# Cache Performance in Java Virtual Machines: A Study of Constituent Phases

Anand S. Rajan, Juan Rubio and Lizy K. John

Laboratory for Computer Architecture
Department of Electrical & Computer Engineering
The University of Texas at Austin
(arajan, jrubio,ljohn@ece.utexas.edu)

## Abstract

This paper studies the cache performance of Java programs in the interpreted and just-in-time (JIT) modes by analyzing memory reference traces of the SPECjvm98 applications with the Latte JVM. Specifically, we study the cache performance of the different components of the JVM (class loader, the execution engine and the garbage collector). Poor data cache performance in JITs is caused by code installation, and the data write miss rate can be as high as 70% in the execution engine. In addition, this code installation also causes deterioration in performance of the instruction cache during execution of translated code. We also believe that there is considerable interference between data accesses of the garbage collector and that of the compiler-translator execution engine of the JIT mode. We observe that the miss percentages in the garbage collection phase are of the order of 60% for the JIT mode; we believe that interference between data accesses of the garbage collector and the JIT execution engine lead to further deterioration of the data cache performance wherever the contribution of the garbage collector is significant. We observe that an increase in cache sizes does not substantially improve performance in the case of data cache writes, which is identified to be the principal performance bottleneck in JITs.

## 1. Introduction

Java[1] is a widely used programming language due to the machine independent nature of bytecodes. In addition to portability, security and ease of development of applications have made it very popular with the software community. Since the specifications offer a lot of flexibility in the implementation of the JVM, a number of techniques have been used to execute bytecodes. The most commonly used modes of execution are interpretation, which interprets the bytecodes and just-in-time compilation, which dynamically translates bytecodes to native code on the fly. A recent development has been the mixed mode execution engine [19], which uses profile based feedback to interpret/compile bytecodes. Other possible modes include hardware execution of bytecodes [20] and ahead-of-time compilation of bytecodes [21]. This paper will focus on the interpreted and JIT modes of execution, which are more widely used by current JVMs.

The speed of executing programs in modern superscalar architectures is not determined solely by the number of instructions executed. A significant amount of the execution time can be attributed to inefficient use of the microarchitecture mechanisms like caches [2]. Even though there have been major strides in the development of fast SRAMS [4] that are used in cache memories, the prevalence of deep superscalar pipelines and aggressive techniques to exploit ILP [12] make it imperative that cache misses are minimized.

Prior studies [10] have established the poor data cache performance of just in time compilers compared to interpreters. Instruction cache performance in JITs has also been shown to be relatively poor. This work attempts to characterize the cache performance in the interpreted and JIT modes by separating the virtual machine into functionally disparate components. This enables us to isolate the component responsible for the major chunk of these misses.

The three distinct phases we examine are the class loader, the execution engine (interpreter or JIT compiler) and the garbage collector. A Java application can use two types of class loaders: a “bootstrap” class loader and user-defined class loaders. The

bootstrap loader loads classes of the Java API and user defined classes in some default way, whereas the user-defined class loaders load classes in custom ways over the course of program execution. The garbage collector [17] determines whether objects on the heap are referenced by the Java application, and makes available the heap space occupied by objects that are not referenced. In addition to freeing unreferenced objects, the garbage collector also combats heap fragmentation.

We repeat our experiment with enhanced cache sizes to examine whether the high miss rates exhibited in data writes are primarily a result of capacity misses and could be done away using larger caches. The remainder of the paper is organized as follows. Section 2 presents the prior research done in this area. Section 3 discusses the experimental methodology, including the benchmarks and tools used for the experiments. Section 4 discusses in detail the results for the various experiments and analyzes them and section 5 offers concluding remarks.

## 2. Related Work

The early works in this area are due to Romer et al. [8], Newhall et al. [7], Hsieh [2,13] et al. etc who studied the impact of interpreted Java programs on microarchitectural resources such as the cache and the branch predictor using a variety of benchmarks. Radhakrishnan et al. [9] analyzed SPECjvm98 benchmarks at the bytecode level while running in an interpreted environment and did not find any evidence of poor locality in interpreted Java code. None of these however examined the behavior of Java with the JIT mode of execution.

There have been quite a few studies looking at the execution characteristics and architectural issues involved with running Java in the JIT mode. Radhakrishnan et al. [10] investigated the CPU and cache architectural support that would benefit such JVM implementations. They concluded that the instruction and data cache performance of Java applications are better than compared to that of C/C++ applications, except in the case of data cache performance of the JIT mode. Radhakrishnan [11] provided a quantitative characterization of the execution behavior of the SPECjvm98 programs, in interpreted

mode and using JIT compilers and obtained similar results to the ones obtained in [10]. None of these studies, however, looked at the different components of the JVM though Kim et al. [14] and Dieckman et al. [15] did examine the performance of the garbage collection phase alone in detail.

