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

Anand S. Rajan
ARM Inc.
arajan@arm.com

Shiwen Hu and Juan Rubio
The University of Texas at Austin
{hushiwen, jrubio}@ece.utexas.edu

Abstract

This paper studies the level 1 cache performance of Java programs by analyzing memory reference traces of the SPECjvm98 applications executed by the Latte Java Virtual Machine. We study in detail Java programs' cache performance of different access types in three JVM phases, under two execution modes, using three cache configurations and two application data sets. We observe that the poor data cache performance in the JIT execution mode is caused by code installation, when the data write miss rate in the execution engine can be as high as 70%. In addition, code installation also deteriorates instruction cache performance during execution of translated code. High cache miss rate in garbage collection is mainly caused by large working set and pointer chasing of the garbage collector. A larger data cache works better on eliminating data cache read misses than write misses, and is more efficient on improving cache performance in the execution engine than in the garbage collection. As application data set increases in the JIT execution mode, instruction cache and data cache write miss rates of the execution engine decrease, while data cache read miss rate of the execution engine increases. On the other hand, impact of varying data set on cache performance is not as pronounced in the interpreted mode as in the JIT mode.

1. Introduction

Java [14] is a widely used programming language due to the machine independent nature of Java bytecodes. In addition to portability, security and ease of development of applications have made it very popular with the software community. Since the JVM specification offers a lot of flexibility in the implementation of a JVM, a number of techniques have been used to execute bytecodes. The most commonly used execution modes 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 [1], which uses profile based feedback to interpret/compile bytecodes. Other possible modes include hardware execution of bytecodes [8] and ahead-of-time compilation of bytecodes [16].

The execution time of a program 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, such as caches [11]. Even though there have been major strides in the development of fast SRAMS [6] that are used in cache memories, the prevalence of deep superscalar pipelines and aggressive techniques to exploit ILP [26] make it imperative that cache misses are minimized.

In this paper, we characterize the performance of level 1 instruction and data caches of Java programs in the interpreted and JIT modes by separating the virtual machine into functionally distinct phases. 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 [24] 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.

Our experiments indicate that the execution engine determines the performance of the L1 instruction cache on most Java applications. While the garbage collector is responsible for most of the data cache misses. The reasons for this behavior are the large working set used during that phase and the pointer chasing present in the garbage collection algorithm. Data cache misses are further categorized as read and write misses, which enables us to study different JVM features and application characteristics that contribute to those two types of data cache misses. We observe that the data cache performance in the JIT mode is worse than in the interpreted mode. The deterioration in the cache performance of data read operations is due to a decrease

in the number of times bytecodes are read. Write operations also suffer from a poor data cache performance as a result of the code installation, which incurs compulsory write misses.

We also analyze the impact of different cache configurations and application data sets on the L1 cache performance. Increasing the size of the data caches improves the performance of the data cache in the execution phase more than in the garbage collection phase. We also observe that changing the data set from s1 to s100 affects the JIT mode and the interpreted mode differently. In the JIT mode, instruction cache and data cache write miss rates of the execution engine decrease, while data cache read miss rate of the execution engine increases. The cache performance of the execution engine for the interpreted mode changes little as the data set increases (except for the compress benchmark). The performance of the data cache for the garbage collection phase deteriorates when the data set increases from s1 to s100, without noticeably affecting the instruction cache performance.

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 to conduct the experiments. Section 4 discusses in detail the results for the various experiments and analyzes them and section 5 concludes with a highlight of the most significant results in this paper.

2. Related work

Early work on the area of Java performance [11,12,15,21] 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. [17] analyzed the 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 using the JIT execution mode.

There have been quite a few studies looking at the execution characteristics and architectural issues involved with running Java in JIT mode. Radhakrishnan et al. [18,19] investigates the CPU and cache architectural support that would benefit both JVM implementations. They concluded that the instruction and data cache performance of Java applications is better than that of C/C++ applications, except in the case of data cache performance of the JIT mode. However, none of these studies looked at the different components of the JVM though Kim et al. [13] and Dieckman et al. [9] examine the performance of the garbage collection phase alone in detail. Shuf et al. [22] analyses the accesses performed by a JVM in terms of stack and heap accesses and correlates them to the resulting data cache and TLB behavior.

3. Methodology

This section introduces the SPECjvm98 benchmarks, the Latte Virtual Machine, the cache simulator and configurations, and the validation of the simulation results.

