A Preliminary Workload Analysis of SPECjvm2008

Hitoshi Oi
The University of Aizu, Aizu Wakamatsu, JAPAN
oi@oslab.biz

Abstract

SPECjvm2008 is a new benchmark program suite for measuring client-side Java runtime environment. It replaces JVM98, which has been used for the same purpose for more than ten years. It consists of 38 benchmark programs grouped into eleven categories and has wide variety of workloads from computation-intensive kernels to XML file processors. In this paper, we present the results of running SPECjvm2008 on three machines that have CPUs with the same microarchitecture and different cache sizes and clock speeds. The result of measurements include instruction and data cache reference and miss rates, and the effect of the multi-threading. We relate these workload parameters to the SPECjvm2008 performance metrics. Roughly speaking, an L2 cache of 1MB sufficiently lows the cache miss rates of SPECjvm2008 and compared to the single-core, 1.5 to 2 times speed-ups are achieved by dual-core executions.

Keywords Java Virtual Machine, Workload Analysis, SPECjvm2008

1 Introduction

SPECjvm2008 is a new benchmark suite from Standard Performance Evaluation Corporation (SPEC) [1]. It evaluates various aspects of a Java Runtime Environment (JRE) and replaces SPEC JVM98 [2] which has been used for the same purpose for more than ten years. During this period, Java applications and JREs (and underlying technologies) have been changed. Accordingly, SPECjvm2008 reflects such changes so that its metrics represents the performance of the current JREs more appropriately.

In this paper, we will present a brief introduction to SPECjvm2008 and analyze its workloads with emphasis on the memory access behavior. We run it on the CPUs with the same microarchitecture and different cache sizes to identify the effect of cache sizes on the performance. Thus far, we have not seen any other work that presents the effect of memory hierarchy to the performance of SPECjvm2008.

The rest of this paper is organized as follows. In the next section, the workload of SPECjvm2008, run rules and performance metrics are described. In Section 3, we present the experimental environment and the results of our performance analysis. In Sections 4 and 5, related work and conclusions of this paper are presented, respectively.

2 SPECjvm2008

In this section, we first describe the workload of SPECjvm2008 and then present its performance metrics and run rules. We do not intend (and are unable here to) fully describe the details of SPECjvm2008 and those who are interested should refer to the documents available at the SPEC web site [1].

2.1 SPECjvm2008 Workload

SPECjvm2008 consists of 38 benchmark programs that are classified into the eleven categories listed in Table 1. *compiler* consists of two benchmarks, compiler.compiler and compiler.sunflow. The former compiles `javac` itself and the latter compiles another benchmark program, sunflow, in SPECjvm2008. *compress* is a compression workload based on Lempel-Ziv method (LZW). This program is ported from SPEC CPU95 [3], but the input is a real data rather than the synthesized one in SPEC CPU95. *crypto* has three sub-benchmark programs of encryption, decryption and sign-verification using different protocols: crypto.aes (AES and DES protocols), crypto.rsa (RSA) and crypto.signverify (MD5withRSA, SHA1withRSA, SHA1withDSA and SHA256withRSA). *derby* is a database benchmark in Java and it is to replace the `db` benchmark in SPEC JVM98. It is designed to represent a more realistic application than db and to stress the BigDecimal library. *mpegaudio* is an mp3 decoding benchmark and corresponds to the benchmark with the same name in JVM98. The mp3 library of mpegaudio in JVM98 has been replaced with JLayer [4]. It evaluates the floating-point operations of the JRE. SciMark is a computational benchmark suite in Java developed by

NIST [5]. It consists of five sub benchmark programs (fft, lu, monte_carlo, sor and sparse). In SPECjvm2008, they are executed with a large data set (32MB) for testing the memory hierarchy and a small (512KB) data set for testing the JVM itself. *serial* operates in a producer-consumer scenario, where the producer serializes primitives and objects from JBoss benchmark and sends them over the socket to the consumer. These data are deserialized at the consumer. In *startup*, a new JVM is started for each benchmark in jvm2008 and it runs one operation of the benchmark. The time from starting up the JVM to the end of the benchmark operation is measured. *sunflow* is a multi-threaded rendering benchmark program. It starts with half the number of threads as the number of logical CPUs, and each of these threads spawns four threads inside the program. *xml* is made of two sub-benchmark programs: *xml.transform* and *xml.validation*. The former evaluates the implementation of *java.xml.transform* of the JRE under test, while the latter uses *java.xml.validation* to compare the XML files and corresponding XML specifications in .xsd files.