## 3. Methodology

### 3.1 Benchmarks

The SPECjvm98 suite of benchmarks [3] is used to obtain the cache performance characteristics of the JVM. This suite contains a number of applications that are either real applications or are derived from real applications that are commercially available. The SPECjvm98 suite allows users to evaluate performance of the hardware and software aspects of the JVM client platform. On the software side, it measures the efficiency of the JVM, the compiler/interpreter, and operating system implementations. On the hardware side, it includes CPU, cache, memory, and other platform specific features. Table 1 provides a summary of these benchmarks used for our experiments.

| Benchmark | Description |
|---|---|
| compress | A popular LZW compression program. |
| jess | A Java version of NASA's popular CLIPS rule-based expert systems |
| db | Data management benchmarking software written by IBM. |
| mpegaudio | The core algorithm for software that decodes an MPEG-3 audio stream. |
| mtrt | A dual-threaded program that ray traces an image file. |
| jack | A real parser-generator from Sun Microsystems. |

Table 1 Description of the SPEC JVM98 benchmarks used

For all benchmarks, SPEC provides three different data sets referred to as s1, s10 and s100. Although the input names may suggest so, SPECjvm98 does not scale linearly. Both the s1 and s100 data sets were used for the experiments though we will present only the results with the s100 data set for clarity. Results for the s1 data set can be obtained from [22].

### 3.2 The Latte Virtual Machine

We used the Latte virtual machine [5] as the target Java Virtual Machine to study the cache performance in each of the distinct phases of a JVM. Latte is the result of a university collaboration project between Seoul National University (Korea) and IBM. It is an open source virtual machine, which was released in Oct 2000 and was developed from the Kaffe open source VM and allows for instrumentation and experimentation. Its performance has been shown to be comparable to Sun's JDK 1.3 (HotSpot) VM.

Latte boasts of a highly optimized JIT compiler targeted towards RISC processors. In addition to classical compiler optimizations like Common Sub-expression Elimination (CSE) and redundancy elimination, it also performs object-oriented optimizations like dynamic class hierarchy analysis. In addition, it performs efficient garbage collection and memory management using a fast mark and sweep algorithm [23].

### 3.3 Tools and Platform

Our study of cache performance of the Latte JVM is performed on the UltraSparc platform running Solaris 2.7 using tracing tools and analyzers. We use the Shade [6] tool suite, which provides user-level program tracing abilities for the UltraSparc machines. Shade is an instruction set simulator and custom trace generator. Application programs are executed and traced under the control of a user-supplied analyzer.

For our performance measurements, we used a cache simulator based on the cachesim5 analyzer provided by the Shade suite of programs. Cachesim models the cache hierarchy for the experiments; it allows the user to specify the number of levels in the cache hierarchy, the size and organization of each of these and the replacement/write

policies associated with them. This analyzer was modified to suit the requirements of our measurements and validated to examine the correctness of these changes. Our cache simulator provides performance results in terms of hit-miss ratios and does not deal with timing issues.

### 3.4 Instrumentation of the JVM

Since the objective is to look at the cache behavior in the different stages of the VM, the source code of Latte was instrumented with *sentinels* that would mark the phases of class-loading, interpretation/compilation and garbage collection. The class-loading phase includes the loading of API classes that is done before the Java application starts to execute method bytecodes. These include the Class class (the base class for all classes in the application), the String class (the class that represents all character strings), the wrapper classes for primitive data types (Integer, Float, Boolean and so on) etc. We do not monitor classes loaded during the course of execution as the structure of the JVM code makes it difficult to do so. The garbage collection phase includes all memory accesses and allocations in the heap in addition to the actual task of "garbage-collection" because this seems to be a more logical classification when it comes to data accesses. We use the default initial heap size of 16MB in all our experiments. The execution phase consists of the process of bytecode interpretation in case of the interpreter and translation of bytecode and installation of native code in the case of JIT compiled mode of execution.

The sentinel code consists of a sequence of native instructions that are easily identifiable by our profiling tools. They are manually inserted in the source code of Latte in the form of high level constructs. The generation of these sequences has been chosen in such a way that these high-level language statements are translated into double word store instructions (STD) in the SPARC assembly code whenever they are encountered. The values stored as part of the store instructions are unique numbers for a particular phase of execution. The beginning of each phase is marked by a sentinel that involves storing 3 128-bit numbers. These numbers were chosen in order to reduce the probability

of the consecutive occurrence of these 3 numbers, thus avoiding false triggers. The occurrence of the sentinel is checked in our analyzer.

### 3.5 Cache Hierarchies

Table 1 lists the cache hierarchies that were chosen for the experiments with the first configuration corresponding to the cache hierarchy on the UltraSparc-1 processor [16]. The first configuration is used for the detailed per-phase analysis of the Latte JVM presented in section 4.

| Configuration | L1 I-cache | L1 D-cache | L2-Unified cache |
|---|---|---|---|
| 1. | 16K, 32 byte blocks, 2-way set associative, write through, LRU. | 16K, 32 byte blocks and 16byte sub-blocks Direct mapped, write through with write-no-allocate | 512K, 64 byte blocks Direct mapped, write back with write on allocate |
| 2. | 64K, 32 byte blocks, 2-way set associative, write through, LRU. | 64K, 32 byte blocks 4 way set associative, write through with write-no-allocate | 1M, 64 byte blocks Direct mapped, write back with write on allocate |
| 3. | 256K, 32 byte blocks, 2-way set associative, write through, LRU | 256K, 32 byte blocks 4-way set associative, write through with write-no-allocate | 2M, 64 byte blocks Direct mapped, write back with write on allocate |

Table 2 Cache Configurations

As mentioned previously, Cachesim5 provides detailed statistics on the references and misses at all levels in the cache hierarchy. It is modified to be able to examine the entire trace of the benchmarks and classify the particular instruction as a load or a store or an ordinary instruction. In addition to the above modification, a flag is set to classify the phase of the JVM execution where the instruction was encountered. This flag is set based on the sentinel values that have been encountered so far. All this classification

information is provided to the cache simulator module. Separate counters are maintained for each of the measurements (references, misses etc) in each of the phases.