3.1. Benchmarks

The SPECjvm98 benchmark suite [4] is used to obtain the cache performance characteristics of the JVM. The 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 the benchmarks used for our experiments.

Table 1. Description of the SPEC JVM98 benchmarks

Benchmark	Description
Compress	A popular LZW compression program.
Jess	A Java version of CLIPS rule-based expert systems
Db	Data management benchmarking software
Mpegaudio	Core algorithm 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.

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 are used for the experiments. More results can be obtained from [20].

3.2. Instrumentation of the Latte virtual machine

A state-of-the-art JVM, Latte [3], is used to study the cache performance of distinct JVM phases. It is an open source virtual machine released in October 1999, and was derived from the Kaffe open source VM [2] and allows for instrumentation and experimentation. Its performance has been shown to be comparable to Sun's JDK 1.3 (HotSpot) VM [25].

Latte includes a highly optimized JIT compiler targeted towards SPARC processors. In addition to classical compiler optimizations such as common sub-expression elimination and loop invariant code motion, it also performs object-oriented optimizations, e.g. dynamic class hierarchy analysis. In addition, it performs efficient garbage collection and memory management using a fast mark and sweep algorithm [24].

The source code of Latte was instrumented with *sentinels* that mark the phases of class-loading, execution and garbage collection. The class-loading phase includes the loading of Java API and user classes. 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 bytecode interpretation in the interpreter execution mode, or translation of bytecode and installation and execution of native code in the JIT execution mode.

3.3. Cache simulator and cache hierarchies

Our cache performance study of the Latte JVM is performed on a Sun UltraSparc platform running Solaris 2.7. The JVM is functionally executed and simulated by Sun's Shade V6 analyzing tool [7]. Shade simulates the entire user program but does not include operating system calls.

For our performance measurements, we used a cache simulator based on Cachesim5, which is provided by the Shade toolset. The cache simulator allows users to specify the number of levels in the cache hierarchy, the size and organization of each cache hierarchy and the replacement/write policies associated with them. The modified cache simulator also recognizes different JVM phases and is validated to ensure the correctness. Our cache simulator provides performance results in terms of cache hit-miss ratios and does not deal with timing issues.

Table 2 lists the cache hierarchies that are chosen for our experiments with the first configuration corresponding to the cache hierarchy of an UltraSparc-1 processor [10]. The first configuration is used for the detailed per-phase analysis of the Latte JVM presented in Section 4, and the cache performance under all three configurations is presented in Section 4.3.

3.4. Validation

The validation of the modified cache simulator is central to our experiments. Each of the benchmarks is executed and the resulting instruction trace is provided to the original Cachesim5 simulator and the total instruction counts, data accesses and misses are counted. The above validation is done with the original JVM that is not instrumented. The benchmarks are then executed and the resulting instruction traces are gathered for the modified Cachesim5 simulator. The statistics obtained in this procedure is compared to those obtained by the original Cachesim5 to ensure the correctness of the modified cache simulator.

Table 2. Cache configurations in the experiments

Config.	L1 I-cache	L1 D-cache	L2-Unified cache
1	16KB, 32 byte blocks, 2-way set associative	16KB, 32 byte blocks Direct mapped, write through with write-no-allocate	512KB, 64 byte blocks Direct mapped, write back with write on allocate
2	64KB, 32 byte blocks, 2-way set associative	64KB, 32 byte blocks 4-way set associative, write through with write-no-allocate	1MB, 64 byte blocks Direct mapped, write back with write on allocate
3	256KB, 32 byte blocks, 2-way set associative	256KB, 32 byte blocks 4-way set associative, write through with write-no-allocate	2MB, 64 byte blocks Direct mapped, write back with write on allocate

With instrumentation applied to the JVM, we needed to validate the results again. The cache statistics gathered for each JVM phase is added up and compared to the numbers obtained in the previous two cases. The numbers agree in all the cases. For the sake of sanity check, the numbers are further compared to those obtained in [19] and there is close correspondence between the numbers obtained in both studies.

4. Results and analysis

This section summarizes the results that characterize the L1 cache performance of SPECjvm98 benchmarks in the various phases of the Latte Virtual Machine in both interpreted and JIT execution modes. Section 4.1 presents the results for the instruction cache performance, while Section 4.2 analyzes the results for the data cache performance in terms of reads and writes. Section 4.3 studies the impact of increased cache sizes on data cache performance by comparing the data cache performance under the three different cache configurations presented in Section 3.3. Section 4.4 studies the impact of varying data set on L1 cache performance.