Category	Description
compiler	Compilation of .java files.
compress	Compression by LZW method.
crypto	Encryption and decryption.
derby	Database focused on BigDecimal.
mpegaudio	Mp3 decoding.
scimark.large	Floating point benchmark with
scimark.small	32MB and 512KB datasets.
serial	Primitive and object (de)serializations.
startup	JVM launch time for each benchmark.
sunflow	Graphics visualization (rendering).
xml	XML transform and validation.

Table 1. SPECjvm2008 Workload Categories

One way to classify this large variety of SPECjvm2008 categories is whether similar benchmarks existed in JVM98 or not. *compress* and *mpegaudio* are inherited from JVM98 with small changes. Similar benchmark programs existed in JVM98 for *compiler, derby sunflow*, but changes are significant. *crypto, scimark, serial, startup* and *xml* are new categories in SPECjvm2008.

2.2 Performance Metrics and Run Rules

There are two categories for running SPECjvm2008: base and peak. The main difference between these categories is whether JVM tuning is allowed or not. In our experiments, we follow the base run rules, i. e. no JVM tuning applied, except `-Xmx` option for JVM which is required to allocate enough heap space for some benchmark programs[1].

The performance metrics of SPECjvm2008 is the number of operations per minute (ops/m) which basically means that how many times the JRE under test can complete the execution of the benchmark programs. The overall ops/m (composite score) is obtained by the the following way. First, for the categories with two or more benchmarks, the geometric means of their ops/m scores are calculated to determine the ops/m of the category. Next, the geometric mean of ops/m scores over the all categories is calculated. This score represents the over all performance of the JRE under test. The execution of each benchmark program consists of two parts: warm-up and iteration. The former part is just to warm-up the JVM and its operations do not count toward the performance metrics. By default, warm-up and iteration parts are two and four minutes long, respectively.

3 Performance Analysis

In this section, we first show the hardware and software components used in the experiments and then present the results and analysis of the measurements.

3.1 Experimental Environment

We use three systems, A, B and C in Table 2. All three systems have CPUs based on the netburst microarchitecture [7]. L1 data cache sizes are 8KB in System A and 16KB in Systems B and C. Instead of a traditional L1 instruction cache, the netburst architecture has a Trace Cache, which stores micro-operations (uOps) decoded from the x86 instructions. All three systems have the same Trace Cache size of 12K uOps. L2 cache sizes for Systems A, B and C are 512KB, 1MB and 2MB, respectively. . The CPUs of Systems B and C are dual-core and only one of them is activated until Section 3.5. Memory access events are monitored by the built-in performance counters and their values are accessed using `oprofile` version 9.3.0 [8].

3.2 SPECjvm2008 Scores

Fig. 1 shows the performance metrics for the composite and each category of SPECjvm2008 benchmark normalized to those of System A. Two horizontal lines are drawn at 1.167 and 1.25 in Fig. 1. These two lines represent the relative clock speed between System A and B, and A and C, respectively. In a simplified model, the performance is proportional to "Clock Speed /CPI". Therefore, if CPI is constant, the relative performance of Systems B and C should be around these horizontal lines. The bars for crypto are

[1] Without -Xmx option, benchmarks such as scimark.large failed by the OutOfMemory error [6].

