A Study of Cache Performance in Java Virtual Machines

by

Anand Sunder Rajan, B.E., M.Sc

Report

Presented to the Faculty of the Graduate School

of The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2002

A Study of Cache Performance in Java Virtual Machines

APPROVED BY

SUPERVISING COMMITTEE

_____

_____

Acknowledgements

I would like to thank Dr. Lizy John for her valuable guidance and advice during the course of this work. I would also like to express my gratitude to Dr Doug Burger for agreeing to be the reader for my report. I also want to thank all the members of the Laboratory for Computer Architecture for all their invaluable help and encouragement.

May 2002

A Study of Cache Performance in Java Virtual Machines

by

Anand Sunder Rajan, M.S.E

The University of Texas at Austin, 2002

SUPERVISOR: Lizy K. John

Java is a widely used programming language due to the portability and machine independent nature of bytecodes. Considering the fact that we have quite a few options available in the execution of Java bytecodes, it is very important to have a clear understanding of the runtime performance aspects in each of the modes. This work attempts to characterize the cache performance of the interpreted, JIT and mixed modes. This study delves deep into the reasons for poor data cache performance in JITs by separating the virtual machine into functionally disparate components and studying cache performance in each of the components. 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. The instruction cache performance in the JIT compilation is always worse than that in the interpreted mode. Data writes exhibit extremely poor performance in JIT modes of execution and the miss rates are on an average 38%. Intelligent translation of Java methods implemented by dynamic profiling in mixed-mode execution engines like HotSpot does not change the overall cache performance of the JVM. We hope that this study serves as a pointer to optimizing specific

sections of the code in the JVM. Our results indicate that the code translation routines of the JIT are good candidates for optimization. We also hope that it would be a guide for architectural enhancements that can mitigate the effect of poor cache performance. An example of such an enhancement could be in the form of directly generating code into an instruction cache that accommodates write operations.

Table of Contents

1. Introduction

Java is a widely used programming language due to the portability and machine independent nature of bytecodes. It is widely recognized as a language for applications deployed over computer networks. In addition to portability, security and ease of development of applications has made it very popular with the software community. Initially, its success was related to the growth of the Internet but now Java technology is expanding in wider areas, such as real-time embedded systems and day-to-day computing.

1.1 The Java Virtual Machine

Java's architecture arises from [20] four distinct but interrelated technologies:

- The Java programming language
- The Java class file format
- The Java API
- The Java virtual machine

When a Java program is run, all these technologies come into play. Source files written in Java are compiled into class files and run on a Java virtual machine. System resources like I/O are accessed inside the program by calling methods in the classes that implement the Java API. A pictorial depiction of this relationship is shown in Figure 1.1.

Together the Java virtual machine and the Java API form a “platform “ for which all Java programs are compiled. Java programs can run on many different kinds of computers, because the Java platform is itself implemented in software. At the heart of Java’s philosophy is the Java virtual machine, which supports all the three prongs of its network-oriented architecture – platform independence, security and network mobility.

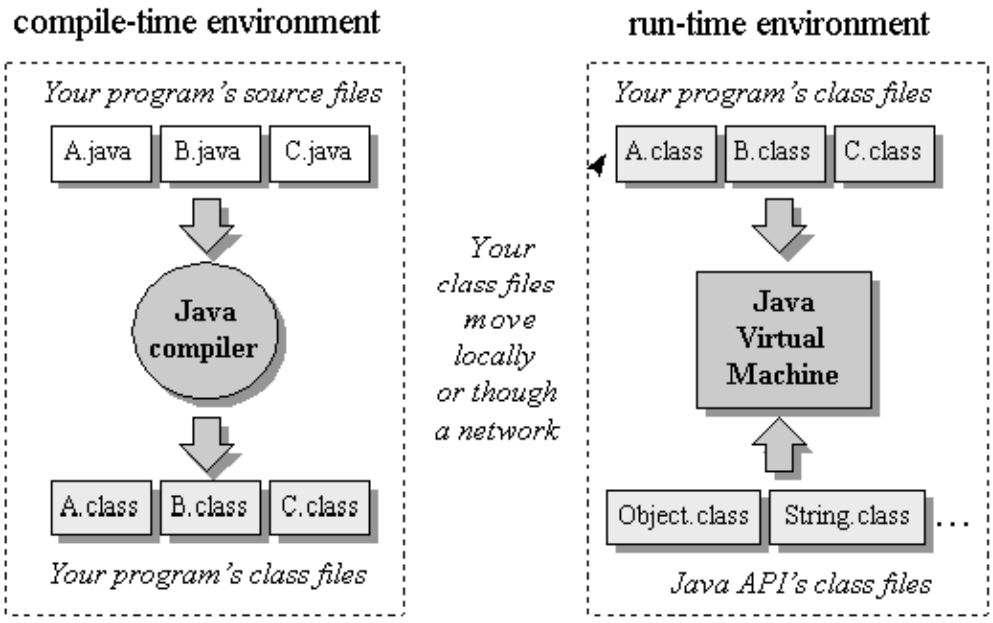

Figure 1.1 The Java Programming Environment

Together the Java virtual machine and the Java API form a “platform “ for which all Java programs are compiled. Java programs can run on many different kinds of computers, because the Java platform is itself implemented in software. At

the heart of Java's philosophy is the Java virtual machine, which supports all the three prongs of its network-oriented architecture – platform independence, security and network mobility.

The Java virtual machine (JVM) is an abstract computer. Its specification defines certain features every JVM must have but leaves many choices to the designers of each implementation. For example, although all JVMs must be able to execute Java bytecodes, they may use any technique to execute them. Also, the specification is flexible enough to enable a JVM to be implemented either completely in software – or to varying degrees in hardware. This flexibility enables a JVM to be implemented on a wide variety of computers and devices.

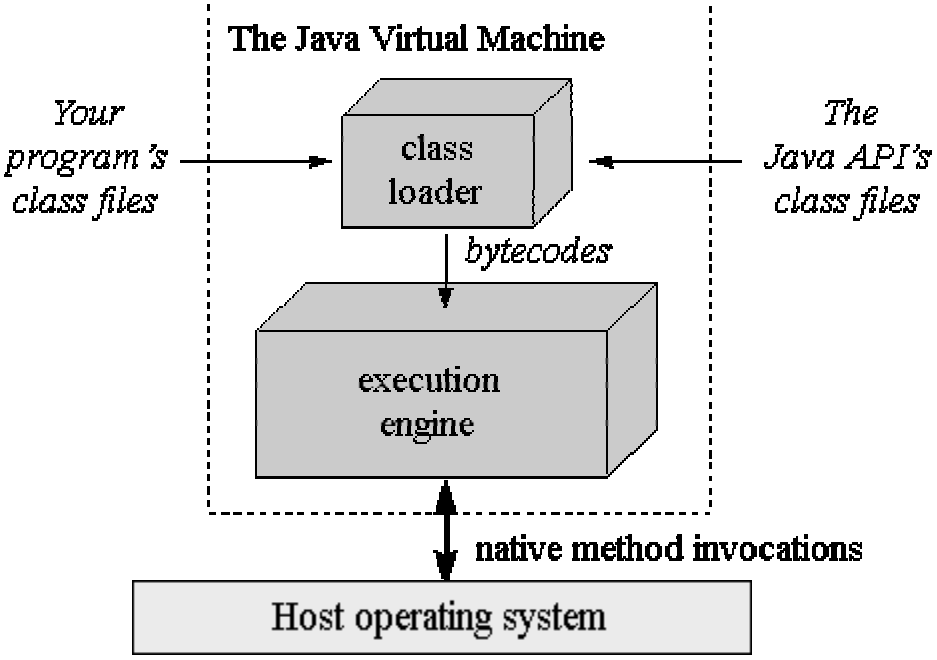

Figure 1.2 The Role of the Class Loader in the JVM

1.1.1 Class Loader Architecture

The class-loader architecture plays an important role in both security and network mobility. Figure 1.2 depicts the role of the class loader in the execution of Java code. The JVM has a flexible class loader architecture that enables a Java application to load classes in custom ways; the class loader is actually a subsystem that involves many class loaders. A Java application can use 2 types of class loaders: a “bootstrap” class loader and user-defined class loaders. The bootstrap loader is part of the implementation of the JVM and loads classes of the Java API and user defined classes in some default way. At run-rime, a Java application can install user-defined class loaders that load classes in custom ways, such as by downloading class files across a network. These user-defined class loaders are written in Java, compiled into class files, loaded into the virtual machine and instantiated just like any other object.

Because of user-defined class loaders, it is not necessary to know all classes that may ultimately take part in a Java application at compile time itself. Some of the API classes need to be loaded before the application can start to execute. 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.

Not needing to know beforehand the classes that would be required by the application is an important feature of Java, and is called dynamic class loading. As we will examine in later, some implementations of the JVM cannot provide this feature. For each class it loads, the JVM keeps track of the class loader used to load this class.

When a loaded class refers to another class, the virtual machine loads the referenced class from the same class loader that originally loaded the referencing class. Thus, the referenced class and the referencing class are dynamically linked.

1.1.2 Java Execution Modes

A JVM's main job is to load class files and execute the bytecodes they contain. The class loader loads only those files that are actually needed by the running program and the bytecodes corresponding to these classes are executed in the execution engine. Since the specifications [1] 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, which uses profile based feedback to interpret/compile bytecodes. Other possible modes include hardware execution and ahead-of-time compilation of bytecodes. Figure 1.3 depicts the various execution modes. We examine the pros and cons associated with each of them.

1.1.2.1 Interpretation

The traditional mode of execution has been interpretation (Figure 1.4) of the bytecodes whereby an interpreter decodes and executes the bytecodes using a software loop. This emulation of the virtual machine is exceedingly slow because the

fetch and decode functionalities of normal program execution (reading and updating program counters, decoding the instruction, transferring control to activities that correspond to the opcode of the decoded instruction etc) are performed in software. Thus, performance [8] of interpreted Java is generally deemed acceptable for small applets but not for any sizeable application.

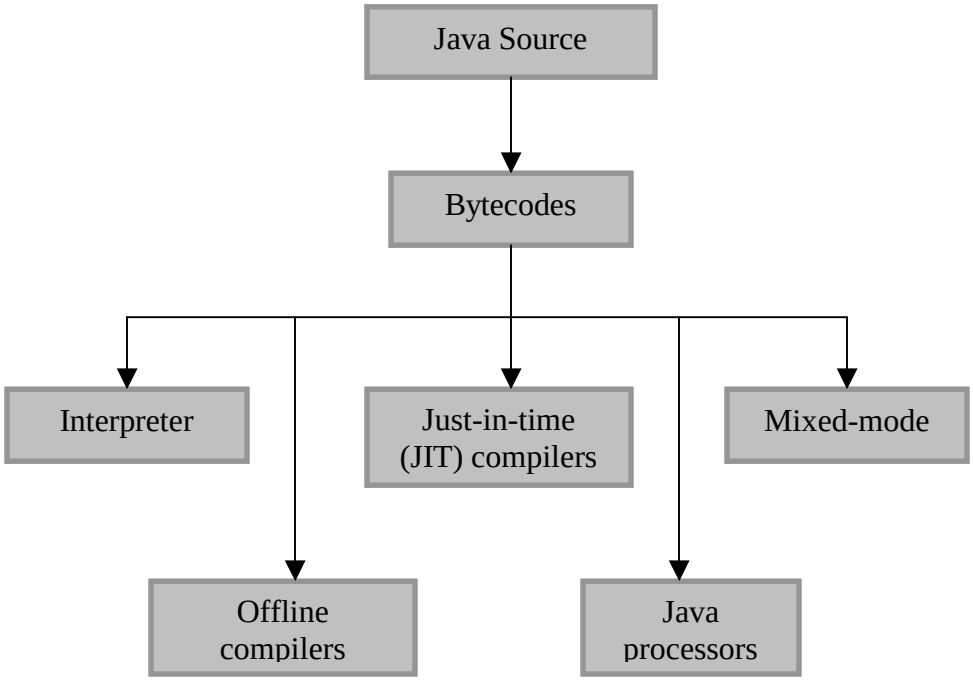

Figure 1.3 Execution Modes for Java Bytecodes.

1.1.2.2 Offline Compilation

It has been suggested that Java bytecodes be compiled offline [14] before they are executed, similar to traditional C/C++ programs. A key advantage of offline analysis is the ability to perform complete flow analysis. This analysis directly enables a number of critical optimizations, as is done in traditional compilers for C/C++ applications. This leads to more efficient execution but suffers from the

drawback of not adhering to the write-once read-anywhere (WORA) philosophy of Java. It is not always possible to support the concept of dynamic class loading though there are some implementations that claim to fully implement dynamic class loading. As a result, this method has been confined to delivering high performance on certain types of specialized server systems such as massively distributed servers and high availability systems.