Class loading contributes almost a constant number of instructions in all benchmarks and its contribution is almost negligible. In fact, the statistics for class loading in every benchmark are more or less constant, and class-

loading accounts for merely 0.01% of the total misses in either execution mode and is therefore not shown in the subsequent tables.

4.1. Instruction cache performance

Figure 1 shows the contribution of the garbage collection and execution phases to the total number of instruction cache accesses. We note that the instruction cache accesses are almost completely dominated by the execution phase in the interpreted mode in all the benchmarks except mtrt. Nevertheless, in the JIT execution mode, the garbage collection has a more significant role on instruction cache accesses than in the interpreted mode. Under both execution modes, garbage collector shows almost no activity in compress and mpegaudio, which reflects the runtime characteristics, such as infrequent heap object allocations and long lifetimes of heap objects [13], of these two benchmarks.

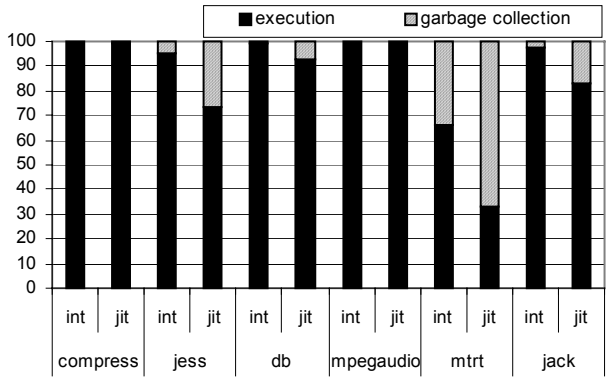


Figure 1. Decomposition of level 1 instruction cache accesses (cache specifications: Configuration 1)

In the interpreted execution mode, the actual interpretation component constitutes the majority of the total instruction count. It ranges from about 143 billion instructions for compress to about 25 billion for mtrt [20]. The overall instruction miss rate is therefore almost the same as the instruction miss rate of the interpretation phase. 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 mode [20]. The decreasing instruction count in the JIT mode is due to the fact that a Java method is translated only once and executed directly afterwards. While in the interpreted mode the interpreter has to be invoked every time to execute the method, which results in a much higher translated native code/bytecode rate than in the JIT mode.

Under the interpreted execution mode, overall instruction miss rate varies from 0.16% (db) to 1.33% (jess) (Table 3) and this good instruction locality is due to that the major part of the interpreter resides in the instruction cache. The interpreter is one large switch statement with about 220 case labels interpreting distinct bytecodes. Since about 40 bytecodes are accessed 90% of the time [17], the case statements for those frequently used bytecodes can fit into the instruction cache.

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: Configuration 1)

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
Mpegaudio (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

The JIT mode has a better overall instruction cache performance than the interpreted mode in four out of six benchmarks. It is mainly due to high method reuse that a small subset of methods is accessed frequently. High method reuse considerably diminishes the impact of non-contiguously placed translated code on instruction cache performance. In compress (the miss rate decreasing from 1.3% in the interpreted mode to 0.07% in the JIT mode) and mpegaudio (the miss rate decreasing from 0.6% in the interpreted mode to 0.2% in the JIT mode), the execution of the translated native code dominates the whole execution time.

In the interpreted mode, garbage collection plays a significant role only in the case of mtrt, where it contributes about 31% of the total number of instruction cache misses, which is mainly due to the fact that mtrt has the most method invocations among all benchmarks. In the JIT mode, the garbage collection phase has relatively more activities than the garbage collection phase in the interpreted mode, especially for jess and mtrt. Moreover, in the JIT mode, the garbage collection phase has better instruction cache locality than the execution phase. And when the instruction cache accesses of the garbage collection phase are substantial, the overall miss rate is brought down due to the low miss rate in the garbage collection phase, which 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 phase 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 except compress and mpegaudio. Under the JIT execution mode, the garbage collection phase contributes about 40-70% of the total data misses in several benchmarks. In compress and mpegaudio, the execution phase once again makes the other phases inconsequential.