### 3.6 Validation

The validation of the modified Cachesim5 is central to our experiments. Each of the benchmarks was run and the resulting instruction trace provided to the original Cachesim5 simulator and the total instruction counts, data accesses and misses were noted. The above was done with no instrumentation whatsoever applied to the JVM. The same benchmarks were now run and the resulting instruction trace provided to the modified Cachesim5 simulator. The statistics obtained in this case were compared to those obtained in the previous case, and there was almost exact agreement in the numbers.

With instrumentation applied to the JVM, we needed to validate the same again. The cache statistics computed for each of the phases into which the JVM had been divided were added up and compared to the numbers obtained in the previous 2 cases. The numbers agreed in all the cases. For the sake of sanity check, the numbers were compared to those obtained in [11] and there was close correspondence between the numbers obtained in both the studies.

## 4. Results and Analysis

This section summarizes the results of this study that characterize the SPECjvm98 benchmarks in terms of the cache performance in the various phases of the execution of the Latte Virtual Machine. Section 4.1 presents results for the instruction cache performance in the interpreted and JIT modes of execution. Sections 4.2.1 and 4.2.2 present the results for the data cache performance in the interpreted and JIT modes of execution for reads and writes respectively.

Section 4.3 presents the effect of increased cache sizes on the data write miss-rates in the JIT by comparing performance using the 3 different cache configurations. The

motivation behind these comparisons is to examine whether the poor data cache performance in the JIT compiled mode was a result of mere capacity misses.

### 4.1 Instruction Cache Performance

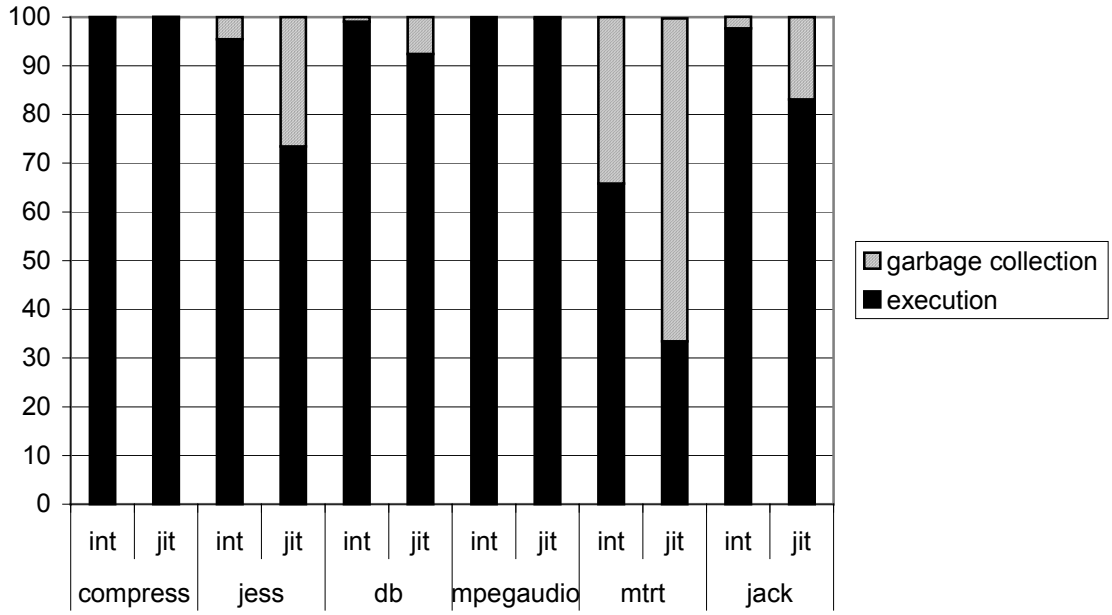


Figure 1 Contribution of garbage collection and execution phases (%) to the total number of instruction accesses. (Cache specifications – 16K, 2-way set associative L1 instruction cache with 32 byte blocks.)

Figure 1 shows the contribution of the garbage collection and execution phases to the total number of instruction accesses. We infer that the overall characteristics of instruction cache performance will be almost completely dominated by the execution phase in the interpreted mode in all benchmarks, except mtrt. But in the JIT mode execution, the garbage collection mode has a significant role to play in all benchmarks, except compress and mpegaudio. This is a direct result of the allocation characteristics of these two benchmarks [14].

For the case of the interpreted execution mode, the actual interpretation component constitutes the majority of the dynamic instruction count. It ranges from about

143 billion instructions for the compress benchmark to about 25 billion for mtrt [23]. The overall instruction miss rate is therefore almost same as for the interpretation phase for the JVM execution. On the other hand, there is almost an 80% to 90% decrease in the dynamic instruction count when moving from the interpreted mode to the JIT [23]. This is because the method bytecodes are translated only once, unlike in the interpreted mode, where the interpreter loop has to be executed every time a method is invoked.

Class loading contributes almost a constant number of instructions to the JVM execution in all the benchmarks and its contribution is not very substantial. In fact, the statistics for class loading in every benchmark are more or less constant. This is attributed to the class-loading phase including only the loading of classes that are required prior to the start of execution of the methods. As far as miss-rates go, class-loading accounts for merely 0.01% of the total misses in either mode of execution and are hence not shown in any of the tables.

