Architectural Issues in Java Runtime Systems

R. Radhakrishnan†, N. Vijaykrishnan‡, L. K. John†, A. Sivasubramaniam‡

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

‡220 Pond Lab
Dept. of Computer Science and Engineering
The Pennsylvania State University
{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 (from resource-rich servers to resource-constrained hand-held/embedded systems). Towards this goal, this paper examines architectural issues, from both the hardware and JVM implementation perspectives. It specifically explores the potential of a smart JIT compiler strategy that can dynamically interpret or compile based on associated costs, investigates the CPU and cache architectural support that would benefit JVM implementations, and examines the synchronization support for enhancing performance, using applications from the SpecJVM98 benchmarks.

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.

Java programs are translated into a machine-independent JVM format (called bytecodes), to insulate them from the underlying machine architecture on which they would eventually execute. It has been suggested [2] that these bytecodes be compiled (offline) before they are executed, similar to traditional C/C++ programs. While there are interesting research issues with this approach, this paper focuses on three other techniques, since offline compilation may not always be possible for Java programs because of dynamic class loading. The architectural impact of (offline) compiled Java applications was studied, and compared to interpreted and C/C++ versions of the same applications in an earlier study [3]. The first, and perhaps the most commonly used, of these techniques is to interpret the bytecodes [4]. The second technique is to dynamically translate/compile the bytecodes into native code at runtime, following which the native code can be executed directly [5, 6, 7]. Such a translator is commonly referred to as a Just-In-Time (JIT) compiler [5]. Finally, and more recently, there has been growing interest to develop hardware runtime support, such as Java processors [8, 9], to execute the bytecodes.

There are pros and cons to each of the above techniques. Specialized hardware support, such as Java processors, though efficient, may not be sufficiently general-purpose. It would be difficult to justify the presence of such support on general-purpose servers. JIT compilers have the potential of significantly lowering the execution times compared to interpreters. However, they may require significantly more resources than interpreters, making them unsuitable for resource-constrained environments such as hand-held devices and embedded systems. They may require several kilobytes of ROM and many more megabytes of RAM [8] than interpreters.

Many previous studies [5, 7, 10, 11, 12] have focussed on enhancing each of the bytecode execution techniques. While the results from these studies are suggestive and useful, it is our belief that no one technique will be universally preferred/accepted over all platforms in the immediate future. 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 run-time environment on general purpose processors. Finally, this three-pronged strategy can also help us design environments that effi-

0-7695-0550-3/00 $10.00 © 1999 IEEE

ciently 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 run-time systems for real applications. Such a study can help us evaluate the pros and cons of the different run-time systems (helping us selectively use what works best in a given environment), isolate architectural and run-time 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:

- Where does the time go in a JIT-based execution (i.e. in translation to native code, or in executing the translated code)? How much better does dynamic compilation (JIT) fare compared to interpreting the bytecodes? 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 mixes of the native instructions that are executed on a general-purpose CPU (such as the SPARC) when executing Java programs (using an interpreter or JIT compiler)? Are these different from those for traditional C/C++ programs? Based on these, can we suggest instructions that should be optimized and functional units that should be provided for implementing an efficient Java runtime system on a general-purpose CPU?
- How do Java executions in JIT and interpreter modes fare with different branch predictors? What is the instruction-level parallelism exhibited by these modes? Based on these, can we suggest architectural support in the CPU (either general-purpose or a specialized Java processor) that can enhance Java executions?
- How does the locality behavior for the JIT and interpreter modes of Java executions compare, and how do they differ from that for traditional C/C++ programs? What cache parameters are suitable for these executions? Does the working set of the translation part of the JIT mode interfere with that for the actual execution?
- How important is synchronization in the Java runtime system? How do we use the synchronization behavior of Java programs to optimize the implementation of the synchronization mechanism?