Table 4. Contribution of each phase to the data cache misses (cache specifications: Configuration 1)

Benchmark	Class Loading	Execution	Garbage Collection	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
Mpegaudio (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

Read accesses in both modes of execution consist of reading method bytecodes and data required by workloads. The difference is that in the JIT mode bytecodes of a method are read for compilation only when the method is first encountered, while in the interpreted mode bytecodes are read every time when the method is invoked. Write accesses in the JIT mode are mostly the result of compiled native code installation, whereas in the interpreted mode, write accesses include mainly Java heap and stack accesses. Data accesses required by applications are similar in both execution modes and affect both modes equally. However, the fundamental differences between the causes of read and write accesses under the two execution modes motivate us to study data cache read and write activities separately.

4.2.1. Read accesses. Table 5 shows the performance results obtained for data cache reads in both the interpreted and JIT modes. There is a drastic reduction in the number of read accesses under the JIT mode. Under the JIT mode, bytecodes of a method are read only when the method is compiled, while under the interpreted mode, bytecodes are read every time when the method is interpreted. Another reason for the reduction is the fact that a large percentage of operations under the interpreted mode involve accessing the stack, which are implemented as loads and stores, while under the JIT mode these stack accessing operations are optimized as register-register operations.

Table 5. Data cache (read) 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: Configuration 1)

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
Mpegaudio (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

Miss rates for the execution phase increase from an average of 3.5% under the interpreted mode to as high as 19% (db) under the JIT mode. Most bytecode read misses under the JIT mode are compulsory misses since they are brought into the data cache the very first time when the method is invoked. As a result, the lowest miss rates will be seen in programs where the actual data required by programs is a large fraction of the total data accesses. Two of such programs are compress (8.78% in the JIT mode and 2.03% in the interpreted mode) and mpegaudio (5.75% in the JIT mode and 0.94% in the interpreted mode). Both compress and mpegaudio operate on large amounts of data.

The garbage collection phases under both execution modes yield high data cache read miss rates. The influence of the garbage collection phase is insignificant in compress and mpegaudio where the garbage collection phase contributes less than 0.4% of the total misses in either mode. For the other benchmarks, the garbage collection phase tends to increase the overall miss rate. The high miss rate of the garbage collection phase is more serious under the JIT mode since under the JIT mode the garbage collection phase's contribution to the overall miss rate is substantial. One of the examples is mtrt, whose overall miss rate is 16% compared to the execution phase miss rate 14%.

Three factors contribute to the high miss rate in the garbage phase. The first factor is the frequent conflict and capacity misses between the data accessed by the garbage collector (references to arrays and objects on the heap) and the data for the execution phase (method bytecodes and application required data). Large working set of the garbage collector also inflicts conflict and capacity misses in the data cache. Finally, frequent pointer chasings by the garbage collector cause compulsory misses in the data cache.

4.2.2. Write accesses. Table 6 shows the results for the write accesses under the interpreted and JIT execution modes. Under the interpreted mode, overall miss rates range from 1.51% (mpegaudio) to 9.53% (jess). Under the JIT mode, we observe much higher write miss rates than those under the interpreted mode, which is primarily caused by the phenomenon of *double-caching*. When the JIT compiler translates bytecodes into native code for the very first time, it incurs compulsory misses when the code is installed in the data cache. In addition, compulsory misses occur when the native code is brought into the instruction cache for execution. These two operations together constitute *double-caching*. Compulsory 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 is translated into 25 native instructions on an average [17].

We observe that the data cache write miss rates in the execution phase of the JIT mode range from 12.5% (db) to about 69.8% (jess). A direct result of the native code installation is that L2 cache is polluted by the installed code. Also, we noted the poorer performance of the instruction cache under the JIT mode when compared to the interpreted mode in Section 4.1. *Double-caching* thus results in the overall poor cache 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 modes reveal them to be extremely high (ranging from 50% to 74%) in all the benchmarks except mtrt, whose miss rate is 4% under the interpreted mode and 39% under the JIT mode. High write miss rate in the garbage collection phase has an adverse effect on the overall miss rates of the benchmarks such as jess (overall miss rate of 9% compared to 5% in the execution phase under the interpreted mode) and jack (overall miss rate of 48% compared to 39% in the execution phase under the JIT mode).

Most of the data writes in the garbage collection phase are due to allocation of objects. As the heap size grows during program execution, more and more of the allocations tend to be compulsory misses, which result in high miss rate. On the other hand, if the heap size is 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. As a result, the initial heap size chosen for a Java program should be highly program dependent.