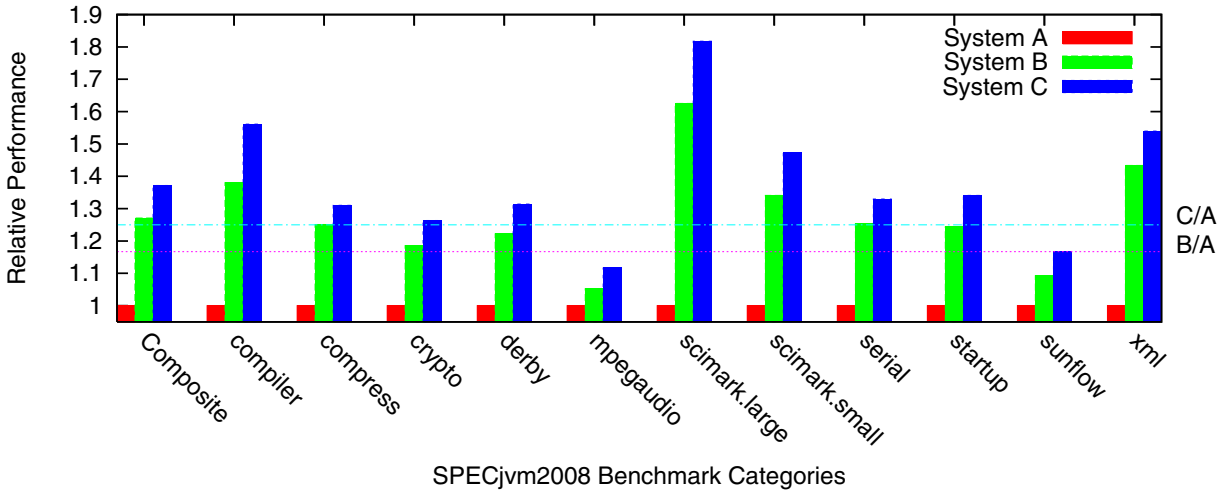

Figure 1. SPECjvm2008 Relative Performance Metrics

System	A	B	C
Processor	Pentium 4	Pentium D	
Clock Speed	2.4GHz	2.8GHz	3GHz
Trace Cache	12K uOps		
L1 Data Cache	8KB	16KB	
L2 Cache	512KB	1MB	2MB
ITLB	128 Entries		
Operating Systems	Linux (CentOS) Kernel 2.6.18		
Java Version	1.6.0_04		
JVM Name	Java HotSpot(TM) Client VM (build 10.0-b19, mixed mode)		

Table 2. Benchmarking Environments

very close to these lines. As we will see in the next subsections, its cache miss rates are small and it is possibly the reason why the performance of crypto follows this simple performance model. The performances of mpegaudio and sunflow are lower than relative clock ratios and they do not seem to take advantages of the faster clock speed and larger cache sizes of Systems B and C. The relative performances of other benchmarks are higher than clock ratios. In particular, scimark.large has achieved 63% and 82% speed-up on Systems B an C, respectively. As it was intended in its design, the working sets of scimark.large do not fit in the small size of L2 cache in System A.

3.3 Trace Cache Miss

Fig. 2 shows the Trace Cache miss rate for each benchmark category. Benchmarks that are considered to be loop-oriented (such as scimark) have very low miss rates since they execute the same decoded x86 instructions in the Trace Cache repeatedly. On the other hand, benchmarks that are considered to analyze the input and take different execution paths have very high miss rates. xml is one of such workloads and its result is consistent with our previous analysis of the XML processor in Java [9], where method invocations are very frequent and each method executes a small number of bytecodes during each invocation. As we will see in the next subsection, Trace Cache miss rates are much lower than L1 data cache miss rates. Also, the differences among three systems are relatively small, which coincide with the fact that all systems have Trace Caches of the same size (12K uOps).

3.4 Data Cache Miss

L2 cache reference rates are presented in Fig. 3. Both the Trace Cache misses and L1 data cache misses cause L2 cache references. By comparing Fig.s 2 and 3, L1 data cache misses are dominant as they occur an order of magnitude more frequently than Trace Cache misses. In compiler, compress, serial, sunflow and xml, we can see significant differences between Systems A and B. A possible explanation for this phenomena is that for these benchmarks, the difference in L1 data cache size (8KB or 16KB) is critical. For benchmarks with large working sets (especially scimark.large), both 8KB and 16KB are too small and they do not make much difference in L2 reference rates. Further investigation is required to confirm this idea.

