A Comparative Study of JVM Implementations with SPECjvm2008

Hitoshi Oi
Department of Computer Science, The University of Aizu
Aizu-Wakamatsu, JAPAN
Email: `oi@oslab.biz`

***Abstract*—SPECjvm2008 is a new benchmark program suite for measuring client-side Java runtime environments. 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 will compare two proprietary Java Virtual Machines (JVMs), HotSpot of Sun Microsystems and JRockit of Oracle, using SPECjvm2008 on three platforms that have CPUs with the same microarchitecture but different clock speed and cache hierarchies. The wide variations of the SPECjvm2008 benchmark categories, together with the differences in hardware configurations of the platforms, reveal the strong and weak points of each JVM implementation.

In the composite SPECjvm2008 performance metrics, JRockit performs 19 to 27% better than HotSpot. This is the results of JRockit's outperforming HotSpot in nine out of eleven categories. However, JRockit is quite weak in JVM initialization as it is revealed from the executions of startup.helloworld; the relative performance of JRockit can be as low as 19% of HotSpot. Another remarkable result is, JRockit runs scimark.monte_carlo much faster (up to 285% of HotSpot) which affects the performance metrics of three categories. The relatively higher performances of JRockit on non-startup benchmarks likely to be the differences in number of x86 instructions executed in JVMs, with exceptions in compiler.* benchmarks. In startup.* benchmarks, the performance differences should also be due to the numbers of x86 instructions executed, but their effects widely vary from benchmark to benchmark.

Keywords Java Virtual Machine, Workload Analysis, Performance Evaluation.

I. Introduction

SPECjvm2008 is a benchmark suite released from Standard Performance Evaluation Corporation (SPEC) [1] in 2008. It evaluates various aspects of a client-side 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 present a comparative study of two Java Virtual Machine (JVM) implementations, HotSpot from Sun Microsystems [3] and JRockit from Oracle [4], using SPECjvm2008. We use three platforms that have CPUs with the same microarchitecture but different clock speeds and cache hierarchies. The wide variations of the SPECjvm2008 workloads, together with the differences in hardware configurations of the platforms, reveal the strong and weak points of each JVM implementation.

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 III, we present the experimental environment and the results of our performance analysis. In Sections IV and V, related work and conclusions of this paper are presented, respectively.

II. SPECjvm2008

In this section, we first describe the workload of SPECjvm2008 and then present its performance metrics and run rules. For complete details of SPECjvm2008, please refer to the documents available at the SPEC web site [1].

A. *SPECjvm2008 Workload*

SPECjvm2008 consists of 38 benchmark programs that are classified into the eleven categories listed in Table I. *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 [5], 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: AES and DES protocols (crypto.aes), RSA (crypto.rsa) and MD5withRSA, SHA1withRSA, SHA1withDSA and SHA256withRSA (crypto.signverify). *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 [6]. It evaluates the floating-point operations of the JRE. SciMark is a computational benchmark suite in Java developed by NIST [7]. 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 SPECjvm2008 and it runs one iteration of the benchmark. The time from starting up the JVM to the end of the benchmark execution 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 & Sub-Benchmarks
compiler	Compilation of .java files. compiler.compiler, compiler.sunflow
compress	Compression by LZW method.
crypto	Encryption and decryption. crypto.aes, crypto.rsa, crypto.signverify
derby	Database focused on BigDecimal.
mpegaudio	Mp3 decoding.
scimark.large scimark.small	Floating point benchmark with 32MB and 512KB datasets. scimark.fft.large, scimark.fft.small scimark.lu.large, scimark.lu.small scimark.monte_carlo, scimark.sor.large scimark.sor.small, scimark.sparse.large scimark.sparse.small
serial	Primitive and object (de)serializations.
startup	JVM launch time for each benchmark. startup.compiler.compiler, startup.compiler.sunflow startup.compress, startup.crypto.aes startup.crypto.rsa, startup.crypto.signverify startup.helloworld, startup.mpegaudio startup.scimark.fft, startup.scimark.lu startup.scimark.monte_carlo, startup.scimark.sor startup.scimark.sparse, startup.serial startup.sunflow, startup.xml.transform startup.xml.validation
sunflow	Graphics visualization (rendering).
xml	XML transform and validation. xml.transform, xml.validation

TABLE I
SPECJVM2008 WORKLOAD DESCRIPTIONS

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.