For the interpreted mode, instruction miss rate varies from 0.16% to 1.33% and this good instruction locality is because the interpreter is one large switch statement with about 220 case labels. But only about 40 distinct bytecodes are accessed 90% of the time [9] and thus the entire loop can fit into the instruction cache. The instruction cache performance in the JIT compilation is always worse than that in the interpreted mode. The reason for this is the fact that the operation of the JIT is for the most part similar to that of a compiler and compilers do not have very good instruction locality (for example, gcc in the SPEC95 suite [18]). In addition, the code installed by the translator need not be contiguously placed in the cache, thus contributing to poorer performance. Exceptions to this are the benchmarks with high method reuse, where a small subset of the methods is accessed; this considerably diminishes the importance of non-contiguously placed translated code. compress (the miss rate decreases from 1.3% to 0.07% as we go from the interpreter to the JIT) and mpegaudio (the miss rate decreases from 0.6% to 0.2% as we go from the interpreter to the JIT) reflect this behavior. In these 2 benchmarks, there is high method reuse and actual execution of the translated code dominates the compilation process.

| Benchmark | Execution Phase | | Garbage Collection Phase | | Overall |
|---|---|---|---|---|---|
| | % Abs. miss | %Total miss | % Abs. miss | %Total miss | % Abs. miss |
| compress (int) | 1.30 | 99.46 | 0.15 | 0.002 | 1.30 |
| (jit) | 0.07 | 98.85 | 0.16 | 1.03 | 0.07 |
| jess (int) | 1.35 | 96.48 | 1.03 | 3.51 | 1.33 |
| (jit) | 1.48 | 85.68 | 0.68 | 14.28 | 1.26 |
| db (int) | 0.16 | 99.35 | 0.10 | 0.62 | 0.16 |
| (jit) | 0.12 | 97.70 | 0.03 | 2.21 | 0.11 |
| mpeg (int) | 0.60 | 99.99 | 0.45 | 0.003 | 0.60 |
| (jit) | 0.18 | 99.51 | 0.31 | 0.48 | 0.18 |
| mtrt (int) | 0.47 | 68.51 | 0.42 | 31.40 | 0.46 |
| (jit) | 1.21 | 54.45 | 0.51 | 45.52 | 0.75 |
| jack (int) | 0.72 | 97.14 | 0.89 | 2.86 | 0.72 |
| (jit) | 1.31 | 95.98 | 0.27 | 4.00 | 1.14 |

Table 3 Instruction cache performance *(% Abs. miss* indicates absolute miss rate in a particular phase and *% Total miss* indicates the contribution of that phase to the total number of misses. Cache specifications – 16K, 2-way set associative L1 instruction cache with 32 byte blocks.)

In the interpreted mode, garbage collection plays a significant role in the case of the mtrt benchmark whereby it contributes about 31% of the total number of misses. But the miss rate is very comparable to that incurred in the interpretation phase and hence there is no effect on the overall miss rate. In the JIT mode, this phase shows a lot more activity especially with the jess and mtrt benchmarks. From Table 3, it is seen that there is more locality seen amongst the instructions in this phase than the compilation/execution phase and this contributes in bringing down the overall miss rate where its contribution to the total number of misses is substantial. This is evidenced in the case of jess (overall miss rate of 1.26% and execution phase miss rate of 1.48%) and mtrt (overall miss rate of 0.75% and execution phase miss rate of 1.21%).

### 4.2 Data Cache Performance

Table 4 shows the contribution of each of the phases to the total data cache misses. We find that once again the contribution of the class-loading phase is quite negligible. The contribution of the garbage collection phase is significant in the interpreted mode in all benchmarks save compress and mpegaudio. In the JIT mode of execution, the garbage collection phase contributes about 40-70% of the total data misses

in some benchmarks. In compress and mpegaudio, the execution phase once again renders the other phases inconsequential.

| Benchmark | Class Loading Contribution | Execution Phase Contribution | Garbage Collection Phase Contribution | Overall Data Cache Miss % |
|---|---|---|---|---|
| compress(int) | 0.001 | 99.84 | 0.15 | 2.98 |
| (jit) | 0.003 | 98.99 | 0.98 | 3.60 |
| Jess (int) | 0.003 | 81.04 | 18.94 | 6.11 |
| (jit) | 0.004 | 58.77 | 41.20 | 24.07 |
| Db (int) | 0.002 | 94.48 | 5.50 | 4.20 |
| (jit) | 0.004 | 86.33 | 13.65 | 19.52 |
| Mpeg (int) | 0.004 | 99.94 | 0.05 | 1.08 |
| (jit) | 0.005 | 99.63 | 0.34 | 11.06 |
| Mtrt (int) | 0.003 | 31.14 | 68.86 | 4.09 |
| (jit) | 0.005 | 28.16 | 71.81 | 21.47 |
| Jack (int) | 0.004 | 84.26 | 15.72 | 3.09 |
| (jit) | 0.007 | 60.82 | 39.14 | 18.87 |

Table 4 Contribution of each phase to the Data Cache Misses of the JVM. Cache specifications – 16K, direct mapped L1 data cache with 32 byte blocks.)

Read accesses in both modes of execution consist of reading method bytecodes and the data required for the execution of these methods. The difference is that, while in the JIT mode method bytecodes are read only the first time the method is invoked, in the interpreted mode they are read every time the method is invoked. Write accesses in the JIT mode are mostly the result of code installation whereas in the interpreted mode, they comprise of stack accesses implemented as stores. Data accesses of the benchmarks are common to both the execution modes and affect both of them equally. Thus, the fundamental difference in the character of read and write accesses in the two modes is the motivation behind our decision to study data cache read and write activity separately.