Fig. 4 shows the L2 cache miss rate for each SPECjvm2008 benchmark category. For compress, crypto, mpegaudio and scimark.large, an L2 cache of 1MB size suffi-

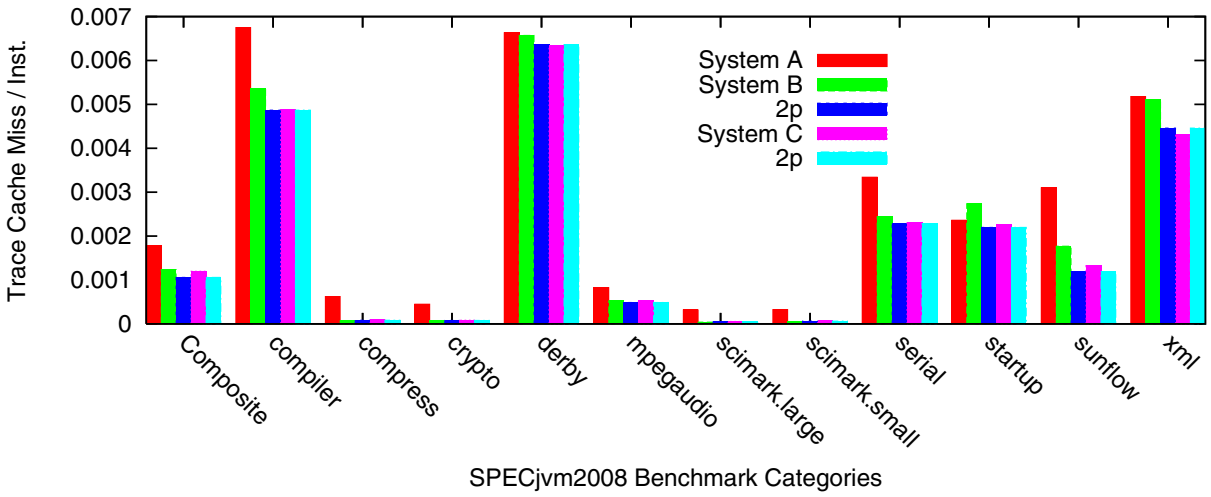

Figure 2. Trace Cache Miss per Instruction. Bars labeled as 2p are for dual-core cases which will be explained in Section 3.5.

ciently lowers the miss rates. On the contrary, the 2MB of L2 cache size further reduces the miss rates of compiler, derby, scimark.small, startup, sunflow and xml. It is interesting to see that while 2MB L2 cache is not very effective for scimark.large, it lowers the L2 cache miss rate of scimark.small from the 1MB case significantly. We may guess that scimark has two different working sets. For both the large (32MB) and the small (512KB) data sets, one of working sets can fit in the 1MB L2 cache (but not in the 512KB L2 cache). However, another working set fits in 2MB (but not 1MB) L2 cache when the data set is small, but not for the large data set size. The L2 miss rates of crypto were quite small. As mentioned in Section 3.2, this seems to be the reason why crypto achieves the speed-up close to the clock frequency ratios between Systems A, B and C.

3.5 Multi-Threading

CPUs of Systems B and C are dual-core with a dedicated L2 cache for each core. By the previous subsection, only one of two cores has been activated to compare them with System A which is based on a single-core CPU. When both cores are turned on, SPECjvm2008 spawns two working threads (except startup which is single-threaded). These two threads may compete the ownership of an L2 cache block, which causes L2 coherence misses. This coherence miss is in addition to other three cache miss categories (compulsory, capacity and conflict, or so-called 3Cs) and should behave differently against the increased cache size. Moreover, even for different memory locations, simultaneous accesses to the main memory by two cores can increase the effective memory access latency. In this subsection, we compare the performance and L2 cache miss rates in dual-core cases to the results in the previous sections.

In Fig.s 2 and 3, bars marked as 2p are the Trace Cache miss rates and L2 cache reference rates for the cases where two cores enabled, respectively. Again, the differences between single and dual core cases are relatively small for these rates. The only exception is serial, in which L2 reference rate is dropped from System B single core to dual core cases.