B. 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 `-Xmx384m` option for HotSpot 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 the number of times the benchmark programs are executed on the JRE under test for a minute. The overall ops/m (composite score) is obtained by the the following way. First, for the category with two or more sub benchmarks, the geometric means of its ops/m scores is 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 overall 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.

III. PERFORMANCE ANALYSIS

In this section, we present the analysis of SPECjvm2008 executed by HotSpot and JRockit on three platforms. First, we present the hardware and software environments used in the measurement. Next, we analyze the relative performance in the SPECjvm2008 performance metrics for the composite score and workload categories and then breakdown them into sub-benchmarks. In Section III-D, we try to relate the relative performance to the number of executed native instructions.

A. Experimental Environment

We use three systems, A, B and C in Table II as the platforms for measurements. All three systems have CPUs based on the netburst microarchitecture [9]. 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. All systems run Linux (CentOS) with kernel version 2.6.18. JVM versions are 1.6.0_12 (HotSpot) and 1.6.0_05 (JRockit).

B. SPECjvm2008 Metrics

Figure 1 shows the relative performance in SPECjvm2008 metrics for two JVMs (JRockit/HotSpot) and their breakdown into eleven workload categories. In the composite scores, JRockit is 19 to 27% faster than HotSpot. JRockit is faster than HotSpot in nine out of eleven categories, except xml on System-A. In fact, xml is the only category for which the faster JVM is different among systems. The relative performance

[1]Without -Xmx option, benchmarks such as scimark.large failed by the OutOfMemory error [8]. We started with the heap size of 128MB and increased it by 128MB until JVM2008 ran without errors.

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 Versions	1.6.0_12 (HotSpot), 1.6.0_05 (JRockit)		

TABLE II
BENCHMARKING ENVIRONMENTS

ranges from 59% (startup on System-A) to 191% (serial on System-A).

Figure 2 shows the relative performance of Systems B and C against A. Two horizontal lines drawn at 1.167 and 1.25 represent the ratios of clock speed of Systems B/A and C/A, respectively. In a simplified model, the performance is proportional to "Clock Speed /Clock per Instruction (CPI)". Therefore, if CPI is constant, the relative performance of Systems B and C should be around these horizontal lines. The benchmarks that perform better than these ratios can be considered to utilize the hardware features (mostly larger L1 and L2 cache sizes) of Systems-B and C. In general, in the composite and most workload categories, Systems-B and C achieve higher relative performance than the clock ratios. Exceptions are mpegaudio on HotSpot and sunflow on both JVMs. In seven out of eleven categories (compiler, crypto, mpegaudio, scimark.large, scimark.small, startup and xml), JRockit achieves better system-relative performance than HotSpot. The relative performance range on System-B is from 105% (mpegaudio on HotSpot) to 170% (scimark.large on JRockit).

C. Sub-Benchmark Breakdown

Figure 3 shows the relative performance between JVMs for non-startup sub-benchmarks. First and most obvious, scimark.monte_carlo runs much faster on JRockit than HotSpot, ranging from 277% (System-C) to 285% (System-A). It should be noted that scimark.monte_carlo is included in three workload categories (scimark.large, scimark.small and startup) and without it JRockit performs almost the same for scimark.* and further worse than HotSpot for startup sub-benchmarks. Both compiler.compiler and compiler.sunflow run javac for the difference java source files. However, while the former runs faster on HotSpot, the latter runs faster on JRockit. We mentioned earlier that xml is the only category for which the faster JVM is different among systems. By the sub-benchmark breakdown, it is found that xml.transform is the source of this characteristic.

Figure 4 shows the sub-benchmark breakdown of the relative performance of JVMs for the startup category. Each sub-benchmark of startup spawns an instance of JVM and then executes a single iteration of other workload within SPECjvm2008. An exception is startup.helloworld, which does not run any SPECjvm2008 workload; it only prints a "Hello World!" message. Therefore, it reveals the performance of JVM initialization itself. For this aspect, HotSpot is clearly better than JRockit; the performance of JRockit is only 19 to 26% of HotSpot. The application execution part of startup.scimark.monte_carlo makes it the only sub-benchmark for which JRockit runs faster than JVM in the startup category. For other startup sub-benchmarks, the relative performance ranges from 36 (startup.scimark.lu on System-A) to 90% (startup.serial on System-B).

