RC23704 (W0508-128) August 23, 2005
Computer Science

IBM Research Report

Characterizing a Complex J2EE Workload: A Comprehensive Analysis and Opportunities for Optimizations

Yefim Shuf
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Ian M. Steiner
Intel Corporation

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Characterizing a Complex J2EE Workload: A Comprehensive Analysis and Opportunities for Optimizations

Yefim Shuf
IBM T.J. Watson Research Center
yefim@us.ibm.com

Ian M. Steiner
Intel Corporation
ian.m.steiner@intel.com

Abstract

While past studies of relatively simple Java benchmarks like SPECjvm98 and SPECjbb2000 have been integral in advancing the server industry, this paper presents an analysis of a significantly more complex 3-Tier J2EE (Java 2 Enterprise Edition) commercial workload, SPECjAppServer2004. Understanding the nature of such commercial workloads is critical to develop the next generation of servers and identify promising directions for systems and software research.

In this study, we validate and disprove several assumptions commonly made about Java workloads. For instance, on a tuned system with an appropriately sized heap, the fraction of CPU time spent on garbage collection for this complex workload is small (<2%) compared to commonly studied client-side Java benchmarks. Unlike small benchmarks, this workload has a rather "flat" method profile with no obvious hot spots. Therefore, new performance analysis techniques and tools to identify opportunities for optimizations are needed because the traditional 90/10 rule of thumb does not apply.

We evaluate hardware performance monitor data and use insights to motivate future research. We find that this workload has a relatively high CPI and a branch misprediction rate. We observe that almost one half of executed instructions are loads and stores and that the data working set is large. There are very few cache-to-cache "modified data" transfers which limits opportunities for intelligent thread co-scheduling. We note that while using large pages for a Java heap is a simple and effective way to reduce TLB misses and improve performance, there is room to reduce translation misses further by placing executable code into large pages.

We use statistical correlation to quantify the relationship between various hardware events and an overall system performance. We find that CPI is strongly correlated with branch mispredictions, translation misses, instruction cache misses, and bursty data cache misses that trigger data prefetching. We note that target address mispredictions for indirect branches (corresponding to Java virtual method calls) are strongly correlated with instruction cache misses. Our observations can be used by hardware and runtime architects to estimate potential benefits of performance enhancements being considered.

1 Introduction

Understanding the characteristics of commercial workloads is vital to the development of hardware and software for future servers. As systems grow more complex, the interactions across different system components become increasingly difficult to understand, predict, and model. To gain insights into how these complex systems work, we need to evaluate realistic commercial workloads and to study the behavior across the software and hardware layers. The data gathered from a running system often reveals the intricacies of system behavior that can be missed on a simulator.

This paper analyzes a commercial server workload, SPECjAppServer2004[1] (*jas2004*), running on a high-end server. *jas2004* exercises the entire execution stack: software components, such as a web server, an application server, and a database; and hardware components, such as a processor, network, memory, and a disk subsystem. This study provides insights into the characteristics of a commercial workload that future system models can target and identifies avenues for improving system performance.

There were several studies of small Java benchmarks to understand Java workload characteristics and identify promising hardware and software optimizations[2, 3, 4, 5, 6, 7]. While using simple benchmarks is convenient, it is important to understand the differences between the characteristics of small benchmarks and large commercial workloads when devising hardware and software optimizations for new servers.

This paper makes the following contributions:

1. It presents a detailed characterization of a complex server workload, including high-level insights and findings on the runtime behavior across the entire execution stack.
2. It debunks common misconceptions about bottlenecks specific to Java-based workloads (such as managed memory overheads), while evaluating the significance of different software and hardware components to the overall performance of a properly tuned system.
3. It identifies bottlenecks and opportunities for performance improvement in software and hardware and guides further optimization efforts. It identifies performance optimizations that would be less effective and points out challenges of optimizing a system for such commercial workloads.

We observe that unlike commonly studied simple Java benchmarks, *jas2004* is characterized by a large multi-megabyte code footprint and a flat runtime profile, in which no single method or function is responsible for more than a small fraction (1-2%) of the overall CPU time. The *jas2004* workload is not intended to stress only the JVM and JIT (Just-In-Time) compiler, and we observe a much smaller percentage of runtime spent in both of these components. Because the CPU time is spread across a wide range of functions and

software components, it is not feasible to devise optimizations that target specific application "hot spots" and expect significant performance benefit. Rather, it will be necessary to develop new optimizations and architectural changes that show benefit across the hardware and software stack. We find that *jas2004* like other commercial workloads exhibits a relatively high CPI (cycles per instruction) which we attribute to large instruction and data working sets, a large proportion of memory instructions, and fairly high branch misprediction rates especially for indirect branches. However, unlike in other transactional workloads like TPC-C, the share of cache-to-cache transfers in *jas2004* is insignificant and this characteristic would make intelligent thread co-scheduling less useful.

This paper continues with an overview of the benchmark workload and the description of our experimental methodology. In Section 4 we describe our experimental results. Section 5 discusses related work, followed by conclusions in Section 6 and ideas for future work in Section 7.

2 Workload: SPECjAppServer2004

jas2004, a multi-tier J2EE (Java 2 Platform Enterprise Edition) benchmark, stresses major components of an enterprise system. Rather than focusing on a single aspect of a system like many past Java server- and client-side benchmarks (e.g., SPECjbb2000, SPECjvm98), *jas2004* is a realistic server application. It incorporates a web container, enterprise Java beans, and it exercises the entire underlying execution stack, including the application server, JVM, database, network and disk I/O, and processors.

As shown in Figure 1, there are two main components of the *jas2004* benchmark, the "Driver" and the "System Under Test" (SUT). The Driver simulates system users and provides benchmark's load, while the SUT is the actual system that is tested by the workload.

The SUT is made up of a web server, an application server, and a database. In our studies, all of the SUT components reside on a single system. The driver sends web requests to the SUT's web server and RMI[1] requests directly to the SUT's application server. The J2EE application server is a sophisticated, large-scale Java application that runs on top of a Java Virtual Machine (JVM) that includes a Just-In-Time Compiler (JIT). The *jas2004* benchmark code executes inside of the application server environment, and one can view *jas2004* as a control application that instructs the application server how to respond to incoming requests.