For the L2 cache miss rates in Fig. 4, however, dual-core cases show differences from single-core cases. Seven out of eleven benchmark categories (compiler, compress, crypto, derby, serial, sunflow and xml), the miss rates for the dual core cases are significantly increased. The worst case is serial whose miss rate is increased to about 600% of the single-core case.

In Fig. 5, relative performance of dual core cases have been added to Fig. 1. Each bar represents the performance normalized to System A, while the number on top of each 2p bar indicates the speed-up from the corresponding single core case. In all benchmark categories (except startup which is single-threaded), very good speed-ups have been achieved (from 1.5 to 2 times faster). The relationship between the speed-ups and the increased L2 cache miss rates are, however, not straightforward for some categories. For example, the L2 miss rate for xml on System C is increased by 82%, but its dual-core execution achieved a speed-up of 80%. In contrast, the increase of L2 miss rate in derby for System C is only 22% but the speed-up is also small (51%). Crypto, mpegaudio and scimark.small are instances which

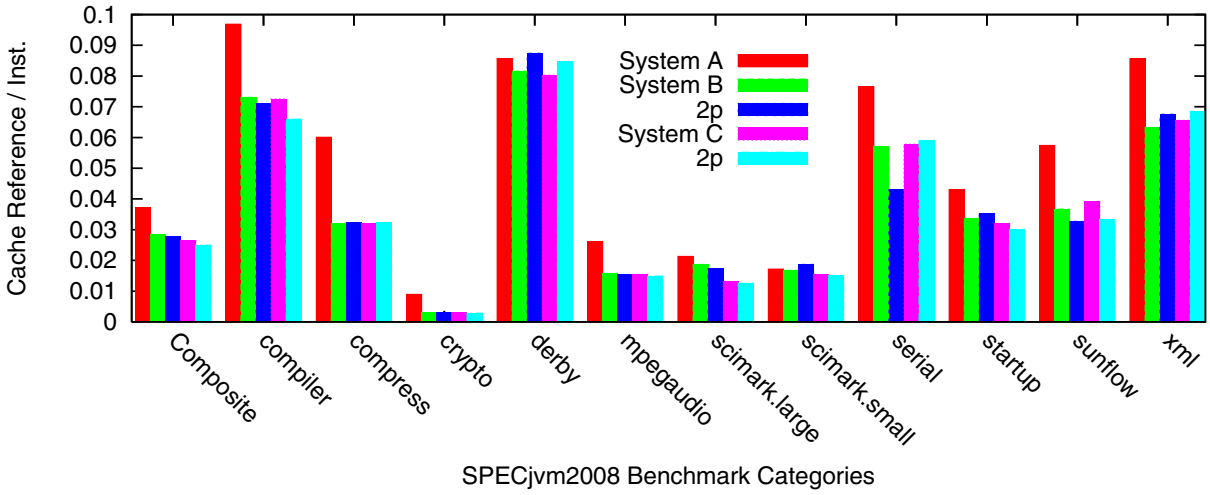

Figure 3. L2 Cache Reference per Instruction

have low L2 miss rates, low (or no) L2 miss increase with dual-core and high speed-up.

In startup, we see small speed-ups (5 to 7%) for both Systems B and C with two cores enabled. While the benchmark itself is single-threaded, startup spends large fraction of time outside JRE (more than 20% in System C). It is possible that this outside JRE part is accelerated by dual core. Again, further investigation is required to confirm this idea.

4 Related Work

For more than ten years, SPEC JVM98 has been used as the standard benchmark for evaluating the Java runtime environment, especially for the client side. There have been many attempts to analyze the performance of JREs and/or JVM98 itself. Gregg et. al. analyzed JVM98 in the bytecode level and obtained statistical data for each workload such as number of method invocations, execution frequency of each bytecode type [10]. [11] is another example of JVM98 analysis. They analyzed the execution of JVM98 on a UltraSPARC based machine and related the bytecode execution behavior with the actually executed SPARC instructions.

While JVM98 was being used, SPEC has also released various server-side Java benchmark programs including SPECjAppServer2004 (J2EE application server) [12], SPECjbb2005 (Java business applications) [13], SPECjms2007 (Java Message Service, message-oriented middleware) [14].