Complete answers to all the above questions is overly ambitious, and beyond the scope of this paper. However, any advance in this direction would be enlightening. 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 [13] benchmarks, available JVM implementations such as JDK 1.1.6 [4] and Kaffe VM 0.9.2 [14] (that have both interpretation and JIT capabilities), and simulation/profiling tools on the Shade [15] environment. All the experiments have been conducted on Sun UltraSPARC machines running SunOS 5.6.

The rest of this paper is organized as follows. The next section gives details on the experimental platform. Section 3 examines the relative performance of JIT and interpreter modes, and explores the benefits of a hybrid strategy. Section 4 investigates some of the questions raised earlier with respect to the CPU and cache architectures, and the synchronization support is presented in Section 5. 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 (*compress, jess, db, javac, mpegaudio, mtrt* and *jack*), 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 for this study since it became evident during the course of the study that when using s100 input set to run the SpecJVM98, the programs run for so long that almost any amount of compilation effort will be amortized. We have also investigated the effect of larger datasets, s10 and s100. The increased method reuse resulted in expected results such as increased code locality, reduced time spent in compilation vs execution, etc, but all major conclusions from the experiments stay valid. The benchmarks were run at the command line prompt, and does not include graphics, AWT or networking. All benchmarks except *mtrt* are single-threaded.

Two popular JVM implementations were used in this study: the Sun JDK 1.1.6 [4] and Kaffe VM 0.9.2 [14]. 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 in detail. The results using KaffeVM are presented for the translate routines. 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 [15] 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 When or whether to translate

Dynamic compilation has been popularly used [5, 16] 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 (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 [10, 17, 7, 6] examining efficiency of the translation procedure and the translated code. Most of the currently available execution environments, such as JDK 1.2 [4] and Kaffe [14] 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 [13] benchmarks (together with a simple HelloWorld program[1]) on the Kaffe environment.

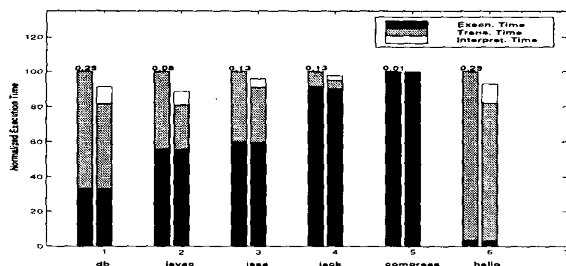


Figure 1: Dynamic Compilation: How well can we do? The first bar for each benchmark is execution time with default JIT mode in Kaffe; second bar is execution time with a smart JIT that uses perfect heuristics

Figure 1 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. 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 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). 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, we find that translating (JIT-ing) the invoked methods significantly outperforms interpreting the JVM bytecodes.

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 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 for each application shows the performance with this oracle in Figure 1, 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 1 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 time. On the other hand, 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-constrained environments. For the

[1] 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.

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

Table 1: Increase in memory usage of JIT compiler compared to interpreter

SpecJVM98 benchmarks, we observe from Table 1 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 [18], 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.

4 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.

4.1 Instruction mix

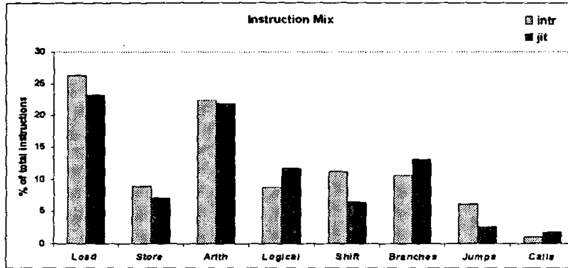


Figure 2: Instruction Mix

Figure 2 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 results are included in [19]. 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, a behavior not significantly different from traditional C and C++ programs [20]. 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 [21, 22]. We provide information on indirect branch frequency in interpreted and JIT execution 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, and due to high frequency of virtual function calls), 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.

