Characterization of Java Applications at Bytecode and Ultra-SPARC Machine Code Levels

Ramesh Radhakrishnan, Juan Rubio and Lizy Kurian John
Electrical and Computer Engineering Department
The University of Texas at Austin, Austin, Texas 78712
{radhakri,jrubio,ljohn}@ece.utexas.edu

Abstract

This paper identifies some of the most important execution characteristics of a recent suite of Java benchmarks (SPEC JVM98) from a bytecode perspective and while running in an interpreted environment on the Sun Ultra SPARC-II. We instrumented the Java Virtual Machine (JVM) to obtain detailed traces and developed a Java bytecode analyzer environment called `Jaba` *to characterize the applications at the bytecode level. Utilizing* `Jaba` *and SPARC profiling tools, we analyze bytecode locality, instruction mix and dynamic method sizes. It is observed that less than 45 out of the 250 Java bytecodes constitute 90% of the bytecode stream. A tri-nodal distribution with peaks of 1, 10 and 27 bytecodes is observed for method size across all benchmarks in the JVM98 suite. For most of the applications one bytecode is seen to translate into approximately 25 SPARC instructions.*

1 Introduction

Java is a general-purpose, concurrent, class based, object-oriented programming language, specifically designed to have as few implementation dependencies as possible. The portability of the code along with its relatively small size have made it an attractive option for the development of applications for heterogeneous environments such as embedded and networked systems. The basic objective of this research is to investigate the characteristics of the Java programming language and its execution in the interpreted environment on modern processor architectures. Measuring the performance of an interpreted language like Java is not relatively straight forward, since there are two ways in which one can characterize the performance. On the one hand we can look at the bytecode level and measure the locality and other characteristics of the bytecodes. On the other hand, we can measure the performance by running these bytecodes through an interpreter, in which case the performance observed is mainly the effectiveness of the interpreter in translating the bytecodes to machine level instructions. In this paper we present some of the most important execution characteristics of a recent suite of Java benchmarks, the SPEC JVM98. The benchmarks are analyzed both from a bytecode perspective and at the SPARC machine code level. Although Java performance studies have been performed by Vijaykrishnan et. al [1], Romer et. al. [2] and Newhall et. al. [3], most of the past studies were based on small, often synthetic Java programs.

2 Experimental Methodology

In order to understand the behavior of a Java application it is necessary to first find a mechanism to monitor the stream of bytecodes that are being executed at a given time. We developed a mechanism that allows the capture of bytecode traces which can then be analyzed off-line to identify specific characteristics of the execution. Our working environment consisted of an Ultra SPARC-II running Solaris 2.6 and the JDK (Java Development Kit) 1.1.6.

The SPEC JVM98 benchmark suite contains eight different tests, out of which five tests are either real applications or derived from real applications that are commercially available. SPEC JVM98 allows users to evaluate performance of the hardware and software aspects of the JVM client platform. On the software side, it measures the efficiency of the JVM, the just-in-time (JIT) compiler, and operating system implementations. On the hardware side, it includes CPU, cache, memory, and other platform-specific features. A summary of the benchmarks is provided in Table 1. The benchmarks can be run using three data sets *s1, s10* and *s100*. We use the data set *s1* for our studies since the static size of the programs were seen to remain approximately the same across the different data sets (data is presented in Section 3.1).

In order to obtain adequate information from the execution of a Java application we instrumented the JVM provided with Sun's JDK 1.1.6. The tracing mechanism was then validated using Sun's tracing JVM on a set of synthetic benchmarks. The analysis of the traces is done off-line to minimize the overhead on the tracing JVM. The analyzer was developed based on the Appli-

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

Table 1: Description of the SPEC JVM98 Benchmarks

Bytecode	**Description**
Loads	loads data from local variables into stack
Stores	the counterpart of loads
Stack	allows for push and pop of operands into the stack, as well as duplication and swap of data in the stack
Constant pool	allocation of elements in the constant pool
ALU	arithmetic (both integer and floating point) and logic instructions
Branches	tests for a condition and changes the program counter based on it
Jump	non-unitary increments or decrements to the program counter
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

Table 2: Classification of Bytecodes

cation Programmer Interface (API) provided by *Shade* [4]. The result was **Jaba**, a **Ja**va **b**ytecode **a**nalyzer library. `Jaba` allows researchers to develop analyzers in an efficient way to study traces generated by the tracing JVM.

The *Shade* tool [4] from Sun is used to obtain SPARC traces for the JVM98 programs (to study the behavior of SPEC JVM98 at the SPARC machine code level). Existing analyzers and new analyzers are used to study the instruction mix of the Java applications. Since Java runs in an interpreted environment, these measurements reflect the combined properties of the interpreter (JVM) and the application (SPEC JVM98).

