Execution Characteristics of Just-In-Time Compilers

Technical Report - TR-990717-01

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

†Laboratory for Computer Architecture
Electrical and Computer Engg Dept
The University of Texas at Austin

‡Dept. of Computer Science and Engineering
Pennsylvania State University
University Park, Pennsylvania
{radhakri,jrubio,ljohn}@ece.utexas.edu
vijay@cse.psu.edu

Abstract

Just-In-Time (JIT) compilers interact with the Java Virtual Machine (JVM) at run time and compile appropriate bytecode sequences into native machine code. Loading and compilation time penalties are incurred at run time. However, the lack of need for lengthy translations yields benefits if methods are reused. In this paper, we provide a quantitative characterization of the execution behavior of the SPEC JVM98 programs, in interpreter mode and using JIT compilers. There has not been an effort to study the interaction of the JIT compilation mode of java execution with architectural features. Such a study is important for the development and improvement of better compilers and virtual machines for executing Java. In our study we observe that the interpreter exhibits better instruction and data cache locality. We also observe differences in branch characteristics for the different modes of execution.

Keywords: Just-in-Time Compilers, Java Language, SPEC JVM98, Program Analysis, Cache Performance, Branch Prediction.

1 Introduction

Java [1] is a widely used programming language largely due to the portability and machine independent nature of bytecodes. These bytecodes can be interpreted, compiled to native code or directly executed on a processor whose Instruction Set Architecture is the bytecode specification. Interpreting the bytecodes, which is the standard implementation of the Java Virtual Machine (JVM) makes execution of programs slow.[1] To improve performance, JIT compilers interact with the JVM at run time and compile appropriate bytecode sequences into native machine code. When using a JIT compiler, the hardware can execute the native code, as opposed to having the JVM interpret the same sequence of bytecodes repeatedly and incurring the penalty of a relatively lengthy translation process. This can lead to performance gains in the execution speed, unless methods are executed less frequently. The time that a JIT compiler takes to compile the bytecodes is added to the overall execution time, and could lead to a higher execution time than an interpreter for executing the bytecodes if the methods that are compiled by the JIT are not invoked frequently.

The JIT compiler performs certain optimizations when compiling the bytecodes to native code. Since the JIT compiler translates a series of bytecodes into native instructions, it can perform some simple optimizations. Some of the common optimizations performed by JIT compilers are data-flow analysis, translation from stack operations to register operations, reduction of memory accesses by register allocation, elimination of common sub-expressions etc. The higher the degree of optimization done by a JIT compiler, the more time it spends in the execution stage. Therefore a JIT compiler cannot afford to do all the optimizations that is done by a static compiler, both because of the overhead added to the execution time and because it has only a restricted view of the program.

This raises the question - when does it make sense to execute Java programs using a JIT compiler? Do all Java applications benefit from a JIT compiler? It is common knowledge that only those applications that make heavy reuse of its methods would benefit from JIT execution. For those programs that spend a lot of time in I/O or garbage collection, using an interpreter would provide comparable if not better execution performance. In non-computational programs where methods are not executed repeatedly, a JIT compiler which would optimize every method it comes across would spend useless time in optimizing methods that are rarely executed again. In such cases an off-line compiler (where the compilation process is separate from the execution

[1] In current implementations of the JVM (JDK 1.1.6 and higher), the JIT compiler is invoked by default

process) would provide faster execution compared to a JIT or interpreted execution [4]. Other modes of execution for Java include compilers that translate the bytecodes to an intermediate language like C [17], and Java processors [11].

1.1 Motivation

There has been a lot of interest generated in the academia and industry research communities for improving Java performance. The SPEC CPU [9] benchmarks have been a common platform for research in architecture and compiler studies, and a means to evaluate the benefits attained from the diverse techniques that are proposed from these studies. Similarly to provide a common platform for Java research, the SPEC consortium released the SPEC JVM98 benchmark suite [8] in August 1998. This would provide the Java research community with the opportunity to use meaningful applications to test their work, as opposed to using small synthetic benchmarks. Despite the limitations of SPEC92 and SPEC95 benchmarks, they helped the architecture and compiler community to evaluate each others work easily and meaningfully [16]. However, no such characterizaton for the SPEC JVM98 benchmark suite has been performed yet. We study in detail the SPEC JVM98 benchmark programs, running them using an interpreter and a JIT. This would provide a base reference for future research that targets hardware or compiler optimizations for Java. We also look at the architectural impact of using the different modes of execution, since execution time is not determined solely by the number of instructions. As mentioned in [3], a significant amount of the execution time can be attributed to ineffective use of the microarchitecture mechanisms, like the cache and branch prediction hardware. The results from such analysis can provide heuristics for newer techniques like the HotSpot technology [12].

