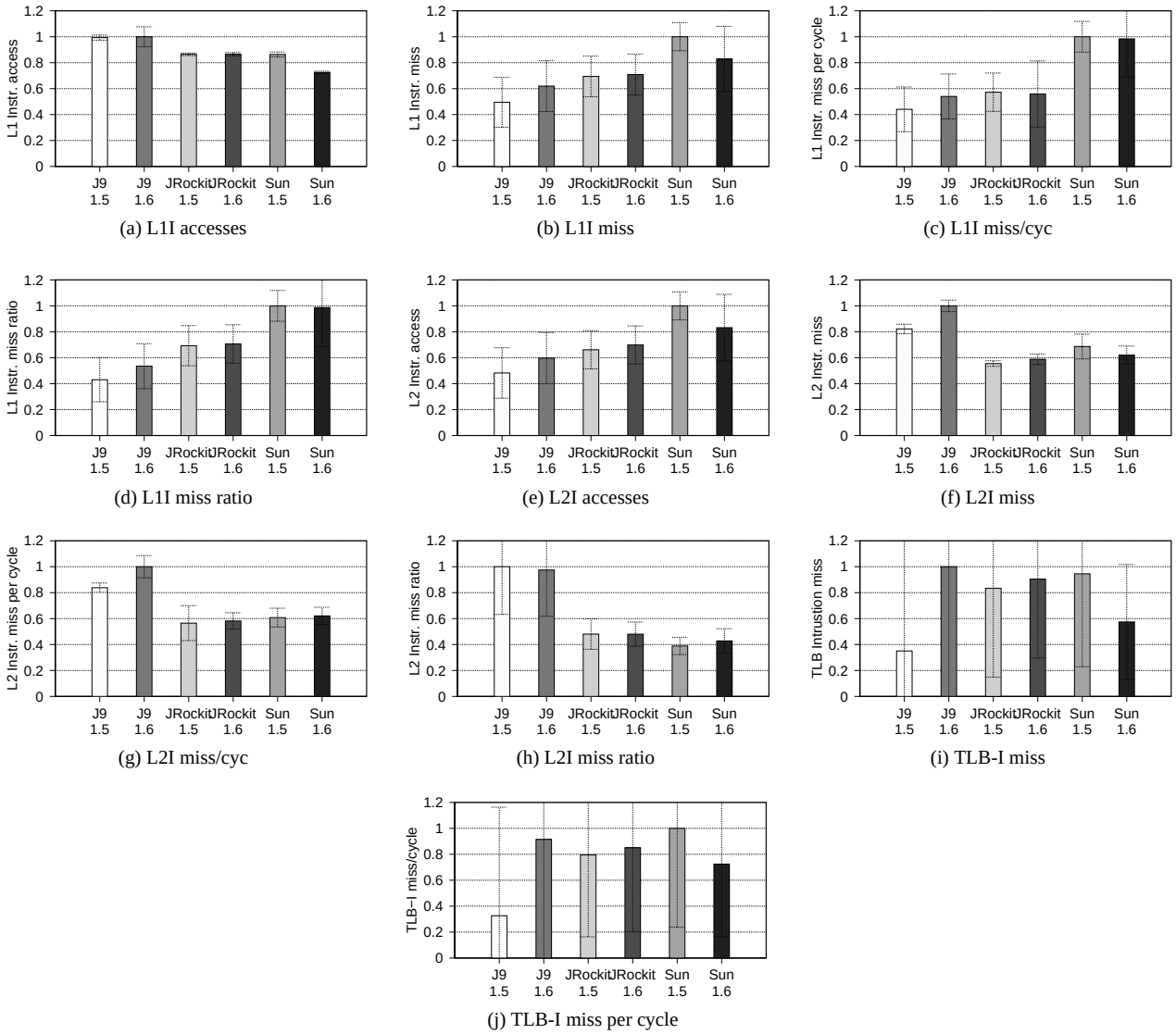
(j) L2D miss ratio (k) L2D LD miss (l) L2D ST miss

Figure 3. Mutator Data Cache Performance For jython

1.6. In these graphs, IPC is measured three ways: 1) x86 instructions per cycle, 2) issued micro-operations per cycle, and 3) retired micro-operations per cycle. Since the hardware decodes x86 instructions into simpler micro-operations and the out-of-order processor we model re-issues on failed speculation, the retired micro-operations per cycle is the most meaningful of these metrics. Figure 9 presents IPC for three SPECjvm98 benchmarks, compress, jess and db. The small differences between perfect and real caches for compress and jess indicates that neither of these benchmarks' IPC suffers significantly with a regular cache. On the other hand, the IPC for db more than doubles with a perfect cache, and suffers particularly poor IPC on our real cache. This result is consistent with conventional wisdom that db is very sensitive to locality [2, 9].