3 Analysis and Results

3.1 Bytecode Level

The first characteristic we looked at is the bytecode instruction mix. The JVM instruction set, being stack-oriented, is significantly different from conventional register based architectures. Hence first we identified a basic classification for the instruction types: Table 2 shows the classification of the bytecode instructions based on the operation that they perform.

Based on the previous classification, the bytecode instruction mix of all programs in the SPEC JVM98 benchmark suite is obtained and presented in Table 3. The total bytecode count ranged from 2 million for *db* to approximately a billion for *compress*. Most of the benchmarks showed similar distributions for the different instruction types. As seen, most of the instructions are of type *load*, which on the average accounts for 35.5% of the total number of bytecodes executed. The next most frequent instructions are *constant pool* and *method calls* with average frequencies of 21% and 11% respectively. From an architectural point of view, this means that in the Java run-time environment, most of the operations consist of transfers of data elements to and from the memory space allocated for local variables and the one allocated for the stack. Comparing it 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, the JVM places greater stress on the memory system. Consequently, we expect that techniques such as instruction folding proposed in [5] for Java processors and instruction combining proposed in [6] for JIT compilers can improve the overall performance of Java applications.

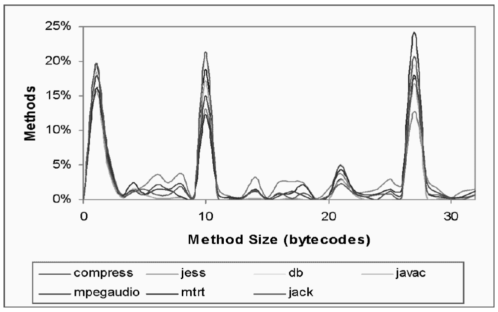

Figure 1: Dynamic Method Size

The second characteristic we studied was the dynamic size of a method. Invoking methods in Java is expensive, as it requires the setting up of an execution environment and a new stack for each new method [7]. Figure 1 shows the method sizes that was seen for the different benchmarks. A tri-nodal distribution is observed, where most of the methods are either 1, 10 or 27 bytecodes long. This seems to be a characteristic of the runtime environment and can be attributed to a frequently used library. However the existence of methods of just one bytecode, indicates the presence of wrapper methods to implement features that exist in the Java language like private and protected methods or *interfaces*. It is therefore a characteristic of the Java language and not of the JVM specification.

Further analysis of the traces show that a few unique bytecodes constitute most of the dynamic bytecode stream. In most benchmarks, fewer than 45 distinct bytecodes constitute 90% of the executed bytecodes. The list of these bytecodes that dominate the dynamic bytecode trace is available in [8]. It is observed that memory access and memory allocation related byte-

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%
Table	0.0%	0.3%	0.3%	0.7%	0.0%	0.7%	0.0%
Total Bytecodes	954990234	8126332	2035798	5958654	115748387	50683565	175740325

Table 3: Instruction Mix at the Bytecode level

Benchmark	# bytecodes
jess	48
db	45
javac	45
mpegaudio	36
mtrt	39
jack	22

Table 4: Number of distinct bytecodes that account for 90% of the dynamic count

codes dominate the bytecode stream of all the benchmarks. This also hints that if the instruction cache can hold the JVM interpreter code corresponding to these bytecodes, the instruction cache performance will be good.

Another bytecode characteristic we studied 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*, which are presented in Table 5. 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. Table 5 also shows that the static size of the benchmarks remain constant across the different data sets, although the dynamic instruction count increases for the bigger data sets.

3.2 SPARC Machine Code Level

Figure 2 shows the instruction mix for the benchmarks. The instructions are broken into control transfer, load and store instructions. Loads are observed to have a high occurrence in all the benchmarks. It must be kept in mind that these loads include loads performed by the Java application to load its data in addition to the loads performed by the JVM to fetch each bytecode during interpretation.

The control transfer instructions can be further classified into branches, jump and call instructions. Figure 3 shows the percentage of these classes of instructions (branch, jump and call). The JVM interpreter is implemented using a large switch statement and every bytecode results in a control transfer to the corre-

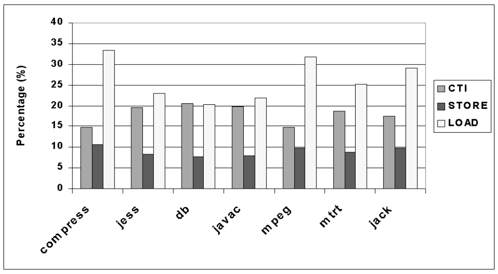

Figure 2: Breakdown of the Instruction Mix into Control Transfer Instructions, Loads and Stores.