Figures 5 and 6 show the relative performance of Systems-B and C against A for sub-benchmarks. Most remarkably, scimark.fft.small runs faster on Systems-B and C up to 368 and 403%, respectively. It is considered that the working set of scimark.fft.small fits into the L1 cache of 16KB of Systems-B and C, but not 8KB of System-A. All sub-benchmarks in scimark.large and scimark.sparse.small also show much higher performance gains on the systems with larger cache sizes. On the contrary, scimark.monte_carlo, scimark.lu.small and scimark.sor.small do not benefit from Systems-B and C. xml.transform utilizes the hardware advantages of Systems-B and C (especially for JRockit). xml.validation also run on Systems B and C but benefits are smaller and both HotSpot and JRockit perform similarly. The performance gain by Systems-B and C for scimark.fft.small and scimark.sparse.small also affect the workload execution part of startup benchmarks.

D. Instruction Count and Performance

One of key factors in the execution efficiency of JVM is the number of native instructions to perform the operations of Java applications. For each sub-benchmark, we count the number of retired x86 instructions for the SPECjvm2008 operation (an iteration of the benchmark program). Let $opsm_h$ and $opsm_j$ be the ops/m metrics of HotSpot and JRockit, respectively. Similarly, let $inst_h$ and $inst_j$ be the number of retired x86 instructions for HotSpot and JRockit. If $opsm_j/opsm_h$ is close to $inst_h/inst_j$, we may attribute the performance difference to the number of the executed native instructions (which indicates the efficiency of the Java bytecode interpretation and/or compilation). If $opsm_j/opsm_h$ divided by $inst_h/inst_j$ is larger than 1 (smaller than 1), it indicates that JRockit (HotSpot) performs more ops/m with fewer x86 instructions.

Figure 7 shows this ratio for each non-startup sub-benchmark. Most of ratios for these sub-benchmark categories in Figure 7 are around 1. This is reasonable because in scimark and other loop-oriented benchmarks, dynamic compilers try to generate compact codes and the number of native instructions in the compiled loop-bodies should have large impacts on the performance. There are several exceptions. For both compiler.compiler and compiler.sunflow, JRockit executes 63 (compiler.sunflow on System-C) to 156% (compiler.compiler on System-A) more x86 instructions than HotSpot. However, compiler.compiler runs faster on HotSpot but compiler.sunflow runs faster on JRockit. Both HotSpot and JRockit perform similarly for scimark.sor.large (Figure 3). However, JRockit runs 33 to 36% fewer x86 instructions for the same number of ops/m for this benchmark. Therefore, the performance and instruction count ratios for scimark.sor.large are significantly larger than 1 (1.46 to 1.54). The same thing can be said for scimark.sor.small.

In Figure 8, the relationships between relative performance and instruction count for the startup benchmarks are presented. The executions of these benchmarks consist of two parts: the JVM initialization and the execution of each base benchmark (e.g. , compiler.compiler for startup.compiler.compiler) for a single iteration. Furthermore, the latter part has two factors: the length of single iteration and the execution speed of the JVM under the base benchmark. Due to these various factors, the relative speed of JVMs cannot be directly attributed to the instruction counts. For example, the number of x86 instructions executed for startup.helloworld on JRockit is up to 577% of HotSpot (on System-C). However, the relative speed of HotSpot for startup.helloworld is only 386% of JRockit and further investigation using other methodologies are required to understand the performance bottlenecks of these JVMs.

IV. Related Work

We previously reported the category-level characteristics of the SPECjvm2008 including the SPECjvm2008 performance metrics, cache reference and miss behavior and the effect of multi-threading [10]. Shiv et. al. performed the detailed measurements of SPECjvm2008 on a Core 2 Duo-based platform [11]. Since they ran SPECjvm2008 on a single platform with a single JVM, it is not possible to judge whether these detailed execution parameters (such as the instruction count for a single iteration of a workload) are (relatively) good or not.

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 the number of method invocations, execution frequency of each bytecode type [12]. [13] is another example of JVM98 analysis. They analyzed the execution of JVM98 on a Ultra-SPARC 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) [14], SPECjbb2005 (Java business applications) [15], SPECjms2007 (Java Message Service, message-oriented middleware) [16].

