Java Runtime Systems: Characterization and Architectural Implications [*]

R. Radhakrishnan†, N. Vijaykrishnan‡, L. K. John†, A. Sivasubramaniam‡, J. Rubio† and J. Sabarinathan†

†Laboratory for Computer Architecture
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712
{*radhakri,ljohn, jrubio, sabarina*}*@ece.utexas.edu*

‡220 Pond Lab
Dept. of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802.
{*vijay,anand*}*@cse.psu.edu*

Abstract

The Java Virtual Machine (JVM) is the corner stone of Java technology, and its efficiency in executing the portable Java bytecodes is crucial for the success of this technology. Interpretation, Just-In-Time (JIT) compilation, and hardware realization are well known solutions for a JVM, and previous research has proposed optimizations for each of these techniques. However, each technique has its pros and cons and may not be uniformly attractive for all hardware platforms. Instead, an understanding of the architectural implications of JVM implementations with real applications, can be crucial to the development of enabling technologies for efficient Java runtime system development on a wide range of platforms. Towards this goal, this paper examines architectural issues, from both the hardware and JVM implementation perspectives. The paper starts by identifying the important execution characteristics of Java applications from a bytecode perspective. It then explores the potential of a smart JIT compiler strategy that can dynamically interpret or compile based on associated costs, and investigates the CPU and cache architectural support that would benefit JVM implementations. We also study the available parallelism during the different execution modes using applications from the SPECjvm98 benchmarks.

At the bytecode level, it is observed that less than 45 out of the 256 bytecodes constitute 90% of the dynamic bytecode stream. Method sizes fall into a tri-nodal distribution with peaks of 1, 9 and 26 bytecodes across all benchmarks. The architectural issues explored in this study shows that when Java applications are executed with a JIT compiler, selective translation using good heuristics can improve performance, but the saving is only 10-15% at best. The instruction and data cache performance of Java applications are seen to be better than that of C/C++ applications, except in the case of data cache performance in the JIT mode. Write misses resulting from installation of JIT compiler output dominate the misses and deteriorate the data cache performance in JIT mode. A study on the available parallelism shows that Java programs executed using JIT compilers have parallelism comparable to C/C++ programs for small window sizes, but falls behind when the window size is increased. Java programs executed using the interpreter have very little parallelism due to the stack nature of the JVM instruction set, which is dominant in the interpreted execution mode. In addition, this work gives revealing insights and architectural proposals for designing an efficient Java runtime system.

[*]A preliminary version of this paper appeared in the International Conference on High Performance Computers and Architecture (HPCA-6)

1 Introduction

The Java Virtual Machine (JVM) [1] is the corner stone of Java technology epitomizing the "write-once run-anywhere" promise. It is expected that this enabling technology will make it a lot easier to develop portable software and standardized interfaces that span a spectrum of hardware platforms. The envisioned underlying platforms for this technology include powerful (resource-rich) servers, network-based and personal computers, together with resource-constrained environments such as hand-held devices, specialized hardware/embedded systems, and even household appliances. If this technology is to succeed, it is important that the JVM provides an efficient execution/runtime environment across these diverse hardware platforms. This paper examines different architectural issues, from both the hardware and JVM implementation perspectives, towards this goal.

Applications in Java are compiled into the byte code format to execute in the Java Virtual Machine (JVM). The core of the JVM implementation is the execution engine that executes the byte codes. This can be implemented in four different ways:

1. An interpreter is a software emulation of the virtual machine. It uses a loop which fetches, decodes and executes the byte codes until the program ends. Due to the software emulation the Java interpreter has an additional overhead and executs more instructions than just the bytecodes.

2. A Just-in-time (JIT) compiler is an execution model which tries to speed up the execution of interpreted programs. It compiles a Java method into native instructions on the fly and caches the native sequence. On future references to the same method, the cached native method can be executed directly without the need for interpretation. JIT compilers have been released by many vendors like IBM [2], Symantec [3] and Microsoft. Compiling during program execution, however, inhibits aggressive optimizations because compilation must only incur a small overhead. Another disadvantage of JIT compilers is the two to three times increase in the object code, which becomes critical in memory constrained embedded

systems. There are many ongoing projects in developing JIT compilers that aim to achieve C++ like performance, such as CACAO [4].

3. Off-line bytecode compilers can be classified into two types: those that generate native code and those that generate an intermediate language like C. Harissa [5], TowerJ [6] and Toba [7] are compilers that generate C code from byte codes. The choice of C as the target language permits the reuse of extensive compilation technology available in different platforms to generate the native code. In bytecode compilers that generate native code directly like NET [8] and Marmot [9], portability becomes extremely difficult. In general, only applications that operate in a homogeneous environment and those that undergo infrequent changes benefit from this type of execution.

4. A Java processor is an execution model that implements the JVM directly on silicon. It not only avoids the overhead of translation of the byte codes to another processor's native language, but also provides support for Java runtime features. It can be optimized to deliver much better performance than a general purpose processor for Java applications by providing special support for stack processing, multi-threading, garbage collection, object addressing and symbolic resolution. Java processors can be cost-effective to design and deploy in a wide range of embedded applications such as telephony and web tops. The picoJava [10] processor from Sun Microsystems is an example of a Java processor.

It is our belief that no one technique will be universally preferred/accepted over all platforms in the immediate future. Many previous studies [11, 12, 13, 10, 14] have focussed on enhancing each of the bytecode execution techniques. On the other hand, a three-pronged attack at optimizing the runtime system of all techniques would be even more valuable. Many of the proposals for improvements with one technique may be applicable to the others as well. For instance, an improvement in the synchronization mechanism could be useful for an interpreted or JIT mode of execution. Proposals to improve the locality behavior of Java execution could be useful in the design of Java processors as well as in the runtime environment on general purpose processors.

Finally, this three-pronged strategy can also help us design environments that efficiently and seamlessly combine the different techniques wherever possible.

A first step towards this three-pronged approach is to gain an understanding of the execution characteristics of different Java runtime systems for real applications. Such a study can help us evaluate the pros and cons of the different runtime systems (helping us selectively use what works best in a given environment), isolate architectural and runtime bottlenecks in the execution to identify the scope for potential improvement, and derive design enhancements that can improve performance in a given setting. This study embarks on this ambitious goal, specifically trying to answer the following questions:

• Do the characteristics seen at the bytecode level favor any particular runtime implementation? How can we use the characteristics identified at the bytecode level to implement more efficient runtime implementations?

• Where does the time go in a JIT-based execution (i.e. in translation to native code, or in executing the translated code)? Can we use a hybrid JIT-interpreter technique that can do even better? If so, what is the best we can hope to save from such a hybrid technique?

• What are the execution characteristics when executing Java programs (using an interpreter or JIT compiler) on general-purpose CPU (such as the SPARC)? Are these different from those for traditional C/C++ programs? Based on such a study, can we suggest architectural support in the CPU (either general-purpose or a specialized Java processor) that can enhance Java executions?

To our knowledge, there has been no prior effort that has extensively studied all these issues in a unified framework for Java programs. This paper sets out to answer some of the above questions using applications drawn from the SPECjvm98 [15] benchmarks, available JVM implementations such as JDK 1.1.6 [16] and Kaffe VM 0.9.2 [17] and simulation/profiling tools on the Shade [18] environment. All the experiments have been conducted on Sun UltraSPARC machines running SunOS 5.6.

1.1 Related Work

Studies characterizing Java workloads and performance analysis of Java applications are becoming increasingly important and relevant as Java increases in popularity, both as a language and software development platform. A detailed characterization of the JVM workload for the UltraSparc platform was done in [19] by Barisone et al. The study included a bytecode profile of the SPECjvm98 benchmarks, characterizing the types of bytecodes present and its frequency distribution. In this paper, we start with such a study and extend it to characterize other metrics such as locality and method sizes as they impact the performance of the runtime environment very strongly. Barisone et al. use the profile information collected from the interpreter and JIT execution modes as an input to a mathematical model of a RISC architecture, to suggest architectural support for Java workloads. Our study uses a detailed superscalar processor simulator, and also includes studies on available parallelism to understand the support required in current and future wide-issue processors.

Romer et al. [20] studied the performance of interpreters and concluded that no special hardware support is needed for increased performance. Hsieh et al. [21] studied the cache and branch performance of interpreted Java code, C/C++ version of the Java code and native code generated by Caffine (a bytecode to native code compiler) [22]. They attribute the inefficient use of the microarchitectural resources by the interpreter as a significant performance penalty, and suggest that an offline bytecode to native code translator is a more efficient Java execution model. Our work differs from these studies in two important ways. First, we include a JIT compiler in this study which is the most commonly used execution model presently. Secondly, the benchmarks used in our study are large real world applications, while the above mentioned study uses microbenchmarks due to the unavailability of a Java benchmark suite at the time of their study. We see that the characteristics of the application used affects favor different execution modes, and therefore the choice of benchmarks used is important.