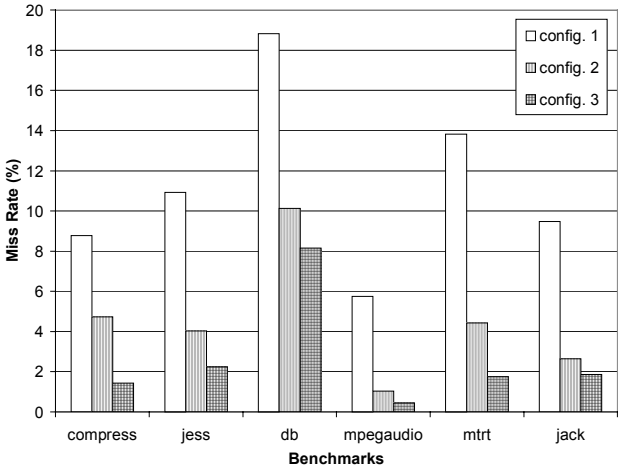
Table 6. Data cache (write) performance *(% Abs. miss* indicates absolute miss rate in a particular phase and *% Total miss* indicates the contribution of that phase to the total number of misses. Cache specifications: Configuration 1)

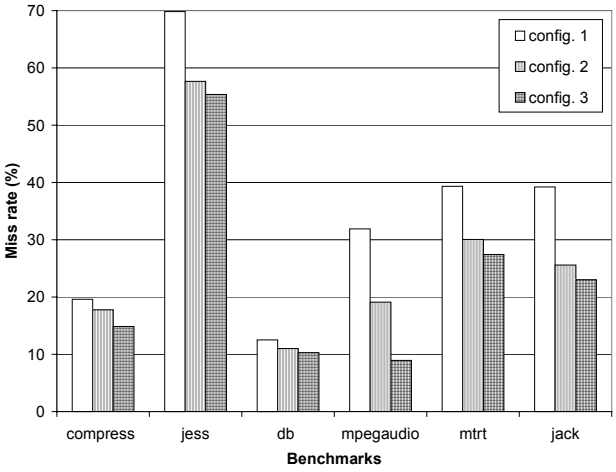
Benchmark	Execution Phase		Garbage Collection Phase		Overall
	% Abs. miss	%Total miss	% Abs. miss	%Total miss	% Abs. miss
Compress (int)	6.18	99.73	74.56	0.26	6.19
(jit)	19.61	97.77	67.37	2.20	19.92
Jess (int)	5.68	55.81	65.32	44.17	9.53
(jit)	69.86	45.72	63.88	54.26	66.48
Db (int)	5.18	86.36	60.59	13.63	5.92
(jit)	12.54	40.34	59.59	59.61	59.61
Mpegaudio (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

The runtime characteristics of benchmarks also account for the high write miss rate in the garbage collection phase. For example, compress spends most of its execution time in loops whereby the lifetimes of objects are less than the duration of a loop, which leads to frequent heap object allocations. The frequent allocations cause more interference between different phases and result in a poor cache performance.

4.3. Cache performance with increased cache sizes

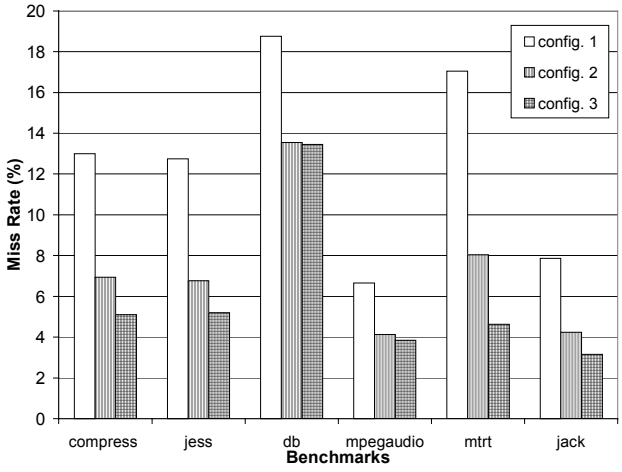
We experiment with different cache sizes to examine if the poor data cache performance in the JIT compiled mode is a result of mere capacity misses. We examine only the miss rates for data cache accesses since programs have very good instruction locality and the overall level 1 cache performance is mainly determined by data cache performance.