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 [19].

For the evaluation of the JREs on embedded platforms, CaffeineMark [18] and GrinderBench [17] are two popular benchmark suites.

V. Conclusions

In this paper, we have compared two Java Virtual Machine implementations, HotSpot and JRockit, with SPECjvm2008. JRockit outperformed HotSpot in nine out of eleven benchmark categories, which result in 19 to 27% higher composite scores of JRockit. scimark.monte_carlo is one of the sources of JRockit's relatively higher performance. It achieves up to 285% of HotSpot's speed and is included in three benchmark categories. Another finding is that the performance of the Java compiler is significantly dependent on the compiled source code: while compiler.compiler runs up to 66% faster on HotSpot, compiler.sunflow runs up to 18% faster on JRockit. Relative performance of two JVMs for non-startup benchmarks were generally close the relative of instruction counts, with exceptions in compiler.*.

More detailed measurements and analysis, such as cache reference and miss behavior, are undergoing and their results should explain the findings in this paper more clearly. We also plan to conduct the experiments on other platforms, such as Core i5/i7 or Atom-based machines to investigate if the characteristics of JVMs reported in this paper are preserved on these platforms.

Acknowledgments

The author would like to Sun Microsystems and Oracle Corporation (especially Tuva Palm) for giving us access to their JVMs and permission to use them for benchmarking.

References

[1] SPECjvm2008, `http://www.spec.org/jvm2008/` .
[2] SPEC JVM98, `http://www.spec.org/jvm98/` .
[3] "Java SE HotSpot at a Glance," http://java.sun.com/javase/technologies/hotspot/
[4] "Oracle JRockit, " http://www.oracle.com/technology/products/jrockit/
[5] SPEC CPU95, `http://www.spec.org/cpu95/`
[6] "MP3 library for the Java Platform," `http://www.javazoom.net/javalayer/javalayer.html`
[7] "Java SciMark 2.0," `http://math.nist.gov/scimark2/`
[8] "SPECjvm2008 Known Issues," `http://www.spec.org/jvm2008/docs/KnownIssues.html`.
[9] Glen Hinton, et. al., "The Microarchitecture of the Pentium? 4 Processor," in *Intel Technology Journal*, vol. 5, issue 1, February 2001.
[10] Hitoshi Oi, "A Preliminary Workload Analysis of SPECjvm2008," in *Proceedings of 2009 International Conference on Computer Engineering and Technology (ICCET 2009)*, Vol. 2, pp13–19, Singapore, January 2009.
[11] Kumar Shiv, et. al., "SPECjvm2008 Performance Characterization," in *Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, pp17–35, 2009.
[12] David Gregg, James Power and John Waldron, "Benchmarking the Java Virtual Architecture," in *Java Microarchitectures*, Kluwer Academic Publishers, April 2002.
[13] 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.
[14] SPECjAppServer2004, `http://www.spec.org/jAppServer2004/` .
[15] SPECjbb2005, `http://www.spec.org/jbb2005/` .
[16] SPECjms2007, `http://www.spec.org/jms2007/` .
[17] "GrinderBench – Professional grade mobile Java benchmarks by EEMBC," `http://www.grinderbench.com/`
[18] "The Embedded CaffeineMark", Pendragon Software Corporation, 1997.
[19] "DataPower: XSLTMark XSLT Performance Benchmark", `http://www.datapower.com/xmldev/xsltmark.html` .