Other studies have explored possibilities of improving performance of the Java runtime system by understanding the bottlenecks in the runtime environment and ways to eliminate them. Some of these studies try to improve the performance through better synchronization mechanisms [23, 24, 25], more efficient garbage collection techniques [26], and understanding the memory referencing behavior of Java applications [27] etc. Improving the runtime system, tuning the architecture to better execute Java workloads and better compiler/interpreter performance are all equally important to achieve efficient performance for Java applications.

The rest of this paper is organized as follows. The next section gives details on the experimental platform. In Section 3 the bytecode characteristics of the SPECjvm98 are presented. Section 4 examines the relative performance of JIT and interpreter modes, and explores the benefits of a hybrid strategy. Section 5 investigates some of the questions raised earlier with respect to the CPU and cache architectures. Section 6 collates the implications and inferences that can be drawn from this study. Finally, Section 7 summarizes the contributions of this work and outlines directions for future research.

2 Experimental platform

We use the SPECjvm98 benchmark suite to study the architectural implications of a Java runtime environment. The SPECjvm98 benchmark suite consists of 7 Java programs which represent different classes of Java applications. The benchmark programs can be run using three different inputs, which are named as s100, s10 and s1. These problem sizes do not scale linearly, as the naming suggests. We use the s1 input set to present the results in this paper and the effects of larger data sets, s10 and s100 has also been investigated. The increased method reuse with larger data sets results in increased code locality, reduced time spent in compilation as compared to execution, and other such issues as can be expected. The benchmarks are run at the command line prompt, and do not include graphics, AWT (graphical interfaces) or networking. A description of the benchmarks is given in Table 1. All benchmarks except *mtrt* are single-threaded. Java is used to build applications that span a wide range, which includes applets at the lower end to server side applications on the high end. The observations cited in this paper hold for those subset of applications which are similar to the SPECjvm98 benchmarks when run with the dataset used in

benchmark	Description
compress	A popular LZW compression program
jess	NASA's popular CLIPS rule-based expert system
db	Data management software written by IBM
javac	The JDK Java compiler from Sun Microsystems
mpegaudio	Algorithm to decode an MPEG-3 audio stream
mtrt	Dual-threaded program that ray traces an image
jack	A parser-generator from Sun Microsystems

Table 1: Description of the SPECjvm98 Benchmarks

this study.

Two popular JVM implementations have been used in this study: the Sun JDK 1.1.6 [16] and Kaffe VM 0.9.2 [17]. Both these JVM implementations support the JIT and interpreted mode. Since the source code for the Kaffe VM compiler was available, we could instrument it to obtain the behavior of the translation routines for the JIT mode in detail. Some of the data presented in Sections 4 and 5 are obtained from the instrumented translate routines in Kaffee. The results using Sun's JDK are presented for the other sections and only differences, if any, from the KaffeVM environment are mentioned. The use of two runtime implementations also gives us more confidence in our results, filtering out any noise due to the implementation details.

To capture architectural interactions, we have obtained traces using the Shade binary instrumentation tool [18] while running the benchmarks under different execution modes. Our cache simulations use the *cachesim5* simulators available in the *Shade* suite, while branch predictors have been developed in-house. The instruction level parallelism studies are performed utilizing a cycle-accurate superscalar processor simulator. This simulator, can be configured to a variety of out-of-order multiple issue configurations with desired cache and branch predictors.

3 Characteristics at the bytecode level

We characterize bytecode instruction mix, bytecode locality, method locality etc in order to understand the benchmarks at the bytecode level. The first characteristic we examine is the bytecode

Bytecode	**Description**	Examples
Loads	loads data from local variables into stack	*dload_2* (load double from local variable 2)
Stores	the counterpart of loads	*astore_1* (store reference into local variable 1)
Stack	allows for push and pop of operands into the stack, as well as duplication and swap of data in the stack	*sipush* (push short) *dup* (duplicate top operand stack word)
Constant pool	allocation of elements in the constant pool Note that the deallocation is performed by the garbage collector, so it does not require dedicated instructions	*new* (create new object) *getfield* (fetch field from object) *ldc* (push item from constant pool)
ALU	arithmetic (both integer and floating point) and logic instructions	*fadd* (add float), *drem* (remainder double) *lshr* (arithmetic shift right long)
Branches	tests for a condition and changes the program counter based on it	*if_acmpeq* (branch if references are equal) *if_icmplt* (branch if less than)
Jump	non-unitary increments or decrements to the program counter	*goto* (branch always)
Method calls	differ from branches and jumps since it is also necessary to allocate a new frame to hold the stack of the method and deallocate for returns	*invokestatic* (invoke a class static method) *ireturn* (return int from method)

Table 2: Classification of Bytecodes

instruction mix of the JVM, which is a stack-oriented architecture. To simplify the discussion, we classify the instructions into different types based on their inherent functionality as shown in Table 2.

Table 3 shows the resulting instruction mix for the SPECjvm98 benchmark suite. The total bytecode count ranges from 2 million for *db* to approximately a billion for *compress*. Most of the benchmarks show similar distributions for the different instruction types. *Load* instructions outnumber the rest, accounting for 35.5% of the total number of bytecodes executed on the average. *Constant pool* and *method call* bytecodes come next with average frequencies of 21% and 11% respectively. From an architectural point of view, this implies that transferring data elements to and from the memory space allocated for local variables and the Java stack dominate. Comparing this with the benchmark 126.gcc from the SPEC CPU95 suite that has roughly 25% of memory access operations when run on a SPARC V.9 architecture, it can be seen that the JVM places greater stress on the memory system. Consequently, we expect that techniques such as instruction folding proposed in [28] for Java processors and instruction combining proposed in [29] for JIT compilers can improve the overall performance of Java applications.

Instruction Group	BENCHMARKS						
	compress	jess	db	javac	mpegaudio	mtrt	jack
Constant Pool	23.3%	21.6%	16.8%	14.6%	17.1%	20.69%	32.0%
Stack	8.8%	3.5%	7.7%	5.8%	7.1%	4.1%	13.5%
Load	34.3%	35.5%	37.8%	37.9%	44.2%	28.2%	30.9%
Store	10.6%	6.6%	8.0%	7.5%	8.3%	3.5%	2.1%
ALU	11.2%	6.1%	8.8%	12.8%	17.1%	7.8%	5.8%
Branch	6.1%	9.6%	10.2%	8.6%	3.4%	5.1%	11.0%
Jump	0.4%	1.1%	1.1%	1.3%	0.4%	0.8%	0.5%
Method Calls	5.4%	15.7%	9.2%	10.8%	2.5%	29.3%	4.1%
Total Bytecodes	954990234	8126332	2035798	5958654	115748387	50683565	175740325

Table 3: Dynamic Instruction Mix at the Bytecode level

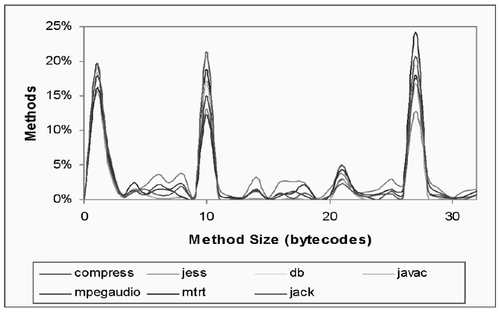


Figure 1: Dynamic Method Size

The second characteristic we examine is the dynamic size of a method[1]. Invoking methods in Java is expensive, as it requires the setting up of an execution environment and a new stack for each new method [1]. Figure 1 shows the method sizes for the different benchmarks. A tri-nodal distribution is observed, where most of the methods are either 1, 9 or 26 bytecodes long. This seems to be a characteristic of the runtime environment itself (and not of any particular application), and can be attributed to a frequently used library. However, the existence of single bytecode methods indicate the presence of wrapper methods to implement specific features of the Java language like private and protected methods or *interfaces*. These methods consist of a control transfer instruction which transfers control to an appropriate routine.

Further analysis of the traces show that a few unique bytecodes constitute the bulk of the dynamic bytecode stream. In most benchmarks, fewer than 45 distinct bytecodes constitute 90% of the executed bytecodes and fewer than 33 bytecodes constitute 80% of the executed bytecodes (Table 4). It is observed that memory access and memory allocation related bytecodes dominate

[1]A java *method* is equivalent to a "function" or "procedure" in a procedural language like C.

Benchmark	# bytecodes 80%	90%	100%
compress	21	30	65
jess	33	48	132
db	33	45	144
javac	31	45	144
mpegaudio	24	36	120
mtrt	26	39	128
jack	10	22	124

Table 4: Number of distinct bytecodes that account for 80%, 90% and 100% of the dynamic instruction stream

	s1		**s10**		**s100**	
benchmarks	calls	methods	calls	methods	calls	methods
compress	17330744	572	18170275	570	99533002	572
jess	414349	1222	5697628	1313	95957670	1375
db	65379	642	1610941	645	91753107	658
javac	213243	1384	2515940	3142	54503910	3325
mpegaudio	954605	843	8289656	846	93046042	844
mtrt	1906112	781	7031487	785	71168982	796
jack	2318110	1230	4621508	1233	39172145	1240

Table 5: Total number of method calls (dynamic) and unique methods for the three data sets

the bytecode stream of all the benchmarks. This also suggests that if the instruction cache can hold the JVM interpreter code corresponding to these bytecodes (i.e. all the cases of the switch statement in the interpreter loop), the cache performance will be better.

