

A Characterization of a Java-Based Commercial Workload on a High-End Enterprise Server

Ian M. Steiner ^{¶ §}

[¶]Center for Reliable High Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana Champaign
isteiner@crhc.uiuc.edu

Yefim Shuf [¶]

[§]IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
yefim@us.ibm.com

ABSTRACT

While past studies with simple Java benchmarks like SPECjvm98 and SPECjbb2000 have been integral in advancing the industry, this paper illustrates some of the characteristics of a more complex and realistic 3-Tier J2EE (Java 2 Enterprise Edition) commercial workload, SPECjAppServer2004.

In the course of this study, we both validate and disprove certain assumptions commonly made by researchers about Java workloads. For instance, on a tuned system having a heap of the appropriate size, the fraction of CPU time spent on garbage collection (GC) for this complex workload is small (<2%) compared to commonly studied benchmarks like SPECjbb2000 and SPECjvm98.

In addition to high-level statistics on garbage collection and the execution profile, detailed hardware performance characteristics, such as the branch misprediction rates and lock contention, are evaluated and used to motivate future research directions. We also use statistical correlation to evaluate and compare the relative significance of different hardware events to the overall performance of the system.

Categories and Subject Descriptors:

C.4 [Performance of Systems]: Design studies
D.3.4 [Programming Languages]: Processors: Incremental compilers, Memory management (garbage collection), Optimization, Runtime environments
B.8.2 [Performance and Reliability]: Performance Analysis and Design Aids

General Terms: Performance, Measurement, Languages

Keywords: performance, Java, JVM, memory management, garbage collection, SPECjAppServer2004, SPECjbb2000, SPECjvm98.

1. INTRODUCTION

Understanding the characteristics of commercial server workloads is vital to the development of future software and hardware. As systems have grown more complex over the years, the intricate interactions inside of a system, along with those that extend across the execution stack, have become increasingly difficult to understand, predict, and model. To understand how these large and complex systems work, it is critical to evaluate realistic applications and systems and to study the various software and hardware layers as well as their interactions. While architectural and compiler exploration requires suitable simulators, it is just as important to evaluate new ideas on actual hardware and to understand the intricacies of end-to-end system performance.

This paper presents both the data and analysis of a commercial server workload running SPECjAppServer2004 [1] (*jas2004*) on an IBM pSeries system configured with a large pool of RAM commonly found in midrange and high-end server systems. *jas2004* exercises the entire execution stack: software components, such as a web server, an application server, and a database; and hardware components, such as the processor, network, memory, and disk subsystems. This study provides insights into the characteristics that future system models can target.

2. WORKLOAD AND SETUP

jas2004, a multi-tier J2EE (Java 2 Platform, Enterprise Edition) benchmark, stresses many of the 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 for a car dealership. It incorporates a web container, enterprise Java beans, database connectivity, and it exercises the entire underlying execution stack, including the application server, JVM, database, system network, disk I/O, and processors. The benchmark code (not including Java libraries) contributes to only ~3% of the system processing time – *jas2004* is essentially a driver that tests all of the software and hardware layers.

Our System Under Test¹ is built on a 4-way 1.2GHz POWER4 IBM pSeries server running AIX 5.2.0, IBM 1.4.2 JVM (configured with a 1GB Heap), a web server (Apache 2.0.48), WebSphere Application Server (WAS) 5.1.0, and DB2 8.1 F6. The JVM has been configured to use 16MB large pages (the default size is 4KB) for the Java heap and selected garbage collection data structures. Large pages, which have been used by many official *jas2004* submissions, have been shown to improve performance of Java applications with a variety of heap sizes, particularly those with larger heaps ($\geq 1\text{GB}$). The JVM uses a flat-heap non-generational mark sweep collector which is optimized for throughput. The database is stored on a RAM disk, which was necessary in order to fully saturate the system that had limited disk resources. We have observed that this exhibits similar characteristics as a disk array.

The following results were collected on a system at 90% utilization, with the benchmark's injection rate (IR) set to 40. It was possible to fully drive the system to 100% (at IR=47), but 40 was used to allow for the various performance monitoring tools, which perturb the execution by a few percent, to be run while still satisfying the *jas2004* response time requirements. Data was collected using `trprof`, hardware performance monitor counters, and the JVM.

¹The System Under Test (SUT) is comprised of all components that are being tested including application servers/containers, database servers, network connections, etc. The driver and supplier emulator are not part of the SUT, and reside on a separate system.

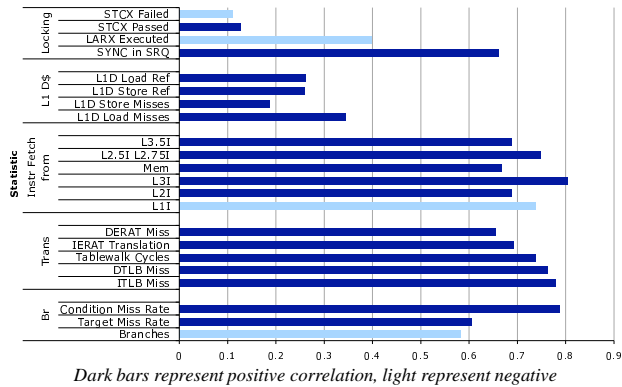


Figure 1: CPI Statistical Correlation (r)

3. RESULTS

GC: In *jas2004*, garbage collection is not as significant as in other older Java benchmarks. Garbage collections were triggered once every 23-25 seconds and only lasted for 300-400ms. This corresponds to ~1.2% of CPU cycles. Of the time spent performing GC, the “mark” phase, which labels all of the reachable objects, is the most significant, representing over 80% of the GC time. The other 20% of the time is spent in the “sweep” phase, which labels and links the free storage. During the 60 minutes studied, no time was spent in the “compaction” phase, which is performed for a variety of reasons including high memory fragmentation. Also, ~80% of the heap was free after each GC.