Compared to JVM98, one of the new categories in SPECjvm2008 is the XML document processing. XSLTMark is another benchmark suite to evaluate the processing of XML documents in Java [17].

For the evaluation of the JREs on embedded platforms, CaffeineMark [16] and GrinderBench [15] are two popular benchmark suites. Studies on local variable access and instruction folding using these benchmarks can be found in [9] and [18], respectively.

5 Conclusions

In this paper, we presented the analysis of the SPECjvm2008 workload with emphases placed on the effect of memory hierarchy. We ran SPECjvm2008 on three systems with netburst-based CPUs and different cache sizes. Roughly speaking, an L2 cache of 1MB per core sufficiently lowers the miss rate of SPECjvm2008, and allows the system to achieve a speed-up higher than the clock speed ratio. In seven out of eleven categories, dual-core executions increased the miss rates up to 600% of the single-core cases. Nevertheless, they achieved the speed-up of 50% to 100% compared to the single core cases.

In this paper, we were concentrated on the memory access behavior of the SPECjvm2008. To identify the potential performance bottleneck, it is necessary to investigate the internal of JREs. Comparing the results obtained from different instances of JREs should also be useful for this purpose. An open question is the behavior of mpegaudio benchmark program. Its cache miss rates (for both instruction and data), were very low. However, it was not accelerated by the faster clock speed as we expected. Some categories in SPECjvm2008 consist of multiple sub-benchmark programs. Scimark is one of such categories and from our previous experiences, behaviors of FFT and MonteCarlo are quite different each other (for example, in the average num-

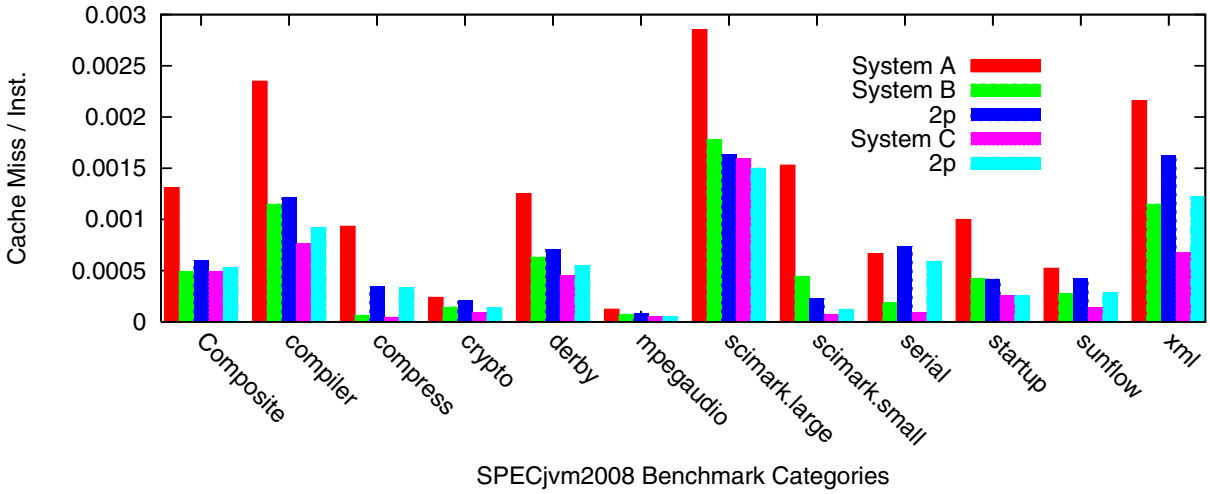

Figure 4. L2 Cache Miss per Instruction

ber of bytecodes executed in an invocation) [9, 18]. Therefore, analyzing sub-benchmark individually should answer to some of the open questions we found in this work.

References

[1] SPECjvm2008, `http://www.spec.org/jvm2008/` .

[2] SPEC JVM98, `http://www.spec.org/jvm98/` .