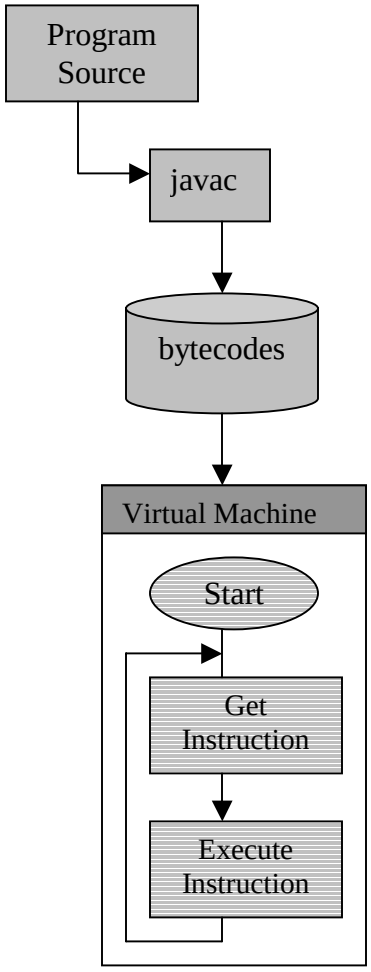

Figure 1.4 The Interpreted Mode of Execution

1.1.2.3 JIT Compilation

The most commonly used mode of execution [16] is just in-time compilation (JIT), which compiles bytecodes on the fly and runs many times faster than an interpreter. It makes a noticeable difference when running an interactive application. Although it falls short of the quality and speed of compiled code, it greatly extends Java's applicability. The JIT compiler (Figure 1.5) will have better performance than the interpreter only when the there is large reuse of methods and suffers from poor data cache performance. In short, the reason for poor data cache performance is the installation of compiled bytecodes. This will be examined in detail over the course of Chapter 4. There has been a quantum improvement in the performance of JITs over the years due to the use of innovative dynamic compilation techniques.

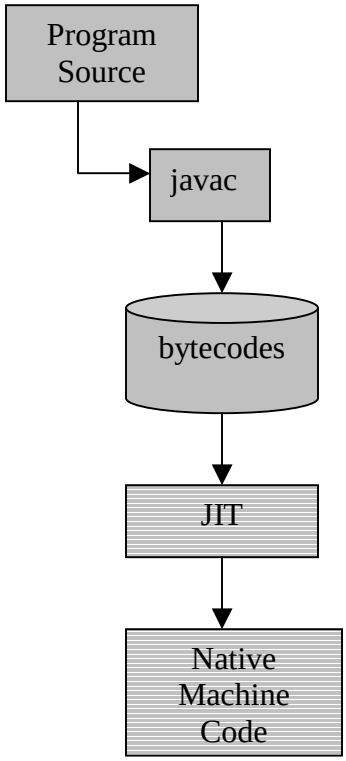

Figure 1.5 The JIT Mode of Execution

1.1.2.4 Hardware Execution

Yet another technique suggested to improve the performance of Java programs has been to use a hardware accelerator or coprocessor [22] that works in conjunction with a microprocessor. Essentially, the effort is to bridge the semantic gap that exists between the bytecodes and the native instructions. Java processors like Sun Microsystems' Java cores and JEM are low-cost hardware engines optimized to directly execute Java code.

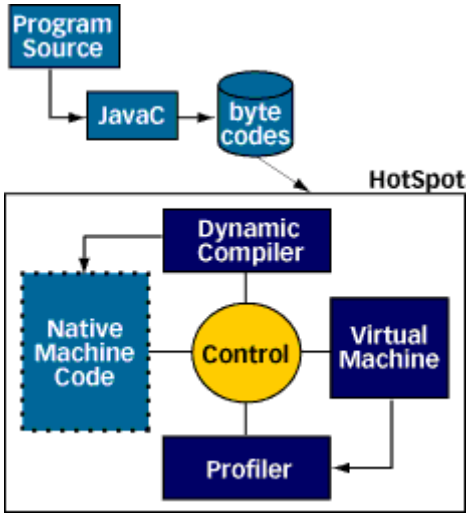

Figure 1.6 Mixed-mode Execution

1.1.2.5 Mixed-mode Execution

The latest and most promising mode of execution is the mixed-mode execution technique (Figure 1.6) epitomized by Sun Microsystems' HotSpot technology [4] and IBM's Jalapeno [17]. This system uses online profiling [18] to identify and compile a performance critical subset of the Java methods, while

interpreting the rest. Online profiling allows the compiler to spend more time on optimizing the frequently used methods. On the other hand, the policy of translating a method as soon as it is encountered in a JIT, leads to precious time being wasted in optimizing rarely used methods, thus resulting in significant increase in execution time [15]. Our results show that the mixed-mode execution engine still suffers from the performance issues that plague the JIT.

1.1.3 The Garbage Collector

When looking at the Java virtual machine, it is also important to look at the garbage collector, which plays a significant role in affecting the performance of the JVM as a whole. The garbage collector [1] 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, thus making this space available for allocating new objects. In addition to freeing unreferenced objects, the garbage collector also combats heap fragmentation. Heap fragmentation occurs during the normal course of execution of the program when new objects are allocated, and unreferenced objects are freed so that free portions of heap memory are left in-between portions occupied by live objects.

Garbage collection relieves the programmer of the burden of freeing allocated memory. The responsibility of de-allocating objects is complicated, and leaving it in the hands of the user can lead to memory leaks and an increase in software-development time. It also helps in ensuring the program integrity by not allowing

users to accidentally or purposely crash the JVM by incorrectly freeing memory. A potential disadvantage of a garbage-collected heap is that it adds an execution overhead that affects the program performance. The garbage collector has to keep track of objects being referenced by the executing program and finalize and free unreferenced objects on the fly. Thus, the execution of the garbage collector falls in the critical path of program execution and tends to increase the execution time. Having to move data structures and objects about the heap tends to pollute the data caches as is seen in our results.

1.2 Objective of Research

Considering the fact that we have quite a few options available in the execution of Java bytecodes, it is very important to have a clear understanding of the runtime performance aspects in each of the modes. Performance analysis has become one of the critical means of designing well-balanced, efficient processor and system organizations. In particular, the importance of evaluating memory sub-system performance cannot be understated. 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 [23] that are used in cache memories, the prevalence of deep superscalar pipelines and aggressive techniques to exploit ILP [24] make it imperative that cache misses are minimized.

This work attempts to characterize the cache performance of the interpreted, JIT and mixed modes. 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 study delves deep into the reasons for poor data cache performance in JITs by separating the virtual machine into functionally disparate components and isolating the component responsible for the major chunk of these misses. More specifically, for our experiments we use the open-source Latte JVM [5] and instrument it to provide detailed cache performance statistics for the different phases of execution of the JVM. The 3 distinct phases we examine are the class loader, the execution engine (it could be the interpreter or the JIT compiler) and the garbage collector. We also repeat our experiments with enhanced cache sizes to see whether our observations are modified with larger caches, which are seen primarily in servers and large computing systems. In addition to the above, we also study the overall cache performance of the HotSpot Server JVM to study if there are any improvements in performance obtained with mixed-mode execution systems.

It is seen that the long speculated theory [10] about poor data cache performance in JITs being caused by compulsory misses incurred during code installation are quite true, and the miss rate can be as high as 70% in the execution engine. In addition, code installation also causes deterioration in performance of the instruction cache during execution of translated code. Also, there is considerable interference between data accesses of the garbage collector and the compiler-

translator execution engine of the JIT compiled mode. This interference between data accesses of the garbage collector and the JIT execution engine leads to further deterioration of the data cache performance wherever the contribution of the garbage collector is significant, thus resulting in miss percentages in the garbage collection phase are of the order of 60%. We observed that an increase in cache sizes provides a performance improvement of 47-83% in the case of data cache reads and about 70% for the instruction cache. The trend is not followed in the case of data cache writes, where the improvement is hardly noticeable. We also find that with the mixed-mode execution style of HotSpot, our problems persist and in some cases, there is deterioration in performance due to alternating between the 2 modes of execution.

1.3 Outline of Report

The remainder of the report is organized as follows. Chapter 2 presents the prior research done in this area. Chapter 3 discusses the experimental methodology, including the benchmarks, JVMs and tools used for the experiments. Chapter 4 discusses in detail the results for the various data sets and JVMs and analyzes them and chapter 5 offers concluding remarks.

2.Background and Motivation

2.1 Previous Research

Due to the promise of Java's write-once run-anywhere capability for Internet applications, there has been a good amount of research towards analyzing the performance of Java in its various modes of execution. Earlier studies focused on the interpreted mode, and one of the first works in this area is due to Romer [8] who measured the MIPSI, Java, Perl and Tcl interpreters running an array of micro and macro benchmarks on the DEC Alpha platform. They concluded that interpreter performance is primarily a function of the interpreter itself and is relatively independent of the application being interpreted. Our experiments will focus only on the Specjvm 98 benchmarks [3] only and will examine cache performance in detail in the components of the interpreter rather than just looking at it as a whole.

Newhall and Miller [7] developed a tool based on a performance measurement model that explicitly represents the interaction between the application and the interpreter. This tool measures the performance of interpreted Java applications and is shown to help application developers tune their code to improve performance. Radhakrishnan et al. [9] analyzed Java applications at the bytecode level while running in an interpreted environment and did not find any evidence of poor locality in interpreted Java code. Our results pertaining to interpreter cache performance agree with the ones obtained in [9]; in addition, we compare these results to the JIT and mixed-mode execution styles.

There has been quite some research on the issue of ahead-of-time translation of Java bytecodes. Proebsting [14] presented Toba, a system for generating stand-alone Java applications and found that Toba-compiled applications execute 1.5-4.2 times faster than interpreted and JIT applications. Hsieh [2,13] studied the impact of interpreters and offline Java compilers on microarchitectural resources such as cache and the branch predictor. They attribute the inefficient use of the microarchitectural resources by the interpreter as a significance performance penalty. They observed in their work that an offline bytecode to native code translator is a more efficient Java execution mode for utilizing the caches and the branch predictors. Our results do not agree with that in the sense that the miss rates seen in the interpreted mode of execution for the instruction as well as data caches are better than for C++ programs.

There have been quite a few studies looking at the execution characteristics and architectural issues involved with running Java in the JIT mode. Most relevant to our experiments is the work of Radhakrishnan [10], which investigates 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. It also speculated that install write misses during the installation of JIT compiler output has a significant effect on the data cache performance in JIT mode. Our results confirm their speculation and in addition, examine their relevance in a mixed-mode execution environment. Radhakrishnan [11] provided a quantitative characterization of the execution behavior of the SPEC

JVM98 programs, in interpreter mode and using JIT compilers and obtained similar results to the ones obtained in [10].

There has been a gradual shift towards exploring mixed mode execution systems that use JIT compilation selectively on only those portions of the Java methods that are frequently executed. Barisone [12] have presented a detailed instruction-level characterization of the overall behavior of the HotSpot Server compiler running the SPEC JVM98 benchmarks and compared the same with the interpreted and JIT compilation modes. Our experiments differ in the sense that in addition to this, we look at cache performance in the components of the JVM. Agesen [15] propose a 3-mode execution engine, comprised of a single interpreter, a fast non-optimizing compiler and a slow, optimizing compiler used in the background with the aim of providing fast startup and high performance simultaneously. Their results show significant performance gains over conventional modes of Java execution. Their results are in terms of execution time and do not look at the performance of caches.

2.2 Motivation

The motivation for this study is the need to examine in detail the cache performance of the JIT compilation mode and understand the reasons for its poor cache performance in comparison to the interpreted mode of execution. Prior work has given conclusive evidence of the poor data cache performance of JIT compiled Java code. It was seen [10] that the overall data cache miss rate is as high as 10-15%

in the JIT mode and not more than 4-8% in the interpreted mode, even though there is a reduction of 20-80% in number of data cache accesses as we move from the interpreted to the JIT mode. This study builds on previous works by examining cache performance in the different stages of execution of Java code namely the class loading, compilation/interpretation and garbage collection phases. This study offers conclusive proof of the speculation that install write misses during the installation of JIT compiler output have a significant effect on the data cache performance in JIT mode [10]. It segregates the cache performance in the 3 phases mentioned and serves as a pointer to optimizing specific sections of the code in the JVM. It could also serve as a guide for architectural enhancements that can mitigate the effect of poor cache performance.