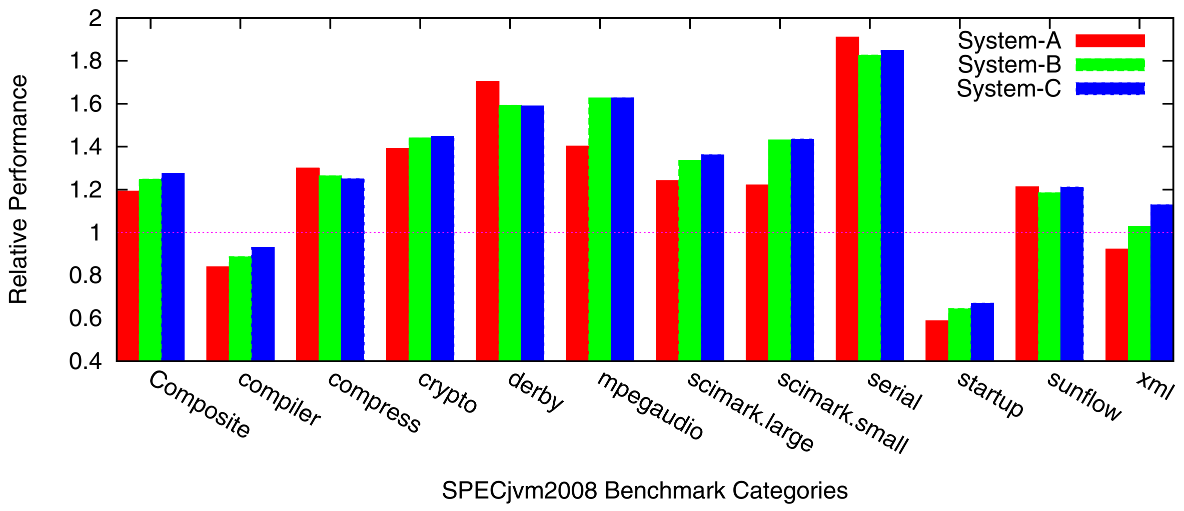

Fig. 1. Relative Performance of JRockit against HotSpot

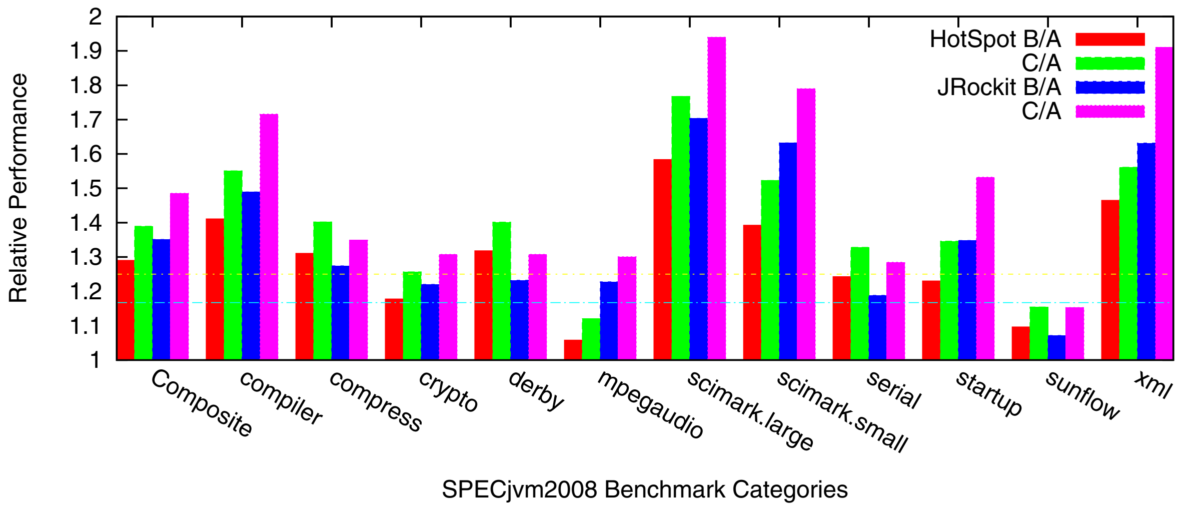

Fig. 2. Relative Performance against System-A

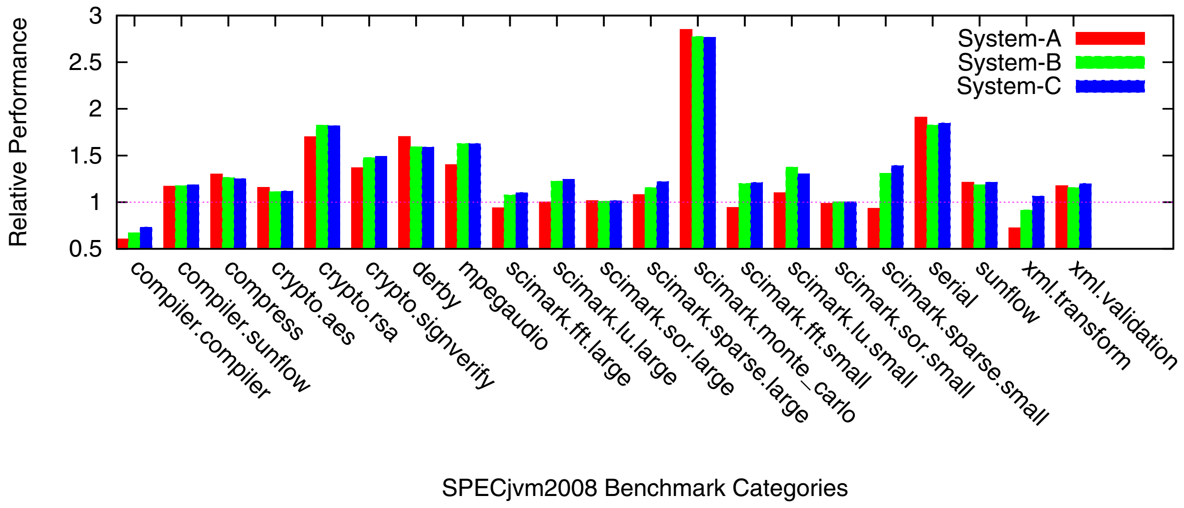