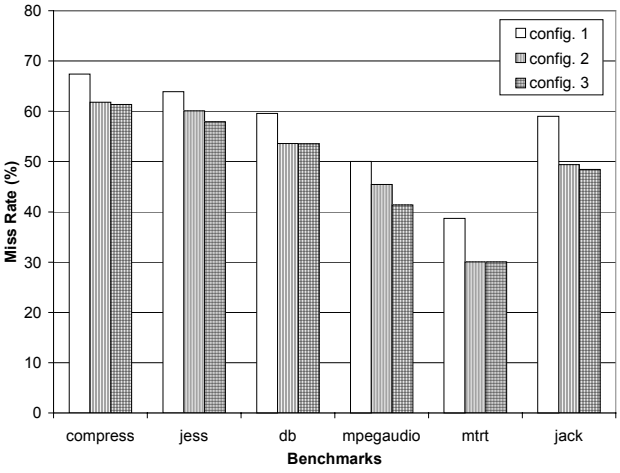




(a) Data cache read misses **(b) Data cache write misses**

Figure 2. Data cache performance of the execution phase under the JIT mode





(a) Data cache read misses **(b) Data cache write misses**

Figure 3. Data cache performance of the garbage collection phase under the JIT mode

As cache size increases, for both execution (Figure 2) and garbage collection (Figure 3) phases, data cache read misses reduce notably, while the reduction in data cache write misses is not very substantial. Since most write misses are compulsory misses (due to native code installation), a larger data cache size helps little on reducing compulsory misses. When data cache changes from configuration 1 to configuration 2, average cache miss reductions are 63% for reads in the execution phase, 23% for writes in the execution phase, 43% for reads in the garbage collection phase, and 12% for writes in the garbage collection phase.

Data cache miss reduction due to a larger cache is more substantial in the execution phase than that in the garbage collection phase, which holds true for both reads and writes. This is mainly due to the facts that the garbage collector's working set, the heap, is too large to fit into the L1 data cache for all three configurations, and write misses caused by pointer chasing are compulsory misses.

Data cache miss reduction diminishes as cache size increases. When data cache changes from configuration 2 to configuration 3, average cache miss reductions are 47% for reads in the execution phase, 16% for writes in the execution phase, 21% for reads in the garbage collection phase, and 3% for writes in the garbage collection phase respectively. All those reductions are smaller than corresponding ones when the data cache changes from configuration 1 to configuration 2.

4.4. Cache performance for the different data sets

As the data set for the SPECjvm98 benchmarks varies, different JVM components perform differently. For instance, the number of classes loaded for the s1 data set

roughly equals those loaded for the s100 data set for all SPECjvm98 benchmarks. Hence the occurrences of class loading and verification vary little as data set changes from s1 to s100. Another example is the JIT compiler. As the data set increases, methods are invoked more times to process the larger data. On the other hand, the number of method compilations holds constant since methods are compiled only in their first invocations. Consequently, the impact of JIT compilation on the overall cache miss rate diminishes.

The cache performance of programs using the s1 and s100 data sets is compared for the execution (Table 7) and garbage collection (Table 8) phases respectively. The L1 cache miss rates are compared as well for both interpreted and JIT execution modes.

Table 7. Execution engine cache miss rates under s1 and s100 data sets (Cache specifications: Configuration 1)

Miss Rates (%)	I-cache		D-cache Read		D-cache Write	
	s1	s100	s1	s100	s1	s100
Compress (int)	0.01	1.3	0.24	2.03	0.75	6.18
(jit)	0.58	0.07	8.03	8.78	25.89	19.61
Jess (int)	1.43	1.35	7.75	4.92	8.18	5.68
(jit)	1.95	1.48	9.46	10.92	61.2	69.86
Db (int)	0.45	0.16	2.05	3.67	5.17	5.18
(jit)	1.79	0.12	8.87	18.83	54.77	12.54
Mpegaudio (int)	1.02	0.6	2.17	0.94	1.66	1.51
(jit)	1.7	0.18	9.01	5.75	47.48	31.91
Mtrt (int)	0.59	0.42	3.18	4.09	3.77	2.44
(jit)	1.78	1.21	9.04	13.83	55.41	39.31
Jack (int)	0.7	0.72	2.64	2.68	2.41	2.34
(jit)	1.78	1.31	10.01	9.48	54.59	39.21

Under the JIT execution mode, the instruction cache performs better when using the s100 data set than the s1 data set since a higher method reuse results in a better instruction locality. The same trend is seen for data cache writes. Since one method needs to be compiled only in its first invocation, increasing the data set does not cause an increase in the number of method compilations. At the same time, application-specific data writes increase as the data set increases. As a result, compulsory misses caused by code installation have a smaller impact over the data cache writes when using s100 as opposed to s1. Contrary to the decreases in instruction and data cache write miss rates, data cache reads perform worse in s100 than in s1. This is a result of more capacity and conflict misses due to larger data set.