Execution Profile: Unlike many other Java server benchmarks, which have targeted a specific component of a JVM, *jas2004* has a flat method profile (Figure 2) that stresses many components across the execution stack [2]. The “hottest” method, a char to byte conversion method, accounted for <1% of the overall execution time. Only 30% of the total execution time is spent executing JIT’d code; half of this time is spent executing the top 224 “hottest” methods out of a total of 8500 executed methods. As such, it is not possible to simply optimize a few of the methods, or to create targeted JIT optimizations in order to achieve significant performance gains. *jas2004* has many difficult to predict branches. Target and condition mispredictions are quite common, with miss rates of 5% and 6% respectively (vs. 1%-2% for “traditional” workloads). Since such a small percentage of the CPU cycles are spent executing benchmark code (~3%), the software base that the workload utilizes is very significant to the overall performance of the benchmark.

CPI and L1 Cache: The *jas2004* workload exhibits a CPI of ~3. Figure 1 shows the statistical correlation [3] of various hardware events to the CPI. Address translation and I Cache misses are shown to be strongly correlated to the CPI (perhaps because those events tend to stall a processor pipeline). Because of limitations in the performance monitoring counters, it was not possible to correlate CPI to D Cache misses across the cache hierarchy. However, despite the fact that there is a high L1 D Cache miss rate, it appears that the workload is not sensitive to these misses (as shown by the low CPI to L1D Load and Store miss correlations). Indeed, it is common for out-of-order processors to be able to hide L1 misses. This workload stresses the memory subsystem, with ~22% of the instructions being (non-speculative) stores and ~31% of the instructions being loads (~30% of which are speculative).

Memory Hierarchy: Figure 3 shows the breakdown of where L1 load data misses are satisfied from. Most of the data are fetched from local L2 and L3 caches, and a relatively smaller fraction of data from remote, higher latency, L2.5 and L3.5 caches. There is a small percentage of fetches from remote L2.5 caches where data is in clean shared state, and an even smaller percentage where data is

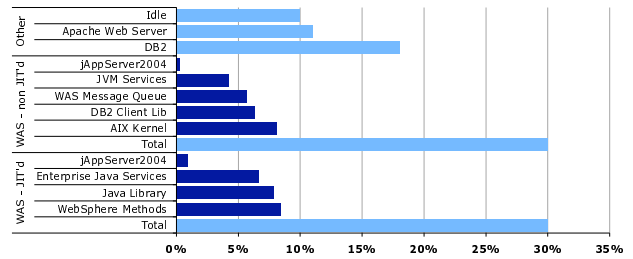


Figure 2: Profile Breakdown - % of Runtime

Cache	Percentage	Cache	Percentage
L2	77%	L2.5 Shared	2%
L3	13%	L3.5	2%
Memory	5%	Other	<1%

L1 D Cache Load Hit Rate = ~86%

Figure 3: L1 Load Miss Data Source Breakdown

in exclusive modified state (in “Other”). It appears that each thread tends to work on its own data rather than multiple threads processing and updating data stored on the same cache line.

Address Translation: Considering the large percentage of memory accesses, one may expect the address translation subsystem to be heavily stressed. The POWER4 processor performs address translation by first checking an effective-to-real address translation (ERAT) table. This table is essentially a cache for the TLB. TLB misses result in expensive page table walks. There are two ERATs, one for instructions and one for data, and a unified TLB. The ERATs miss less than once every 100 cycles. The TLB is able to pick up about 75% of these misses as well, resulting in relatively few page table walks. This, however, shows that the ERATs are heavily stressed and not able to hold the translation working set. Because address translation is strongly correlated to CPI (Figure 1), it may be useful to increase the ERAT sizes to improve overall performance. Note that disabling large pages reduces DTLB hits by 25% and, because of the increased pressure on the unified TLB, reduces ITLB hits by 15%. The ERATs do not support large pages.

Locking: POWER4 implements locks using LARX (load and reserve) and STCX (store conditional) operations. *jas2004* executes these instructions less than once every 600 and 700 instructions respectively. the STCX instruction only fails ~0.01% of the time.

4. SUMMARY

jas2004 is a complex workload with a large instruction footprint and flat profile that heavily stresses the memory subsystem. This is in contrast to SPECjvm98 and SPECjbb2000, which have been shown to exhibit code “Hot Spots” and much better L1 cache hit rates [4]. Garbage collections consume less than ~2% of cycles, much less than many other Java benchmarks like SPECjbb2000 and SPECjvm98 [5]. Branch prediction, instruction fetch, and address translation are all strongly correlated to the CPI of the workload.

5. REFERENCES

- [1] Standard Performance Evaluation Council, *SPECjAppServer2004 Benchmark*, 2004. <http://www.spec.org/jAppServer2004/>.
- [2] M. Stoodley, “Challenges to improving the performance of middleware applications.” 3rd Workshop on Managed Runtime Environments, 2005.
- [3] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind, “Vertical profiling: Understanding the behavior of object-oriented applications,” in *Proc. of OOPSLA*, 2004.
- [4] 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 *SIGMETRICS*, June 2001.
- [5] S. Blackburn, P. Cheng, and K. McKinley, “Myths and Realities: The performance impact of garbage collection,” in *SIGMETRICS*, June 2004.