Lastly, the results obtained in this detailed characterization are compared to the cache performance tradeoffs seen with the mixed-mode execution engine. This serves to highlight the performance gains associated with moving to an execution mode that optimizes more selectively than the pure JIT compilation mode.

3. Experimental Methodology

In this section, we describe the tools that were used for the study and the different benchmarks that were studied.

3.1 Tools and Platform

Our study of cache performance of the Latte JVM was performed on the UltraSparc platform running Solaris 2.7 using tracing tools and analyzers. Sun Microsystems provides Shade [6], a 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 the cachesim5 analyzer provided by the Shade suite of programs. Cachesim5 is used to model 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. A trace of instructions is input to the analyzer and the output provides detailed information on:

a) Total number of accesses.

b) Total number of misses as absolute numbers and as a percentage of accesses (read, write and combined).

c) Percentage of cache blocks that were valid, dirty, unused etc.

d) Other write statistics based on write policies specified. For example, dirty write backs due to data writes, total write backs etc.

This analyzer was modified to suit the requirements of our measurements and validated to examine the correctness of these changes. It must be added that this analyzer is not cycle accurate and thus, timing issues are not considered in our experiments.

3.2 The Latte Virtual Machine

We used the Latte virtual machine as the target Java Virtual Machine to study the cache performance in each of the distinct phases of a JVM. Latte [5] 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 claims to perform efficient garbage collection and memory management using a fast mark and sweep algorithm. It makes exception handling more efficient by using on-demand translation of exception handlers.

3.3 The HotSpot VM

We used the HotSpot Client VM 1.3.1 [4] from Sun Microsystems to compare our results for the traditional execution mode JVMs to the more recent mixed-mode execution mode JVMs. It has been specially tuned to maximize peak operating speed and is intended for executing long-running server applications, for which having the fastest possible operating speed is generally more important than having a fast startup time or smaller runtime memory footprint.

Mixed-mode execution is expected to solve the problems of JIT compilation by taking advantage of an interesting property of most programs. Virtually all programs spend the vast majority of their time executing a small minority of their code. Rather than compiling method-by-method, just in time, the Java HotSpot VM runs the program immediately using an interpreter and analyzes the code as it runs to detect the critical "hot spots" in the program. It then focuses the attention of a global native-code optimizer on the hot spots. By avoiding compilation of infrequently executed code (most of the program), the Java HotSpot compiler is expected to devote much more attention to the performance-critical parts of the program, without necessarily increasing the overall compilation time. This hot-spot monitoring is continued dynamically as the program runs, so that it literally adapts its performance on-the-fly to the needs of the user.

The HotSpot client VM contains an advanced adaptive compiler that supports many of the same types of optimizations performed by optimizing C++ compilers, as well as some optimizations that cannot be done by traditional compilers, such as

aggressive inlining across virtual method invocations. Adaptive optimization technology is very flexible in its approach, and typically outperforms even advanced static analysis and compilation techniques.

3.4 Instrumentation of the JVM

Since the objective was 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 sentinel generation code has been chosen in such a way that these high-level language statements are translated into double word store (STD) instructions 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 such a manner that the probability of the consecutive occurrence of these 3 numbers is almost negligible. The occurrence of the sentinel is checked in the modified Cachesim5 analyzer.

3.5 Benchmarks

The SPEC JVM98 suite of benchmarks [3] was 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 SPEC JVM98 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.

No.	Benchmark	Description and Source
1.	compress	A popular LZW compression program.
2.	Jess	A Java version of NASA's popular CLIPS rule-based expert systems
3.	Db	Data management benchmarking software written by IBM.
4.	mpegaudio	The core algorithm for software that decodes an MPEG-3 audio stream.
5.	Mtrt	A dual-threaded program that ray traces an image file.
6.	Jack	A real parser-generator from Sun Microsystems.

Table 3.1: Description of the SPEC JVM98 benchmarks used

Both the s1 and s100 data sets were used for the experiments.

3.6 Cache Hierarchies and Modification of Cachesim5

The following cache hierarchies were chosen with the first configuration corresponding to the cache hierarchy on the UltraSparc-1 processor [19]:

a) Configuration 1: This configuration has a 16K L-1 Instruction Cache with block size of 32 bytes and no sub-blocking. It is 2-way associative, follows an LRU replacement policy and the write policy is write through. The L-1 Data Cache is 16K with block size of 32 bytes and sub-blocks of 16 bytes. It is direct mapped and the write policy is write-through with no write-on-allocate. The L-2 Unified Cache is 512K with block size of 64 bytes and no sub-blocking. It is direct mapped and the write policy is write-back with write-on-allocate.

b) Configuration 2: This configuration has a 64K L-1 Instruction Cache with block size of 32 bytes. It is 2-way associative, follows an LRU replacement policy and the write policy is write through. The L-1 Data Cache is 64K with block size of 32 bytes and no sub-blocking. It is 4-way associative and the write policy is write-through with no write-on-allocate. The L-2 Unified Cache is 1M with block size of 64 bytes and no sub-blocking. It is direct mapped and the write policy is write-back with write-on-allocate.

c) Configuration 3: This configuration has a 256K L-1 Instruction Cache with block size of 32 bytes. It is 2-way associative, follows an LRU replacement policy and the write policy is write through. The L-1 Data Cache is 256K with block size of 32 bytes. It is 4-way associative and the write policy is write-through with no write-on-allocate. The L-2 Unified Cache is 2M with block size of 64 bytes and no sub-blocking. It is direct mapped and the write policy is write-back with write-on-allocate.

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.

3.7 Validation of Modified Cachesim5

The validation of the modified Cachesim5 was 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 [12] and there was close correspondence between the numbers obtained in both the studies.

4. Results and Analysis

This chapter summarizes the results of this study that characterizes the SPEC JVM98 benchmarks in terms of the cache performance in the various stages of the execution of the Latte Virtual Machine. In addition, we also present the results of the same experiments with the HotSpot VM.

4.1 Metrics and Data Sets

Cache performance was evaluated for the s1 as well as the s100 data sets of the SPEC JVM benchmarks with the 3 different cache configurations specified in the previous chapter. A larger data set does not imply a change in the static structure of the program; it increases the number of dynamic instructions executed by increasing the amount of method reuse, which is analogous to increasing the loop indices of the program. By increase in method reuse, we mean the number of times a method is invoked is increased greatly. Table 4.1 presents the number of methods and the number of dynamic method calls in the s1 and s100 data sets for each of the benchmarks.

As mentioned earlier, the results of the experiments are in the form of cache performance statistics for the 3 phases of the execution of the Latte JVM:

a) Class Loading

b) Execution Engine – Interpretation / Compilation

c) Garbage Collection

Benchmark	s1 data set		s100 data set	
	Calls	Methods	Calls	Methods
compress	17.33M	577	14.56M	449
jess	414,349	1222	95.96M	1375
db	65,379	642	91.75M	658
mpegaudio	954,605	843	93.05M	844
mtrt	1.91M	781	71.17M	796
jack	2.32M	1230	39.17M	1240

Table 4.1 Number of methods and dynamic method calls for s1 and s100 data sets.

4.2 Organization of Results

We were unable to instrument and segregate phases in the HotSpot JVM because the VM does not build on our hardware platform, Solaris 2.7, upon modification to the source. This problem is expected to be corrected in all future releases. Hence, the results tabulated for this virtual machine will be for the VM as a whole. In all other tabulations of the results, we will present absolute numbers for instructions or data references and misses in each of the phases. The miss rate will be expressed as the percentage of cache accesses in that particular phase that missed. For clarity, only the s100 data set results are presented in this chapter. The s1 data set results are presented in the appendix.

The organization of the results is as follows. Section 4.3 presents results with the Latte VM for the instruction cache performance in the interpreted and JIT modes of execution. Sections 4.4 and 4.5 present the results for the data cache performance in the interpreted and JIT modes of execution for reads and writes respectively. Read accesses in the both modes of execution consist of reading method bytecodes and the

data required for the execution of these methods. The crucial 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 differences 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.

Section 4.6 presents the effect of increased cache sizes on the performance in the Latte VM by comparing performance using 3 different cache configurations. The motivation behind these comparisons is to examine whether the poor data cache performance in the JIT compiled mode of the Latte VM was a result of mere capacity misses. For completeness, we also present the trends with the instruction cache accesses and the data cache reads.

Section 4.7 presents cache performance statistics for the 3 configurations with the HotSpot Server VM. We wanted to examine if poor data cache performance applies to newer JVMs too that use mixed-mode execution and is not localized to traditional JVMs using JIT compiled modes of execution alone. Again, for completeness we present results for instruction cache accesses as well as data cache reads.

We will use the following convention to assist in following the results more clearly. Each table in this section will be referred to by a tuple of the form <JVM, execution mode, operation on cache, cache configuration>. JVM can take on values Latte or HotSpot. Execution mode can take on values int (interpreted mode of execution), jit (JIT mode of execution) or mix (mixed mode execution). Operation on cache takes on values ins (instruction access), dr (data read) and dw (data write). Cache configuration can take on values 1,2 or 3 indicating configurations 1,2 and 3 respectively. For example, <Latte, jit, dr, 2> indicates results for the Latte VM in the JIT mode of execution for data read operations with Cache Configuration 2.

4.3 Instruction Cache Performance

4.3.1 Interpreted Mode of Execution (<Latte, int, ins, 1>)

In the case of the Instruction Cache, it is seen from Table 4.2 (<Latte, int, ins, 1>) that 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. The overall instruction miss rate is therefore almost same as for the interpretation phase for the JVM execution. 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 tuple 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. Classes loaded on demand are classified as being part of the execution phase.

Instruction miss rate here varies from 0.16% to 1.33% and this good instruction locality is due to the fact that 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 be fit into the instruction cache. Garbage collection plays a significant role in the case of the mtrt benchmark whereby it contributes about 33% of the instructions executed. 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.

compress			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	503.06K	8661	1.72
Interpretation	143.27G	1.86G	1.30
Garbage Collection	25.77M	40K	0.15
Total	143.30G	1.87G	1.31

jess			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	503.06K	8661	1.72
Interpretation	28.82G	388.84M	1.35
Garbage Collection	1.37G	14.15M	1.03
Total	30.19G	403.01M	1.33

db			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	503.06K	8653	1.72
Interpretation	53.53G	86.14M	0.16
Garbage Collection	520.41M	539.97K	0.10
Total	54.05G	86.70M	0.16

mpegaudio			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	503.06K	8653	1.72
Interpretation	128.65G	781.22M	0.60
Garbage Collection	6.62M	29.67K	0.45
Total	128.66G	781.27M	0.61

mtrt			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	503.06K	8663	1.72
Interpretation	13.20G	5.58M	0.42
Garbage Collection	25.42G	121.54M	0.47
Total	38.63G	177.38M	0.46

jack			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	503.06K	8661	1.72
Interpretation	38.82G	279.59M	0.72
Garbage Collection	916.52M	8.22M	0.89
Total	39.73G	287.83M	0.72

Table 4.2 Instruction Cache Performance for the s100 data set with the Latte VM for the interpreted mode of execution.

4.3.2 JIT Mode of Execution (< Latte, jit, ins, 1>)

Table 4.3 < Latte, jit, ins, 1> shows the instruction cache performance numbers seen with the JIT compilation mode. It is very clear that there is almost an 80% to 90% decrease in the dynamic instruction count when moving from the interpreted mode to the JIT. 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. 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 [23]). In addition, the code installed by the translator need not be contiguously placed in the cache contributing to poorer performance. Exceptions to this are the benchmarks with larger footprints; compress (the miss rate decreases from 1.3% to 0.07%) and mpegaudio (the miss rate decreases from 0.6% to 0.2%), where there is high method reuse and actual execution of the translated code dominates the compilation process.

As before, the contribution of the class loading stage is not considerable and the actual compilation stage dominates. But the garbage collection stage shows a lot more activity here with the jess and mtrt benchmarks. But the absolute instruction misses in this phase show no particular trend. It is clear 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 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%).