4.2 Branch prediction

Benchmark	2-BIT	BHT	Gshare	GAp
comp(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 2: 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. Indirect jumps and virtual function calls that are abundant in Java applications, especially in the interpreted mode of execution, can complicate the task of predicting the outcome of these control instructions. Table 2 illustrates the branch misprediction rates for four different branch prediction schemes including a simple 2-bit predictor [23], 1 level Branch History table (BHT) [23], Gshare [24] and a two-level predictor indexed by PC (described as GAp by Yeh and Patt [25]. The first level predictor has 2K entries and the second level predictor (where applicable) has 256 entries. The Branch Target Buffer (BTB) contains 1K entries. The Gshare predictor uses 5 bits of global history. The branch predictors get sophisticated as we go from left to right in Table 2. 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 indirect jumps used to implement virtual method calls and 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 [22], [26]) should be used.

4.3 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 3 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. (These parameters were chosen to keep the caches similar to those on state-of-the-art microprocessors.) Instruction cache performance in interpreted mode is extremely 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. Locality studies of Java bytecodes [27] illustrates that less than 20% of distinct bytecodes account for 90% of the dynamic bytecode stream in all the programs. In fact, the aforementioned study showed that 15 unique bytecodes accounted for 60% to 85% of the dynamic bytecode stream of the SpecJVM98 programs, and 22 to 48 distinct bytecodes constituted 90% of the bytecode stream. 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 [20]) 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 1), the I-cache misses are more dominant.

Benchmark	I-Cache		D-Cache	
	Refs	Misses	Refs	Misses
compress(intr)	10425M	84398	5365M	21M
(jit)	1385M	218101	751M	43M
jess(intr)	259M	179545	81M	2.3M
(jit)	188M	616962	45M	3.7M
db(intr)	86M	82873	24M	751592
(jit)	75M	232046	17M	1.2M
javac(intr)	199M	140239	59M	1.8M
(jit)	167M	469143	40M	3.3M
mpeg(intr)	1314M	92439	544M	1.9M
(jit)	264M	355896	101M	3.2M
mtrt(intr)	1531M	252370	521M	8.6M
(jit)	942M	522692	230M	16M
jack(intr)	2668M	124563	1033M	11M
(jit)	986M	1.0M	298M	15M

Table 3: Cache Performance for the SpecJVM98

This table shows the number of references and misses for the instruction and data cache. Cache size= 64K bytes, block size= 32 bytes, I-cache is 2-way and D-cache is 4-way set-associative. M - indicates million.

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 3. 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 3).

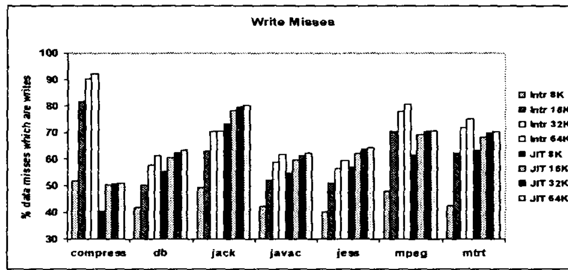


Figure 3: Percentage of Data Misses that are Writes. Cache used is direct mapped with line size of 32 bytes

Figure 4 illustrates the average 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.

One noticeable fact in Table 3 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.

We have further isolated the cache behavior during the translation part and the rest of the JIT execution. The cache behavior of the translate portion is illustrated in Figure 5. The translation related instruction cache misses contribute to around 30% (except *jack* and *mtrt*) of all the instruction cache misses. Also, it was found that the instruction cache locality is better within the translate routines (miss rate of 1.1% for *db*) than the rest of the JIT (miss rate of 1.5% for *db*). This is due to the high code reuse exhibited by the code generation routines within translate. The same methods are invoked for translating specific bytecodes that occur many times. On the other hand, the data 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.

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 5). 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 [27]. An optimization to lower the penalty of write misses during code generation and installation is discussed later in Section 6.