### 4.2.1 Read Accesses

Table 5 shows the performance numbers obtained for data cache reads in the interpreted and JIT modes of execution. There is a drastic reduction in the number of read accesses in the JIT mode. This is a result of the method bytecodes being read only the first time the method is invoked, rather than being read every time the method is invoked,

as was the case in the interpreted mode. Another reason is the fact that a large percentage of operations in the interpreted mode involve accessing the stack, which are implemented as loads and stores. On the other hand, these are optimized as register-register operations in the JIT execution mode.

Miss rates for the execution phase increase from an average of 3.5% to as high as 18% when we move from the interpreted mode of execution to the JIT mode. The reason for this is the fact all bytecode read misses that occur are cold misses since they are brought into the data cache the very first time the method is invoked. As a result, the lowest miss rates will be seen in benchmarks where the actual data required by the benchmark program is a large fraction of the total data accesses. This is indeed so in compress, which applies the same compression algorithm on a large amount of data (miss rate of 8.78% in JIT and 2.03% in the interpreter) and mpegaudio, where large MPEG-3 audio streams are decoded using the same algorithm (miss rate of 5.75% in JIT and 0.94% in interpreter).

One of the notable points is the fact that there is a very high miss rate seen in the case of the data reads in the garbage collection phase for both modes of execution. The influence of the garbage collection phase is not felt in the compress and mpegaudio benchmarks where it contributes less than 0.4% of the total misses in either mode. We observe that in most of the other cases, the garbage collection phase tends to increase the overall miss rate. Most of the objects referenced in Java programs are short-lived [14] and in case of benchmarks like jess and jack, they are less than 32K in size. Yet, the miss rates in the garbage collection phase for jess (about 15% in the interpreted mode and 13% in the JIT mode) and jack (about 8% in the interpreted mode and JIT modes) are high. This leads us to believe that there are frequent conflict and capacity misses between the data accessed by the garbage collector (data structures and objects on the heap) and the data for the execution phase (method bytecodes for the interpreter and predominantly translated code for the JIT mode).

The problem is more serious in the JIT where the size of the translated code is very large; it has the potential to cause the net data cache performance to deteriorate

when the garbage collection phase's contribution is substantial. This behavior is profoundly expressed in mtrt (the overall miss rate is 4.23% compared to the miss rate of 4.09% for the execution phase) and jess (the overall miss rate is 5.08% compared to the miss rate of 3.72% for the execution phase).

| Benchmark | Execution Phase | | Garbage Collection Phase | | Overall |
|---|---|---|---|---|---|
| | % Abs. miss | %Total miss | % Abs. miss | %Total miss | % Abs. miss |
| Compress (int) | 2.03 | 99.94 | 16.14 | 0.05 | 2.03 |
| (jit) | 8.78 | 99.62 | 12.99 | 0.36 | 8.79 |
| Jess (int) | 4.92 | 95.26 | 14.74 | 4.73 | 5.08 |
| (jit) | 10.92 | 82.30 | 12.74 | 17.68 | 11.19 |
| Db (int) | 3.67 | 98.06 | 19.39 | 0.02 | 3.72 |
| (jit) | 18.83 | 95.94 | 18.76 | 4.04 | 18.82 |
| Mpeg (int) | 0.94 | 99.98 | 3.88 | 0.01 | 0.94 |
| (jit) | 5.75 | 99.75 | 6.66 | 0.21 | 5.75 |
| Mtrt (int) | 4.09 | 33.55 | 4.31 | 66.44 | 4.23 |
| (jit) | 13.83 | 33.18 | 17.04 | 66.79 | 15.84 |
| Jack (int) | 2.68 | 97.89 | 8.24 | 2.09 | 2.72 |
| (jit) | 9.48 | 90.03 | 7.86 | 9.91 | 9.29 |

Table 5 Data cache (reads) performance *(% Abs. miss* indicates absolute miss rate in a particular phase and *% Total miss* indicates the contribution of that phase to the total number of misses. Cache specifications – 16K, direct mapped L1 data cache with 32 byte blocks.)

### 4.2.2 Write Accesses

Table 6 shows the results for the write accesses seen with the interpreted and JIT modes of execution. In the interpreted mode, overall miss rates range from 1.51% for mpegaudio to 9.53% for the case of jess. In the JIT mode, we observe the effects of installation of native code, which leads to the phenomenon of *double-caching*. When the JIT compiler translates method bytecodes into native code for the very first time, it has to incur compulsory misses when the code is installed in the data cache. In addition, compulsory misses will be incurred when the native code is brought into the instruction cache for execution. These two operations together constitute *double-caching*. Write misses are also seen when the method bytecodes are read into the data cache on invocation of the method for the very first time but this is not as profound as in the case of code installation because each bytecode translates into 25 native instructions on an average [10].

In terms of actual results, we observe that the miss rates in the execution phase of the JIT mode range from 12.5% for db to about 69.8% for jess. A direct result of the installation of native code is that more data read misses are seen in both the execution and garbage collection phases due to conflict with this installed code (this was examined in 4.1). Also, we noted the poorer performance of the instruction cache in the JIT mode when compared to the interpreted mode in 4.2. *Double-caching* thus results in the overall poor performance of JIT compilers, which renders them less effective under memory constraints even though the speedup over the interpreted mode is appreciable.