It was speculated [9] that the instruction cache performance in the JVMs with JIT mode of execution would be poorer than in the interpreted mode due to the poor

locality exhibited by all compilers. But we do not see a clear trend between the overall miss rates of the interpreted and JIT modes in our results. It was also expected that since the compiled code for the methods are small, and do not have large basic block sizes, there would be frequent breaks in the instruction run causing a large number of misses. One would expect that as a result of frequent breaks in instruction run, the miss rates would decrease as we moved to the larger data sets because the execution of the application code after translation into the native form would start dominating in the larger data sets and also there would be better method reuse. This is confirmed by the results seen for the s1 and s100 data sets in the JIT mode. For the s1 data set (Appendix B1), the compilation phase shows consistently higher miss rates (from 1.5% to 1.9%) when compared to the s100 data set (from 0.1% to 1.3%).

compress			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	1011	0.495
Translation + Execution	9.56G	6.89M	0.07
Garbage Collection	44.95M	71,966	0.16
Total	9.60G	6.97M	0.07

jess			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	1011	0.495
Translation + Execution	3.98G	58.76M	1.48
Garbage Collection	1.44G	9.8M	0.68
Total	5.42G	68.58M	1.26

db			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	1011	0.49
Translation + Execution	6.59G	7.66M	0.12
Garbage Collection	536.47M	172,968	0.03
Total	7.13G	7.84M	0.11

mpegaudio			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	1009	0.49
Translation + Execution	8.94G	16.31M	0.18
Garbage Collection	26.11M	79,887	0.31
Total	8.97G	16.39M	0.18

mtrt			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	1011	0.49
Translation + Execution	1.29G	15.73M	1.21
Garbage Collection	2.56G	13.15M	0.51
Total	3.86G	28.89M	0.75

jack			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	1011	0.49
Translation + Execution	4.68G	61.34M	1.31
Garbage Collection	952.15M	2.56M	0.27
Total	5.63G	63.91M	1.14

Table 4.3 Instruction Cache Performance for the s100 data set with the Latte VM in the JIT mode of execution.

4.4 Data Cache Performance – Read Accesses

4.4.1 Interpreted Mode of Execution (< Latte, int, dr, 1>)

Table 4.4 (< Latte, int, dr, 1>) shows the performance numbers obtained for data cache reads in the interpreted mode of execution. One of the notable points is the fact that there is a very high miss rate seen in the case of the data read misses in the

garbage collection phase. It has the potential to cause the net data cache performance to deteriorate when its contribution is not negligible and this deterioration of data cache performance is indeed the case. 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). In terms of absolute misses, mtrt shows a profound increase due to the influence of the garbage collection phase (an increase in overall miss rate from 4.09% to 4.23% translates to 18 million additional misses). This high miss rate can probably be attributed to frequent conflict and capacity misses between the data accessed by the garbage collector (data structures and objects on the heap) and the method bytecodes that are interpreted in the execution phase.

When compared to the data reads for the s1 data set (Appendix A2), the data reads in the s100 phase for the interpretation phase are considerably lower. This can be attributed to the fact that there is greater method reuse in the latter case and the most frequently used methods are cached. This effect is most pronounced in the case of the db and jess benchmarks (which have the greatest increase in method reuse), where the decrease in miss-rates is 3.2% (8.1% to 4.9%) and 1.8% (5.5% to 3.8%) respectively.

compress			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	97.89K	9029	9.22
Interpretation	45.10G	919.52M	2.03
Garbage Collection	3.02M	486.7K	16.14
Total	45.11G	920.05M	2.04

jess			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	97891	9037	9.23
Interpretation	8.51G	419.06M	4.92
Garbage Collection	141.12M	20.79M	14.74
Total	8.65G	439.89M	5.08

db			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	97891	9039	9.23
Interpretation	15.23G	558.32M	3.67
Garbage Collection	56.57M	109.68K	19.39
Total	15.29G	569.32M	3.72

mpegaudio			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	97891	9029	9.22
Interpretation	39.21M	371.41M	0.94
Garbage Collection	886.42K	34,369	3.88
Total	39.22G	371.49M	0.95

mtrt			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	97891	9034	9.23
Interpretation	4.21G	172.15M	4.09
Garbage Collection	7.91G	340.86M	4.31
Total	12.11G	513.04M	4.23

jack			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	97891	9033	9.23
Interpretation	11.81G	316.66M	2.68
Garbage Collection	82.19M	6.77M	8.24
Total	11.90G	323.46M	2.72

Table 4.4 Data Cache (Read) Performance for the s100 data set with the Latte VM in the interpreted mode of execution.

4.4.2 JIT Mode of Execution (< Latte, jit, dr, 1>)

Table 4.5 (<Latte, jit, dr, 1>) presents the performance numbers for the JIT compiled mode. Comparing the results to the interpreted mode results seen in Table 4.4 (<Latte, int, dr, 1>), we observe that there is a drastic reduction in the number of read accesses. As seen previously, 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. This is most clearly demonstrated in the long running benchmarks, compress and mpeg, which show 85-90% reductions in data-read accesses. 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 stream of MPEG-3 audio streams are decoded using the same algorithm (miss rate of 5.75% in JIT and 0.94% in interpreter).

Garbage collection activity shows a marginal increase in the JIT mode; this is something we are unable to explain. The garbage collector performs very poorly, and in most of the benchmarks, causes an increase in the overall miss rate by almost 2% in some cases (mtrt). This can be attributed to the fact that there is considerable amount of interference with the code installed by the compiler and this manifests itself wherever the contribution of the garbage collector is substantial as is the case in mtrt. When compared to the interpreted mode, there is no clear trend exhibited in the absolute misses as well as the miss rates and may be considered to be highly program dependent.

compress			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	5578	11.95
Translation + Execution	2.51G	220.13M	8.78
Garbage Collection	6.08M	790,016	12.99
Total	2.52G	220.95M	8.79

jess			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	5578	11.94
Translation + Execution	831.40M	90.74M	10.92
Garbage Collection	152.96M	19.49M	12.74
Total	984.64M	110.26M	11.19

db			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	5578	11.94
Translation + Execution	1.39G	262.15M	18.83
Garbage Collection	58.96M	11.06M	18.76
Total	1.45G	273.24M	18.82

mpegaudio			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	5578	11.94
Translation + Execution	1.90G	109.60M	5.75
Garbage Collection	3.56M	237,477	6.66
Total	1.91G	109.87M	5.75

mtrt			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	5578	11.94
Translation + Execution	322.65M	44.79M	13.83
Garbage Collection	529.17M	90.17M	17.04
Total	852.09M	134.99M	15.84

jack			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	5578	11.95
Translation + Execution	663.38M	62.87M	9.48
Garbage Collection	88.12M	6.92M	7.86
Total	751.78M	69.83M	9.29

Table 4.5 Data Cache (Read) Performance for the s100 data set with the Latte VM in the JIT mode of execution.

4.5 Data Cache Performance – Write Accesses

4.5.1 Interpreted Mode of Execution (Latte, int, dw, 1>)

Table 4.6 (<Latte, int, dw, 1>) shows the results for the write accesses seen with the interpreted mode of execution. Overall, miss rates here range from 1.51% for mpeg to 9.53% for the case of jess. An examination of the absolute misses and the miss rate for the execution individual phases reveals that garbage collection miss rates are extremely high (ranging from 50% to 74%) in all the benchmarks save mtrt, where the miss rate is about 4.27%. 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 9.53% compared to 5.68% in the execution phase) and jack (overall miss rate of 4.12% compared to 2.34% in the execution phase). In all the other benchmarks, the deterioration in performance is not much due to the preponderance of the execution phase. One can attribute this to the fact that all write misses in the garbage collection phase are brought about by deallocation of objects on the heap and updating of data structures, a broad percentage of whose accesses would be cold misses. In non-generational garbage collectors as in Latte, most misses due to allocation are write misses, and fetch useless garbage that will immediately be overwritten by the initializing writes that create objects

compress			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	27628	11065	40.05
Interpretation	13.29G	822.28M	6.18
Garbage Collection	2.92M	2.18M	74.56
Total	13.30G	824.49M	6.19

jess			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	27628	11089	40.13
Interpretation	2.43G	138.42M	5.68
Garbage Collection	167.7M	109.54M	65.32
Total	2.60G	248M	9.53

db			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	27628	11092	40.14
Interpretation	4.18G	216.81M	5.18
Garbage Collection	564.9M	34.22M	60.59
Total	4.24G	251.06M	5.92

mpegaudio			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	27628	11052	40
Interpretation	11.83G	179.02M	1.51
Garbage Collection	486.85K	242.92K	49.89
Total	11.84G	179.29M	1.51

mtrt			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	27628	11084	40.12
Interpretation	1.32G	32.39M	2.44
Garbage Collection	2.61G	111.47M	4.27
Total	3.93G	143.89M	3.66

jack			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	27628	11097	40.16
Interpretation	3.54G	8.29M	2.34
Garbage Collection	112.11M	67.80M	57.70
Total	3.66G	150.80M	4.12

Table 4.6 Data Cache (Write) Performance for the s100 data set with the Latte VM in the interpreted mode of execution.

4.5.2 JIT Mode of Execution (<Latte, jit, dw, 1>)

One of the most significant results of these sets of experiments has been to conclusively prove the effect of "double-caching". This refers to the effect that is seen when the JIT compiler translates method bytecodes into native code for the very first time and has to incur compulsory misses when it is installed in the data cache. In addition, compulsory instruction cache misses will be incurred when the native code is brought into the instruction cache for execution. 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 translates into 25 native instructions on an average [10].

In terms of actual results, we observe that the miss rates in the execution phase range from 12.5% for db to about 69.8% for jess. The overall effect is compounded by the poor performance in the garbage collection phase, where the miss rates range from 39% to about 68%. 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 4.4.2 with <

Latte, jit, dr, 1>). Also, we noted the poorer performance of the instruction cache in the JIT mode when compared to the interpreted mode in 4.3.2 (<Latte, int, dw, 1>). This particular effect 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. Table 4.7 (<Latte, jit, dw, 1>) provides the complete results of cache performance with the JIT mode.

We also note that when compared to the s1 data set, the overall miss rate is much lower. It ranges from 26.3% (compress) to 59% (jess) in the case of the s1 data set. This decrease is attributed to increased method reuse; the cost of translation and installation of native code is amortized over the frequent invocation of the methods.

compress			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	7292	55.21
Translation + Execution	571.03M	111.97M	19.61
Garbage Collection	3.75M	2.52M	67.37
Total	574.86M	114.52M	19.92

jess			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	7292	55.21
Translation + Execution	129.99M	90.81M	69.86
Garbage Collection	168.72M	107.79M	63.88
Total	298.79M	198.64M	66.48

db			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	7292	55.21
Translation + Execution	183.55M	23.02M	12.54
Garbage Collection	57.09M	34.02M	59.59
Total	240.72M	57.07M	23.71

mpegaudio			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	7292	55.21
Translation + Execution	484.37M	154.52M	31.91
Garbage Collection	1.36M	681,804	50.04
Total	485.81M	155.23M	31.95

mtrt			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	7292	55.21
Translation + Execution	59.62M	23.44M	39.31
Garbage Collection	216.59M	83.79M	38.69
Total	276.28M	107.26M	38.82

jack			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	7292	55.21
Translation + Execution	131.23M	51.45M	39.21
Garbage Collection	112.97M	66.64M	58.99
Total	244.28M	118.12M	48.35

Table 4.7 Data Cache (Write) Performance for the s100 data set with the Latte VM in the JIT mode of execution.

4.6 Cache Performance with Increased Cache Sizes

We experimented with larger cache sizes to examine if the poor data cache performance in the JIT compiled mode of the Latte VM was a result of mere capacity misses. For completeness, we also studied the trends for the instruction cache performance and data cache reads. The results obtained (with the s100 data set) for

the instruction cache performance, data reads and data writes for the 3 cache configurations are presented in this section. 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. Absolute misses and overall statistics for the s100 data set are tabulated in Appendices C and D for <Latte, jit, ins/dr/dw, 2>) and <Latte, jit, ins/dr/dw, 3>) respectively.

For the case of the instruction cache, the miss rate is greatly reduced in the execution phase with an increase in cache size for jess, mtrt and jack, as we move from <Latte, jit, ins, 1>) to <Latte, jit, ins, 2>). But the increase is not so profound when we move to <Latte, jit, ins, 3>). Figure 4.1 plots the execution phase miss rates for each of the benchmarks with the 3 different cache configurations.

Similar behavior is exhibited for the garbage collection phase; wherever garbage collection contributes significantly to instruction cache accesses (jess, mtrt and jack), we see considerable improvement. Figure 4.2 plots the miss rates in the garbage collection phase for each of the benchmarks with the 3 different cache configurations.