Fig. 3. Sub-Benchmark Relative Performance of JRockit against HotSpot

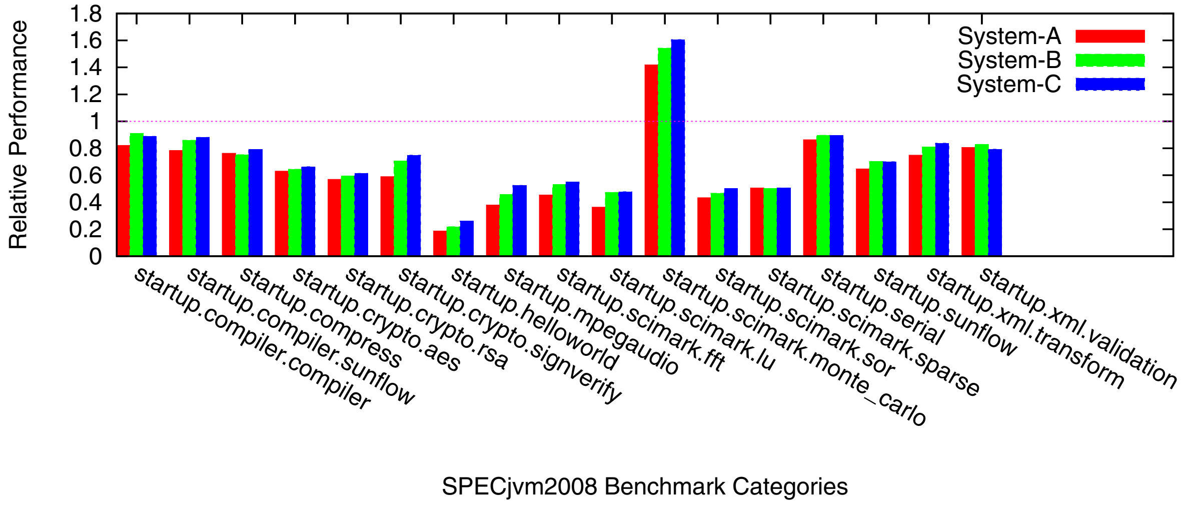

Fig. 4. Startup Sub-Benchmark Relative Performance of JRockit against HotSpot

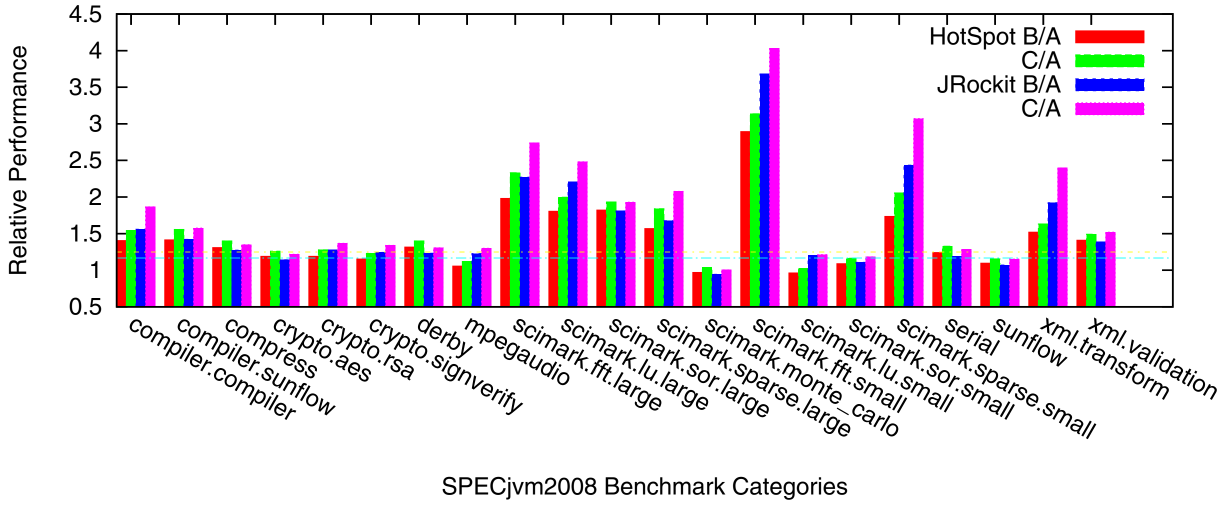