The driver is run on a separate system, and does not factor into the performance or results of the benchmark. The rate at which the driver sends requests to the SUT is called an "Injection Rate" (IR), which is preconfigured and is constant throughout execution. Because busier servers tend to have larger data sets, the IR also controls the size of the initial database on the test system. A higher IR requires higher network bandwidth between the driver and the SUT.

The benchmark performance metric is *jAppServer2004 Operations per Second* (JOPS). On a tuned system, there are approximately 1.6 JOPS executed per IR. JOPS is the workload specific performance metric, and cannot be compared to JOPS from past SPECjAppServer benchmarks or results from other transaction processing benchmarks. For a benchmark run to "pass", the run must meet certain response time requirements [1]. Namely, 90% of web and RMI requests need to be satisfied in under 2 seconds and 5 seconds, respectively.

As shown in Section 4.1.2, relatively little CPU time is spent in the benchmark code. Instead, the workload tests the software and hardware infrastructure. This behavior is different from that of benchmark programs in the SPECint and SPECfp suites, where most of the runtime is spent in the benchmark's compiled code. The behavior also differs from past Java benchmarks such as SPECjbb2000 and the SPECjvm98 suite, which were intended to stress a JVM rather than provide a realistic whole-system workload.

3 Experimental Setup

3.1 System Hardware and Software

We analyze *jas2004* workload on a SUT consisting of a single server (Figure 1). Such a configuration is well-suited for high-end servers that can run all of the workload software components (a web server, an application server, and a database server) on a single system or even a single logical partition (LPAR)[8]. A single server deployment is considerably easier to manage and this configuration tends to deliver excellent performance.

Our SUT is an IBM System p server[9] running the AIX operating system, a JVM, a web server, the WebSphere application server, and the DB2 DBMS (Figure 1). We use a 1GB Java heap – a reasonable heap size for business class servers that tend to have a lot of memory (16GB in our case). The AIX and the JVM have been configured to use large pages (16MB compared to the default size of 4KB) for the Java heap and selected garbage collector data structures. Large pages improve performance of long-running Java applications with a variety of heap sizes and especially applications with multi-Gigabyte heaps (by 10%). Many official *jas2004* results submitted to SPEC by IBM, Sun, and other vendors use large pages. Our JVM uses a flat-heap non-generational mark-sweep-compact garbage collector that is optimized for throughput. In our experiments, because of a small number of hard disks, we primarily use an operating system managed RAM disk to hold the database files, but we also cross-checked out data on the same system by adding more disks to ensure our observations still hold.

J2EE application execution requires an application server. We use WebSphere Application Server (WAS)[10]. The application server and J2EE applications that execute in its en-

[1] Remote Method Invocation: A Java technology that facilitates the creation of distributed systems by allowing for the methods of remote objects to be invoked from other JVMs

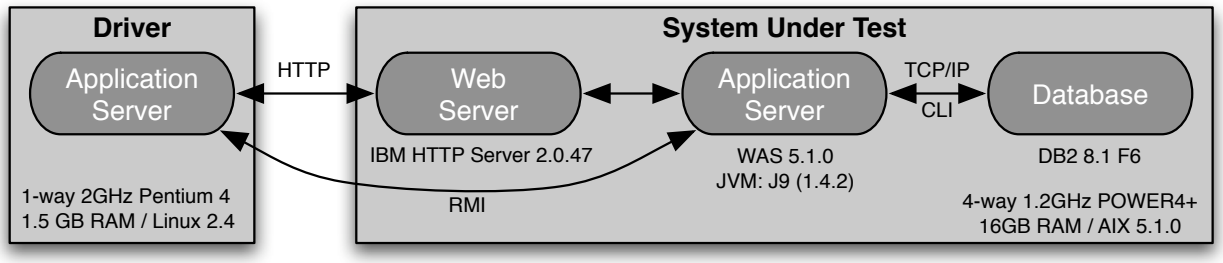


Figure 1: System Setups

vironment run on top of a Java Virtual Machine (JVM) that incorporates a Just-In-Time (JIT) compilation framework (for code optimization), a garbage collector (for memory management), and other components. This paper focuses on a recently released IBM JVM (J9 1.4.2). In the course of our study, we also evaluated another IBM JVM (Sovereign 1.4.1). While performance characteristics of these two JVMs are not identical, the general trends that we present in this paper resemble closely those that we have seen with Sovereign JVM.

The application server code interfaces with a database to provide data for web requests and RMI. We use the most current version of IBM DB2[11] database. The database and the underlying file system need to timely provide the data to achieve high system utilization. To collect meaningful data from hardware performance monitor counters, it is necessary to realize close to 0% I/O wait and idle times. On our system, we could accomplish this by using a RAM disk or by adding more disks to hold the database data.

3.2 Data Collection Tools

Data from various components of the system and its software stack have been collected using a suite of standard performance tools. For example, vmstat provides a high-level view of system performance with information about CPU utilization and memory.

The hardware performance monitor (HPM) of the POWER4 processor along with the hpmstat tool make it possible to collect accurate, sampled hardware data from counters that simultaneously track various processor events. The data presented in this paper correspond to user-level processes only, where the application server, database, and benchmark execute.

Another tool used in this study, tprof, in conjunction with information emitted at runtime by the JIT compiler, provides function-level runtime profiling. This allows for the identification of potential bottlenecks and "hot spot" sections of code. It also reveals which components of the software stack are important and can help one gauge the potential of future optimization efforts. Finally, the JVM had an instrumentation for collecting garbage collection statistics (using the *verbosegc* flag).

3.3 Experimental Methodology

We first tuned the system and removed "spurious" bottlenecks. We initially looked at the high-level characteristics across the hardware and software layers. Tuning WebSphere, DB2, and filesystem parameters, helped us get a better understanding of the high-level bottlenecks and characteristics of the workload. When tuning, we strived for a higher throughput, lower GC time, and lower idle and I/O times. We then used hpmstat to gain further insights into the low-level hardware characteristics of the system. We paid close attention to how the trends we observed in low-level hardware data mapped back to the high-level system behavior. In Section 4, we present the information in the order that we evaluated it – starting with the high-level information and then looking deeper at the low-level characteristics with careful attention to how they provide insights into our initial observations. Finally, we use statistical correlation to more accurately understand the relationships between different hardware events and CPI[12, 13].

Much of the hardware performance counter data presented in the following section were collected during a single long benchmark execution, but not at the same time – the HPM facility of POWER4 allows for a group of eight hardware counters to be active at any given time. The runtime profile of the workload is fairly constant so the collected data is representative of the entire run. Because only one group of counters can be collected at once, one cannot correlate the data across different groups of counters.