2 Related Work

Previous research has mainly concentrated on analyzing the interpreted behavior of Java execution. Romer, et.al. investigated the interaction of interpreters with modern architectures and demonstrated that interpreter performance is relatively independent of the application [6]. They also concluded that specialized hardware support for interpreted environments was not essential and that performance improvements could be achieved through software means. Newhall and Miller developed a tool based on a performance measurement model that explicitly represents the interaction between the application and the interpreter [10]. This tool measures the performance of interpreted Java applications and is shown to help application developers tune their code to improve performance. Hsieh, et.al. studied the impact of interpreters and offline Java compilers on microarchitectural resources such as the cache and the branch predictor [3]. They attribute the inefficient use of the microarchitectural resources by the interpreter as a significant performance penalty. It is also observed in their work that an offline bytecode to native code translator is a more efficient Java execution model for utilizing the caches and branch predictors. Vijaykrishnan et.al. reported the behavior of Java bytecode execution for a Java processor and proposed architectural features based on them [13]. However, there has not been an effort to study the interaction of the Just-In-Time compilation mode of Java execution with the architectural features. This study is important to provide hints in the development and improvement of HotSpot Java compilers [12]. The HotSpot compilers choose between the JIT and interpreter mode of execution selectively to improve performance. Hence understanding the underlying characteristics of both the JIT and the interpreter and their interaction with the architecture is essential.

3 Experimental Methodology

We use the SPEC JVM98 benchmark suite [8] to examine the performance of the different approaches for running Java programs. The SPEC JVM98 benchmark suite contains eight different tests, five of which are either real applications or derived from real applications that are commercially available. SPEC JVM98 allows users to evaluate performance for the hardware and software aspects of the JVM platform. On the software side, it measures the efficiency of 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 are the arguments used to specify the data set to be used when running the benchmarks). All our benchmarks were run at the command line prompt, as opposed to being run from an applet in order to capture the the program characteristics.

The Java interpreter used is the Sun JDK Version 1.1.3 running under SunOS 5.6 on an UltraSPARC-II machine. The JIT compilers used are `sunwjit` and `sunwjit_opt`,[2] both available from the Sun website [7]. `sunwjit_opt` is a more optimized JIT as compared to

[2]The `sunwjit` is the default JIT used in JDK 1.1.3. and `sunwjit_opt` was a pre-release version. The default JIT used in the JDK 1.1.6 and JDK 1.1.7 platform is `sunwjit_opt`. We believe that our observations regarding the architectural impact of JIT compilation would also hold for the JDK 1.2 platform.

Benchmark	Description
compress	A popular LZW compression program.
jess	A Java version of NASA's popular CLIPS rule-based expert systems.
db	Data management benchmarking software written by IBM.
javac	The JDK Java compiler from Sun Microsystems.
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 real parser-generator from Sun Microsystems.

Table 1: Description of the SPEC JVM98 Benchmarks

`sunwjit` and does more aggressive code inlining and common sub-expression elimination. This additional functionality in `sunwjit_opt` could increase its compilation time by up to 25%.

We used the Shade binary instrumentation tool [5] to obtain traces while running the benchmarks under different execution modes. Our simulation model is based on a modern, dynamically scheduled, superscalar processor microarchitecture. It consists of a 64K first-level I-Cache with 32 byte blocks and 2-way associativity. The first-level D-Cache is also 64K with 32 byte blocks and 4-way associativity. The second-level cache is a 1M direct mapped cache with a block size of 128 bytes. We model different branch predictors to compare the predictability of branches for the different modes of execution. A 256-entry 2-level (GAp) Predictor and the Gshare predictor using 5 bits of global history are used, along with simple prediction schemes such as Backward Branch Taken Forward Not Taken (BTF!T), a Global 2-Bit predictor and a One level branch history table which are used to evaluate the predictability of branches.