Table 5 presents the number of unique methods, and the frequency of calls to those methods. The number of methods and the dynamic calls are obtained at runtime by dynamically profiling the application. Hence only methods that execute at least once have been counted. Table 5 also shows that the static size of the benchmarks remain constant across the different data sets (since the number of unique methods does not vary), although the dynamic instruction count increases

benchmarks	**s1**	**s10**	**s100**
compress	30298	31878	174008
jess	340	4339	16392
db	102	2498	139442
javac	154	800	16392
mpegaudio	1132	9799	89787
mtrt	2440	8957	89408
jack	1885	3748	110244

Table 6: Method Reuse Factor for the different data sets

for the bigger data sets (due to increased method calls). The number of unique calls has an impact on the number of indirect call sites present in the application. Looking at the three data sets, we see that there is very little difference in the number of methods across data sets.

Another bytecode characteristic we look at is the *method reuse* factor for the different data sets. The *method reuse* factor can be defined as the ratio of *method calls* to *number of methods* visited at least once. It indicates the locality of methods. The method reuse factor is presented in Table 6. The performance benefits that can be obtained from using a JIT compiler is directly proportional to the *method reuse* factor, since the cost of compilation is amortized over multiple calls in JIT execution.

The higher number of method calls indicate that the method reuse in the benchmarks for larger data sets would be substantially more. This would then lead to better performance for the JITs (as observed in the next section). In Section 5 we show that the instruction count when the benchmarks are executed using a JIT compiler is much lower than when using an interpreter for the s100 data set. Since there is higher method reuse in all benchmarks for the larger data sets, using a JIT results in better performance over an interpreter. The bytecode characteristics described in this Section help in understanding some of the issues involved in the performance of the Java runtime system (presented in the remaining part of the paper).

4 When or whether to translate

Dynamic compilation has been popularly used [11, 30] to speed up Java executions. This approach avoids the costly interpretation of JVM bytecodes, while sidestepping the issue of having to pre-compile all the routines that could ever be referenced (from both the feasibility and performance angles). Dynamic compilation techniques, however, pay the penalty of having the compilation/translation to native code falling in the critical path of program execution. Since this cost is expected to be high, it needs to be amortized over multiple executions of the translated code. Or else, performance can become worse than when the code is just interpreted. Knowing when to dynamically compile a method (using a JIT), or whether to compile at all, is extremely important for good performance. To our knowledge, there has not been any previous study that has examined this issue in depth in the context of Java programs, though there have been previous studies [13, 31, 12, 4] examining efficiency of the translation procedure and the translated code. Most of the currently available execution environments, such as JDK 1.2 [16] and Kaffe [17] employ limited heuristics to decide on when (or whether) to JIT. They typically translate a method on its first invocation, regardless of how long it takes to interpret/translate/execute the method and how many times the method is invoked. It is not clear if one could do better (with a smarter heuristic) than what many of these environments provide. We investigate these issues in this section using five SPECjvm98 [15] benchmarks (together with a simple HelloWorld program[2]) on the Kaffe environment.

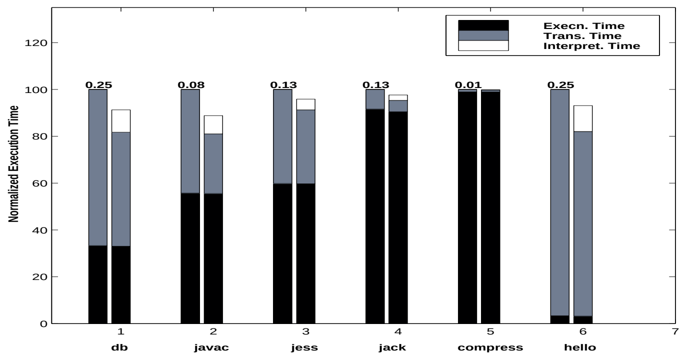


Figure 2: Dynamic Compilation: How well can we do? The first bar for each benchmark is execution time with default JIT mode in Kaffe; second bar is execution time with a smart JIT that uses perfect heuristics

Figure 2 shows the results for the different benchmarks. All execution times are normalized with respect to the execution time taken by the JIT mode on Kaffe. On top of the JIT execution bar is given the ratio of the time taken by this mode to the time taken for interpreting the program using Kaffe VM. As expected (from the method reuse characteristics for the various benchmarks), we find that translating (JIT-ing) the invoked methods significantly outperforms interpreting the JVM bytecodes for the SPECjvm98. The first bar, which corresponds to execution time using the default JIT, is further broken down into two components, the total time taken to translate/compile

[2]While we do not make any major conclusions based on this simple program, it serves to observe the behavior of the JVM implementation while loading and resolving system classes during system initialization.

the invoked methods and the time taken to execute these translated (native code) methods. The considered workloads span the spectrum, from those in which the translation times dominate such as *hello* and *db* (because most of the methods are neither time consuming nor invoked numerous times), to those in which the native code execution dominates such as *compress* and *jack* (where the cost of translation is amortized over numerous invocations).

The JIT mode in Kaffe compiles a method to native code on its first invocation. We next investigate how well the smartest heuristic can do, so that we compile only those methods that are time consuming (the translation/compilation cost is outweighed by the execution time) and interpret the remaining methods. This can tell us whether we should strive to develop a more intelligent selective compilation heuristic at all, and if so, what is the performance benefit that we can expect. Let us say that a method i takes I_i time to interpret, T_i time to translate, and E_i time to execute the translated code. Then, there exists a crossover point $N_i = T_i/(I_i - E_i)$, where it would be better to translate the method if the number of times a method is invoked $n_i > N_i$, and interpret it otherwise. We assume that an oracle supplies n_i (the number of times a method is invoked) and N_i (the ideal cut-off threshold for a method). If $n_i < N_i$, we interpret all invocations of the method, and otherwise translate it on the very first invocation. The second bar in Figure 2 for each application shows the performance with this oracle, which we shall call *opt*. It can be observed that there is very little difference between the naive heuristic used by Kaffe and *opt* for *compress* and *jack* since most of the time is spent in the execution of the actual code anyway (very little time in translation or interpretation). As the translation component gets larger (applications like *db*, *javac* or *hello*), the *opt* model suggests that some of the less time-consuming (or less frequently invoked) methods be interpreted to lower the execution time. This results in a 10-15% savings in execution time for these applications. It is to be noted that the exact savings would definitely depend on the efficiency of the translation routines, the translated code execution and interpretation.

The *opt* results give useful insights. Figure 2 shows that by improving the heuristic that is employed to decide on when/whether to JIT, one can at best hope to trim 10-15% in the execution

Benchmarks	Execution Time	Translation Time	% translation phase
compress	30357117894	770898443	2.53
db	9485628139	1034621136	10.9
javac	13994202232	1869633297	13.3
mtrt	65396263261	5931042361	9.06
mpeg	20591664167	1130043309	5.49
jack	40959418061	1163206309	2.83

Table 7: Breakdown of execution time for the s10 dataset

time. It must be observed that the 10-15% gains observed can vary with the amount of method reuse and the degree of optimization that is used. For example, we observed that the translation time for the Kaffe JVM accounts for a smaller portion of overall execution time with larger data sets (7.5% for s10 dataset (shown in Table 7) as opposed to the 32% for s1 dataset). Hence, reducing the translation overhead will be of lesser importance when execution time dominates translation time. However, as more aggressive optimizations are used the translation time can consume a significant portion of execution time for even larger datasets. For instance, the base configuration of the translator in IBM's Jalapeno VM [32] takes negligible translation time when using the s100 data set for javac. However, with more aggressive optimizations about 30% of overall execution time is consumed in translation to ensure that the resulting is code is executed much faster[32]. Thus, there exists a trade-off between reducing the amount of time spent in optimizing the code and the amount of time spent in actually executing the optimized code.

For the s1 dataset we find that a substantial amount of the execution time is spent in translation and/or executing the translated code, and there could be better rewards from optimizing these components. This serves as a motivation for the rest of this paper which examines how these components exercise the hardware features (the CPU and cache in particular) of the underlying machine, towards proposing architectural support for enhancing their performance.

While it is evident from the above discussion that most methods benefit from JIT compilation, resource constraints may force us to choose an interpreted JVM. Large memory space required by JIT compilers has been considered to be one of the issues limiting their usage in resource-

benchmark	compress	jess	db	javac	mpeg	mtrt	jack
% increase	10.3	33.5	25.7	22.1	16.3	30.2	26.6

Table 8: Increase in memory usage of JIT compiler compared to interpreter

constrained environments. For the SPECjvm98 benchmarks, we observe from Table 8 that the memory size required by the JIT compiler is 10-33% higher than that required for the interpreter. It is to be noted that there is a more pronounced increase for applications with smaller dynamic memory usage [27], such as *db*. The memory overhead of JIT can thus be more significant in smaller (embedded) applications. Due to the different constraints imposed on JVM implementations, it is rather difficult to preclude one implementation style over the other. As a result, we include both interpreters and JIT compilers in our architectural studies, in the rest of this paper.