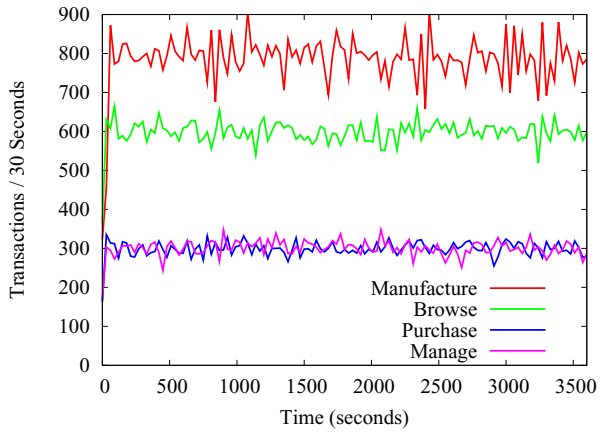
Experiments were performed during benchmark runs of at least 30 minutes. The system profiles tend to stabilize after less than 5 minutes; therefore, it is possible to collect steady-state data relatively quickly. The experiments presented here were done on the J9 JVM at IR40. The experiments had a 5 minute ramp-up time and a 2 minute ramp-down time. The paper presents only steady state data.

4 Results and Observations

4.1 High-level Characteristics

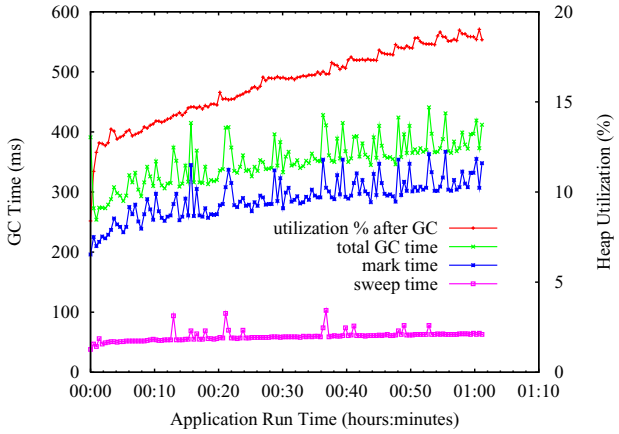
At IR47, we could maintain a close to 100% CPU utilization when using a RAM Disk for the DB2 database, with 80% of the CPU time spent in user-level code and 20% in the operating system.[2] In hard-disk based runs, we used hard disks for the database, but because of our limited disk resources (2 disks), it was not possible to achieve high system utilization because the "I/O wait times" (i.e., an idle CPU with an outstanding I/O re-

[2] At IR40, a system running with J9 JVM experiences a load level of 90%. This setting of IR is used so that (1) we can compare J9 JVM performance with that of Sovereign JVM (the latter has a higher CPU utilization at the same IR) and (2) so that the overhead of `tprof` does not overload the system, which could lead to the benchmark workload not meeting response time deadlines.



The order of the lines in the legend corresponds to the order they appear in the graph.

Figure 2: Benchmark Throughput



The order of the lines in the legend corresponds to the order they appear in the graph.

Time Between GC (sec)	25-28
GC Time (ms)	300-400
Average Percent of Runtime	1.3%

Figure 3: Garbage Collection Statistics

quest) would grow dramatically, causing the response time to grow and the benchmark to fail. We confirmed experimentally that having multiple hard disks for the database was equivalent to using a RAM disk for the data that we collected, and continued to use a RAM disk. The workload tends to enter a steady state within 5 minutes. After that, the hardware characteristics, including the system load, are roughly constant with the exception of a periodic behavior that can be correlated (as we have verified) to garbage collections.

Figure 2 shows transaction throughput rates during a 60 minute run. The graph shows that the transaction rate for each of the four different types of requests stabilizes relatively quickly, and remains fairly constant throughout execution. This behavior makes it possible to gather meaningful HPM data over time.

4.1.1 Garbage Collection

Figure 3 shows the characteristics of the GC for J9 JVM during a 60 minute run. Each data point represents one GC run. Note that the order of the lines corresponds to the order that they appear in the legend. Whether we use J9 JVM or Sovereign JVM, little CPU time is spent on garbage collection (GC). We used a reasonably large heap size (1GB), which is common for server-side workloads, but much larger than heap sizes used in many past studies with benchmarks like SPECjvm98.

The most interesting observation is the short garbage collection time. Some proponents of unmanaged code argue that a GC is unacceptably inefficient and causes major performance degradations. On the contrary, in this very complex and large-scale workload less than 2% of the CPU time is spent in GC, with collections occurring infrequently and for only a short amount of time.

Of the time spent performing GC, the "mark" phase (which identifies all reachable objects in the heap region being collected), is the most significant one, representing over 80% of the GC time. The other 20% of the GC time is spent in the "sweep" phase, which identifies the free storage. During the 60 minutes studied, there was no "compaction", which is usually performed to reduce memory fragmentation. Considering that much of GC time is spent on marking, a traversal order that respects locality during marking can reduce GC pause times and is a viable opportunity for optimization.

At the end of a 60 minute benchmark run, less than 200MB (20%) of the heap contained reachable objects. However, the amount of used heap memory is still increasing at $\sim$1MB per minute. The mark and sweep phases of garbage collection do not collect very small chunks of memory that are actually free. Such chunks of memory, which are termed "dark matter," are freed either in a compaction phase or when neighboring objects are freed (resulting in a larger chunk of free space). It appears that the "live" heap size grows largely due to this relatively insignificant fragmentation. Because the heap was not highly fragmented, and because much of the heap could be reclaimed by just performing the mark and sweep phases, there was no compaction.

Interestingly, GCs occur at a fairly predictable rate – once every 25-28 seconds. This is a convenient characteristic, as it allows us to easily correlate periodic spikes of different sampled runtime events with garbage collections. We will use this observation in Section 4.2 to evaluate how system characteristics change during GC.

4.1.2 Method Profile

Using the tprof tool we can compare CPU requirements of different software components since they all execute on the same system. This also gives us some guidance for capacity planning for a larger scale system. On our system, the WebSphere application server consumes twice as many CPU cycles as a web server and DB2 combined.