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 6. 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 6. 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.

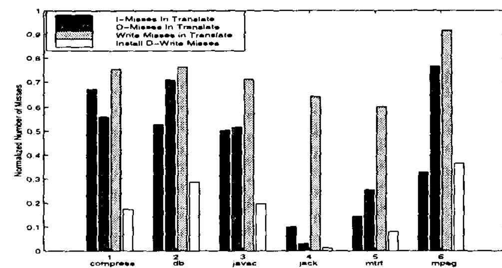


Figure 5: Cache Misses within Translate Portion. Cache configuration used : 4-way set associative, 64K DCache with a line size of 32 bytes and 2-way set associative, 64K ICache with a line size of 32 bytes.

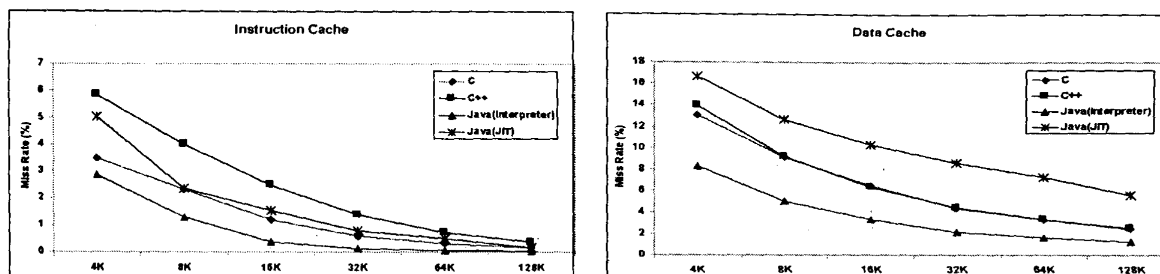


Figure 4: Miss rates for C, C++ and Java workloads for (i) Instruction Cache (ii) and Data Cache. The miss rates for C and C++ are obtained from studies by Calder et. al

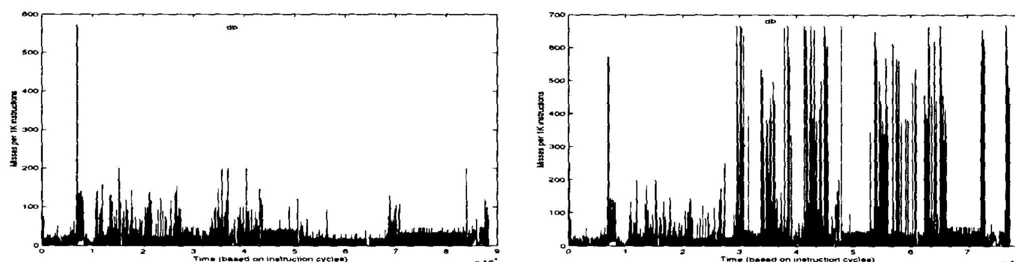


Figure 6: Miss rate variation in D-Cache for *db* during code execution in (i) Interpreter Mode and (ii) JIT Compiler Mode. Cache configuration used : 4-way set associative, 64K Cache with a line size of 32 bytes.

4.3.1 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 7 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 misses in instruction caches, however, data caches display a different behavior (illustrated in Figure 8). 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 can be explained using method locality and bytecode size information. Prior research on method locality and size distribution [27] 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 [12], 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 [18]. 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.

4.4 ILP Issues

Instruction level parallelism support is becoming increasingly predominant in current microprocessors. An investigation of ILP issues for Java runtime systems can not only enlighten us on the suitability of this support on general-purpose microprocessors for Java programs, but can also help us incorporate such support in specialized Java processors. It should be remembered that these general purpose processors have not really been tuned for the object-oriented nature of Java or the stack architecture of the JVM. In order to draw some insight into ILP related issues, we have simulated the execution of SpecJVM98 programs on a superscalar processor simulator [2].