Cache performance under the interpreted mode varies little as data set increases. Under the interpreted mode, bytecodes of a method are interpreted whenever the method is invoked, and those bytecodes are treated as data. Hence in both s1 and s100 data sets, the interpreter dominates instruction cache accesses; bytecode accesses and stack operations of the interpreter dominate data cache reads; and stack operations of the interpreter dominates data cache writes. As a result, data set sizes have little impact on the cache performance in the interpreted mode.

Table 8. Garbage collection cache miss rates under s1 and s100 data sets (Cache specifications: Configuration 1)

Miss Rates (%)	I-cache		D-cache Read		D-cache Write	
	s1	s100	s1	s100	s1	s100
Compress (int)	0.04	0.15	3.3	16.14	52.17	74.56
(jit)	0.25	0.16	6.73	12.99	51.49	67.37
Jess (int)	0.73	1.03	4.04	14.74	59.8	65.32
(jit)	0.38	0.68	4.92	12.74	50.99	63.88
Db (int)	0.05	0.1	0.96	19.39	47.15	60.59
(jit)	0.28	0.03	5.49	18.76	47.99	59.59
Mpegaudio (int)	0.51	0.45	4.17	3.88	49.89	49.89
(jit)	0.31	0.31	6.64	6.66	50.07	50.04
Mtrt (int)	0.29	0.47	13.29	4.31	65.68	4.27
(jit)	0.17	0.51	10.52	17.04	62.62	38.69
Jack (int)	0.69	0.89	6.37	8.24	60.35	57.7
(jit)	0.29	0.27	6.18	7.86	57.21	58.99

As the data set increases from s1 to s100, data cache read and write miss rates for the garbage collection phase increase under both the interpreted and JIT modes (Table 8). This is mainly due to the increase in capacity and conflict misses of the garbage collector that comes as a result of the larger data set residing in the L1 data cache. On the other hand, instruction cache miss rates for the garbage collector on both execution modes vary little as the data set changes.

5. Conclusion

At the heart of Java technology lays 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 L1 cache performance of the JVM. We study in detail the cache performance of the different access types (instruction read, and data read and write) for Java applications. We look at three JVM phases (class loading, execution engine, and garbage collection), under two execution engines (interpretation and JIT compilation), using three cache configurations and two application data sets (s1 and s100) of the SPECjvm98 benchmark [4]. This multi-dimensional study enables us to analyze the impact of different JVM and application features on cache performance, and we find that JIT compilation's double-caching and pointer chasing together with the large working set of garbage collection affect cache performance in several aspects. The major observations of the paper are as follows:

- For most Java applications, L1 instruction cache performance is mainly determined by the execution engine.
- Garbage collection in the JIT execution mode is more frequent than in the interpreted mode since in JIT mode compiled native code also resides in the heap and increases the memory footprint substantially.
- Both data cache read and write miss rates in the garbage collection are much higher than those in the execution engine. Two factors account for the high miss rate in the garbage collection phase. First, garbage collector's working set (the heap) is too large to fit into the L1 data cache, which means that garbage collector has more conflict and capacity misses than the execution engine. Second, the pointer chasing algorithm used in the garbage collector module causes many compulsory misses in the data cache.
- Data cache performance in the JIT mode is worse than in the interpreted mode. The performance deterioration of data cache reads in the JIT mode is due to the decreasing of method bytecode locality since in the JIT mode bytecodes are read into the L1 data cache only for compilation, while in the interpreted mode bytecodes of a method are fetched into the L1 data cache whenever the method is interpreted. As the result of decreasing bytecode reads in the L1 data cache, data cache read miss rate increases. The performance deterioration of data cache writes in the JIT mode is due to compiled code installation, which incurs compulsory write misses.
- As cache size increases, data cache performance in the execution phase improves more than in the garbage collection phase, and a larger data cache is more effective on eliminating read misses than write misses in both the execution engine and the garbage collection.
- The impact that a change in the data set has over cache performance varies depending on the JVM phase and cache access type. In JIT mode, as the data set changes from s1 to s100, the instruction cache and data cache write miss rates of the execution engine decrease, while data cache read miss rate of the execution engine increases. Contrary to the notable cache performance differences in the JIT mode, in the interpreted mode, cache performance of the execution engine varies little as data set increases on most applications. For the garbage collection in either execution mode, data cache performance deteriorates when data set increases from s1 to s100, while varying data set affects little the instruction cache performance.