Figure 4 shows the breakdown of CPU cycles and is based on the data collected during the last 5 minutes of a 60 minute long run. Such a long run was necessary to ensure that most "important" WebSphere and *jas2004* Java methods had

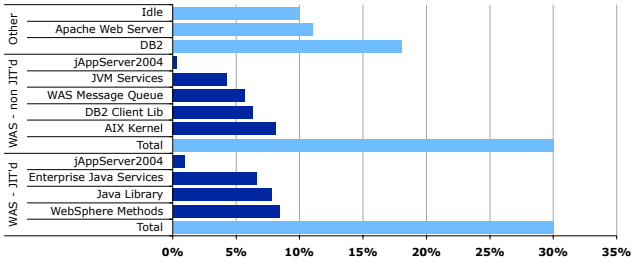


Figure 4: Profile Breakdown - % of Runtime

a chance to be profiled by the JVM runtime and then be JIT-compiled into machine code at high optimization levels. The compiler performed fairly aggressive method inlining. The two WAS related sections in Figure 4 – "WAS non JITed" and "WAS JITed" – correspond to the Java process running the WebSphere Application Server (much of which is written in Java). This process includes JITed Java code, interpreted Java code, other library code, and kernel time related to the process. We drill down into the statistics for the JITed code to get further details.

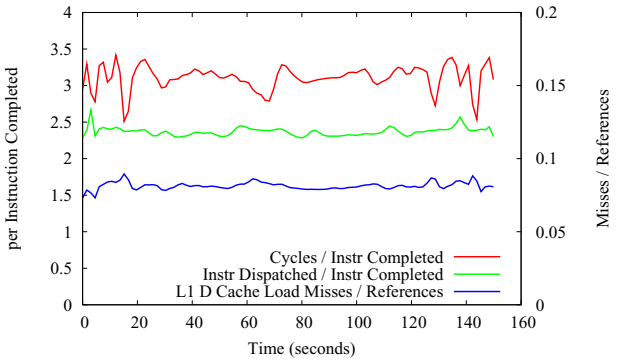
It appears that only 2% of CPU cycles was spent executing code from the *jas2004* benchmark itself. Therefore, the performance of the application code itself is relatively unimportant to the overall system performance. What is significant is how it utilizes application server features.

The method profile is very flat – the "hottest" method, a char to byte conversion method, accounted for $<1\%$ of the overall execution time. Approximately 50% of the JITed code execution time is spent in 224 methods (out of the 8500 methods). Considering that the profile is so "flat" (i.e., there is no relatively small number of methods that are responsible for 90% of execution time – so the 10/90 rule does not apply) it is not feasible to optimize a handful of methods, or to have very targeted JIT compiler optimizations in order to achieve sizeable performance gains with *jas2004*. Hence, new approaches for analyzing common patterns of instructions across methods are needed to identify opportunities for improving the quality of JIT compiled code.

About 76% of the JIT compiled code is made up of WebSphere, Enterprise Java Services, and Java Library code. The Java Library profile is not as flat as the overall profile but still does not contain any obvious hot spots. Because the "important-at-runtime" code is available to the system providers prior to deployment, this presents an opportunity for potential pre-compilation and optimization prior to execution. Unlike in many past Java application server benchmarks, half of the WAS runtime was spent running code that was not JIT compiled. This includes time in the OS code, DB2 and Message Queue library code, and the J9 VM (including the JIT itself). Hence, optimizations across software components can be beneficial.

4.2 Low-level Characteristics

One of the most important characteristics of a workload that hardware designers pay attention to is CPI (processor cycles



The order of the lines in the legend corresponds to the order they appear in the graph.

Figure 5: CPI, Speculation Rate, and L1 Miss Rate

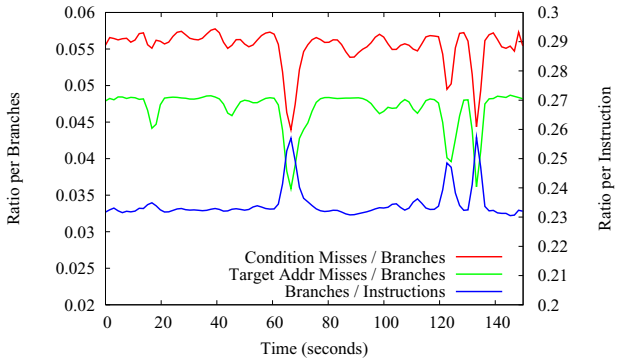
per instruction). Figure 5 shows relatively high CPI ($\sim$3) on a tuned, loaded system. The idle system has CPI of $\sim$0.7 (not shown). One contributing factor to the high CPI is a high speculation rate (shown as Instructions Dispatched / Completed). Namely, for every 5 instructions dispatched, only slightly more than 2 are retired. This high speculation rate is not entirely due to branch mispredictions, which cause wrong-path execution and misspeculation; high CPI also incorporates instructions being dispatched multiple times before their execution. For example, in POWER4, a load instruction is retried every 7 cycles on a DERAT miss until the translation information is available in the TLB. Therefore, high DERAT and TLB miss rates can also lead to higher speculation rates and, as a result, higher CPI.

One may expect to see different levels of CPI during garbage collection: possibly higher due to the size of the memory region examined by GC, or possibly lower because of better GC code locality. Generally, we do not see a strong correlation between CPI or speculation rates and GCs.

4.2.1 Branch Prediction

One of the major culprits for high CPI and speculation rates in superscalar processors have been branch mispredictions. Despite the advanced branch prediction hardware of POWER4 processors, Figure 6 shows a misprediction rate of approximately 6% on the conditions of branches and of 5% on the targets of indirect branches. It shows more branches and fewer mispredictions on a periodic rate that matches up with Garbage Collection. This result is consistent with the nature of GC codes, which tend to contain tighter loops and more predictable branches.

Polymorphism, an important feature of Java and other object-oriented languages, results in compilation of object-based methods with virtual method tables and indirect branches. If virtual method calls on varying object types are unpredictable in a dynamic instruction stream, one would expect to see many target mispredictions in the BTB. A large instruction working set may contain more branches than the prediction hardware can maintain, resulting in more mispredictions and higher CPI. We find that there is a strong correlation between target address mispredictions and instruction



The order of the lines in the legend corresponds to the order they appear in the graph.

Figure 6: Branch Prediction

cache misses. Target address mispredictions can cause useless instructions to be fetched into caches and useful instruction to be evicted. Consequently, a compiler optimization that reduces the number of call sites with indirect branches by converting them to relative branches (in common cases) can improve performance.