An examination of the garbage-collection phase miss rates in both the modes reveals them to be extremely high (ranging from 50% to 74%) in all the benchmarks save mtrt, where the miss rate is about 4% for the interpreted mode and 39% for the JIT mode. This has an adverse effect on the overall miss rate of the benchmark; this is seen in the case of jess (overall miss rate of 9% compared to 5% in the execution phase for the interpreted mode) and jack (overall miss rate of 48% compared to 39% in the execution phase in the JIT mode). Most of the data writes in this phase are due to allocation of objects. As the heap size grows during program execution, more and more of the allocations tend to be cold misses resulting in high miss rates. On the other hand, if the heap size were to be small, the garbage collector would be invoked more frequently, leading to more interference between the installed native code and the heap objects in the JIT mode [14]. As a result, the initial heap size chosen for a Java program would need to be highly program dependent.

Another reason for the high miss rates in the garbage collection phase is the object allocation behavior of our benchmarks. For example, compress executes in loops whereby the lifetimes of objects are less than the duration of a loop and jess and jack have very short object lifetimes [14], thus leading to frequent allocation. This frequent allocation leads to more interference between the phases and poorer cache performance.

| Benchmark | Execution Phase | | Garbage Collection Phase | | Overall |
|---|---|---|---|---|---|
| | % Abs. miss | %Total miss | % Abs. miss | %Total miss | % Abs. miss |
| compress (int) | 6.18 | 99.73 | 74.56 | 0.26 | 6.19 |
| (jit) | 19.61 | 97.77 | 67.37 | 2.20 | 19.92 |
| jess (int) | 5.68 | 55.81 | 65.32 | 44.17 | 9.53 |
| (jit) | 69.86 | 45.72 | 63.88 | 54.26 | 66.48 |
| db (int) | 5.18 | 86.36 | 60.59 | 13.63 | 5.92 |
| (jit) | 12.54 | 40.34 | 59.59 | 59.61 | 59.61 |
| Mpeg (int) | 1.51 | 99.85 | 49.89 | 0.14 | 1.51 |
| (jit) | 31.91 | 99.54 | 50.04 | 0.44 | 31.95 |
| mtrt (int) | 2.44 | 22.51 | 4.27 | 77.47 | 3.66 |
| (jit) | 39.31 | 21.85 | 38.69 | 78.12 | 38.82 |
| Jack (int) | 2.34 | 54.90 | 57.70 | 44.96 | 4.12 |
| (jit) | 39.21 | 43.56 | 58.99 | 56.42 | 48.35 |

Table 6 Data cache (writes) performance *(% Abs. miss* indicates absolute miss rate in a particular phase and *% Total miss* indicates the contribution of that phase to the total number of misses. Cache specifications – 16K, direct mapped L1 data cache with 32 byte blocks.)

### 4.4 Cache Performance with Increased Cache Sizes

We experimented with larger cache sizes to examine if the unacceptably poor data cache performance in the JIT compiled mode of the Latte VM was a result of mere capacity misses. We examine only the miss rates for the execution and garbage collection phases in this section, since the contribution of the class loader phase is not substantial.

Referring to Figure 2, we find that the reduction in miss rate in the execution phase is not very substantial. The miss rates still range from 14.8% in the case of compress (19.6% in configuration 1 and 17.7% in configuration 2) to 55.3% in the case of jess (69.8% in the configuration 1 and 57.6% in configuration 2). This implies that however much we may increase the size of the data cache, we are not able to recover from the performance penalty imposed by the compulsory misses in the execution phase (installation of translated code).