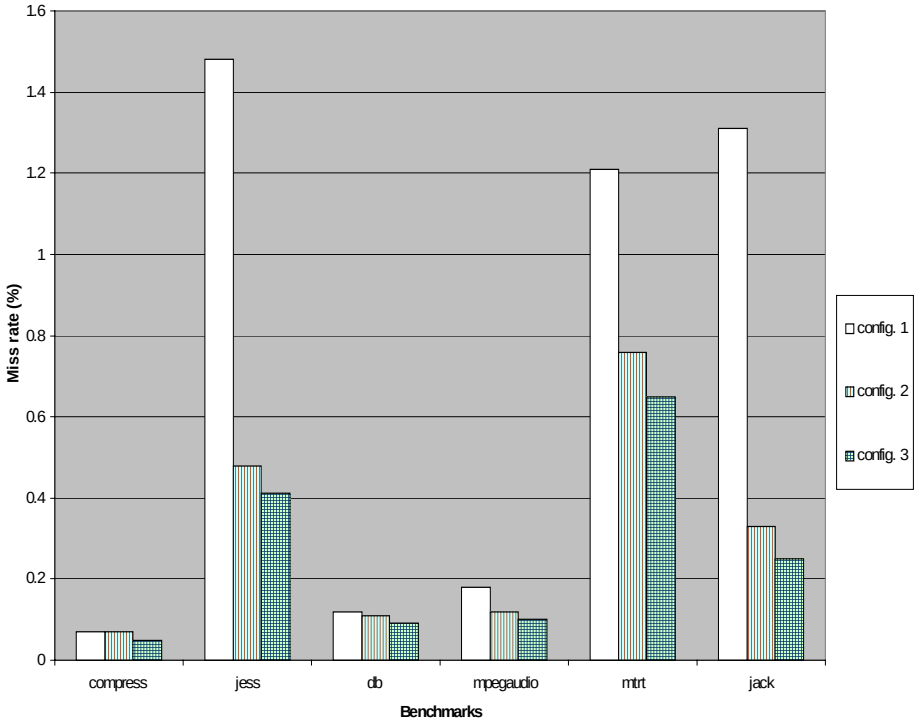

Figure 4.1 Instruction Cache - Miss Rates for Execution Phase (Latte VM)

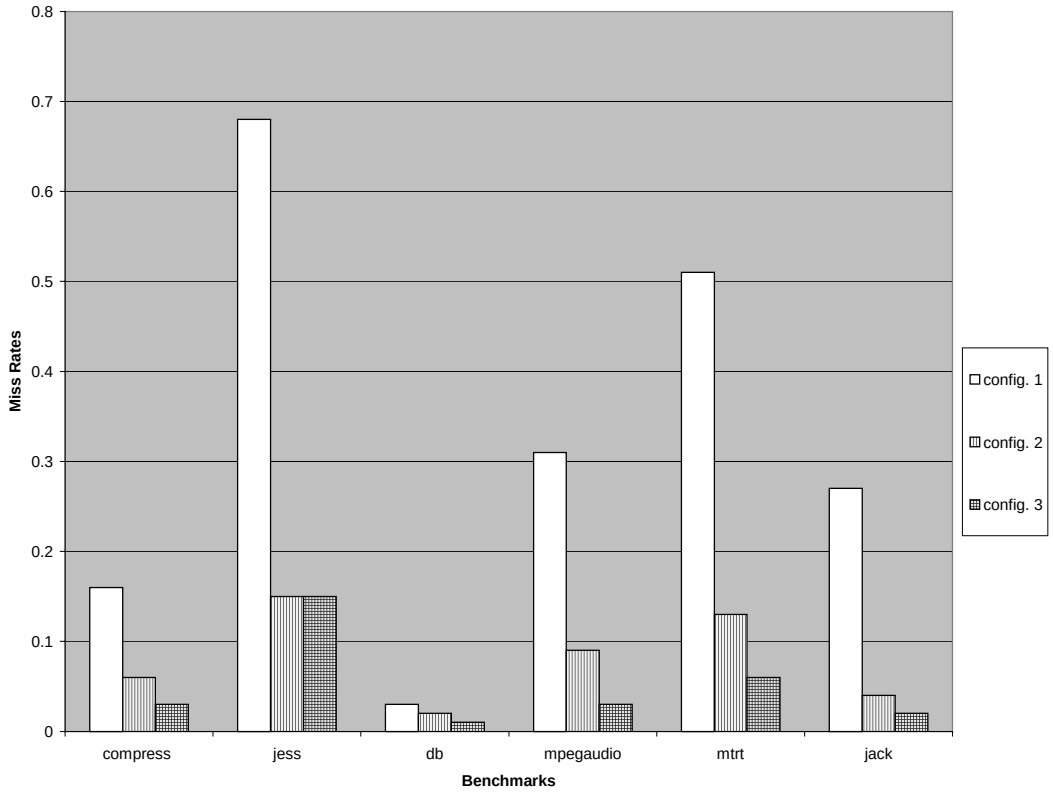

Figure 4.2 Instruction Cache - Miss Rates for Garbage Collection Phase (Latte VM)

With data cache reads, there is considerable performance gain. For the execution phase, there is considerable drop in miss rates for all the benchmarks as we move from <Latte, jit, dr, 1> to <Latte, jit, dr, 2>. Performance gains are still quite good as we move to <Latte, jit, dr, 3>, where compress shows the greatest reduction in miss rate. Figure 4.3 shows a comparison of the miss rates for each of the configurations. The garbage collection phase also benefits from increased cache sizes, especially in mtrt (where garbage collection is a major component). Here we see a reduction in miss rate from 17.04% in <Latte, jit, dr, 1> to 4.64% in <Latte, jit, dr,

2>. Figure 4.4 presents these results for all the benchmarks in each of the configurations.

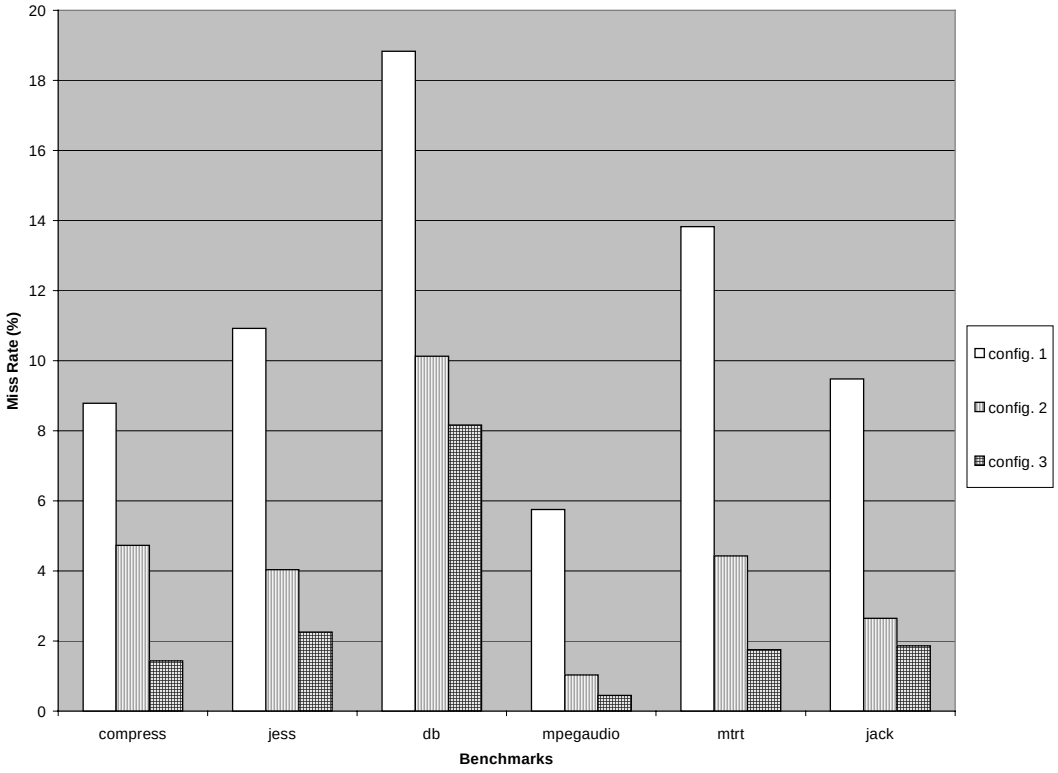

Figure 4.3 Data Cache Reads - Miss Rates for Execution Phase (Latte VM)

The scenario is quite different when one looks at the data writes. Referring to Figure 4.5, we find that the reduction in miss rate in the execution phase is not very substantial. The miss rates still range from 14.83% in the case of compress (19.63% in the <Latte, jit, dw, 1> and 17.78% in <Latte, jit, dw, 2>) to 55.3% in the case of jess (69.8% in the <Latte, jit, dw, 1>) and 57.6% in <Latte, jit, dw, 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) and the garbage collection phase (allocation of objects).

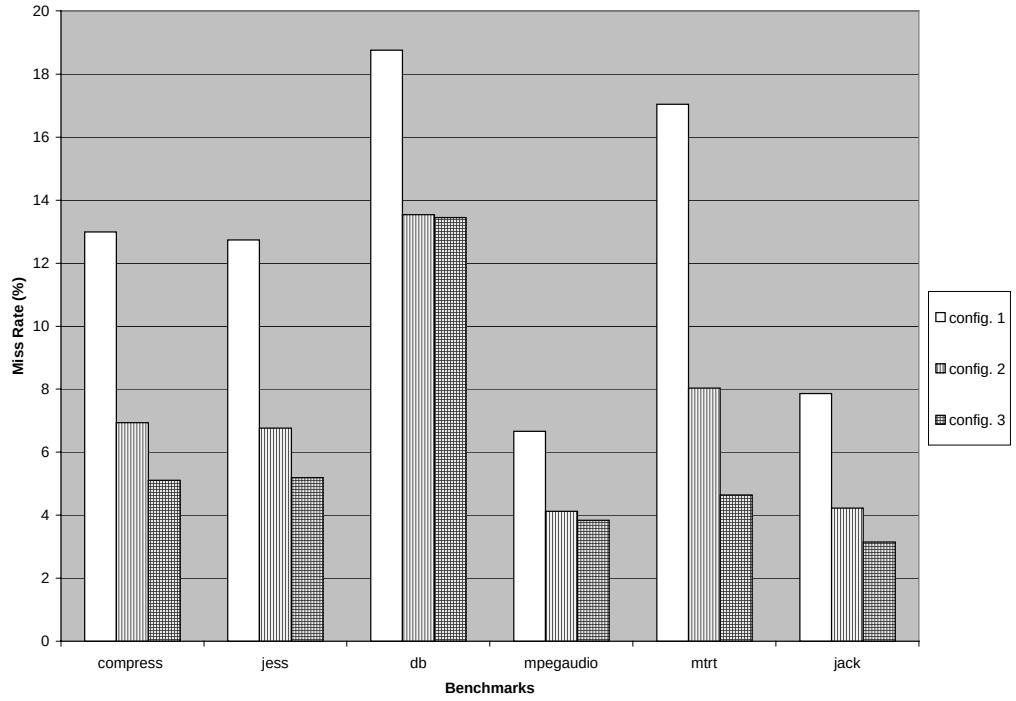

Figure 4.4 Data Cache Reads - Miss Rates in Garbage Collection Phase (Latte VM)

In the garbage collection phase, the performance is far worse. jess (where the garbage collector executes a number of data writes) has a miss rate of 57.9% in <Latte, jit, dw, 3>) down from a miss rate of 63.9% in <Latte, jit, dw, 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). Figure 4.6 presents these results.