4. Conclusion

The nondeterminism, complexity and size of modern JVMs makes it hard to apply standard tools for microarchitectural performance evaluation, such as cycle accurate simulation and hardware performance counters. In this work-in-progress report, we identify new methodologies for evaluating stock production JVMs while minimizing sources of nondeterminism and isolating the behavior of the JVM runtime from the application. Our work builds on prior approaches that isolate the JVM runtime and perform statistically rigorous JVM performance evaluation. In particular, we introduce a multi-invocation, multi-iteration methodology which provides measurements of multiple executions of a benchmark that are not perturbed by JIT compilation and that have the same starting state. We show how these methodologies can be used to gather a large number of metrics from hardware performance counters and to evaluate different hardware scenarios in simulation. We show that these new tools allow us to gain new insights into JVM performance issues on a single benchmark, and are therefore likely to to be useful more broadly in comparing JVM performance and in understand and evaluating future hardware innovations.



(a) L1I accesses (b) L1I miss (c) L1I miss/cyc

(d) L1I miss ratio (e) L2I accesses (f) L2I miss

(g) L2I miss/cyc (h) L2I miss ratio (i) TLB-I miss

(j) TLB-I miss per cycle

Figure 4. Mutator Instruction Cache Performance For jython

Acknowledgements

Special thanks to our IBM colleagues Julian Wang, Brian Hall, Frank O'Connell, Daryl Maier, Derek Inglis, and Kevin Stoodley who helped us with this work.

References

[1] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications*, pages 47–65, Minneapolis, MN, October 2000.

[2] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *Proceedings of the ACM Conference on Measurement & Modeling Computer Systems*, pages 25–36, NY, NY, June 2004.

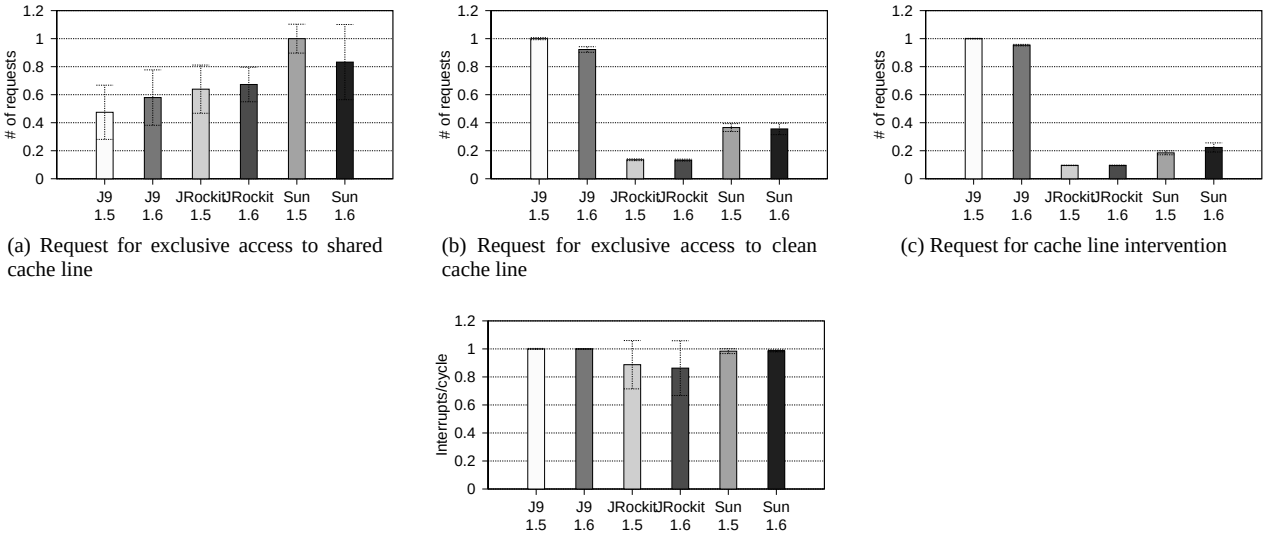
[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, Oct. 2006. ACM Press.

[4] S. M. Blackburn and A. Hosking. Barriers: Friend or foe? In *The International Symposium on Memory Management*, pages 143–151, Oct. 2004.

[5] S. M. Blackburn, S. Singhai, M. Hertz, K. S. McKinley, and J. E. B. Moss. Pretenuring for Java. In *ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications*, pages 342–352, Tampa, FL, Oct. 2001. ACM.

[6] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Supercomputing*, page 42, Washington, DC, USA, 2000. IEEE Computer Society.