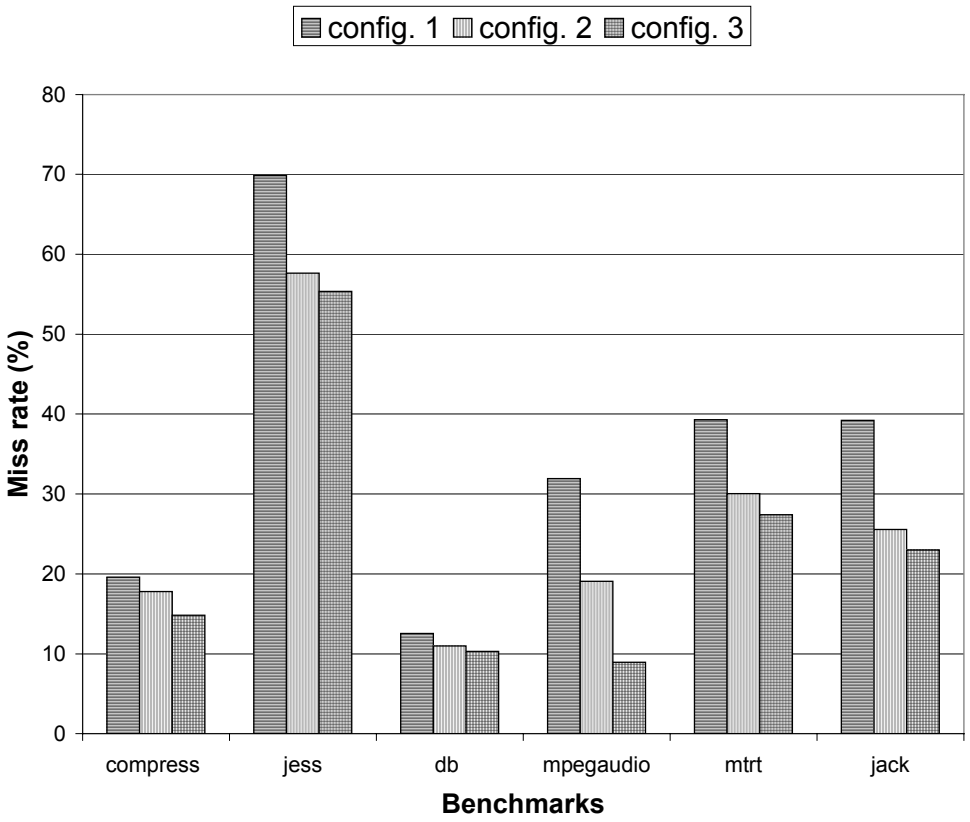


Figure 2 Data Cache Writes- Miss Rates for Execution Phase of the JIT Mode. (In config. 1, the L1 data cache is 16K, direct mapped and has 32-byte block size. In config. 2, the L1 data cache is 64K, 4-way set associative and has 32-byte block size. In config. 3, the L1 data cache is 256K, 4-way set associative and has 32-byte block size.)

In the garbage collection phase (figure 3), the performance is far worse. jess (where the garbage collector executes a number of data writes) has a miss rate of 57.9% in configuration 3) down from a miss rate of 63.9% in configuration 1), which is still significantly high. This trend is seen in all the benchmarks, though the miss rates are slightly lower than the values seen for db, ranging from 30% (mtrt) to 54% (db). Again, we see that larger caches are not the solution to offset the inherent poor performance seen in JVMs due to garbage collectors and their interaction with the execution phase.

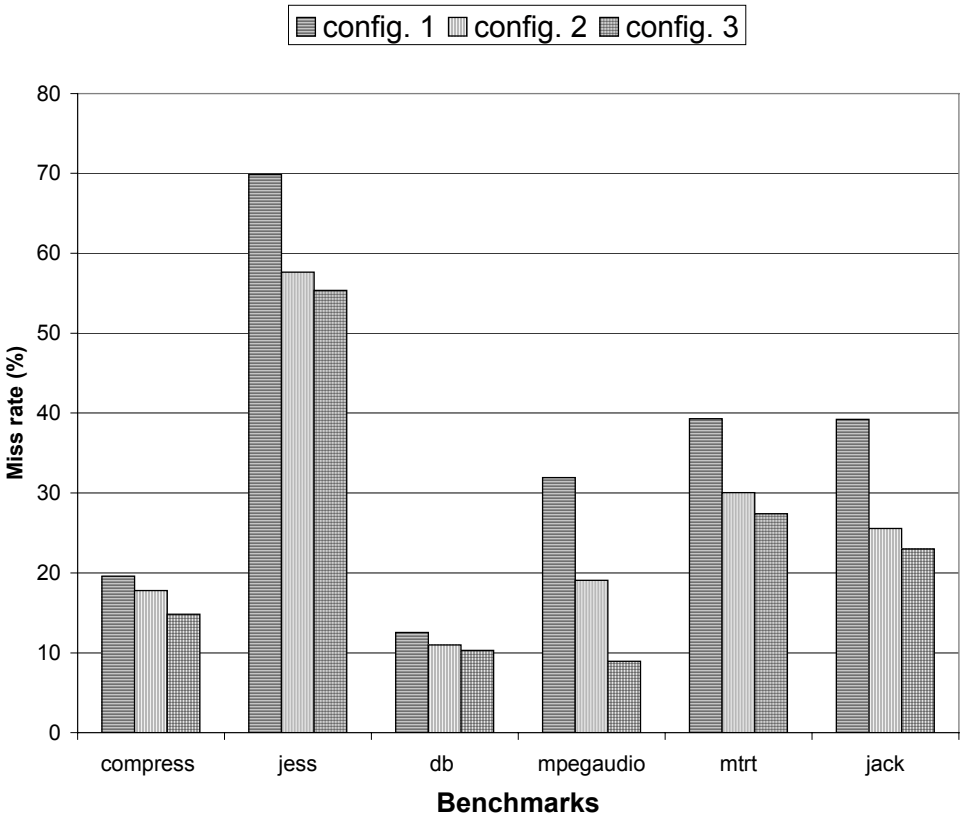


Figure 3 Data Cache Writes - Miss Rates in Garbage Collection Phase of the JIT mode. (In config. 1, the L1 data cache is 16K, direct mapped and has 32-byte block size. In config. 2, the L1 data cache is 64K, 4-way set associative and has 32-byte block size. In config. 3, the L1 data cache is 256K, 4-way set associative and has 32-byte block size.)

## 5.Conclusion

At the heart of Java technology lies the Java virtual machine. The design of efficient JVM implementations on diverse hardware platforms is critical to the success of Java technology. An efficient JVM involves addressing issues in compilation technology, software design and hardware-software interaction.