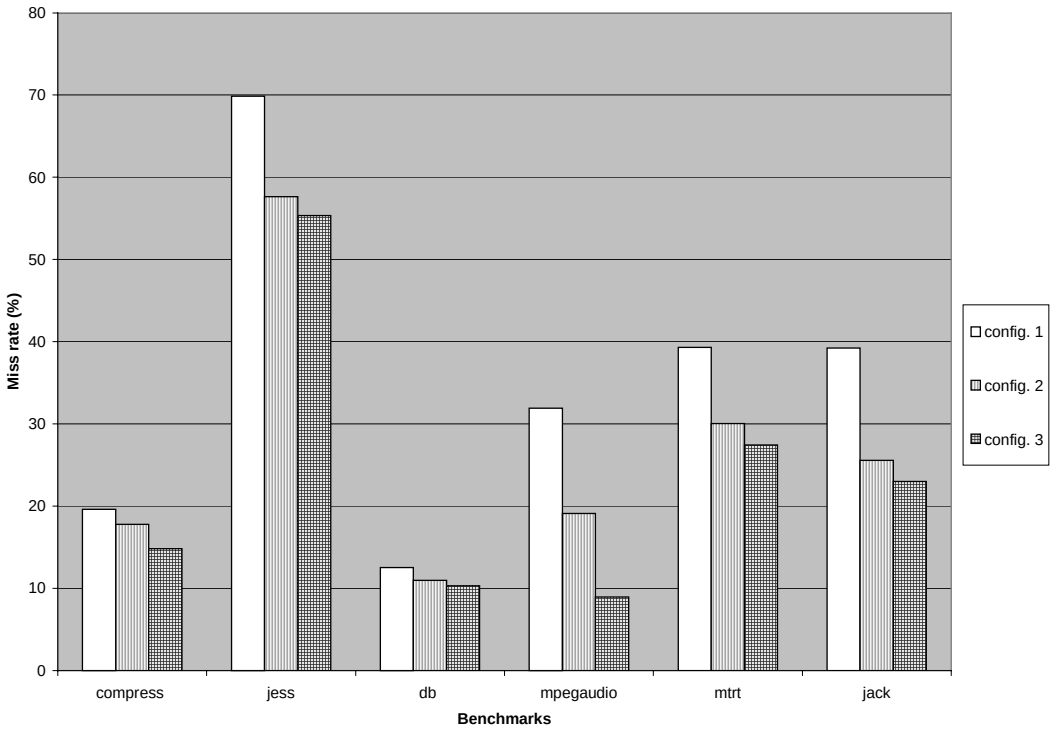

Figure 4.5 Data Cache Writes- Miss Rates for Execution Phase (Latte VM)

4.7 Cache Performance Results for the HotSpot Server VM

It was decided to compare the detailed results obtained with the Latte VM to the performance numbers obtained using Sun's HotSpot Client VM 1.3.1. This was done to confirm that the effects of poor data cache performance is not localized to traditional JVMs using JIT compiled modes of execution alone, and applies to newer JVMs too that use mixed-mode execution. We were not able to generate per-

phase results for the HotSpot VM because the VM would not build on the Solaris platform we were using. Hence, it was not possible to instrument the VM in order to obtain per phase results. In addition, the mtrt benchmark terminated with illegal memory access errors and we have not included it in our results. We studied the overall results with the same cache configurations that were used to study the per-phase behavior of the Latte VM, and our numbers indicate that the trends observed in the Latte VM are observed here too.

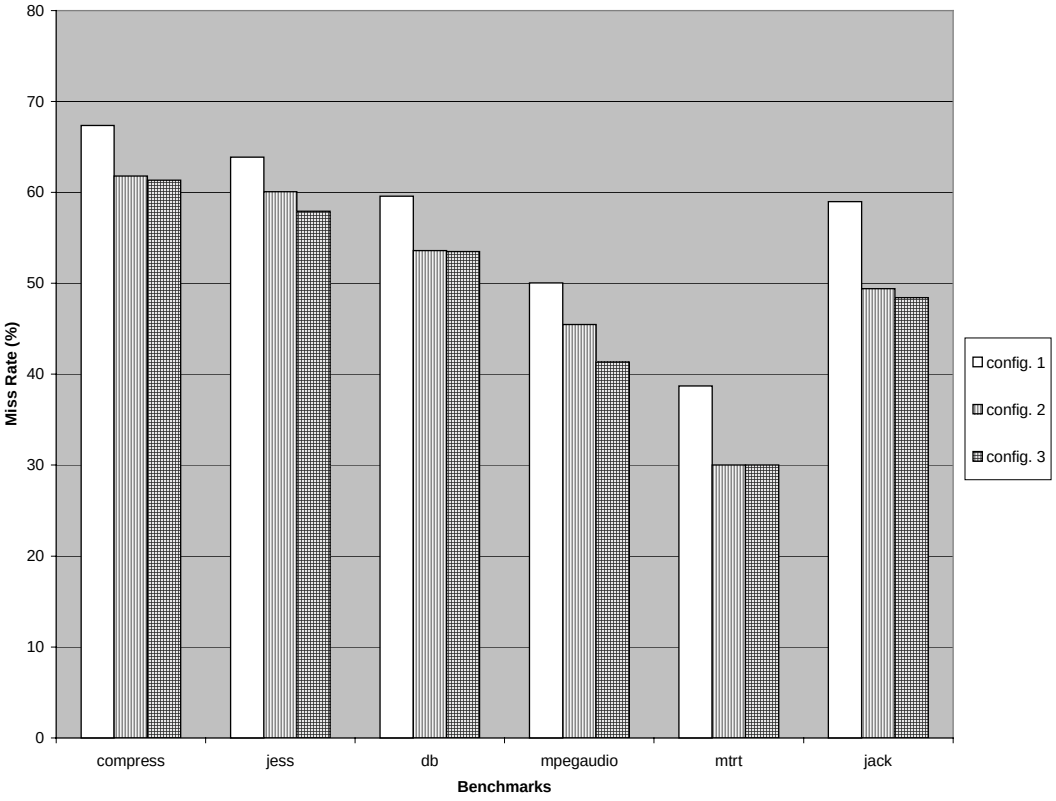

Figure 4.6 Data Cache Writes - Miss Rates in Garbage Collection Phase (Latte VM)

The instruction cache performance seen in the HotSpot Server VM is presented in Figure 4.7. The results seen here were very similar to what was seen with the Latte VM, though the absolute number of instruction and data cache accesses is much higher in the HotSpot VM. This high instruction miss rate can probably be attributed to overheads incurred in transforming from interpreted mode to the JIT mode and vice-versa when handling different methods. Continuous profiling of methods is also an overhead that cannot be neglected. There is considerable reduction in the instruction cache miss rates when we go from <HotSpot, mix, ins, 1> to <HotSpot, mix, ins, 3>; the largest improvements are seen in the case of jack (3.38% to 0.26%) and mpegaudio (from 0.97% to 0.04%). In general, improvements seen with increasing cache size are more prominent for the HotSpot VM and directly comparing miss rates for <HotSpot, mix, ins, 3>, we find that overall miss rates are lower for the HotSpot VM when compared to the Latte VM. Detailed results for Configurations 1,2 and 3 are presented in Appendix E.

A large improvement is seen for all the benchmarks with respect to data cache read misses when we go from <HotSpot, mix, dr, 1> to <HotSpot, mix, dr, 3>. Figure 4.8 presents the comparison results for data cache read-access misses with all the cache configurations. The Latte VM also showed considerable improvement when we increased the cache size. As we saw for the instruction cache, improvements seen with increasing cache size are more prominent for the HotSpot VM. For example, in compress, the miss rate with the Latte VM for data cache reads falls from 8.79% to

1.44% as we move from <HotSpot, mix, dr, 1> to <HotSpot, mix, dr, 3>, whereas it falls from 9.18% to 1.12% with the HotSpot VM, under the same conditions.

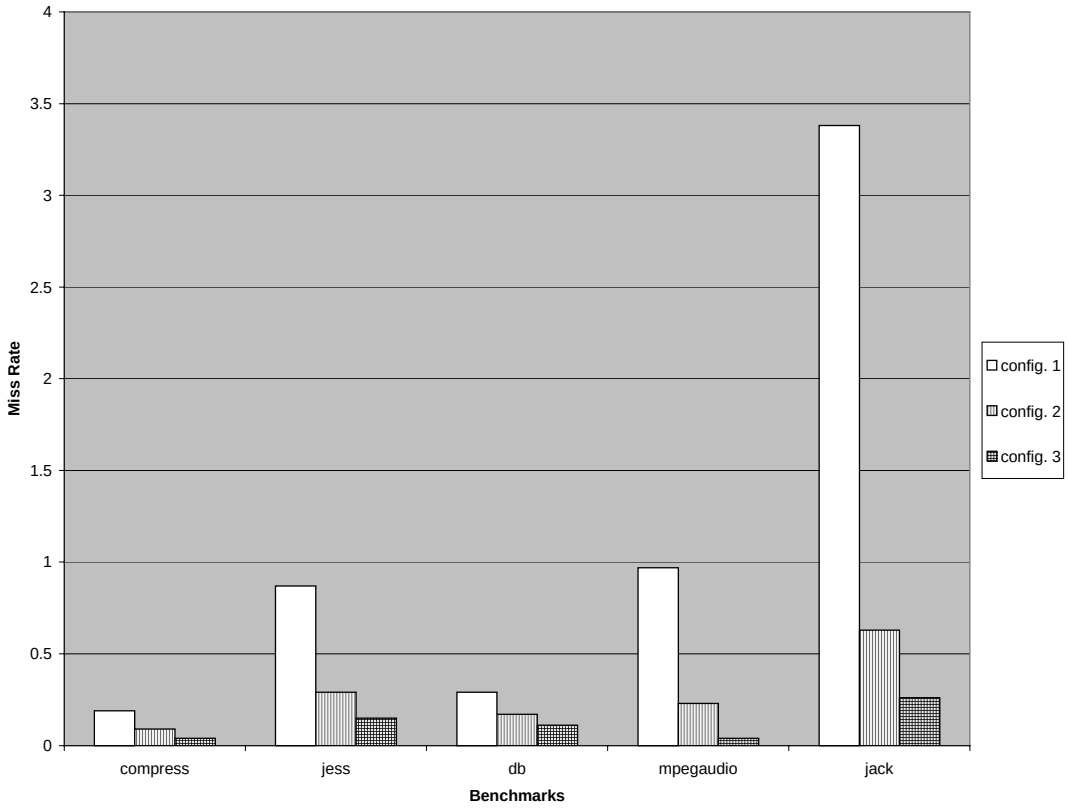

Figure 4.7 Instruction Cache – Miss Rates (HotSpot VM)

The data cache write-access miss rates are slightly higher in the case of the HotSpot VM. It shows a minimum of 35.27% for the case of compress (19.9% in <Latte, jit, dw, 3>) and a maximum of 65.6% for the case of jess (66.5 in <Latte, jit, dw, 1>) in <HotSpot, mix, dw, 3>. This shows that our conclusion which states that data cache write misses do not decrease substantially in JIT compiled execution engines are valid even if a policy of selective translation of methods is followed.

Thus, the penalty imposed by installation of code in the translation stage of the execution is not offset even with the use of an intelligent policy for method compilation.

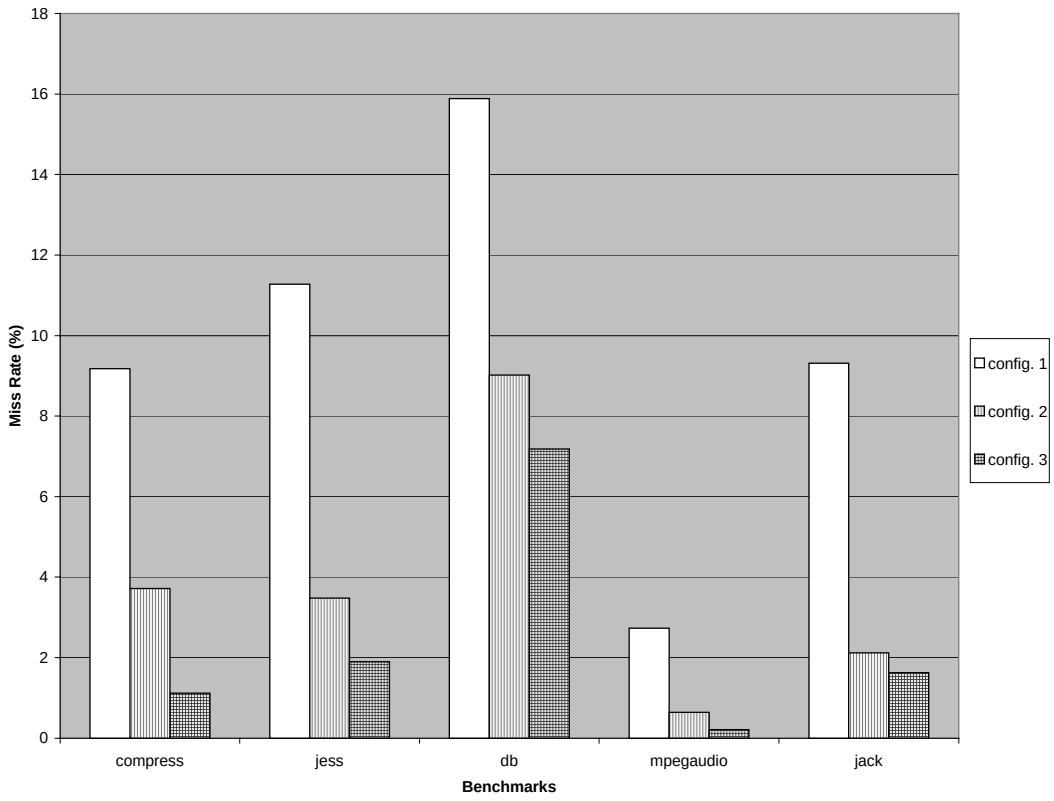

Figure 4.8 Data Cache Reads– Miss Rates (HotSpot VM)

Data cache write miss rates do not decrease to acceptable values with increased cache sizes; in the case of mpegaudio there is a reduction from 29.7% to 13.2% and in the case of db there is a reduction from 65.6% to 57.2% when we move from <HotSpot, mix, dw, 1> to <HotSpot, mix, dw, 3>. In fact, as we move from

<HotSpot, mix, dw, 2> to <HotSpot, mix, dw, 3> the reduction in miss rate is insignificant. The data cache miss rates seen even with Configuration 3's caches sizes are still very high and range from 13% to 58%, and conclusively prove that the performance penalty that we incur with on-the-fly compilation of methods cannot be done away with merely increasing cache sizes. Figure 4.9 provides the comparison charts for the 3 configurations.