5 Architectural issues

Understanding the underlying characteristics of Java applications in their various modes of execution, and in comparison to code in other languages/paradigms is extremely important to develop an efficient run-time environment for Java. In order to answer some of the questions raised earlier in Section 1 we have conducted detailed studies on the instruction mix of SPECjvm98 programs in interpreter and JIT-compiled modes of execution. We also study the cache performance, branch predictor performance, and the instruction level parallelism of these programs, at the native SPARC code level.

5.1 Instruction mix

Table 9 shows the dynamic instruction count for the SPECjvm98 benchmarks when run using the s1 data sets. Table 9 shows the number of bytecodes that were executed, as well the number of native instructions executed when the benchmarks were interpreted or run using the JIT compiler.

Figure 3 shows a summary of the results on the instruction mix, computed cumulatively over all the SPECjvm98 programs. The individual application mixes exhibit a similar trend, and the

benchmarks	bytecodes	interpreter	JIT
compress	954990234	10425544771	1385557173
jess	8126332	259702575	188581319
db	2035798	86844476	75959798
javac	5958654	199254389	167556898
mpeg	115748387	1314397268	264879641
mtrt	50683565	1531909956	942593921
jack	175740325	2668899901	986682716

Table 9: Instruction count at the bytecode level, and for the interpreterd and JIT compiled execution modes

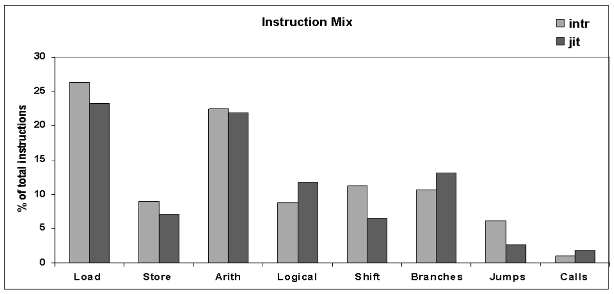


Figure 3: Instruction Mix at the SPARC code level

results are included in [33]. Execution in the Java paradigm, either using the interpreter or JIT compiler, results in 15% to 20% control transfer instructions and 25% to 40% memory access instructions [3]. Although memory accesses are observed to be frequent in the instruction stream in both modes, it is 5% more frequent in the interpreted mode in comparison to the JIT compiler. In interpreted mode, a large percentage of operations involve accessing the stack, which translate to loads and stores in the native code. Contrary to this, in the JIT compiler mode, many of these stack operations are optimized to register operations, resulting in a reduction in the frequency of memory operations in the instruction mix. While this is to be expected, our experiments quantify the percentage reduction.

Past studies have shown that Java applications (and object oriented programs in general) contain indirect branches with a higher frequency than SPECint programs [35, 36]. Figure 3 provides information on indirect branch frequency (i.e. Jumps) for the interpreted and JIT execution

[3]This instruction mix is not significantly different from that of traditional C and C++ programs such as those used in [34]

modes. Comparing the two execution modes, the interpreter mode has higher percentage of indirect jumps (primarily due to register indirect jumps used to implement the switch statement in the interpreter), while the code in the JIT compiler case has higher percentage of branches and calls. JIT compilers optimize virtual function calls by inlining those calls, thereby lowering the number of indirect jump instructions. A combination of a lower number of switch statements that are executed and inlining of the method calls, results in more predictable behavior of branches in the JIT mode, as illustrated in the next section.

5.2 Branch prediction

Benchmark	Mode	2-BIT	BHT	Gshare	GAp
compress	Intr	60.23	34.69	34.90	35.41
	JIT	28.31	9.26	8.97	8.93
jess	Intr	46.39	19.13	19.37	18.66
	JIT	38.16	13.05	12.74	12.88
db	Intr	43.97	17.12	16.82	16.69
	JIT	39.64	12.82	12.70	12.81
javac	Intr	44.74	18.04	17.90	17.39
	JIT	39.06	12.92	12.18	12.47
mpeg	Intr	52.54	31.61	33.29	31.59
	JIT	37.47	12.24	11.88	12.16
mtrt	Intr	48.78	14.54	13.17	13.42
	JIT	41.43	11.62	9.20	10.44
jack	Intr	56.26	28.31	28.78	27.92
	JIT	35.68	12.78	12.26	12.65

Table 10: Branch misprediction rates for four predictors

The predictability of branches in Java applications along with the suitability of traditional branch predictors, is examined in this section. Virtual function calls that are abundant in Java applications, and indirect jumps abundant in the interpreted mode of execution, can complicate the task of predicting the outcome of these control instructions. Table 10 illustrates the branch misprediction rates for four different branch prediction schemes including a simple 2-bit predictor [37], 1 level Branch History table (BHT) [37], Gshare [38] and a two-level predictor indexed by PC (described as GAp by Yeh and Patt [39]. The one level predictor and GShare predictor have 2K entries and the two level predictor has 256 rows of 8 counters each (3 bits of correlation).

The Gshare predictor uses 5 bits of global history. The Branch Target Buffer (BTB) contains 1K entries. The branch predictors get sophisticated as we go from left to right in Table 10. The simple 2-bit predictor has been included only for validation and consistency checking. As expected from trends in previous research, among the predictors studied, Gshare or GAp has the best performance for the different programs. The major trend observed from our experiments is that the branch prediction accuracy in interpreter mode is significantly worse than that for the JIT compiler mode. This is a direct implication of the control transfer instruction mix in the interpreter and JIT compile modes. The interpreter mode results in a high frequency of indirect control transfers due to the switch statement for case by case interpretation. The accuracy of prediction for the Gshare scheme is only 65 to 87% in interpreter mode and 88 to 92% in the JIT compiler mode. Thus, it may be concluded that branch predictor performance for Java applications is significantly deteriorated by the indirect branches abundant in the interpreter mode, whereas execution with the JIT compiler results in performance comparable to that of traditional programs. To summarize, if Java applications are run using the JIT compiler, the default branch predictor would deliver reasonable performance, whereas if the interpreter mode is used, a predictor well-tailored for indirect branches (such as [36], [40]) should be used.

5.3 Locality and cache performance

In addition to examining the locality/cache behavior of Java executions in the following discussion, we also examine how the coexistence of the JVM and the application being executed affects the locality behavior of the entire execution. We perform a detailed study of the cache behavior, looking at the entire execution in totality, as well as the translation and execution parts (of the JIT mode) in isolation.

Table 11 illustrates the number of references and misses for the L1 instruction and data cache in the interpreter and JIT compiled modes. Both instruction and data caches are of 64K bytes size and have a block size of 32 bytes. The instruction cache is 2-way set associative and the data cache is 4-way set associative. Instruction cache performance in interpreted mode is extremely

Benchmark	Mode	**I-Cache** Refs	Misses	Rate	**D-Cache** Refs	Misses	Rate
compress	Intr	10425M	84398	0.001	5365M	21M	0.394
	JIT	1385M	218101	0.013	751M	43M	5.828
jess	Intr	259M	179545	0.068	81M	2.3M	2.910
	JIT	188M	616962	0.321	45M	3.7M	8.298
db	Intr	86M	82873	0.094	24M	751592	3.097
	JIT	75M	232046	0.300	17M	1.2M	6.764
javac	Intr	199M	140239	0.069	59M	1.8M	3.099
	JIT	167M	469143	0.274	40M	3.3M	8.335
mpeg	Intr	1314M	92439	0.007	544M	1.9M	0.356
	JIT	264M	355896	0.134	101M	3.2M	3.232
mtrt	Intr	1531M	252370	0.016	521M	8.6M	1.654
	JIT	942M	522692	0.054	230M	16M	7.239
jack	Intr	2668M	124563	0.005	1033M	11M	1.069
	JIT	986M	1.0M	0.102	298M	15M	5.341

Table 11: Cache Performance for the SPECjvm98

This table shows the number of references, misses and miss rate for the instruction and data cache. Despite the redundancy, number of misses and % misses are both provided to glean the increase in absolute number of misses in JIT compiler case compared to interpreter, despite the drastic reduction in total number of references. Cache size= 64K bytes, block size= 32 bytes, I-cache is 2-way and D-cache is 4-way set-associative. M - indicates million.

good with hit-rates higher than 99.9% in all benchmarks. The interpreter is a switch statement with approximately 220 cases for decoding each bytecode. The excellent instruction locality in interpreted mode stems from the fact that the entire switch statement or at least the most frequently used parts of it nicely fit into state-of-the-art cache sizes. Prior research showed that that 15 unique bytecodes on the average, account for 60% to 85% of the dynamic bytecodes [33]. It was also observed that 22 to 48 distinct bytecodes constituted 90% of the dynamic bytecodes executed. These factors result in a small working set for the interpreter.

The instruction cache performance in JIT compiler mode is inferior to instruction cache performance in interpreter mode. Dynamically compiled code for consecutively called methods may not be located in contiguous locations. Rather than bytecode locality, it is method locality, method footprint, and working set properties of the JIT compiler code that determine the instruction cache performance for the execution of code generated in the JIT mode. Compilers typically result in poor cache performance (as exemplified by *gcc* in the SPEC suite [34]) and compilation process is a major component of the JIT mode. For applications like *db*, *jess* and *javac* which spend a significant amount of time in the translation part (Figure 2), the I-cache misses are more dominant.