Fig. 5. Sub-Benchmarks: Relative Performance against System-A

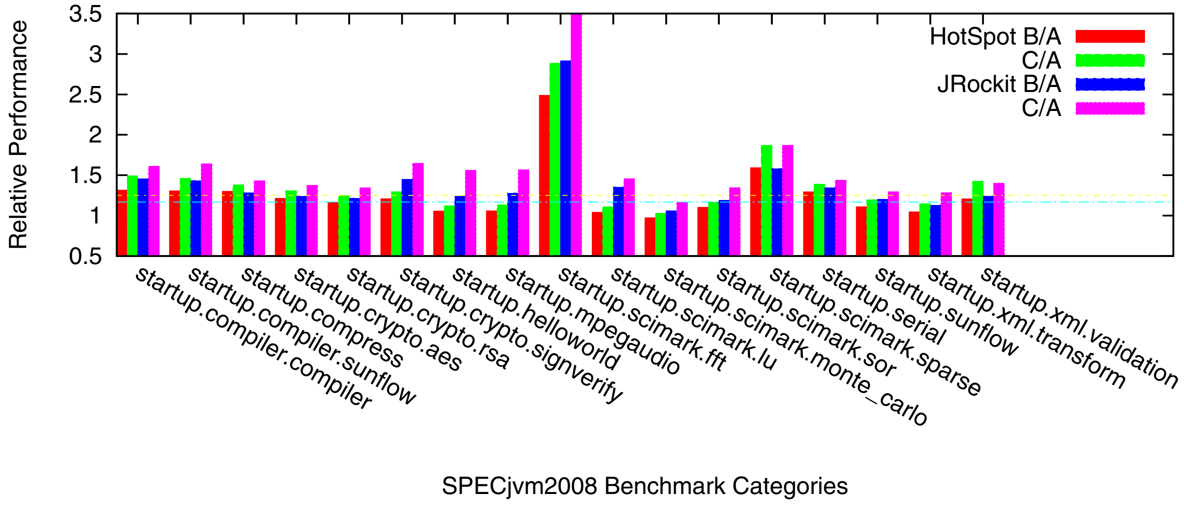

Fig. 6. Startup: Relative Performance against System-A

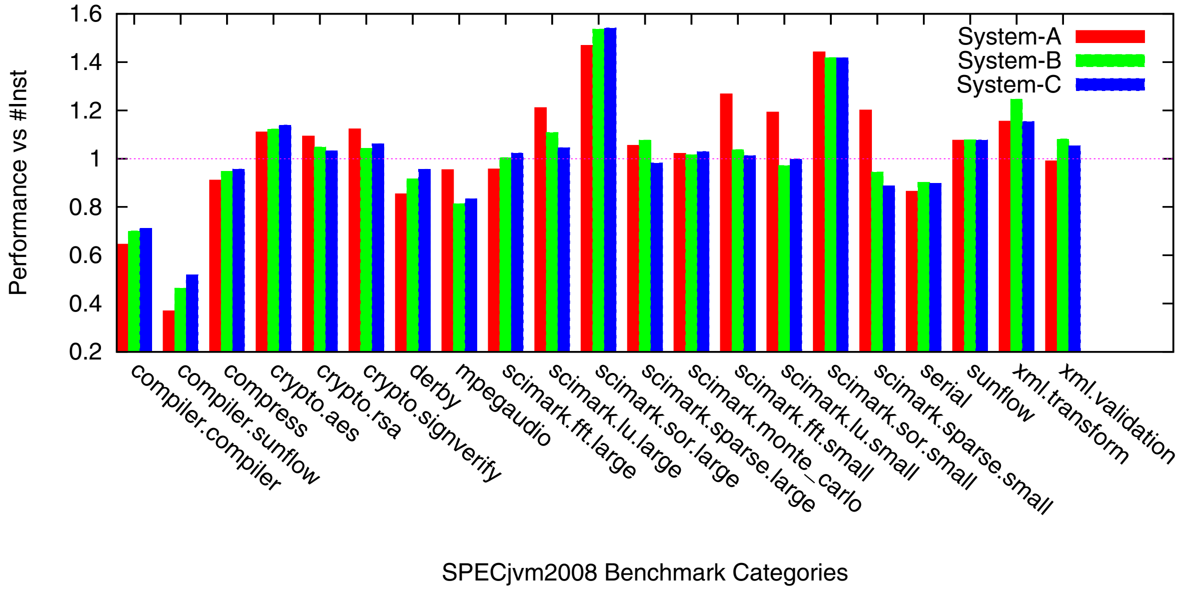

Fig. 7. Sub-Benchmarks: Relative Performance and Instruction