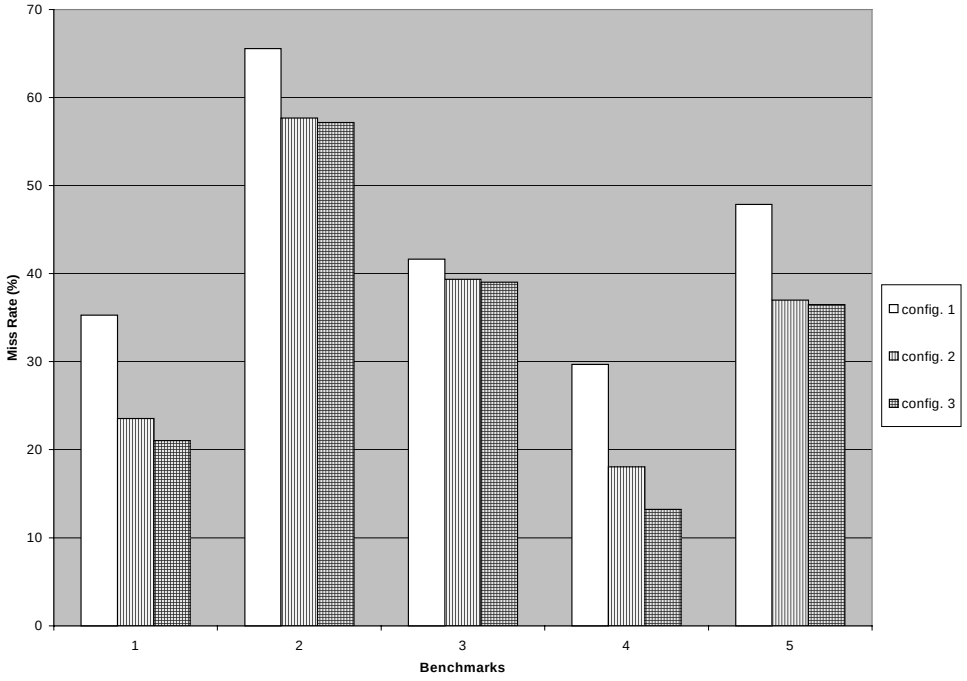

Figure 4.9 Data Cache Writes – Miss Rates (HotSpot VM)

5.Conclusion

Java's architecture paves the way for network-oriented software architectures that take full advantage of Java's support for network mobility of code and objects. 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 and their behavior compared to that of mixed-mode execution engines that use dynamic profiling to intelligently mix both the traditional execution modes.

The major findings from our research 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 compilation is always 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 is considerable interference between the GC 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. For the case of the s1 data set, the average is about 49%. Poor data cache performance is the result of compulsory misses resulting from the installation of translated code. The garbage collector also performs very poorly owing to conflict and capacity misses and results in even further deterioration when its contribution is substantial.
- With increased cache sizes, the instruction cache miss rates and data read miss rates are decreased substantially, but this 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.

- Intelligent translation of Java methods implemented by dynamic profiling in mixed-mode execution engines like HotSpot does not change the overall cache performance of the JVM.

Appendix A1

Instruction Cache Performance in Interpreted Mode – s1 data set (Configuration 1)

compress			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	208,888	1,157	0.55
Interpretation	10.98M	1.17M	0.01
Garbage Collection	7.33M	3,142	0.04
Total	10.99M	1.20M	0.01

jess			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	208,888	1,484	0.71
Interpretation	163.19M	2.32M	1.43
Garbage Collection	12.93M	94,719	0.73
Total	180.72M	2.48M	1.37

db			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	208,888	1,162	0.56
Interpretation	38.45M	172,298	0.45
Garbage Collection	6.06M	3,125	0.05
Total	49.44M	199,488	0.40

mpegaudio			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	208,888	1,484	0.71
Interpretation	1.32G	13.48M	1.02
Garbage Collection	6.62M	33,584	0.51
Total	1.33G	13.57M	1.02

mtrt			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	208,888	1,484	0.71
Interpretation	890.14M	5.26M	0.59
Garbage Collection	45.99M	135,713	0.29
Total	941.13M	5.45M	0.58

jack			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	208,888	1,484	0.71
Interpretation	2.31G	16.20M	0.70
Garbage Collection	59.15M	407,911	0.69
Total	2.37G	16.66M	0.70

Appendix A2

Data Cache (Reads) Performance in Interpreted Mode– s1 data set (Configuration 1)

compress			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	47,216	3,767	7.98
Interpretation	3.45G	8.31M	0.24
Garbage Collection	981,860	32,414	3.30
Total	3.46G	8.39M	0.24

jess			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	47,216	5,610	11.88
Interpretation	42.53M	3.29M	7.75
Garbage Collection	1.46M	58,849	4.04
Total	44.95M	3.43M	7.63

Db			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	47,216	3,770	7.99
Interpretation	9.58M	196,578	2.05
Garbage Collection	798,828	7,685	0.96
Total	11.38M	246,829	2.17

mpegaudio			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	47,216	5,609	11.88
Interpretation	399.49M	8.66M	2.17
Garbage Collection	886,515	36,979	4.17
Total	401.47M	8.78M	2.19

mtrt			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	47,216	5,610	11.88
Interpretation	270.85M	8.62M	3.18
Garbage Collection	4.77M	634,053	13.29
Total	276.61M	9.33M	3.37

jack			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	47,216	5,610	11.88
Interpretation	700.56M	18.48M	2.64
Garbage Collection	5.45M	347,533	6.37
Total	700.70M	18.91M	2.68

Appendix A3

Data Cache(Writes) Performance in Interpreted Mode–s1 data set (Configuration 1)

compress			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,403	4,448	33.18
Interpretation	1.02G	7.66M	0.75
Garbage Collection	600,105	313,075	52.17
Total	1.03G	8.01M	0.79

jess			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,403	7,201	53.72
Interpretation	11.58M	947,078	8.18
Garbage Collection	1.21M	722,713	59.80
Total	13.06M	1.73M	13.26

Db			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,403	4,427	33.03
Interpretation	2.67M	138,049	5.17
Garbage Collection	459,389	216,826	47.15
Total	3.39M	396,027	11.66

mpegaudio			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,403	7,232	53.96
Interpretation	120.16M	2.00M	1.66
Garbage Collection	486,840	242,897	49.89
Total	120.94M	2.31M	1.91

mtrt			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,403	7,201	53.72
Interpretation	82.61M	3.12M	3.77
Garbage Collection	5.22M	3.43M	65.68
Total	88.09M	6.60M	7.49

jack			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,403	7,201	53.72
Interpretation	209.81M	5.05M	2.41
Garbage Collection	6.96M	4.20M	60.35
Total	217.04M	9.31M	4.29

Appendix B1

Instruction Cache Performance for JIT– s1 data set (Configuration 1)

compress			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	1011	0.49
Compilation + Translation	1.02G	5.94M	0.58
Garbage Collection	21.12M	53,240	0.25
Total	1.05G	6.01M	0.57

jess			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	1011	0.49
Compilation + Translation	604.56M	11.81M	1.95
Garbage Collection	35.61M	134,602	0.38
Total	641.58M	1.95M	1.86

Db			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	1011	0.49
Compilation + Translation	343.32M	6.16M	1.79
Garbage Collection	20.27M	57,475	0.28
Total	364.98M	6.22M	1.71

mpegaudio			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	1011	0.49
Compilation + Translation	603.47M	10.24M	1.70
Garbage Collection	26.05M	79,365	0.31
Total	630.91M	10.33M	1.64

mtrt			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	1011	0.49
Compilation + Translation	603.88M	10.72M	1.78
Garbage Collection	64.54M	107,739	0.17
Total	669.81M	10.83M	1.62

jack			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	1011	0.49
Compilation + Translation	819.37M	14.59M	1.78
Garbage Collection	80.88M	231,918	0.29
Total	901.64M	14.83M	1.64

Appendix B2

Data Cache (Reads) Performance for JIT– s1 data set (Configuration 1)

compress			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	5,582	11.95
Compilation + Translation	242.73M	19.49M	8.03
Garbage Collection	2.93M	197,447	6.73
Total	245.93M	19.72M	8.02

jess			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	5,579	11.94
Compilation + Translation	110.40M	10.44M	9.46
Garbage Collection	4.70M	231,441	4.92
Total	115.38M	10.71M	9.28

Db			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	5,579	11.94
Compilation + Translation	61.27M	5.43M	8.87
Garbage Collection	2.80M	153,597	5.49
Total	64.34M	5.62M	8.73

mpegaudio			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	5,582	11.95
Compilation + Translation	108.35M	9.76M	9.01
Garbage Collection	3.55M	236,097	6.64
Total	112.18M	10.03M	8.95

mtrt			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	5,579	11.94
Compilation + Translation	114.89M	10.38M	9.04
Garbage Collection	7.42M	780,105	10.52
Total	122.57M	11.20M	9.14

jack			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	5,579	11.94
Compilation + Translation	135.51M	13.57M	10.01
Garbage Collection	8.59M	530,932	6.18
Total	144.38M	14.14M	9.79

Appendix B3

Data Cache (Writes) Performance for JIT– s1 data set (Configuration 1)

compress			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	7,293	55.22
Compilation + Translation	55.28M	14.31M	25.89
Garbage Collection	1.18M	608,256	51.49
Total	56.54M	14.95M	26.44

jess			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	7,262	54.98
Compilation + Translation	22.33M	13.67M	61.20
Garbage Collection	2.11M	1.08M	50.99
Total	24.53M	14.78M	60.23

db			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	7,262	54.98
Compilation + Translation	12.75M	6.98M	54.77
Garbage Collection	1.06M	509,339	47.99
Total	13.89M	7.52M	54.13

mpegaudio			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	7,293	55.22
Compilation + Translation	23.26M	11.04M	47.48
Garbage Collection	1.36M	681,184	50.07
Total	24.70M	11.75M	47.58

mtrt			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	7,262	54.98
Compilation + Translation	24.23M	13.43M	55.41
Garbage Collection	5.97M	3.74M	62.62
Total	30.28M	17.19M	56.77

jack			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	7,262	54.98
Compilation + Translation	27.61M	15.07M	54.59
Garbage Collection	7.81M	4.47M	57.21
Total	35.50M	19.57M	55.12

Appendix C1

Instruction Cache Performance for JIT– s100 data set (Configuration 2)

compress			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	800	0.39
Compilation + Translation	9.55G	6.92M	0.07
Garbage Collection	44.95M	28,509	0.06
Total	9.60G	6.95M	0.07

jess			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	800	0.39
Compilation + Translation	3.97G	19.34M	0.48
Garbage Collection	1.44G	2.22M	0.15
Total	5.42G	21.58M	0.39

db			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	795	0.39
Compilation + Translation	6.59G	6.99M	0.11
Garbage Collection	536.49M	82,977	0.02
Total	7.13G	7.08M	0.10

mpegaudio			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	791	0.38
Compilation + Translation	8.95G	10.30M	0.12
Garbage Collection	26.11M	24,546	0.09
Total	8.97G	10.33M	0.12

mtrt			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	800	0.39
Compilation + Translation	1.29G	9.81M	0.76
Garbage Collection	2.56G	3.37M	0.13
Total	3.86G	13.19M	0.34

jack			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	800	0.39
Compilation + Translation	4.67G	15.21M	0.33
Garbage Collection	952.15M	331,439	0.04
Total	5.63G	15.54M	0.28