The data cache performance of Java applications is worse than its instruction cache performance, as is the case for normal C/C++ programs. However, data locality in the interpreted mode is better than the locality in the case of the JIT compiler. In the interpreter mode, each time a method is executed, the bytecodes are accessed from the data cache and decoded by the interpreter. The intended code is thus treated as data by the interpreter, in addition to the actual data accessed by the application, resulting in a lower miss rate overall (code usually has better locality than data). The benchmark data and benchmark bytecodes will be allocated and accessed from the data cache. Two benchmarks, *compress* and *mpeg*, exhibit significant method reuse and yield excellent data cache hit ratios in the interpreter mode, because the footprint can be entirely captured in the cache. In contrast, the JIT compiler translates the bytecodes fetched from the data cache into native code before the first execution of the method. Therefore the subsequent invocations of the method do not access the data cache (they access the I-cache) for bytecodes. This results in a drastic reduction of total data cache references from interpreter mode to JIT mode as illustrated in Table 11. The number of data references in the JIT compiler case is only 20% to 80% of the reference count in the interpreter case. Of the total data cache misses in the JIT mode, 50 to 90% of misses at 64K cache size are write misses (see Figure 4). We later demonstrate that the majority of these result from code installation in the translate phase (Figure 6).

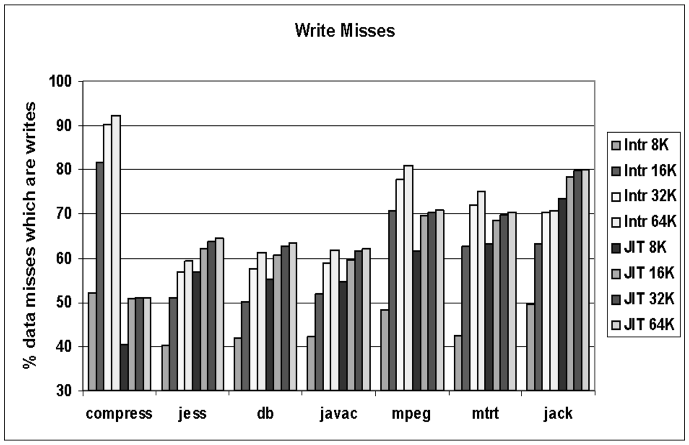
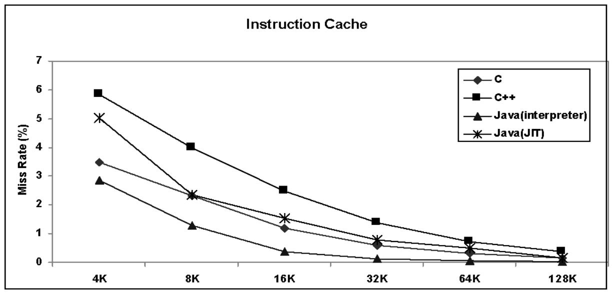


Figure 4: Percentage of Data Misses that are Writes. Cache used is direct mapped with line size of 32 bytes



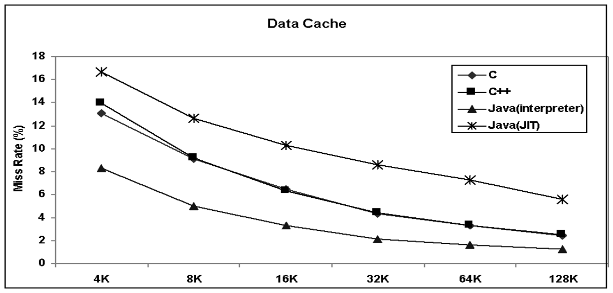


Figure 5: Miss rates for C, C++ and Java workloads for (i) Instruction Cache (ii) and Data Cache. The miss rates for C and C++ are obtained from studies by Calder et. al

Figure 5 illustrates the average [4] cache miss rates of SPECjvm98 workloads in comparison to the SPECint programs and several C++ programs. For both instruction and data caches, the interpreter mode exhibits better hit rates than C, C++ and the JIT mode of execution. The behavior during execution with the JIT compiler is closer to that of traditional C and C++ programs for the instruction cache. In the case of data cache, the miss rates for the JIT mode of execution are highest among the different workloads. It may be inferred that the behavior of Java applications are predominantly dependent on the execution mode rather than the object-oriented nature of the language i.e. the results depend more on whether they are run in interpreter or JIT mode rather than on the fact that they are object-oriented. This observation might have been influenced by the fact that the SPECjvm98 benchmarks are not very object-oriented.

One noticeable fact in Table 11 is that the absolute number of misses (instruction and data) in the JIT compiler mode is higher than the number of misses in the interpreter mode, despite the reduction in total instruction count and data cache reference count. There are two factors that can be attributed to this - code generation and installation of translated code performed by the JIT compiler. Both these operations can result in a significant number of misses, which we show by studying the behavior of these references in isolation.

cache behavior during the translation part and the rest of the JIT execution. The cache behavior of the translate portion is illustrated in Figure 6. The translation related instruction cache misses contribute to more than 30% (except *jack* and *mtrt*) of all the instruction cache misses. The data

[4]All averages presented are arithmetic means unless specified otherwise.

cache misses of the translate routines do not exhibit any general trends, and are dependent on the application. For *mpeg*, *compress* and *db* benchmarks, the data cache exhibits a better locality in the code outside the translate routine. While *compress* benefits from high spatial locality operating on sequential elements of large files, *db* benefits from reuse of a small database to perform repeated data base operations. For *javac*, it was found that the code within and outside translate exhibit a similar cache behavior (miss rates of 5.5 and 5.3% inside and outside translate). This can be ascribed to *javac* being a complier and the executed code performing the same type of operations as the translate routine.

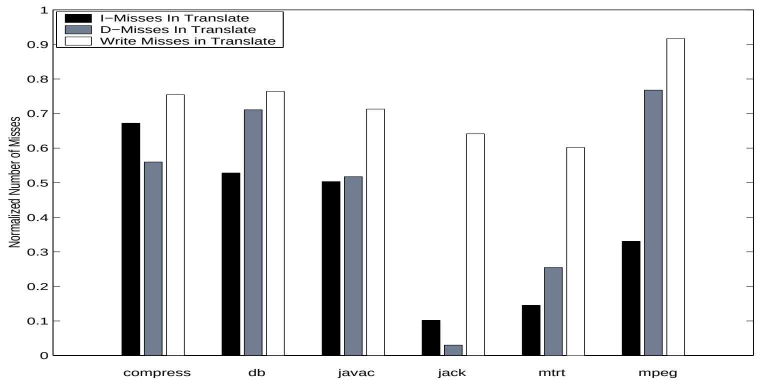


Figure 6: Cache Misses within Translate Portion for the s1 dataset. Cache configuration used : 4-way set associative, 64K DCache with a line size of 32 bytes and 2-way set associative, 64K ICache with a line size of 32 bytes. The first bar shows the I-Cache misses in translate relative to all I-Cache misses, the second bar shows the D-Cache misses in translate relative to all D-Cache misses, and the third bar shows the D-Cache write misses in translate relative to overall D-Cache misses in translate.

The data cache misses in the translate portion of the code contribute to 40-80% of all data misses for many of the benchmarks. Among these, the data write misses dominate within the translate portion and contribute to 60% of misses during translate (see the third bar for each benchmark in Figure 6). Most of these write misses were observed to occur during the generation and installation of the code. Since, the generated code for the method is written to memory for the first time, it results in compulsory misses in the D-Cache. One may expect similar compulsory misses when the bytecodes are read during translation. However, they are relatively less frequent than the write misses since 25 native (SPARC) instructions are generated per bytecode on an average [41]. An optimization to lower the penalty of write misses during code generation and installation is discussed later in Section 6. The cache behavior during the translation process for the s10 data size is given in Figure 7. We observe that the write misses in translate still contribute to a significant portion of the performance penalty.

We also studied the variation in the cache locality behavior during the course of execution for different benchmarks in the interpreter and JIT compiler modes. The results for *db* can be observed in Figure 8. The miss rates in the interpreter mode show initial spikes due to the class loading at the start of the actual execution. However, there is a fairly consistent locality for the rest of the code. In contrast, there are a significantly larger number of spikes in the number of misses during execution in the JIT mode. This can be attributed to the compilation part of the JIT compiler which results in significant number of write misses. A clustering of these spikes can be observed in the JIT mode in Figure 8. This is due to a group of methods that get translated in rapid succession. Also, we observed that for the *mpeg* benchmark the clustered spikes in the JIT mode are restricted to the initial phase of algorithm as there is significant reuse of the same methods.