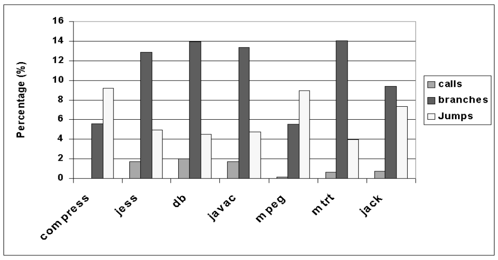

Figure 3: Breakdown of the Control Transfer Instructions into Branches, Calls and Jumps.

sponding case where the routine to decode the bytecode is defined by the JVM. Table 6 shows the number of SPARC instructions that are generated per bytecode. One bytecode is seen to result in an average of 25 SPARC machine level instructions (10 to 42 in the various benchmarks). We also show the number of control transfer instructions, loads and stores that were generated per bytecode in Table 6. It is seen that one bytecode results in an average of 4.7 control transfer instructions and 8 memory access instructions.

4 Conclusions

The Java language paradigm is gaining wide acceptance as a mechanism to transmit portable code over a network. The JVM is the component of the Java language specification responsible for the portability of the code. In this paper, we identified several important execution characteristics of a suite of Java bench-

	s1		s10		s100	
benchmarks	calls	methods	calls	methods	calls	methods
compress	17330744	577	18170275	578	14566857	449
db	65379	642	1610941	645	91753107	658
jack	2318110	1230	4621508	1233	39172145	1240
mpegaudio	954605	843	8289656	846	93046042	844
jess	414349	1222	5697628	1313	95957670	1375
javac	213243	1384	2515940	3142	54503910	3325
mtrt	1906112	781	7031487	785	71168982	796

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

Benchmarks	Inst	CTI	loads	stores
compress	10.91	1.60	3.64	1.15
jess	31.93	6.24	7.32	2.66
db	42.64	8.71	8.65	3.26
javac	33.37	6.58	7.32	2.64
mpegaudio	11.35	1.65	3.61	1.09
mtrt	30.22	5.63	7.61	2.67
jack	15.18	2.63	4.42	1.46

Table 6: Instructions Executed per Bytecode

The total dynamic instructions (Inst), control transfer instructions (CTI), loads and stores generated for each bytecode is shown.

marks both from a bytecode perspective and at the SPARC machine code level. The analysis at bytecode level points to processor features required in a Java processor or in a software JVM, while the analysis at the SPARC machine code level indicates the need for (or against) specialized mechanisms for general purpose processors interpreting Java code. The bytecode analysis also helps to explain many observations at the machine code level. The major observations based on our investigation are summarized below.

1. It is observed that less than 45 out of the 250 Java bytecodes constitute 90% of the bytecode stream. Optimizations targeted at these bytecodes may prove to be effective, although Amdahl's law has to be considered.

2. One bytecode was seen to translate into approximately 25 SPARC instructions on the average. On an average, each bytecode results in 8 memory access instructions and more than 4 control transfer instructions.

3. The instruction mix showed that there was a high frequency of memory references. The average percentage of loads was 35% at the bytecode level and 26% at the SPARC machine code level. The large percentage of load instructions stresses the importance of a good memory system design in Java systems.

4. The most common dynamically invoked methods were seen to be either 1, 10 or 27 bytecodes long. It was also seen that 45% of all dynamic methods were less than 9 bytecodes or 16 bytes long.

We believe that this paper provides some valuable insight into the behavior of Java applications on modern processors. The analysis at the SPARC machine code level is the combined effect of the JVM interpreter and the Java applications and the observations should not be interpreted as being solely due to Java language features.

Acknowledgments

We appreciate the support by Sun Microsystems on this project. A special acknowledgment is made to the Java Development team for giving us access to the source code for the JDK 1.1.6. The authors are supported by the National Science Foundation under Grants EIA-9807112 and CCR-9796098, the State of Texas Advanced Technology Program grant #403, and by Dell, Intel, Microsoft and IBM.

References

[1] N.Vijaykrishnan, N.Ranganathan, and R.Gadekarla, "Object-Oriented Architectural Support for a Java Processor," in *Proceedings of ECOOP'98, the 12th European Conference on Object-Oriented Programming*, 1998.

[2] 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.

[3] T. Newhall and B. Miller, "Performance Measurement of Interpreted Programs," in *Proceedings of Euro-Par*, 1998.

[4] Robert F. Cmelik and David Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling, SMLI TR-93-12," tech. rep., Sun Microsystems Inc, 1993.

[5] J. O'Conner and M. Tremblay, "PicoJava-I: The Java Virtual Machine in Hardware," in *Proceedings of Micro*, March 1997.

[6] HotSpot: A new breed of virtual machine, http://www.javaworld.com/jw-03-1998/jw-03-hotspot.html?030998.

[7] F. Y. T. Lindholm, *The Java Virtual Machine Specification.* Addison Wesley, 1997.

[8] R. Radhakrishnan, J. Rubio and L. John, "Chacterization of Java Applications at Bytecode and Machine Code Levels," Tech. Rep. TR-LCA080599. http://www.ece.utexas.edu/projects/ece/lca/ps/TR-LCA080599.ps.