[7] L. Eeckhout, A. Georges, and K. De Bosschere. How Java programs interact with virtual machines at the microarchitecture level. In *ACM*



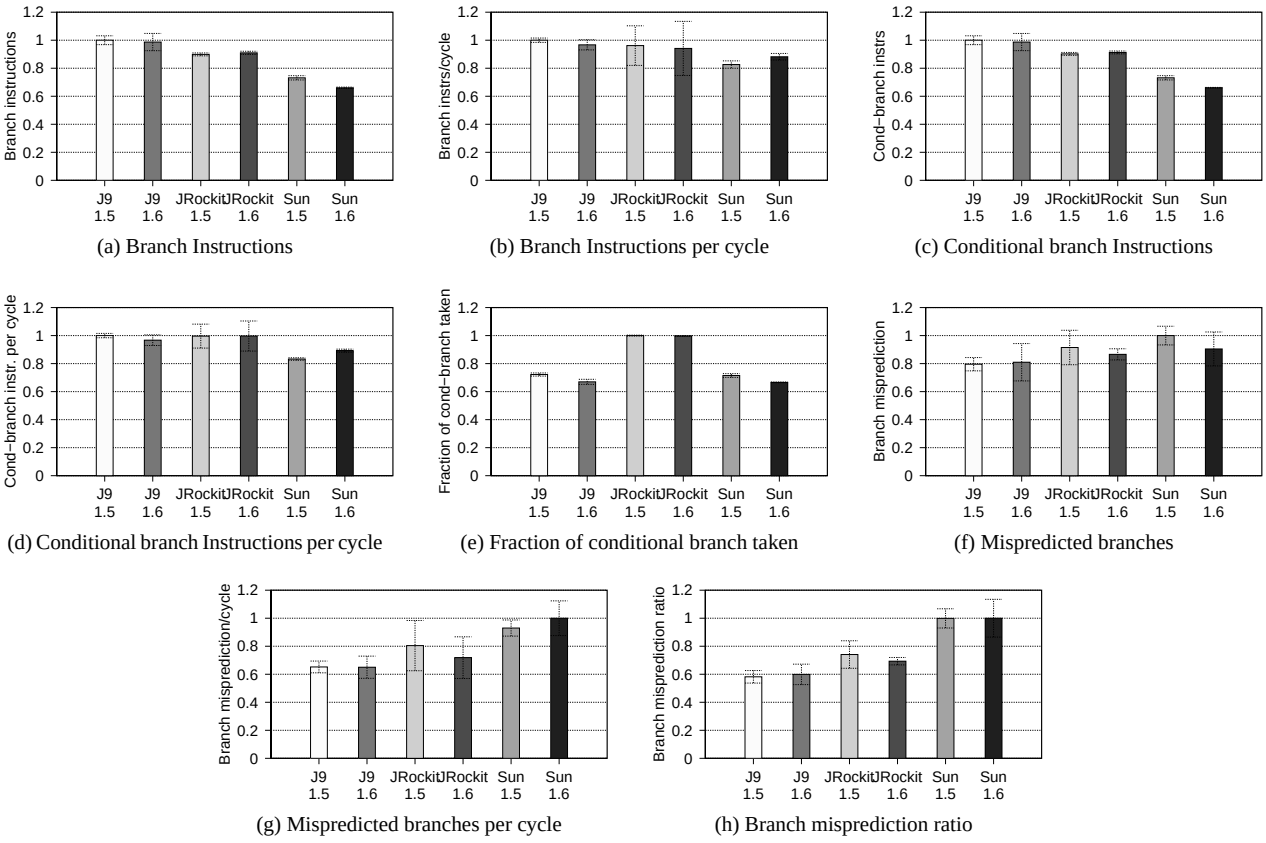
(a) Request for exclusive access to shared cache line