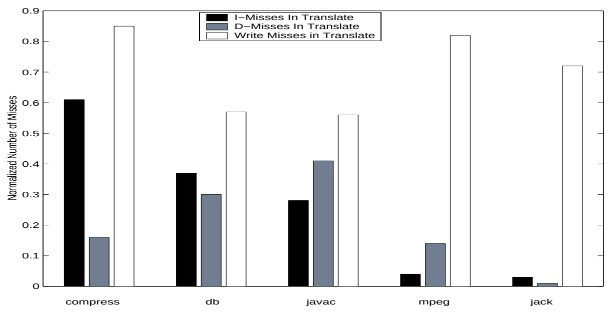


Figure 7: Cache misses within the translate portion for the s10 dataset. Cache configuration used : 4-way set associative, 64K DCache with a line size of 32 bytes and 2-way set associative, 64K ICache with a line size of 32 bytes. The first bar shows the I-Cache misses in translate relative to all I-Cache misses, the second bar shows the D-Cache misses in translate relative to all D-Cache misses, the third bar shows the D-Cache write misses in translate relative to overall D-Cache misses in translate.

5.4 Other observations from cache studies

The cache performance of SPECjvm98 applications were studied over a wide range of cache sizes, block sizes and associativity. Figure 9 illustrates that increasing associativity produces the expected effect of reducing misses, and the most pronounced reduction is when associativity is increased from 1 to 2. Increasing the line size also produces the usual effect of reducing cache

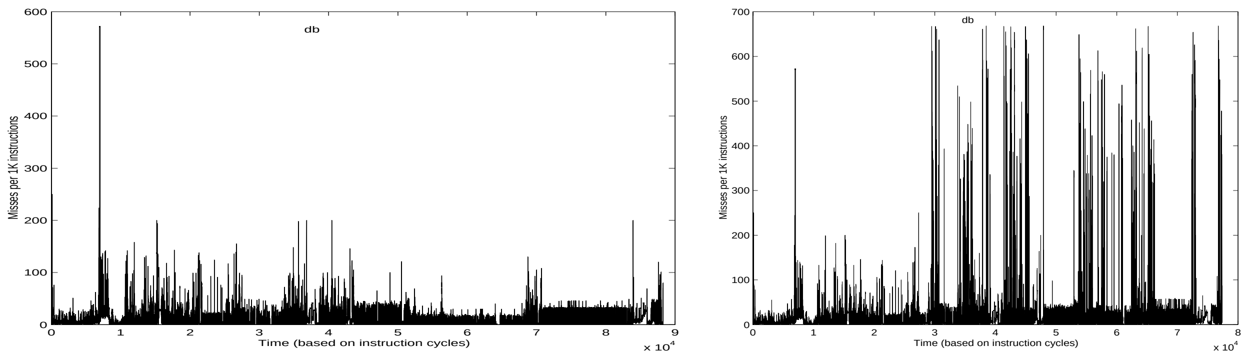


Figure 8: Miss rate variation in D-Cache for *db* during code execution in (i) Interpreter Mode and (ii) JIT Compiler Mode. Cache configuration used : 4-way set associative, 64K Cache with a line size of 32 bytes.

misses in instruction caches, however, data caches display a different behavior (illustrated in Figure 10). For interpreted code, in 6 out of the 7 benchmarks, a small data cache block size of 16 bytes is seen to have the least miss rate for the data cache. On the other hand, for execution with the JIT compiler, a block size of 32 or 64 bytes is better than 16 bytes in a majority of the cases. The increase in data cache miss rates when the line size is increased during interpreted execution can be explained using method locality and bytecode size information. Prior research on method locality and size distribution [41] showed that 45% of all dynamically invoked methods were either 1 or 9 bytecodes long. Since average bytecode size has been shown to be 1.8 bytes [14], 45% of all methods can be expected to be less than 16 bytes long. Therefore, unless methods invoked in succession are located contiguously, increasing line sizes beyond 16 bytes (or 32 at the most) cannot capture further useful future references, explaining the data cache behavior of the interpreted code. The data cache in the JIT compiler mode is affected by the size of the objects accessed by the applications. While mean object sizes of individual objects range from 16 to 23 bytes for the SPECjvm98 benchmarks, the commonly used character arrays range between 26 and 42 bytes [27]. Thus, line sizes of either 32 or 64 bytes provide the best locality for most of the benchmarks.

Layout of translated code installed by the JIT compiler can have a large impact on miss behavior. We are not aware of the details on the techniques used by Kaffe or JDK to optimize code layout. Dynamically generated code layout can thus be an interesting area for further research.

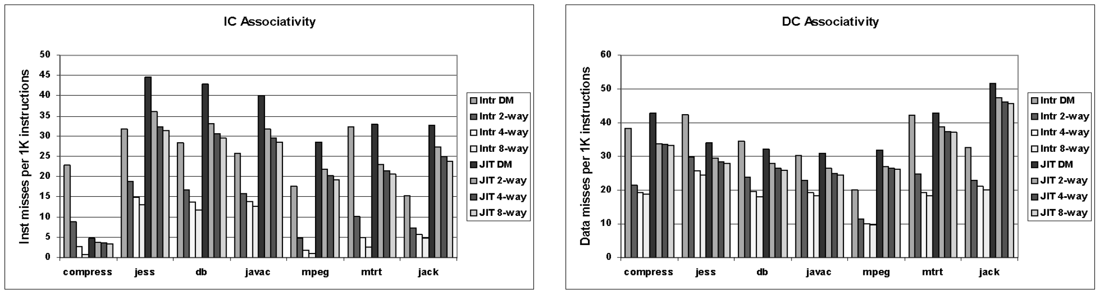


Figure 9: Effect of increasing the associativity of the (i) instruction cache and (ii) data cache. Cache configuration used: 8K Cache with 32 byte line size and associativity of 1, 2, 4 and 8.

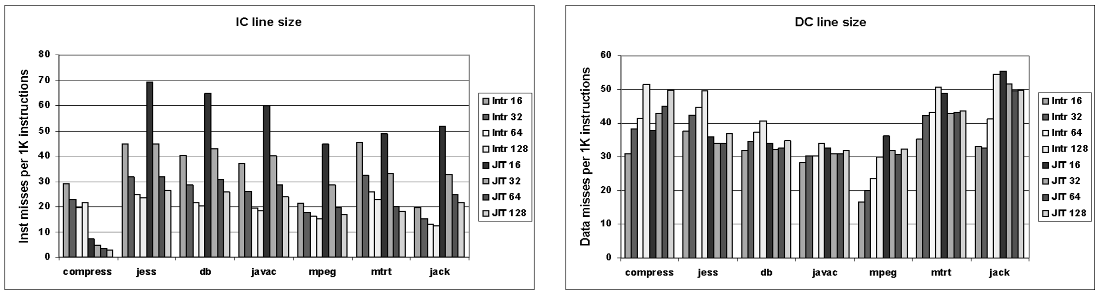


Figure 10: Effect of changing line size on the (i) instruction cache and (ii) data cache. Cache configuration used: 8K direct mapped Cache with line sizes of 16, 32, 64 and 128 bytes.

5.5 Limits of Available Parallelism

In order to understand the instruction level parallelism issues involving the stack-oriented Java code, we investigated the limits of available parallelism in Java workloads. We also compared the ILP of the Java benchmarks to SPEC95 applications, and several C++ programs. We use dynamic dependence analysis in order to compute the limits of ILP as in previous parallelism investigations [42, 43]. First, we construct a Dynamic Dependency Graph (DDG) which is a partially ordered, directed, and acyclic graph, representing the execution of a program for a particular input. The executed operations comprise the nodes of the graph and the dependencies

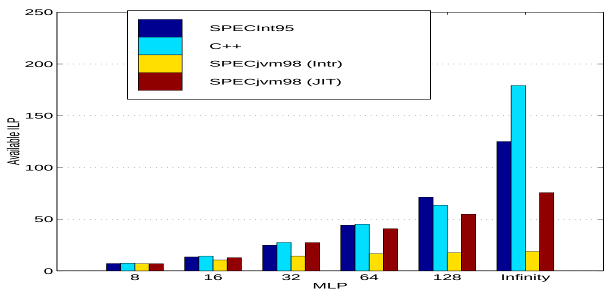


Figure 11: Average Available Parallelism of SPECInt, C++ and SPECjvm98 for Different Machine Level Parallelism (MLP)

realized during the execution form the edges of the graph. The edges in the DDG force a specific order on the execution of dependent operations - forming the complete DDG into a weak ordering of the programs required operations. A DDG, which contains only data dependencies, and thus is not constrained by any resource or control limitations, is called a dynamic data flow graph. It lacks the total order of execution found in the serial stream; all that remains is the weakest partial order that will successfully perform the computations required by the algorithms used. If a machine were constructed to optimally execute the DDG, its performance would represent an upper bound on the performance attainable for the program. In our study, first, the critical path length defined as the height of the scheduled DDG (the absolute minimum number of steps required to evaluate the operations in the scheduled DDG) is determined. The available parallelism is computed by dividing the total number of nodes by the critical path length. Machine Level Parallelsim (MLP) is the maximum number of nodes that can actually be scheduled in a cycle given the constraints of the machine. It is analogous to the number of functional units that can be used simultaneously. However we do not restrict the type of functional unit, i.e. they can all be adders or multipliers or a mix of different units. To give an upper bound on the available parallelism, an available MLP of infinity was considered but MLP of 8, 16, 32, 64 and 128 were also studied for comparative purposes (See Table 12). The latency of all operations is set to be 1 cycle. Perfect memory disambiguation and perfect branch prediction are assumed. We consider only true dependencies (or RAW dependencies) while scheduling instructions. Hence this is the