Appendix C2

Data Cache (Reads) Performance for JIT– s100 data set (Configuration 2)

compress			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	3,789	8.11
Compilation + Translation	2.50G	11.87M	4.73
Garbage Collection	6.08M	421,684	6.94
Total	2.51G	119.12M	4.74

jess			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	3,784	8.10
Compilation + Translation	831.26M	33.52M	4.03
Garbage Collection	152.96M	10.35M	6.77
Total	984.49M	43.89M	4.46

db			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	3,789	8.11
Compilation + Translation	1.39G	141.03M	10.13
Garbage Collection	58.97M	7.99M	13.54
Total	1.45G	149.04M	10.27

mpegaudio			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	3,789	8.11
Compilation + Translation	1.90G	19.56M	1.03
Garbage Collection	3.56M	147,282	4.13
Total	1.91G	19.72M	1.03

mtrt			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	3,784	8.10
Compilation + Translation	322.65M	14.28M	4.43
Garbage Collection	529.17M	42.54M	8.04
Total	852.09M	56.84M	6.67

jack			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	3,784	8.10
Compilation + Translation	663.17M	17.50M	2.64
Garbage Collection	88.12M	3.72M	4.23
Total	751.56M	21.24M	2.83

Appendix C3

Data Cache (Writes) Performance for JIT– s100 data set (Configuration 2)

compress			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	4,514	34.18
Compilation + Translation	571.03M	101.52M	17.78
Garbage Collection	3.75M	2.31M	61.81
Total	574.86M	103.86M	18.07

jess			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	4,517	34.20
Compilation + Translation	129.96M	74.92M	57.64
Garbage Collection	168.72M	101.33M	60.06
Total	298.76M	176.26M	58.99

db			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	4,491	34.0
Compilation + Translation	183.55M	201.3M	10.97
Garbage Collection	57.09M	30.59M	53.59
Total	240.72M	50.74M	21.08

mpegaudio			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	4,514	34.17
Compilation + Translation	484.37M	92.45M	19.09
Garbage Collection	1.36M	619,160	45.45
Total	485.81M	93.09M	19.16

mtrt			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	4,517	34.20
Compilation + Translation	59.61M	17.91M	30.03
Garbage Collection	216.59M	65.51M	30.24
Total	276.28M	83.44M	30.20

jack			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	4,517	34.20
Compilation + Translation	131.13M	33.52M	25.56
Garbage Collection	112.97M	55.79M	49.39
Total	244.18	89.34M	36.59

Appendix D1

Instruction Cache Performance for JIT– s100 data set (Configuration 3)

compress			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	699	0.34
Compilation + Translation	9.56G	5.27M	0.05
Garbage Collection	44.94M	15,373	0.03
Total	9.60G	5.29M	0.06

jess			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	699	0.34
Compilation + Translation	3.98G	16.38M	0.41
Garbage Collection	1.44G	2.14M	0.15
Total	5.42G	18.53M	0.34

db			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	699	0.34
Compilation + Translation	6.59G	5.79M	0.09
Garbage Collection	536.47M	45,358	0.01
Total	7.13G	5.84M	0.08

mpegaudio			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	694	0.34
Compilation + Translation	894.79M	9.02M	0.10
Garbage Collection	26.09M	7,820	0.03
Total	8.97G	9.03M	0.10

mtrt			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	699	0.34
Compilation + Translation	1.29G	8.42M	0.65
Garbage Collection	2.56G	1.52M	0.06
Total	3.86G	9.94M	0.26

jack			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	204,255	699	0.34
Compilation + Translation	4.67G	11.48M	0.25
Garbage Collection	952.15M	228,302	0.02
Total	5.63G	11.71M	0.21

Appendix D2

Data Cache (Reads) Performance for JIT– s100 data set (Configuration 3)

compress			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	3,601	7.71
Compilation + Translation	2.50G	35.82M	1.43
Garbage Collection	6.08M	310,433	5.11
Total	2.51G	36.15M	1.44

jess			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	3,600	7.71
Compilation + Translation	831.26M	18.69M	2.25
Garbage Collection	152.96M	7.94M	5.19
Total	984.49M	26.64M	2.71

db			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	3,600	7.71
Compilation + Translation	1.39G	113.62M	8.16
Garbage Collection	58.96M	7.93M	13.45
Total	1.45G	121.56M	8.37

mpegaudio			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	3,601	7.71
Compilation + Translation	1.90G	8.30M	0.44
Garbage Collection	3.56M	136,762	3.84
Total	1.91G	8.45G	0.44

mtrt			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	3,601	7.71
Compilation + Translation	322.50M	5.63M	1.75
Garbage Collection	529.17M	24.55M	4.64
Total	851.95M	30.20M	3.55

jack			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	46,696	3,601	7.71
Compilation + Translation	663.42M	12.32M	1.86
Garbage Collection	88.12M	2.78M	3.15
Total	751.81M	15.11M	2.01

Appendix D3

Data Cache (Writes) Performance for JIT– s100 data set (Configuration 3)

compress			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	4,330	32.79
Compilation + Translation	571.03M	84.69M	14.83
Garbage Collection	3.74M	2.30M	61.34
Total	574.86M	87M	15.13

jess			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	4,324	32.74
Compilation + Translation	129.96M	71.93M	55.34
Garbage Collection	168.72M	97.73M	57.92
Total	298.76M	169.67M	56.79

db			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	4,324	32.74
Compilation + Translation	183.54M	18.91M	10.30
Garbage Collection	57.09M	30.54M	53.49
Total	240.72M	49.47M	20.55

mpegaudio			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	4,330	32.78
Compilation + Translation	484.36M	43.32M	8.94
Garbage Collection	1.36M	563,437	41.36
Total	485.80M	43.90M	9.04

mtrt			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	4,330	32.78
Compilation + Translation	59.56M	16.33M	27.42
Garbage Collection	216.58M	65.02M	30.02
Total	276.23M	81.36M	29.46

jack			
Stage	**References**	**Misses**	**Percentage Misses**
Class Loading	13,207	4,330	32.78
Compilation + Translation	131.24M	30.19M	23.00
Garbage Collection	112.97M	54.68M	48.41
Total	224.29M	84.89M	34.75

Appendix E1

Cache Performance for HotSpot VM– s100 data set (Configuration 1)

compress			
Cache Operation	**References**	**Misses**	**Percentage Misses**
Instruction	13.23G	24.84M	0.19
Data Read	3.26G	299.74M	9.18
Data Write	1.09G	388.37M	35.27

jess			
Cache Operation	**References**	**Misses**	**Percentage Misses**
Instruction	5.17G	45.08M	0.87
Data Read	942.36M	106.29M	11.28
Data Write	314.15M	205.95M	65.55

Db			
Cache Operation	**References**	**Misses**	**Percentage Misses**
Instruction	7.72G	22.94M	0.29
Data Read	1.80G	286.48M	15.89
Data Write	674.18M	280.74M	41.64

mpegaudio			
Cache Operation	**References**	**Misses**	**Percentage Misses**
Instruction	16.11G	156.22M	0.97
Data Read	3.92G	106.88M	2.73
Data Write	1.24G	367.06M	29.67

jack			
Cache Operation	**References**	**Misses**	**Percentage Misses**
Instruction	4.07G	137.70M	3.38
Data Read	716.30M	66.70M	9.31
Data Write	357.99M	171.36M	47.87

Appendix E2

Cache Performance for HotSpot VM– s100 data set (Configuration 2)

compress			
Cache Operation	**References**	**Misses**	**Percentage Misses**
Instruction	13.24G	11.33M	0.09
Data Read	3.27G	121.35M	3.72
Data Write	1.09G	256.81M	23.53

jess			
Cache Operation	**References**	**Misses**	**Percentage Misses**
Instruction	5.16G	14.70M	0.29
Data Read	941.19M	32.78M	3.48
Data Write	313.79M	181.02M	57.68

Db			
Cache Operation	**References**	**Misses**	**Percentage Misses**
Instruction	7.72G	13.41M	0.17
Data Read	1.80G	162.47M	9.02
Data Write	673.91M	265.14M	39.34

mpegaudio			
Cache Operation	**References**	**Misses**	**Percentage Misses**
Instruction	16.09G	36.72M	0.23
Data Read	3.92G	25.26M	0.64
Data Write	1.24G	223.19M	18.06

jack			
Cache Operation	**References**	**Misses**	**Percentage Misses**
Instruction	4.07G	25.70M	0.63
Data Read	716.09M	15.15M	2.12
Data Write	357.91M	132.36M	36.98

Appendix E3

Cache Performance for HotSpot VM– s100 data set (Configuration 3)

compress			
Cache Operation	**References**	**Misses**	**Percentage Misses**
Instruction	13.18G	5.58M	0.04
Data Read	3.26G	36.49M	1.12
Data Write	1.09G	228.56M	21.02

jess			
Cache Operation	**References**	**Misses**	**Percentage Misses**
Instruction	5.18G	7.72M	0.15
Data Read	944.01M	17.96M	1.90
Data Write	315.09M	180.06M	57.15

Db			
Cache Operation	**References**	**Misses**	**Percentage Misses**
Instruction	7.72G	8.46M	0.11
Data Read	1.80G	129.46M	7.18
Data Write	673.83M	263.08M	39.04

mpegaudio			
Cache Operation	**References**	**Misses**	**Percentage Misses**
Instruction	16.08G	6.97M	0.04
Data Read	3.92G	8.28M	0.21
Data Write	1.23G	163.15M	13.21

jack			
Cache Operation	**References**	**Misses**	**Percentage Misses**
Instruction	4.07G	10.71M	0.26
Data Read	716.81M	11.65M	1.63
Data Write	358.01M	130.56M	36.47

References

1. F.Y.T. Lindholm, The Java Virtual Machine Specification, MA: Addison Wesley, 1997.
2. Cheng-Hsueh A. Hsieh, Marie Conte et al., *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. The Java HotSpot Performance Engine Architecture, http://java.sun.com/products/hotspot/whitepaper.html
5. Byung-Sun Yang, Soo-Mook 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. Robert F. Cmelik and David 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, 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 Lizy 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. Ramesh Radhakrishnan, N. Vijaykrishnan, Lizy 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. Ramesh Radhakrishnan, Juan Rubio, Lizy 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. A. Barisone, F. Belliotti et al., *Ultrasparc instruction level characterization of Java virtual machine workload*, 2nd Annual Workshop on Workload Characterization (WWC) for Computer System Design, pages 1--24. Kluwer Academic Publishers, 1999.
13. Cheng-Hsueh A. Hsieh, Wen-mei 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. T. Proebsting, G. Townsend et al., *Toba: Java for applications a way ahead of time* (WAT) compiler, Proceedings of the Third Conference on Object--Oriented Technologies and Systems, 1997.
15. Ole Agesen and David Detlefs, *Mixed Mode Bytecode Execution*, Sun Microsystems Inc., Technical Report TR-2000-87.
16. Timothy Cramer, Richard Friedman et al., *Compiling Java just in time*, IEEE Micro, May 1997.
17. Burke, M., et al., *The Jalapeno dynamic optimizing compiler for Java*, ACM Java Grande Conference, June 1999.
18. M. Arnold, S. Fink, D. Grove et al., *Adaptive Optimization in the Jalapeno JVM*, OOPSLA 2000, October 2000.
19. Tim Horel and Gary Lauterbach, *UltraSPARC-III: Designing Third-Generation 64-Bit Performance*, IEEE Micro, 1999.
20. Bill Venners, *Inside the Java 2 Virtual Machine*, McGraw Hill, 2000.
21. M. O'Connor and M. Tremblay, picoJava-I: The Java virtual machine in hardware, IEEE Micro, pp 45-53, Mar 1997.
22. B. Calder, D. Grunwald et al., *Quantifying Behavioral Differences Between C and C++ Programs*, Vol. 2, No. 4, pp 313-351, 1994.

23. Chandrakasan, Bowhill, Fox, *Design of High-Performance Microprocessor Circuits*, IEEE Press, 2000.
24. Tse-Yu Yeh and Yale Patt, *A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution*, IEEE Micro, 1992.

VITA

Anand Sunder Rajan was born in New Delhi, India on January 26, 1978, the son of Srinivasan Kannan and Vasanthi Kannan. After completing his work at National Public School, Bangalore, India, he entered the Birla Institute of Technology and Science in Pilani, India. He received the degrees of Bachelor of Engineering in Computer Science and Master of Science in Physics in June, 2000. In August, 2000, he entered The Graduate School at the University of Texas.

Permanent Address: 476A, 13th Cross, 28th Main,
1st Phase, J.P. Nagar, Bangalore 560078,
India.

This report was typed by the author.