This study has focused on understanding the cache performance of the JVM as a whole and the contribution of its main functional components, namely the class loader, the execution engine and the garbage collector, to this overall behavior. This was done

for the most common implementations of the JVM – the JIT and the interpreter. The conclusions of the paper are as follows:

- The JIT mode of execution of bytecodes results in a large reduction in the number of native instructions executed but the price to be paid is in the form of poor cache performance. This is reflected in instruction caches as well as data read and data write operations.
- The instruction cache performance in the JIT mode is worse than that in the interpreted mode. This is to some extent a result of the poor instruction locality inherent in compiler applications and the nature of Java methods, which result in non-contiguous pieces of native code. But the major contribution seems to be from compulsory misses incurred when the translated code is brought into the instruction cache.
- The garbage collector demonstrates good instruction locality but its performance deteriorates in the JIT mode of execution due to conflict misses with the translated code. This behavior is carried to data reads too, where there appears to be considerable interference between the garbage collector's data structures and the objects on the one hand and the translated code on the other hand.
- Data writes exhibit extremely poor performance in JIT modes of execution and the miss rates are on an average 38% for the s100 data sets. Poor data cache performance is the result of compulsory misses resulting from the installation of translated code.
- The garbage collector also performs very poorly in the JIT mode and results in further deterioration of data cache performance when its contribution to data accesses is substantial. Previous works have shown that the optimal heap size for Java programs is highly dependent on their allocation characteristics and a non-optimal heap size leads to poor data cache performance. Additionally, we feel that interference between heap objects and the data accesses of the execution phase also contributes to poorer data cache performance.

- With increased cache sizes, a significant reduction is not seen with data cache writes that have been evidenced to be the performance bottlenecks. The average miss rate over all the benchmarks is still about 30%, an improvement of 8% over the original cache hierarchy configuration and this provides substantial evidence to state that the data cache write misses are not merely the result of capacity misses.

## 6. References


1. T.Lindholm and F.Yellin, The Java Virtual Machine Specification, MA: Addison Wesley, 1997.
2. C.A.Hsieh, M.Conte, T.L.Johnson, J.C. Gyllenhaal and W.W. Hwu, *A Study of Cache and Branch Performance Issues with Java on Current Hardware Platforms*, Proceedings of COMPCON, Feb 1997, pp 211-216.
3. SPEC JVM98 Benchmarks, http://www.spec.org/osg/jvm98
4. A.Chandrakasan, W.Bowhill, F.Fox, *Design of High-Performance Microprocessor Circuits*, IEEE Press, 2000.
5. B.Yang, S.Moon et al., *LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation*, International Conference on Parallel Architectures and Compilation Techniques, October 1999.
6. R.F.Cmelik and D. Keppel, Shade: *A Fast Instruction Set Simulator for Execution Profiling*, Sun Microsystems Inc., Technical Report SMLI TR-93-12, 1993.
7. T. Newhall and B.Miller, *Performance Measurement of Interpreted Programs*, Proceedings of Euro-Par '98, 1998.
8. T.H.Romer, D.Lee, G.M.Voelker, A.Wolman, W.A.Wong, J.Baer, B.N. Bershad, H.M.Levy et al., *The Structure and Performance of Interpreters*, Proceedings of ASPLOS VII, 1996, pp. 150-159.

9. R.Radhakrishnan, J.Rubio and L.K.John, *Characterization of Java Applications at Bytecode and Ultra-SPARC Machine Code Levels*, Proceedings of IEEE International Conference on Computer Design, pages 281-284, 1999.
10. R.Radhakrishnan, N.Vijaykrishnan, A.Sivasubramaniam, L.K.John et al., *Architectural Issues in Java Runtime Systems*, Proceedings of the International Symposium on High Performance Computer Architecture, pages 387-398, 2000.
11. R.Radhakrishnan, J.Rubio, L.K.John and N.Vijaykrishnan, *Execution Characteristics of JIT Compilers***,** Department of Electrical and Computer Engineering, University of Texas at Austin, Technical Report TR-990717-01.
12. T.Yeh and Y.Patt, *A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution*, IEEE Micro, 1992.
13. C.A.Hsieh, J.C.Gyllenhaal and W.W.Hwu et al., *Java Bytecode to native code translation: The Caffeine Prototype and Preliminary Results*, Proceedings of the 29th Annual Workshop on Microprogramming, December 1996.
14. J.Kim and Y.Hsu, *Memory system behavior of Java programs: Methodology and analysis*, ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pages 264-274, June 2000.
15. S.Dieckman and U.Holzle, *A study of the allocation behavior of the SPECjvm98 Java benchmarks*, ECOOP98, pages 92-115, July 1998.
16. T.Horel and G.Lauterbach, *UltraSPARC-III: Designing Third-Generation 64-Bit Performance*, IEEE Micro, Nov 1999.
17. B.Venners, *Inside the Java 2 Virtual Machine*, McGraw Hill, 2000.
18. B.Calder, D.Grunwald and B.Zorn, *Quantifying Behavioral Differences Between C and C++ Programs*, Journal of Programming Languages, Vol. 2, No. 4, pp 313-351, 1995.
19. The Java HotSpot Performance Engine Architecture, http://java.sun.com/products/hotspot/whitepaper.html
20. M.O'Connor and M.Tremblay, picoJava-I: The Java virtual machine in hardware, IEEE Micro, pp 45-53, Mar 1997.

21. T.Proebsting, G.Townsend, P.Bridges, J.H.Hartman, T.Newsham and S.A.Waterson, *Toba: Java for applications a way ahead of time* (WAT) compiler, Proceedings of the Third Conference on Object--Oriented Technologies and Systems, 1997.
22. A. Rajan and L.K.John, *A Study of Cache Performance in Java Virtual Machines*, The University of Texas at Austin, 2002.
*23*. P. Wilson, *Uniprocessor Garbage Collection Techniques*, International Workshop on Memory Management*,* Sep 1992.