Benchmarks	Available ILP (for MLPs of 8 ... Infinite)					
SPEC Int95	8	16	32	64	128	Infinity
compress	6.94	13.68	21.9	37.65	65.89	265.01
gcc	7.26	13.94	25.56	40.79	58.15	92.14
go	7.15	14.11	27.26	48.79	70.23	83.81
li	7.47	14.25	25.35	47.17	70.15	72.48
m88ksim	6.45	11.09	21.50	45.59	104.43	133.36
ijpeg	7.44	14.67	28.46	48.01	71.86	8465.51
C++	8	16	32	64	128	Infinity
deltablue	7.28	15.15	30.79	60.48	111.28	270.97
eqn	7.42	14.18	26.54	37.96	43.51	272.09
idl	7.57	14.59	29.05	48.31	63.49	277.01
ixx	7.2	13.83	26.92	51.32	76.05	111.71
richards	7.56	13.78	24.83	35.84	55.99	125.57
SPECjvm98 (Intr)	8	16	32	64	128	Infinity
jess	6.69	11.02	16.17	19.54	20.78	22.83
db	6.75	11.59	19.54	28.55	33.7	41.06
javac	6.66	10.67	15.55	19.82	21.89	25.13
mpeg	7.27	10.11	12.86	14.63	15.19	15.39
mtrt	6.76	10.26	12.92	14.06	14.38	15.82
jack	7.02	10.85	19.15	28.87	31.69	34.24
SPECjvm98 (JIT)	8	16	32	64	128	Infinity
jess	6.92	12.78	23.57	39.59	47.72	49.33
db	6.67	12.31	23.30	43.76	70.46	96.90
javac	6.67	12.22	22.97	42.05	65.33	86.23
mpeg	7.30	13.71	25.12	45.73	76.49	174.10
mtrt	7.04	13.05	23.34	34.24	40.04	86.51
jack	7.00	12.99	21.22	40.44	47.88	50.47

Table 12: Available Instruction Level Parallelism of Different Benchmarks for MLP of 8, 16, 32, 64, 128 and the Unbounded Machine. SPECjvm98 Benchmarks are Invoked with an Interpreter and a JIT compiler.

absolute limit of parallelism that can potentially be exploited from that program, with the best of renaming, etc. More details on these experiments can be found in [44]. Table 12 shows that Java code exhibits less ILP in comparison to all other workloads analyzed. The average available parallelism (in terms of the harmonic mean of the observations) of the four different suites of programs for different window sizes is summarized in Figure 11 .

With infinite MLP, the mean ILP is 125 for the SPECInt benchmarks) and 175 for the C++ programs. The mean ILP is 20 for the Java programs when interpreted and 40 when invoked through the JIT compiler. The extremely low ILP of the interpreted Java programs, even with no other control or machine constraints, can be attributed to the stack-based implementation of the Java Virtual Machine (JVM). The stack nature of the JVM imposes a strict ordering on the execution of the bytecodes. The JIT compiler optimizes away most of these stack accesses and converts them to register operations. Hence we see a higher ILP for the same benchmarks when executed using a JIT compiler.

6 Architectural Implications

We have looked at a spectrum of architectural issues that impact the performance of a JVM implementation, whether it be an interpreter or a JIT compiler. In the following discussion, we briefly summarize our observations, review what we have learned from these examinations, and comment on enhancements for the different runtime systems. More importantly, we try to come up with a set of interesting issues, that are, perhaps, worth a closer look for future research.

Even though our profiling of the JVM implementations for many of the SPECjvm98 benchmarks shows that there is a substantial amount of time spent by the JIT compiler in translation, it appears that one cannot hope to save much with a better heuristic than compiling a method on its first invocation (10-15% saving at best with an ideal heuristic). Rather, the effort should be expended in trying to find a way of tolerating/ hiding the translation overhead. We also found that one cannot discount interpretation in an ad hoc manner, since it may be more viable in a resource-constrained (memory in particular) environment.

An examination of the architectural interactions of the two runtime alternatives, has given us useful insights. It has been perceived that Java (object-oriented programs in general) executions are likely to have substantial indirect branches, which are rather difficult to optimize. While we find this to be the case for the interpreter, the JIT compilers seem sufficiently capable of performing optimizations to reduce the frequency of such instructions. As a result, conventional two-level branch predictors would suffice for JIT mode of execution, while a predictor optimized for indirect branches (such as [36]) would be needed for the interpreted mode. The instruction level parallelism available in interpreted mode is seen to be lower than while using a JIT compiler. The parallelism in the interpreted mode is heavily influenced by the stack-oriented nature of the

JVM ISA. This affects the performance as one moves to wider superscalar machines. We find that the interpreter exhibits better locality for both instructions and data, with substantial reuse of a few bytecodes. The I-cache locality benefits from the interpreter repeatedly executing the native instructions corresponding to these bytecodes, and D-cache locality is also good since these bytecodes are treated as data. In general, the architectural implications of a Java runtime system is seen to be more dependent on the mode of execution (interpreter or JIT) rather than the object-oriented nature of Java programs.

Figure 2 shows that a significant component of the execution time is spent in the translation to native code, specifically for applications like *db*, *javac* and *jess*. A closer look at the miss behavior of the memory references of this component in Section 5 shows that this is mainly due to write misses, particularly those that occur in code generation/installation. Installing the code will require writing to the data cache, and these are counted as misses since those locations have not been accessed earlier (compulsory misses). These misses introduce two kinds of overheads. First, the data has to be fetched from memory into the D-cache before they are written into (on a write-allocate cache, which is more predominant). This is a redundant operation since the memory is initialized for the first time. Second, the newly written instructions will then be moved (automatically on instruction fetch operations) from the D-cache to the I-cache (not just causing an extra data transfer, but also potentially double-caching). To avoid some of these overheads, it would be useful to have a mechanism wherein the code can be generated directly into the I-cache. This would require support from the I-cache to accommodate a write operation (if it does not already support it), and preferably a write-back I-cache. It should also be noted that for good performance, one should be careful to locate the code for translation itself such that it does not interfere/thrash with the generated code in the I-cache. We are looking into the possibility of reusing the recently translated code in subsequent translations (so that translation can be speeded up). It was also suggested earlier in Section 5 that it may be a worthwhile effort to look into issues of translated code location (perhaps using associations), to improve locality during subsequent executions.

Figure 2 shows that there are applications, like *compress*, and *jack*, in which a significant portion of the time is spent in executing the translated code. One possible way of improving these applications, is to generate highly optimized code (spending a little more time to optimize code will not degrade the performance). Another approach is to speed up the execution of the generated code. This could involve hardware and systems software support for memory management, synchronization and class resolution/loading. We are currently in the process of isolating the time spent in these components, and their interactions.

There is a common (and interesting) trait in the *compress*, *jack* and *mpeg* applications, where the execution time dominates and a significant portion of this time is spent in certain specific functions. For instance, *compress* and *mpeg* employ a standard set of functions to encode all the data. The benchmark *jack* scans the data, looking for matching patterns. If we are to optimize the execution of such functions, then we can hope for much better performance. We are currently trying to identify such commonly employed functions (for at least certain application domains), so that we can configure hardware cores using reconfigurable hardware (such as Field Programmable Gate Arrays) on-the-fly (similar to how JIT dynamically opts to compile-and-execute rather than interpret).

7 Conclusions and future work

The design of efficient JVM implementations on diverse hardware platforms is critical to the success of Java technology. An efficient JVM implementation involves addressing issues in compilation technology, software design and hardware-software interaction. We began this exercise with an exploration of Java workload characterizations at the bytecode level. Then we investigated how well a dynamic compiler can perform by using intelligent heuristics at runtime. The scope for such improvement is observed to be limited, and stresses the need for investigating sophisticated compilation techniques and/or architectural support features. This study has focused on understanding the influence of hardware-software interactions of the two most common JVM implementations (interpreter and JIT-compiler), towards designing architectural support for efficient execution of Java programs. The major findings from our research are the following:

- At the bytecode level, 45 out of the 255 bytecodes constitute 90% of the dynamic bytecode stream. The excellent bytecode locality observed for the SPECjvm98 applications and the distribution seen for the method sizes help in explaining the locality and cache performance observed for the JIT compiled and interpreted applications.

- When Java applications are executed with a JIT compiler, selective translation using good heuristics can improve performance. However, even an oracle can improve performance by only 10-15% for the SPECjvm98 applications. Further improvement necessitates improving the quality of the translated code or architectural enhancements.

- The instruction and data cache performance of Java applications are better compared to that of C/C++ applications, except in the case of data cache performance in the JIT mode.

- Except using smaller block sizes for data caches or using branch predictors specially tailored for indirect branches, we feel that optimizing caches and branch predictors will not have a major impact on performance of interpreted Java execution.

- Write misses resulting from installation of JIT compiler output has a significant effect on the data cache performance in JIT mode. Certain enhancements, such as being able to write to the instruction cache, or using special buffers could be useful during dynamic code generation.