Acknowledgments

This research is partially supported by the National Science Foundation under grant numbers 0113105, 9807112, and by Tivoli, Motorola, Intel, IBM and Microsoft Corporations.

References

[1] The Java HotSpot Performance Engine Architecture, http://java.sun.com/products/hotspot/whitepaper.html.

[2] Kaffe. http://www.kaffe.org/.

[3] LaTTe: An Open-Source Java Virtual Machine and Just-in-Time Compiler. http://latte.snu.ac.kr/

[4] SPEC JVM98 Benchmarks, http://www.spec.org/osg/jvm98.

[5] B. Calder, D. Grunwald, and B. Zorn, "Quantifying Behavioral Differences Between C and C++ Programs", *Journal of Programming Languages*, Vol. 2, No. 4, 1995, pp. 313-351.

[6] A. Chandrakasan, W. Bowhill, and F. Fox, *Design of High-Performance Microprocessor Circuits*, IEEE Press, 2000.

[7] R. F. Cmelik and D. Keppel, "Shade: A Fast Instruction Set Simulator for Execution Profiling", *Technical Report SMLI TR-93-12,* Sun Microsystems Inc., 1993.

[8] M. O'Connor and M.Tremblay, "PicoJava-I: The Java Virtual Machine in Hardware*", IEEE Micro*, March 1997, pp. 45-53.

[9] S. Dieckman and U. Holzle, "A study of the Allocation Behavior of the SPECjvm98 Java Benchmarks", *ECOOP98*, pp. 92-115.

[10] T. Horel and G. Lauterbach, "UltraSPARC-III: Designing Third-Generation 64-Bit Performance", *IEEE Micro*, May-June. 1999, pp. 73-85.

[11] 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.

[12] C. A. Hsieh, J. C. Gyllenhaal, and W. W. Hwu, "Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results", *Proceedings of the 29th Annual International Symposium on Microarchitecture*, 1996, pp. 90-97.

[13] J. Kim and Y. Hsu, "Memory System Behavior of Java Programs: Methodology and Analysis", *Proceedings of the Joint International Conference on Measurement and Modeling of Computer System*, 2000, pp. 264-274.

[14] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison Wesley, 1997.

[15] T. Newhall and B.Miller, "Performance Measurement of Interpreted Programs", *Proceedings of Euro-Par '98*, 1998.

[16] 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.

[17] 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*, 1999, pp. 281-284.

[18] R. Radhakrishnan, N. Vijaykrishnan, A. Sivasubramaniam, and L. K. John, "Architectural Issues in Java Runtime Systems", *Proceedings of the International Symposium on High Performance Computer Architecture*, 2000, pp. 387-398.

[19] R. Radhakrishnan, J. Rubio, L. K. John, and N. Vijaykrishnan, "Execution Characteristics of JIT Compilers", *Technical Report TR-990717-01*, University of Texas at Austin, 1999.

[20] A. Rajan, "A Study of Cache Performance in Java Virtual Machines", *Master's Thesis*, University of Texas at Austin, 2002.

[21] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J. Baer, B. N. Bershad, and H. M. Levy, "The Structure and Performance of Interpreters", *Proceedings of ASPLOS VII*, 1996, pp. 150-159.

[22] 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", *Proceedings of the Joint International Conference on Measurement & Modeling of Computer Systems*, 2000, pp. 194-205.

[23] B.Venners, *Inside the Java 2 Virtual Machine*, McGraw Hill, 2000.

[24] P. Wilson, "Uniprocessor Garbage Collection Techniques", *International Workshop on Memory Management*, Sep. 1992, pp. 16-18.

[25] B. Yang, S. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioglu, and E. Altman, "LaTTe: A Java VM Just-in-time Compiler with Fast and Efficient Register Allocation", *International Conference on Parallel Architectures and Compilation Techniques*, October 1999, pp. 128-138.

[26] T.Yeh and Y.Patt, "A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution", *IEEE Micro*, 1992.