Poor branch prediction can result in spurious loads, which can evict useful data from caches, resulting in reduced memory subsystem performance. However, Figure 5 shows no correlation between the speculation rate and the L1 performance. This is further confirmed by using statistical correlation – a correlation of only 0.1 (Section 4.3).

4.2.2 Address Translation

The POWER4 processor (PowerPC architecture) utilizes three different types of addresses:

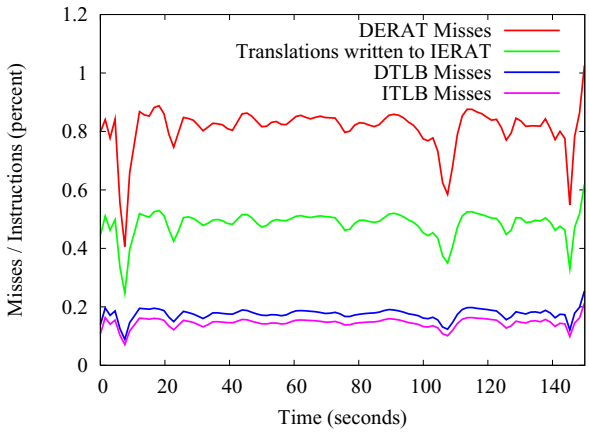
- Effective (EA): application's address space,
- Virtual (VA): unified address space, and
- Real (RA): used to access physical memory.

To perform fast EA to RA translation, there exist two effective-to-real address translation (ERAT) tables (separate for instructions and data) that are accessed in parallel with processor's L1 caches. The L1 caches are indexed on EA and tagged on RA. ERAT misses trigger TLB reads, which, along with a segment-lookaside buffer lookup, take at least 14 cycles to translate EA to RA.

Figure 7 shows the ratios of Data and Instruction ERAT and TLB misses per instruction. The top two lines correspond to DERAT and IERAT, while the bottom two lines correspond to DTLB and ITLB. Lower ratios yield better performance.

One cause of high CPI can be a relatively high DERAT miss rate. For *jas2004*, more than 100 instructions retire between DERAT misses. Upon a DERAT miss, the TLB is able to satisfy requests in 75% of cases. Hence, there are some pages in a translation working set that the ERAT cannot maintain

During GC, we see 2-3 orders of magnitude fewer ITLB and DTLB misses per instruction. Note that the graph in Figure 7 has been fitted using Bezier smoothing, and the peaks that are shown actually correspond to events that last for 0.2 to 0.3 seconds – the time for running GC.



The order of the lines in the legend corresponds to the order they appear in the graph.

Figure 7: TLB Miss Frequency

The data presented here is for a JVM that uses 16MB large pages for the Java heap, along with standard 4K pages for the rest of memory (instructions and data). Enabling large pages increases DTLB hit rates by 25%, and because of the reduced pressure on the unified TLB, ITLB hit rates also increase by 15%. This demonstrates that the usage of large pages can have a significant effect on the TLB performance of this workload.

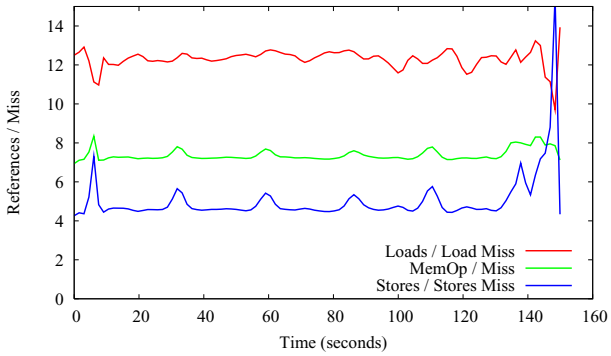
While the address translation performance is relatively good, there is room for improving ERAT hit rates by implementing object locality optimizations. Increasing the sizes of ERATs so that they could better maintain working sets could further improve overall performance. We will see later that the address translation performance is correlated to the CPI of the system. Therefore, improvements in this area will likely result in higher system performance. Further, utilizing large pages for JIT compiled code and other components of execution the stack will lead to additional performance improvements.

4.2.3 The Memory Hierarchy

We observe that for approximately every 2 instructions that are retired, either a load or store is sent to the L1 cache. In other words, there is a L1 cache access every 6 cycles. There are 4.5 retired instructions per store, and 3.2 retired instructions per load. Hence, *jas2004* is a memory-intensive workload.

The L1 DCache on POWER4 is a 2-way, 32KB cache with a FIFO replacement policy. The L1 DCache is a subset of the L2 cache (the coherence point in the system) and has a a write-through policy. L1 store misses do not evict L1 lines; rather, they write data to the L2 cache. This prevents stores from evicting "useful" data from the L1 DCache.

Figure 8 shows the L1 load and store miss rates over time. The L1 DCache misses about once every 12 loads and about once every 5 stores, averaging about a 14% miss rate overall. This is similar to miss rates on modern integer benchmarks, but is much higher than past Java benchmark results[14]. During GC, we observed a lower miss rate for stores but relatively no change to load miss rates. There is also no significant correlation between the rates of memory operations to GC activity (data not shown).



The order of the lines in the legend corresponds to the order they appear in the graph.

Figure 8: L1 Data Cache Performance

On our POWER4-based server[9], two processor cores ("siblings") reside on a chip and share an on-chip L2 cache. Four such chips (i.e., 8 processor cores) can reside on a single multi-chip module (MCM)[15], each of which has an L3 cache attached to it (L3 is off-chip while its directory and control are on-chip). Each processor can request and receive data residing in other non-sibling processor's L2s. If the data is loaded from off-chip L2 residing on the same MCM, it is labeled as loaded from L2.5. Similarly, if the data is loaded from L2 residing on a different MCM, it is labeled as coming from L2.75. *Shared* and *Modified* refer to the MESI state that a cache line was in when it was read.[3] The L3.5 cache is a L3 cache attached to a different MCM.

There are approximately 1 in 7 loads that miss in L1 DCache. Figure 9 shows where the loads that miss in the L1 are satisfied from. In the experiment, $\sim 10^9$ instructions are retired per 0.1 second sampling period. Due to limitations in the HPM system[16], it is not possible to collect L1 statistics at the same time as those shown in these graphs.