Figure 9 illustrates the rate of execution of the benchmarks in interpreted and JIT compiler modes and

[2]The simulation configurations used are described in [28].

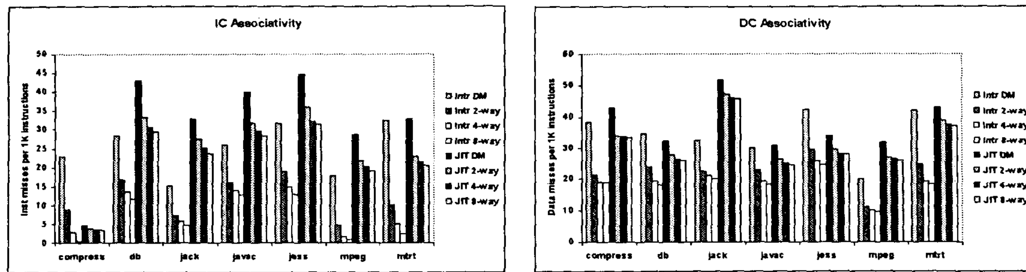


Figure 7: Effect of increasing the associativity of the (i) instruction cache and (ii) data cache. Cache configuration used: 8K Cache with 32 byte line size and associativity of 1, 2, 4 and 8.

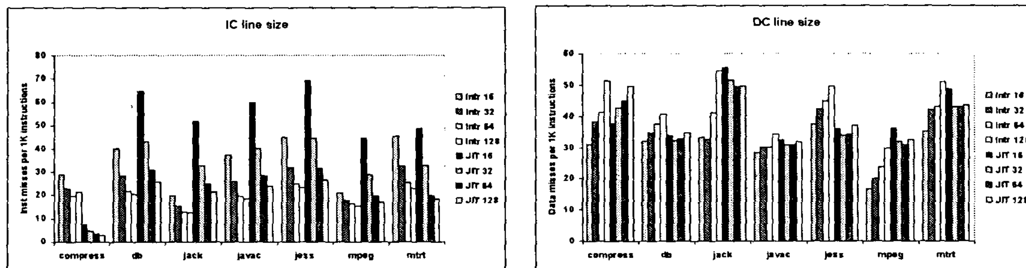


Figure 8: Effect of changing line size on the (i) instruction cache and (ii) data cache. Cache configuration used: 8K direct mapped Cache with line sizes of 16, 32, 64 and 128 bytes.

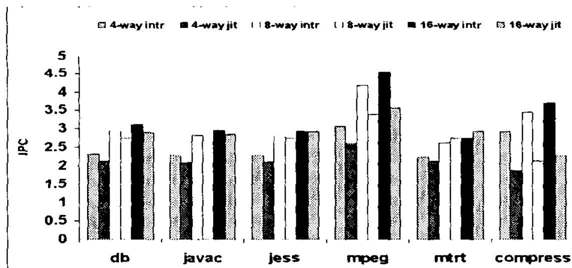
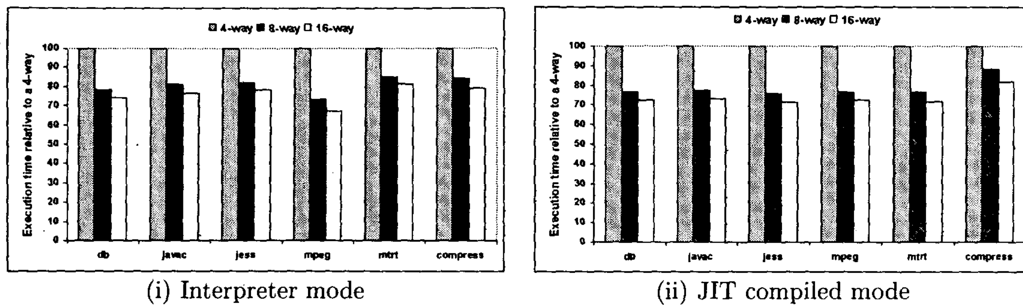