[3] SPEC CPU95, `http://www.spec.org/cpu95/`

[4] "MP3 library for the Java Platform," `http://www.javazoom.net/javalayer/javalayer.html`

[5] "Java SciMark 2.0," `http://math.nist.gov/scimark2/`

[6] "SPECjvm2008 Known Issues," `http://www.spec.org/jvm2008/docs/KnownIssues.html`.

[7] Glen Hinton, et. al., "The Microarchitecture of the Pentium? 4 Processor," in *Intel Technology Journal*, vol. 5, issue 1, February 2001.

[8] "OProfile - A System Profiler for Linux (News)," `http://oprofile.sourceforge.net/news/` .

[9] Hitoshi Oi, "Local Variable Access Behavior of a Hardware-Translation Based Java Virtual Machine," to appear in *Journal of Systems and Software*, Elsevier.

[10] David Gregg, James Power and John Waldron, "Benchmarking the Java Virtual Architecture," in *Java Microarchitectures*, Kluwer Academic Publishers, April 2002.

[11] Ramesh Radhakrishnan, Juan Rubio and Lizy Kurian, "Characterization of Java applications at bytecode and ultra-SPARC machine code levels," in *Proceedings of the 1999 IEEE International Conference on Computer Design (ICCD)*, pp281–284, 1999.

[12] SPECjAppServer2004, `http://www.spec.org/jAppServer2004/` .

[13] SPECjbb2005, `http://www.spec.org/jbb2005/` .

[14] SPECjms2007, `http://www.spec.org/jms2007/` .

[15] "GrinderBench – Professional grade mobile Java benchmarks by EEMBC," `http://www.grinderbench.com/`

[16] "The Embedded CaffeineMark", Pendragon Software Corporation, 1997.

[17] "DataPower: XSLTMark XSLT Performance Benchmark", `http://www.datapower.com/xmldev/xsltmark.html` .

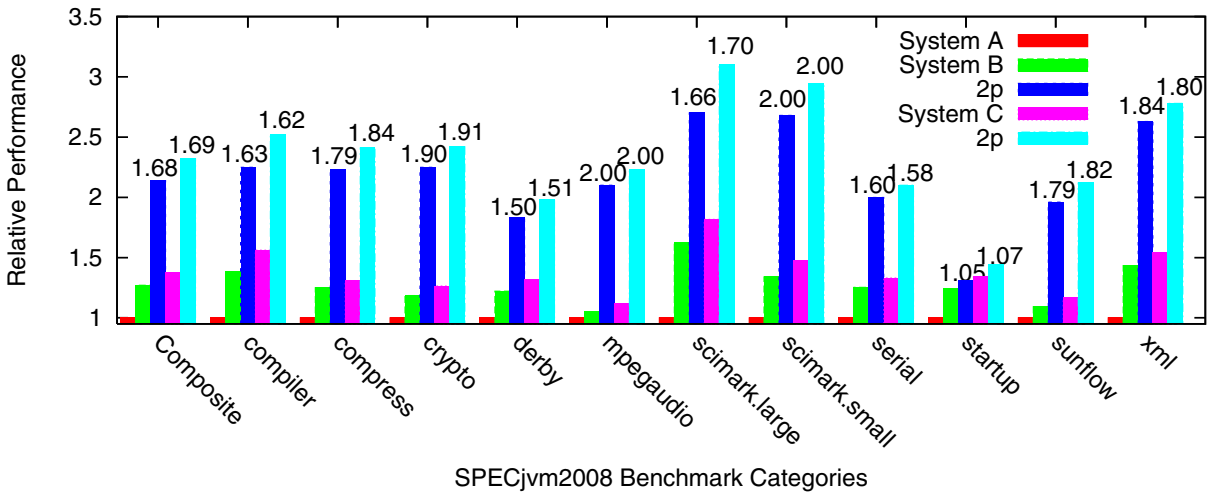

Figure 5. SPECjvm2008 Relative Performance including Dual-Core Cases. Each bar represents the relative performance against System A, while the number on each 2p bar is the speed-up from the corresponding single-core case.

.

[18] Hitoshi Oi, "Instruction Folding in a Hardware Translation-Based Java Virtual Machine," in *The Journal of Instruction-Level Parallelism*, vol10, June 2008.