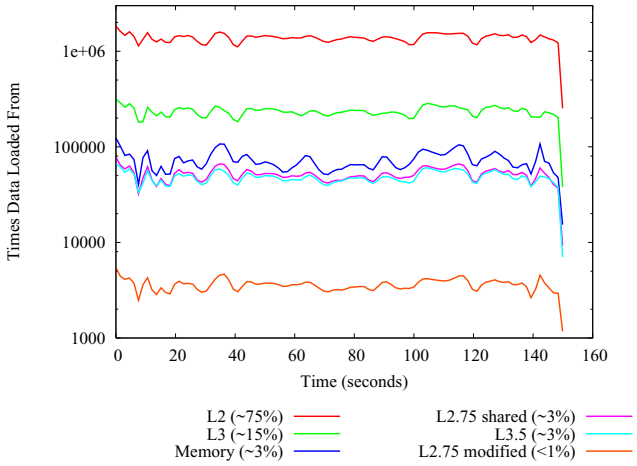
This graph illustrates that L2 cache is stressed by this workload and has only a 75% hit rate. The majority of the remaining hits come from L3 and memory, with some hits coming from from L2.75 shared and L3.5. There is very little L2.75 modified traffic.

jas2004 exhibits very little *modified* traffic across threads. Further, the data suggest that there may be little benefit from intelligent co-scheduling of threads. Reducing the data footprint of the workload could reduce pressure on L2 cache. A bigger L2 could provide some performance boost. Alternatively, a lower latency to L3 could also deliver sizeable performance benefits.

4.2.4 Locking, Contentions, and SYNC Cost

To achieve high performance in a multi-threaded application, it is essential to limit contentions for executing critical blocks of code. POWER4 has two instructions that can be used for performing locking operations – LARX and STCX – called "load and reserve" and "store conditional", respectively.[4] These instructions are used to implement an atomic "read-modify-write" operation. A LARX instruction is a load operation that also creates a reservation on a memory location from which it loads data. The reservation is to be used by a STCX instruction that follows.

[3] On our 4 CPU (4 cores) benchmark system, there are 2 MCMs, each having a two core processor enabled (with a single L2 cache shared by the 2 cores) and one L3 cache. Because our system has only a single "live" L2 on each MCM, we see no L2.5 traffic.



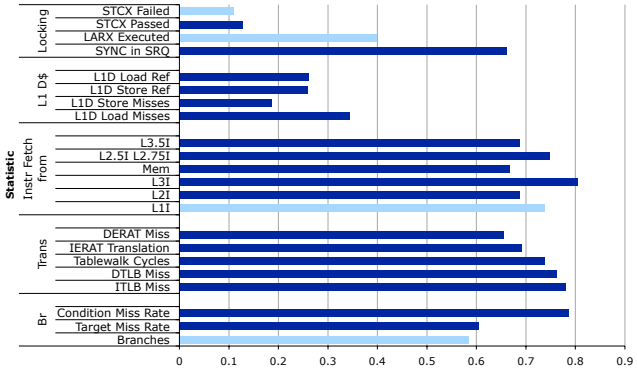
From top to bottom, the lines represent: L2, L3, Memory, L2.75 shared, L3.5, and L2.75 modified.

Figure 9: Data Loaded From (after an L1 miss)

A LARX instructions is executed about once every 600 instructions of user-level code. While 0.16% of instructions may not appear to be significant, the overhead due to locking can be noticeable. Assuming each LARX instruction is surrounded by 20 additional instructions for performing a lock acquisition, one can estimate that ~3% of instructions are spent acquiring locks. This number could be even higher because of spinning on contended locks. The amount of time spent in the pthread_mutex_lock function is a good estimate for lock contentions in a system. We see ~2% of all (user + kernel) cycles spent in this function, which suggests that despite frequent lock acquisition there is relatively little lock contention or spinlocking.

On POWER4, several different SYNC instructions (ISYNC, SYNC, LWSYNC) are used in the synchronization code to ensure proper ordering and that all previous instructions have completed before the next instruction is initiated. For example, when a SYNC instruction completes, all storage accesses initiated prior to the SYNC instruction are complete. A SYNC instruction itself can take a long time to complete and is considered to be costly. However, it has been optimized on POWER4. We measured that the fraction of cycles when a SYNC request is in the SRQ (store reorder queue) is less than 1%. Hence, the overhead of SYNC instructions is fairly small for user-level processes including WebSphere and there is little room for improvement. In contrast, during the execution of privileged code, the fraction of cycles when a SYNC request is in the SRQ is approximately 7% (data not shown). GC contains far fewer SYNC instructions compared to the frequency of these instructions during the execution of mutator (Java) threads.

[4] The ISA instructions are LWARX/STWCX and LDARX/STDCX.



Dark bars represent positive correlation, light represent negative

Figure 10: CPI Statistical Correlation (r)

4.3 CPI Correlation

Quantifying a relative correlation of processor events with CPI can yiled useful insights about system behavior and bottlenecks. Statistical correlation allows one to quantify how close two sets of sampled data are to being linearly related. Correlation can be calculated as follows:

$$\frac{\Sigma(x-\bar{x})(y-\bar{y})}{\sqrt{\Sigma(x-\bar{x})^2\Sigma(y-\bar{y})^2}}$$

The formula yileds a number between -1 and 1, where negative values represent an inverse correlation, and where the absolute value represents how closely two data sets are correlated.

Figure 10 shows correlations of several different hardware events and execution characteristics to the overall CPI of the system (for the user-level code including WebSphere). The figure shows that there is no single characteristic or a hardware event that is perfectly correlated with CPI but several events are correlated fairly strongly, and this suggest that there are some opportunities for optimization. We will review them in detail.

There is stronger correlation between load misses and CPI relative vs. between store misses and CPI. This observation confirms that load misses have a more negative impact on performance than store misses. Interestingly, overall, there is no strong correlation between L1 DCache load and store misses (i.e., L1D Load Miss and L1D Store Miss) and CPI. We also see little correlation between CPI and load and store miss rates. The data suggests that the L2 latency is sufficiently short for this workload, and the front-end is capable of supplying useful work while L1 misses are being serviced. The fact that the L3 cache satisfies about 15% of L1 misses suggests that the L2 cannot maintain the entire working set. Increasing the size of the L2 cache can improve performance.