- The instruction level parallelism available in JIT mode is seen to be higher than while using an interpreter. JIT optimizations which convert stack computations to register based operations, expose more parallelism which can be exploited by wide superscalar machines.

The topics that seem to hold the most promise for further investigation are new architectural mechanisms for hiding the cost of translation during JIT. Techniques for achieving this may also be used in conjunction with dynamic hardware compilation (one could visualize this as hardware translation instead of compilation that is done by a traditional JIT compiler) of Java bytecodes using reconfigurable hardware. Another important direction, that has not been addressed in this paper, is on providing architectural support for compiler optimization, such as those undertaken in [45]. For example, a counter could track the number of hits associated with an entry in the branch target buffer. When the counter saturates, it can trigger the compiler to perform code inlining optimization that can replace the indirect branch instruction with the code of the invoked method. Of course, we may need some mechanism to monitor the program behavior changes to undo any optimizations that may become invalid later. It has also been observed that it would be worthwhile investigating the translated code location issues towards improving the locality during subsequent execution.

In this work, we were able to study the translation part of the JVM in isolation. Further investigation is necessary to identify the impact of the other parts of the JVM such as the garbage collector, class loader, class resolver and object allocator on the overall performance and their architectural impact. The key to an efficient Java virtual machine implementation is the synergy between well-designed software, an optimizing compiler, supportive architecture and efficient run-time libraries. This paper has looked at only a small subset of issues with respect to supportive architectural features for Java, and there are several issues that are ripe for future research.

Acknowledgments This research is supported in part by the National Science Foundation under Grants EIA-9807112, CCR-9796098, Career Award MIPS-9701475, CCR-0073419 and by Sun Microsystems.

References

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

[2] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani, "Overview of the IBM Java Just-in-Time Compiler," *IBM Systems Journal*, vol. 39, no. 1, pp. 175–193, 2000.

[3] "Symantec Cafe."
http://www.symantec.com/cafe.

[4] A. Krall, "Efficient JavaVM just-in-time compilation," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 54–61, 1998.

[5] G. Muller, B. Moura, F. Bellard, and C. Consel, "Harissa: a flexible and efficient Java environment mixing bytecode and compiled code ," in *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems (COOTS)*, pp. 1–20, 1997.

[6] "Tower Technology Corporation, Austin, TX."
http://www.towerj.com.

[7] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson, “Toba: Java for applications - a way ahead of time (wat) compiler,” Tech. Rep. Technical Report, Department of Computer Science, University of Arizona, Tucson, 1997.

[8] M. T. C. C. A. Hsieh, T. L. Johnson, J. C. Gyllenhaal, and W. W. Hwu, “ Optimizing NET compilers for improved Java performance ,” *IEEE Computer*, pp. 67–75, June 1997.

[9] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi, “Marmot: An optimizing compiler for java,” Tech. Rep. MSR-TR-99-33, Microsoft Research, 1999.

[10] M. O’Connor and M. Tremblay, “ picoJava-I: The Java virtual machine in hardware ,” *IEEE Micro*, pp. 45–53, March-April 1997.

[11] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko, “ Compiling Java just in time ,” *IEEE Micro*, vol. 17, pp. 36–43, May-June 1997.

[12] A. Adl-Tabatabai, M. Cierniak, G. Lueh, V. M. Parakh, and J. M. Stichnoth, “Fast effective code generation in a just-in-time Java compiler,” in *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pp. 280–290, June 1998.

[13] T. Newhall and B. Miller, “Performance measurement of Interpreted Programs,” in *Proceedings of Euro-Par’98 Conference*, September 1998.

[14] N. Vijaykrishnan, N. Ranganathan, and R. Gadekarla, “Object-oriented architectural support for a Java processor,” in *Proceedings of the 12th European Conference on Object-Oriented Programming*, pp. 430–455, July 1998.

[15] “SPEC JVM98 Benchmarks.”
http://www.spec.org/osg/jvm98/.

[16] “Overview of Java platform product family .”
http://www.javasoft.com/products/OV_jdkProduct.html.

[17] “Kaffe Virtual Machine.” http://www.transvirtual.com.

[18] R. F. Cmelik and D. Keppel, “Shade: A fast instruction-set simulator for execution profiling,” Tech. Rep. SMLI TR-93-12, Sun Microsystems Inc, 1993.

[19] A. Barisone, F. Bellotti, R. Berta, and A. De Gloria , “Instruction Level Characterization of Java Virtual Machine Workload,” in *Workload Characterization for Computer System Design* (L. John and A. Maynard, eds.), pp. 1–24, 1999.

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

[21] C. A. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhaal, and W. W. Hwu, "A study of the cache and branch performance issues with running Java on current hardware platforms," in *Proceedings of the IEEE Compcon '97*, pp. 211–216, 1997.

[22] C. A. Hsieh, J. C. Gyllenhaal, and W. W. Hwu, "Java bytecode to native code translation: the Caffeine prototype and preliminary results," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 90–97, 1996.

[23] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano, "Thin locks: featherweight synchronization in Java," in *Proceedings of ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pp. 258–268, June 1998.

[24] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, and Y. Ramakrishna, "An Efficient Meta-Lock for Implementing Ubiquitous Synchronization," in *Proceedings of OOPSLA 1999*, 1999.

[25] A. Krall and M. Probst, "Monitors and exceptions: How to implement Java efficiently ," in *Proceedings of ACM 1998 Workshop on Java for High-Performance Computing*, pp. 15–24, 1998.

[26] O. Agesen, D. Detlefs, and J. E. B. Moss, "Garbage collection and local variable type-precision and liveness in Java Virtual Machines," *Programming Languages Development and Implementation*, pp. 269–279, 1998.

[27] S. Deickmann and U. Holzle, "A study of the allocation behavior of the SPECjvm98 Java benchmarks," in *Proceedings of the European Conference on Object Oriented Programming*, July 1999.

[28] H. McGhan and M. O'Connor, " PicoJava: A direct execution engine for Java bytecode ," *IEEE Computer*, pp. 22–30, October 1998.

[29] D. Griswold, "The Java HotSpot Virtual Machine Architecture ," March 1998. Sun Microsystems Whitepaper.

[30] U. Holzle, "Java on Steroids: Sun's high-performance Java implementation," in *Proceedings of HotChips IX*, August 1997.

[31] T. Newhall and B. Miller, "Performance measurement of dynamically compiled Java executions," in *Proceedings of the 1999 ACM Java Grande Conference*, June 1999.

[32] Manish Gupta, "Optimizing Java Programs: Challenges and Opportunities, Presented at the Second Annual Workshop on Hardware Support for Objects and Microarchitectures for Java ," September 2000. Available at http://www.sun.com/labs/people/mario/iccd2000whso/.

[33] R. Radhakrishnan, J. Rubio, L. John and N. Vijaykrishnan, "Execution characteristics of just-in-time compilers," Tech. Rep. TR-990717, The University of Texas at Austin, 1999. http://www.ece.utexas.edu/projects/ece/lca/ps/tr990717.ps.

[34] B. Calder, D. Grunwald, and B. Zorn, " Quantifying behavioral differences between C and C++ programs ," *Journal of Programming Languages*, vol. 2, no. 4, 1994.

[35] K. Driesen and U. Holzle, "The Cascaded Predictor: Economical and Adaptive Branch Target Prediction," in *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 249–258, 1999.

[36] K. Driesen and U. Holzle, "Accurate Indirect Branch Prediction," in *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 167–178, June 1998.

[37] J. L. Hennesy and D. A. Patterson, *Computer Architecture, A Quanitative Approach.* Morgan Kauffman, 1996.

[38] S. McFarling, "Combining branch predictors," Tech. Rep. WRL Technical Note TN-36, DEC Western Resaech Laboratory, June 1993.

[39] T. Yeh and Y. Patt, "A Comparison of dynamic branch predictors that use two levels of branch history," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 257–266, 1995.

[40] N. Vijaykrishnan and N. Ranganathan, "Tuning Branch Predictors to support Virtual Method Invocation in Java," in *Proceedings of the 5th USENIX Conference of Object-Oriented Technologies and Systems*, pp. 217–228, 1999.

[41] R. Radhakrishnan, J. Rubio, and L. John, "Characterization of Java applications at the bytecode level and at UltraSPARC-II Machine Code Level," in *Proceedings of International Conference on Computer Design*, October 1999.

[42] T. M. Austin and G. S. Sohi, "Dynamic dependency analysis of ordinary programs," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 342–351, May 1992.

[43] R. Sathe and M. Franklin, "Available parallelism with data value prediction," in *Proceedings of the 5th International Conference on High Performance Computing (HiPC-98)*, pp. 194–201, April 1998.

[44] J. Sabarinathan, "A study of instruction level parallelism in contemporary computer applications ," December 1999. Masters Thesis, The University of Texas at Austin.

[45] M. C. Merten, A. R. Trick, C. N. George, J. Gyllenhaal, and W. W. Hwu, " A hardware-driven profiling scheme for identifying pr ogram hot spots to support runtime optimization," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp. 136–147, 1999.