(b) Request for exclusive access to clean cache line

(c) Request for cache line intervention

(d) Hardware Interrupts per cycle

Figure 5. Mutator Cache Coherency and Hardware Interrupts For jython



(a) Branch Instructions

(b) Branch Instructions per cycle

(c) Conditional branch Instructions

(d) Conditional branch Instructions per cycle

(e) Fraction of conditional branch taken

(f) Mispredicted branches

(g) Mispredicted branches per cycle

(h) Branch misprediction ratio

Figure 6. Mutator Branch Behavior For jython

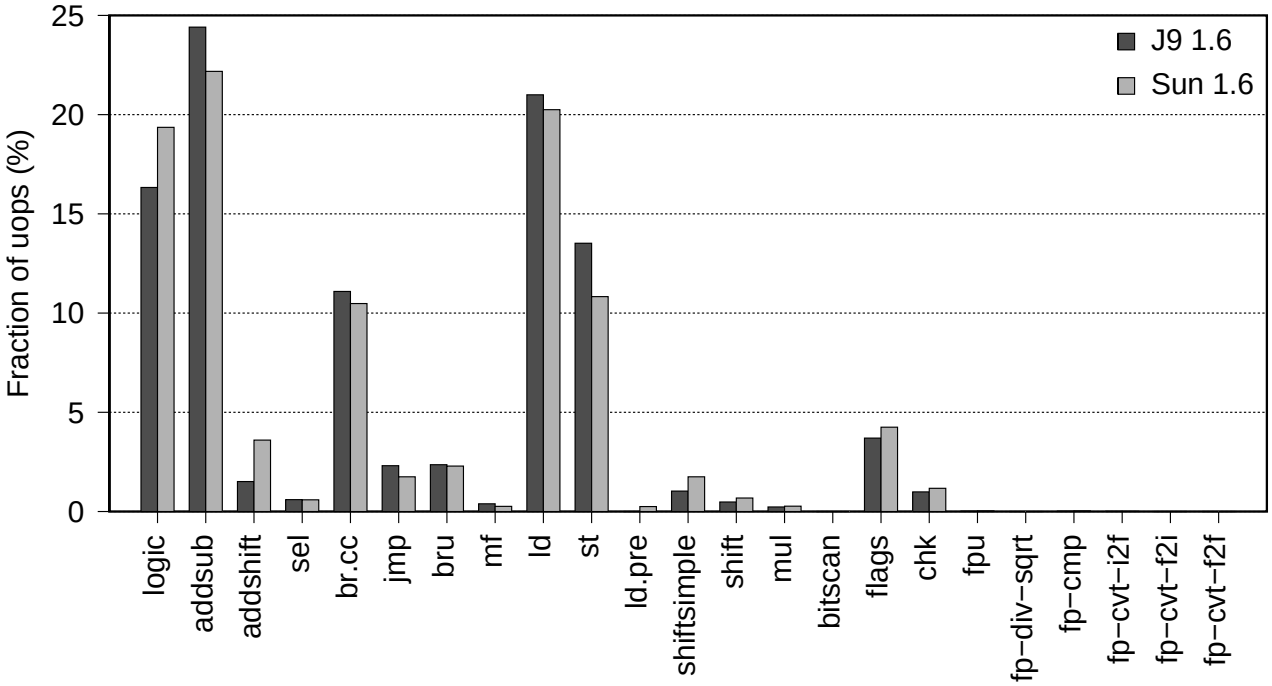


Figure 7. Micro Operation Mix for jython

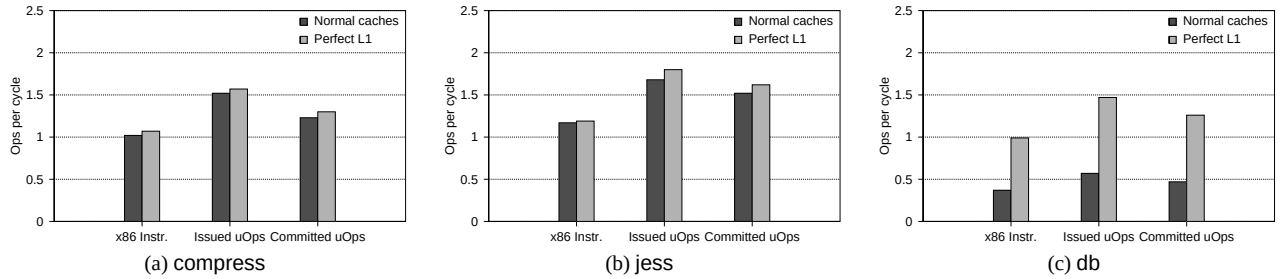


Figure 9. Effect of a Perfect Cache on IPC

Conference on Object–Oriented Programming, Systems, Languages, and Applications, pages 169–186, Anaheim, CA, October 2003.

[8] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications*, pages 57–76, New York, NY, USA, Oct. 2007. ACM.

[9] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving program locality. In *ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications*, pages 69–80, Vancouver, BC, 2004.

[10] K. Ogata, T. Onodera, K. Kawachiya, H. Komatsu, and T. Nakatani. Replay compilation: Improving debuggability of a Just-in-Time compiler. In *ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications*, pages 241–252, 2006.

[11] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

[12] Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001.

[13] Sun Microsystems, Inc. JVMTI: JVM Tool Interface. `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html`.

[14] M. T. Yourst. PTLsim: a cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS 2007: 2007 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 23–34. IEEE, Apr. 2007.