We observe strong correlation between CPI and the number of prefetching requests from the POWER4 sequential prefetcher (L1D Prefetches, L2 Prefetches, D$ Prefetch Stream Alloc.). More prefetching requests are generated and new prefetching streams are allocated as a result of a sequence of L1 misses (a burst of misses). The data suggests that a single L1 DCache miss is often satisfied out of L2 (in 75% of cases as discussed earlier) and its impact can be hidden (POWER4 can have about 100 instructions in flight), but a burst of L1 DCache misses would have much stronger correlation with CPI because a sequence of misses would be more likely to slow down a processor pipeline.

A "speculation rate" (instructions dispatched for each instruction completed) is not correlated strongly with CPI perhaps because other factors have much more impact on CPI. The bar labeled "Cyc w/ Instr. Comp." corresponds to the number of cycles when at least one instruction was completed. A higher count of such cycles corresponds to lower CPI which is reflected by a negative correlation. Similarly, when more instructions are fetched from L1 ICache, the CPI is likely to be lower, which is also reflected by a negative correlation. At the same time, when more instructions are fetched from deeper levels of memory hierarchy (L2, L3, memory), the processor is more likely to stall, resulting in a higher CPI and positive correlation. By reducing instruction footprint one can reduce CPI. The code footprint of this workload is large and is larger than that of other commercial benchmark workloads that stress a memory system (e.g., TPC-C). The fact that the instruction cache miss data are stronger correlated with CPI than the data cache miss data implies that the workload is more sensitive to instruction fetch performance. It may be interesting to evaluate an alternative L2 replacement policy that gives instruction entries a lower probability of being chosen for eviction than data entries.

While we observed that a SYNC request was in SRQ in less than 1% of cycles, we still see a strong correlation between SYNCs and CPI. As discussed, SYNC instructions cause earlier instructions to be completed prior to the execution of later instructions, which stalls the flow of a processor pipeline. The impact of these instructions could be larger on a larger system.

Translation misses are strongly correlated with CPI. This is in spite of the fact that we use large pages for the Java heap. It appears that by improving ERAT and TLB hit rates (e.g., by placing executable and JIT compiled code to large pages, or increasing ERAT sizes) one can reduce CPI.

Branch prediction rates, particularly the conditional misses, are strongly correlated to CPI. This is to be expected considering high branch misprediction rates that we observed for this workload (Figure 6). The number of branches seems to have no correlation with the target address mispredictions (correlation of -0.07). There is some correlation between the number of conditional misses and the number of branches (0.43).

Because of certain limitations in the hardware performance counters, it is not possible to correlate CPI with various data cache counts presented in Figure 9 (such as data fetches from L2, L3, and memory).

Overall, the data shows that it is difficult to identify any major components of the architecture that need "drastic" improvement for this type of workload. However, reducing instruction and data cache footprint, code reordering, reducing target address mispredictions, increasing L2 capacity, and reducing translation misses by exploiting large pages can improve performance. Considering that a large proportion of instructions are loads and stores, low latency accesses to the L1 data cache remain important for performance.

5 Related Work

There has been several performance studies of Java client and Java server applications, particularly with the SPECjvm98 suite, SPECjbb2000, and later with previous versions of SPECjAppServer (namely, SPECjAppServer2002 and its predecessor, ECPerf). Earlier Java benchmarks were generally focused on stressing a JVM, a JIT compiler, and a GC, but unlike SPECjAppServer2004, were not designed to emulate all components of a large server application. Small Java benchmarks did not portray a realistic view of a real server workload.

Cain et al.[17] implemented the TPC-W benchmark in Java, and evaluated its memory characteristics on a 6-processor IBM RS/6000. This benchmark is similar to the *jas2004* benchmark in that it intends to simulate a commercial system including a web front-end and a database. However, it does not incorporate a J2EE application server. They found that the majority of memory stalls resulted from L2 load misses, and that a large number of these misses were serviced by cache-to-cache transfers. *jas2004* also suffers from many L2 misses, but only a fraction of L2 misses are satisfied from remote L2 caches.

Stoodley[18] reported on the difference between *jas2004* and simple benchmarks like SPECjbb2000 and SPECjvm98 applications. Their study of *jas2004* also finds that the workload has a flat method profile and low synchronization overhead.

Karlsson et al. [7, 19, 20] compared performance of SPECjAppServer2001[5] with SPECjbb2000. SPECjAppServer2001, like *jas2004*, is a multi-tier benchmark and is intended to better emulate a real commercial application. They investigated the effects of scaling (through a combination of simulation and real-system studies) on a memory subsystem.

Li et al.[4] use a complete system simulation to characterize SPECjvm98 benchmarks. They find that a large amount of runtime is spent in the OS kernel, and trace the cause back to software-managed TLB miss handling routines. We see a relatively good TLB performance with *jas2004* on POWER4 which has a hardware-managed TLB. In our experiments, a much larger fraction of time was spent in user-level processing.

SPECjbb2000 and VolanoMark were compared with integer benchmarks by Seshadri et al. [5]. They demonstrate higher L1 hit rates and slightly higher L2 hit rates in SPECjbb2000, lower CPI values, similar branch behavior, and lower levels of speculation.

Shuf et al. [14] evaluated the memory behavior of the SPECjvm98 suite and the SPECjbb2000 benchmark to identify how a JVM, a GC, and a JIT compiler can be optimized and improved. They created memory access traces on an instrumented JVM and then simulated a memory system of a processor. Some of their simulation results were then validated using performance monitor counters on a PowerPC604e system. Unlike *jas2004*, the benchmarks studied in that paper had obvious "hot" methods and "hot"instance fields and a much lower L1 data cache miss rate. Some benchmarks had high TLB miss rates.

We observed a small GC overhead when running *jas2004*. Blackburn et al. [2] evaluated several GC techniques on the SPECjvm98 suite and SPECjbb. In their experiments the heap sizes were considerably smaller and a large percentage of runtime was spent in GC.

Hauswirth et al.[12] used statistical correlation to perform "vertical profiling" to collect correlation statistics across data from different performance monitoring tools. In[13] they explore techniques for automating this process.

[5] referred to as ECperf in their paper

6 Conclusions

The paper makes the following contributions:

- It presents a detailed performance analysis of the SPECjAppServer2004 workload (*jas2004*) running on a high-end server with a state-of-the-art application server, a JVM, and DBMS.
- It discusses challenges in analyzing complex enterprise applications.
- It highlights the differences between the characteristics of small Java benchmark workloads and a large *jas2004* workload and points out that these significant differences require the development of new approaches for performance analysis.
- It identifies performance bottlenecks and opportunities for performance improvements.
- It uses statistical correlation to evaluate the relationship between different hardware events and the CPI of the system, which aids in diagnosing performance bottlenecks.