Figure 9: Instruction execution rate of the SpecJVM98 benchmarks

points to the available parallelism in the programs. A noticeable observation is that the Instructions Per Cycle (IPC) is higher in the interpreter mode in comparison to the JIT compiler mode. We attribute this to two reasons: (i) the superior cache performance of interpreter mode compared to JIT compiler mode (as observed in Section 4.3) and (ii) the inherent unoptimized nature of interpreted code which leads to significant overlap and concurrency while using dynamic scheduling and optimization techniques in modern microprocessors. As observed in Section 4.2, the JIT compiler has a superior branch predictor performance compared to the interpreter. This enables the JIT to overcome some of the deterioration in IPC performance caused by poor cache performance. Hence, the JIT compiler mode achieves an IPC not significantly worse than that in interpreter mode. The corresponding normalized execution times for CPUs with different issue widths are shown in Figure 10.

While the absolute IPC of the interpreter mode is better, the progressive improvement with larger issue processors becomes smaller for the interpreter as can be observed for *jess* and *mtrt* in Figure 9. These results can be attributed to the poor target prediction for the switch construct used in the interpreters. This construct is used to jump to the corresponding interpretation code for the bytecode to be executed. The IPC is initially higher for the interpreter as the instructions for interpreting a single bytecode could be optimized by removing false dependencies imposed by the stack based processing. However as issue width increases some of the simple bytecodes such as *iadd*, *isub* [1] would be completely executed in a cycle and fetching the next bytecode is a bottleneck. This bottleneck is caused by the inaccuracy in predicting the target of the switch construct. This observation indicates an interesting possibility for improving the structure of the interpreters. The picoJava processor [11] employs a folding optimization in which commonly occurring sequences of bytecodes of length 2, 3, and 4 are combined in a single execution cycle. An interpreter



(i) Interpreter mode

(ii) JIT compiled mode

Figure 10: Performance for SpecJVM98 programs running in (i) Interpreter mode and (ii) JIT compiled mode normalized to performance on a 4-way configuration.

code that identifies these sequences of bytecodes can mitigate the effect of inaccurate target prediction and scale better. We also expect the scaling of interpreters to improve with architectural support features such as trace caches to exploit cache locality of the instructions and indirect branch predictors to improve the target prediction of switch constructs. This raises an interesting question: Can we actually do more efficient software interpretation with these enhancements as compared to hardware interpretation performed by stack-based Java Processors[11, 29]?

5 Synchronization Issues

Having examined the hardware issues that are important in the interpretation, or translation-execution parts of the JVMs, we next look at another important issue that can help reduce some of the time spent in the execution/interpretation components shown in Figure 1. Runtime support in the form of synchronization mechanisms, as well as class loading and resolution can play a crucial role in execution/interpretation overheads (they are common to an interpreter as well as a JIT-based execution). We examine one of these issues, namely synchronization, which is a common operation in most Java programs [30]. Even if a user-program is single-threaded, the class libraries create special threads to handle finalization [1] and weak references [31]. Hence, it is important for the runtime system to implement synchronization efficiently i.e. provide a scalable solution when several threads contend for the synchronization mechanism while minimizing the overheads in the uncontended cases.