Count. Each bar represents the relative performance divided by the inverse of relative instruction count. See Section III-D for more detail.

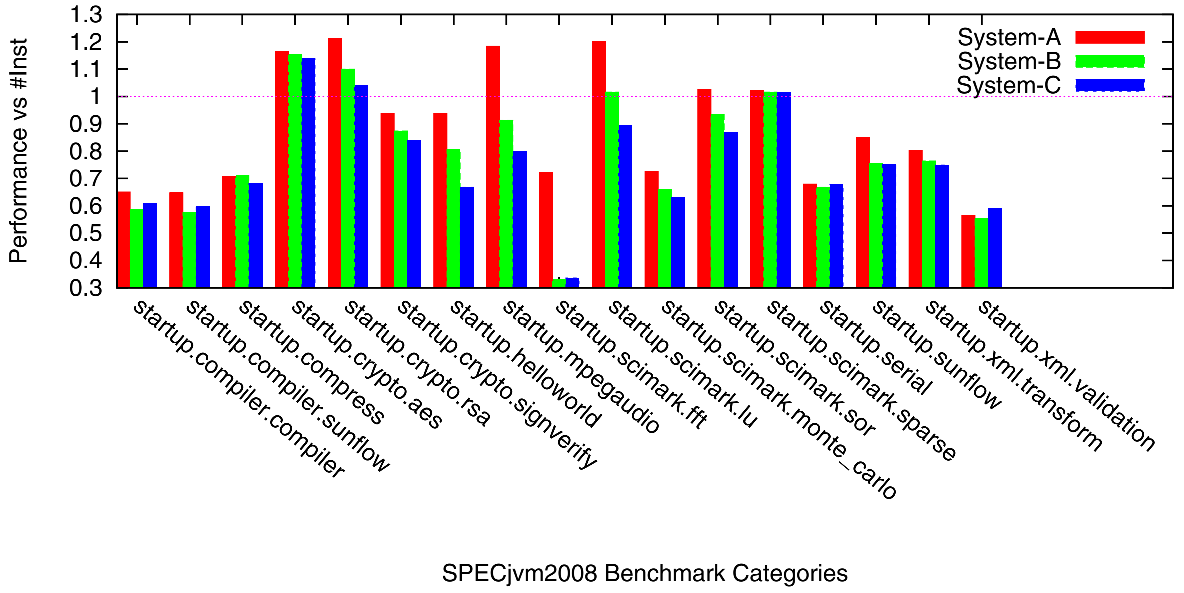

Fig. 8. Startup: Relative Performance and Instruction Count