At a high-level, *jas2004* is a complex workload that is considerably difficult to setup and tune. However, it reaches a steady state fairly quickly and has a uniform performance throughout a run. This workload behavior is conducive to collecting meaningful data from hardware performance monitors. The *jas2004* workload exhibits a very flat execution profile and does not contain "hot spot" kernels like some of the smaller Java benchmarks studied in the past[14]. This presents many challenges to JIT compiler writers and creates a demand for novel profiling and code analysis techniques. Moreover, a rather significant fraction of runtime is spent not in the JIT compiled code, namely the OS and the middleware code. Therefore optimizations across OS and middleware can be beneficial.

Garbage collection contributes very little to the overall runtime of the workload (whether we use J9 JVM or Sovereign JVM). In a separate study, we observed a similar small GC runtime overhead with Trade6, another J2EE workload. Our GC results differ from previous Java benchmarking experiments because most previous studies used small Java benchmarks that spend more than 90% of time in JVM and JITed code and used relatively small heap sizes.

Unlike small Java benchmarks, this workload has a considerably larger code footprints, flat profile, no hot spots. Its executable and JIT compiled code cannot fit into a L2 cache. The observed branch misprediction rate is relatively high. Therefore optimizations like placing executable and JIT compiled code into large pages, better code reordering, better prediction of indirect branches are good directions for improving performance of this workload.

Unlike small Java applications, the *jas2004* workload has a larger fraction of loads and stores and a higher CPI. Larger data footprints cannot be accommodated by a L2 cache. A bigger L2 cache or a lower latency of accesses to a L3 cache could help improve system performance.

7 Future Work

An evaluation of the effects of scaling the number of processors on performance will be interesting as the industry moves to designs with many processor cores. Another direction is to analyze *jas2004* workload on relatively inexpensive blade systems and to place a web server, an application server and a DBMS onto a cluster of interconnected blades. Hauswirth et al. [13] devised techniques for correlating data across different components of the execution stack. The application of these techniques would further improve our understanding of *jas2004* workload. It would be interesting to compare the characteristics of the recently released DaCapo benchmarks [21] with *jas2004*.

References

[1] Standard Performance Evaluation Council, *SPECjAppServer2004 Benchmark*, 2004. `http://www.spec.org/jAppServer2004/`.

[2] S. Blackburn, P. Cheng, and K. McKinley, "Myths and Realities: The performance impact of garbage collection," in *Proc. of the 2004 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2004)*, June 2004.

[3] B. Cahoon and K. S. McKinley, "Data flow analysis for software prefetching linked data structures in Java," in *Proc. of the 10th International Conference on Parallel Architectures and Compilation Techniques (PACT 2001)*, (Barcelona, Spain), September 2001.

[4] T. Li, L. K. John, V. Narayanan, A. Sivasubramaniam, J. Sabarinathan, and A. Murthy, "Using complete system simulation to characterize SPECjvm98 benchmarks," in *Proc. 2000 International Conference on Supercomputing (ICS 2000)*, (Santa Fe, New Mexico), pp. 22–33, May 2000.

[5] P. Seshadri, L. John, and A. Mericas, "Workload characterization of Java server applications on two PowerPC processors," in *Proc. of the Third Annual Austin Center for Advanced Studies Conference*, February 2002.

[6] X. Huang, J. E. B. Moss, K. S. McKinley, S. M. Blackburn, and D. Burger, ""dynamic simplescalar: Simulating Java virtual machines"," Tech. Rep. TR-03-03, University of Texas at Austin, February 2003.

[7] M. Karlsson, K. Moore, E. Hagersten, and D. Wood, "Memory characterization of the ECperf benchmark," in *Proc. of the 2nd Annual Workshop on Memory Performance Issues (WMPI 2002)*, May 2002.

[8] J. Jann, L. M. Browning, and R. S. Burugula, "Dynamic reconfiguration: Basic building blocks for autonomic computing on IBM pSeries servers," *IBM Systems Journal*, vol. 42, no. 1, 2003.

[9] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy, "POWER4 system microarchitecture," *IBM Journal of Research Development*, vol. 46, no. 1, 2001.

[10] E. N. Herness, R. J. High, and J. R. McGee, "WebSphere Application Server: A foundation for on demand computing," *IBM Systems Journal*, vol. 43, no. 2, 2004.

[11] C. M. Saracco, M. A. Roth, and D. C. Wolfson, "Enabling distributed enterprise integration with WebSphere and DB2 Information Integrator," *IBM Systems Journal*, vol. 43, no. 2, 2004.

[12] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind, "Vertical profiling: Understanding the behavior of object-oriented applications," in *Proc. of OOPSLA 2004*.

[13] M. Hauswirth, A. Diwan, P. F. Sweeney, and M. Mozer, "Automating vertical profiling," in *Proc. of OOPSLA 2005*.

[14] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh, "Characterizing the memory behavior of Java workloads: A structured view and opportunities for optimizations," in *Proc. of the Joint International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS 2001 / Performance 2001)*, 2001.

[15] J. U. Knickerbocker *et al.*, "An advanced multichip module (MCM) for high-performance UNIX servers," *IBM Journal of Research Development*, vol. 46, no. 6, 2001.

[16] H. H. Hunter and R. Nair, "Refining performance monitor design," in *Workshop on Complexity-Effective Design (WCED)*, June 2004.

[17] H. W. Cain, R. Rajwar, M. Marden, and M. H. Lipasti, "An Architectural Evaluation of Java TPC-W," in *Proc. 7th International Symposium on High Performance Computer Architecture*, January 2001.

[18] M. Stoodley, "Challenges to improving the performance of middleware applications." Presented at 3rd Workshop on Managed Runtime Environments, 2005.

[19] M. Karlsson, E. Hagersten, K. E. Moore, and D. A. Wood, "Exploring processor design options for java-based middleware.," in *Proc. of 34th International Conference on Parallel Processing (ICPP 2005)*, June 2005.

[20] M. Karlsson, K. E. Moore, E. Hagersten, and D. A. Wood, "Memory system behavior of java-based middleware.," in *Proc. of HPCA 2003*, 2003.

[21] S. Blackburn et al, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," in *Proc. of OOPSLA 2006*.