Java provides the monitor synchronization construct to user programs for guarding accesses to shared data. One has to keep space and time efficiency, together with scalability in multiprocessor environments, in mind when implementing this construct in the JVM. Sun's JDK 1.1.6 [4] targets a space-efficient implementation, but is not very scalable or time-efficient. It maintains an auxiliary open hashing data structure (with 128 buckets), called a monitor cache, that leads to all the (monitor) locks associated with the objects. When a thread invokes a method in a synchronized object, the handle address of the object is used to hash into the monitor cache to find the lock associated with the object (which may require a linked-list traversal within the corresponding bucket). Further, the entire monitor cache data structure is itself locked before the lock for the object can be located. While such an implementation is space-efficient (proportional to the actual number of monitors used/referenced), there is a lot of overhead in accessing a lock for an object, which can become significant specifically in the uncontended case. We can classify synchronized object accesses by a thread into four cases: (a) attempting to lock an unlocked object, (b) attempting to lock an object that has already been locked by the same thread (recursive) with a recursion depth less than 256, (c) same as case (b) with a recursion depth greater than or equal to 256, and (d) attempting to lock an object already locked by another thread. Of these, only (d) is the contended case. The rationale behind the separation of the recursion depth into two separate cases (with a threshold of 256) will be explained shortly. An examination of the synchronization behavior of the SpecJVM98 benchmarks with respect to these four cases is shown in Figure 11 (i). It can be observed that most synchronization accesses in these benchmarks fall under cases (a) and (b). This indicates that the original JDK implementation of the monitor cache would be inefficient for these benchmarks.

There have been attempts to improve the performance of the synchronization operations using cooperative thread scheduling [32], or by making special provision for faster execution of single-threaded user programs [33, 34]. We will consider a more general alternative that modifies the original JDK implementation by trading some of the space saving for time. For instance, in addition to the monitor cache structure, we can devote 24 bits within each object for the purpose of locking. These 24 bits can be used to implement what is called a thin lock mechanism as proposed by Bacon et al. [35]. One of these bits indicates whether a thin or the traditional (fat) lock is being used. Of the remaining 23 bits, 8 are used to track the recursion level (up to 256) of the locking, and 15 bits are devoted to maintain the identifier of the thread currently holding the lock. In case (a) described above, the invok-

ing thread uses the thin lock, increments the recursion level, and sets its thread identifier in the 15 bits. If case (b) occurs (which is detected by comparing the thread identifier with that stored in the thin lock) when the thin lock bit is on, the thread simply increments the recursion level. If the recursion level exceeds 256, the thread indicates that the traditional fat lock (using the monitor cache structure) should be used for subsequent accesses. Cases (c) and (d) would default to using the traditional fat lock structure. Such a thin lock mechanism can help lower the overheads in the common cases ((a) and (b)). This modified synchronization mechanism has been compared with the original JDK implementation. The results for five of these benchmarks are shown in Figure 11. The reduction in monitor cache access overheads and avoidance of locking the monitor cache itself in the common case leads to nearly two fold improvement in the speed of synchronization operations. Synchronization operations amount to around 10-20% (while it may be lower as a percentage for the interpreter mode) of the overall execution time in the JIT mode.

It should be noted that the space overhead of the proposed thin lock implementation is rather high. Given that many objects typically tend to be small, the extra space that is required can make it a less attractive option. Further, we are adding the extra space for all objects, regardless of whether they are synchronized or not (only around 8% of objects are accessed in synchronized mode). One possible way of alleviating these problems is to use a two bit implementation of thin locks (without requiring 24 bits), and optimize only case (a) accesses. Figure 11, which suggests that more than 80% of synchronization accesses fall in the case (a) category, is another motivating reason for this alternative.

6 Architectural Implications

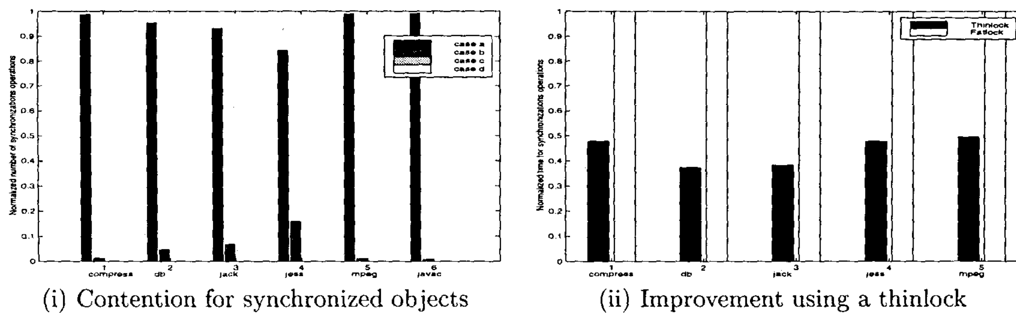
We have looked at a spectrum of architectural issues that impact the performance of a JVM implementation, whether it be an interpreter or a JIT compiler. In the following discussion, we briefly summarize our observations, review what we have learned from these examinations, and comment on enhancements 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 [22]) would be needed for the interpreted mode. The instruction level parallelism available in interpreted mode is seen to be higher than while using a JIT compiler. However, lack of predictability of indirect branches resulting from interpreter switch construct affects the performance as one moves to wide 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 are more dependent on the mode of execution (interpreter or JIT) rather than the object-oriented nature of Java programs.

Figure 1 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 4 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 4 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 1 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



(i) Contention for synchronized objects (ii) Improvement using a thinlock

Figure 11: Synchronization behavior of SpecJVM98

possible way of improving these applications, is to generated highly optimized code (spending a little more time to optimize code will not hurt these applications). 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. We have looked into one issue with respect to synchronization in this paper, and plan to examine the others in our future work apart from optimizing synchronization performance further.

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-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 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:

- 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 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, could be useful during dynamic code generation.
- The instruction level parallelism available in interpreted mode is seen to be higher than while using a JIT compiler. However, lack of predictability of indirect branches resulting from interpreter switch construct affects the performance as one moves to wide superscalar machines.
- Synchronization using a thin lock structure improves performance two-fold compared to using the monitor cache structures in JDK 1.1.6. It is also observed that a single bit per object would be sufficient to implement the thin lock mechanism to minimize space overheads and still speed-up 80% of all synchronization operations in the SpecJVM98 benchmarks.

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 optimiza-

tion, such as those undertaken in [36]. 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 and focus on the implications of the synchronization mechanism on performance. 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 runtime libraries. This paper has looked at only a small subset of issues with respect to supportive architectural features for Java, and there are a lot of 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, and by Sun Microsystems, Dell, Intel, Microsoft and IBM. We acknowledge the comments and suggestions by Doug Burger on a draft of this paper.

References

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

[2] 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.

[3] 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.

[4] "Overview of Java platform product family ." http://www.javasoft.com/products/OV_jdkProduct.html.

[5] 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.

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

[7] 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.

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

[9] N. Vijaykrishnan, *Issues in the Design of a Java Processor Architecture.* PhD thesis, College of Engineering, University of South Florida, Tampa, FL 33620, July 1998.

[10] T. Newhall and B. Miller, "Performance measurement of Interpreted Programs," in *Proceedings of Euro-Par'98 Conference*, September 1998.

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

[12] 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.

[13] "SPEC JVM98 Benchmarks." http://www.spec.org/osg/jvm98/.

[14] "Kaffe Virtual Machine." http://www.transvirtual.com.

[15] 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.

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

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

[18] 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.

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

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

[21] 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.

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

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

[24] S. McFarling, "Combining branch predictors," Tech. Rep. WRL Technical Note TN-36, June 1993.

[25] 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.

[26] 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.

[27] 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.

[28] R. Radhakrishnan, N. Vijaykrishnan, L. John and A. Sivasubramanium, "Architectural issues in java runtime systems," Tech. Rep. TR-990719, 1999. http://www.ece.utexas.edu/projects/ece/lca/ps/tr990719.pdf.

[29] A. Wolfe, " First Java-specific chip takes wing ," *Electronic Engineering Times*, 22 April 1997. http://www.techweb.com/wire/news/1997/09/0922java.html.

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

[31] S. M. Inc., "Java2 on-line documentation." http://java.sun.com/products/jdk/1.2/docs/api/index.html.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.