4 Instruction Usage

Execution characteristics of object-oriented languages like C++ and Java have been observed to be different from that of procedural languages like C and FORTRAN [2] [3]. Object-oriented languages are known to make more calls and execute higher number of indirect branches than other languages [2]. A hybrid language like C++, which is not fully object-oriented has shown significant differences in terms of basic block size, number of instructions per function and call frequency with respect to C. One finds significant differences in the above cases since one is measuring different languages. We expect to see significant differences between the instruction usage when Java is interpreted, and when it is executed using a JIT compiler. This is because execution behavior is a function not only of the source language characteristics, but also of the compiler technology and execution model.

Benchmark	Dynamic Instructions	Control Transfer Inst			Computational Inst			Memory	
		Branches	Calls	Jumps	Arith	Logical	Shift	Load	Store
compress(intr)	10425544771	5.57	0.01	9.17	20.15	5.48	14.95	33.35	10.56
(jit)	1385557173	5.61	0.23	2.50	13.49	10.31	7.71	40.61	6.77
(opt)	1313126925	6.59	0.33	2.74	15.28	10.48	8.47	34.90	7.41
jess(intr)	259702575	12.85	1.72	4.96	22.94	10.27	9.04	22.93	8.34
(jit)	188581319	15.21	2.87	3.44	23.78	13.05	5.34	16.61	7.59
(opt)	280097308	14.98	2.66	3.13	25.74	12.74	6.96	16.58	6.77
db(intr)	86844476	13.91	2.00	4.50	24.60	11.14	8.13	20.29	7.65
(jit)	75959798	15.73	2.73	3.14	24.98	13.19	5.54	16.04	7.61
(opt)	129192079	15.50	2.53	2.88	27.71	12.59	7.04	16.23	6.38
javac(intr)	199254389	13.35	1.69	4.71	24.55	10.32	8.96	21.93	7.89
(jit)	167556898	15.90	2.50	2.91	25.52	13.13	5.62	16.55	7.35
(opt)	280372380	15.14	2.40	2.78	26.80	12.81	7.77	16.57	6.43
mpeg(intr)	1314397268	5.54	0.13	8.94	21.55	4.97	15.74	31.80	9.65
(jit)	264879641	6.91	0.85	1.45	17.37	8.52	9.96	31.89	6.49
(opt)	310981176	10.21	1.36	1.92	21.99	10.17	8.53	24.01	7.58
mtrt(intr)	1531909956	14.04	0.67	3.96	22.54	11.50	9.38	25.18	8.85
(jit)	942593921	17.81	1.31	2.41	26.28	14.06	5.76	18.31	6.17
(opt)	953517625	17.79	1.43	2.49	26.63	14.01	6.33	18.14	5.49
jack(intr)	2668899901	9.36	0.70	7.31	20.64	7.80	12.57	29.10	9.62
(jit)	986682716	14.69	2.31	2.88	21.65	10.48	5.47	22.31	7.96
(opt)	1037794742	15.12	2.35	2.86	23.08	10.91	6.18	19.37	7.93

Table 2: Instruction Usage Characteristics for the SPEC JVM98

This table shows the instruction characteristics for the different benchmarks. All the numbers are obtained by running the benchmarks using -s1 option.

4.1 Instruction Counts

Table 2 shows a marked difference in instruction usage for the interpreter and JIT mode of execution for the various benchmarks. This proves that dynamic instruction usage is heavily dependent upon the mode of execution, and varies between different modes of execution even for the same language. For each benchmark, an (intr) by its side indicates that the program was interpreted, a (jit) refers to using the `sunwjit` JIT compiler and (opt) indicates that the `sunwjit_opt` compiler was used to execute the particular program. The first column of Table 2 shows the total number of dynamic instructions executed by the processor for the different execution modes. The `sunwjit_opt` JIT compiler executes lower instructions compared to the other modes for *compress*, which is the largest benchmark. In three benchmarks *mpeg*, *mtrt*

Benchmark	Dynamic Instruction Count -s1	-s10	-s100
jess(intr)	259702575	1883530288	37278365517
(jit)	188581319	597550362	16917095279
(opt)	280097308	662273384	16320933197
db(intr)	86844476	2563156018	71418597792
(jit)	75959798	1954427130	39318582607
(opt)	129192079	1960581764	36489640798
javac(intr)	199254389	1686869000	44062969591
(jit)	167556898	1057087495	23210316813
(opt)	280372380	1239521688	24215526605

Table 3: Dynamic instruction count for all the three data sets

In this table the dynamic instruction counts for the different data sets are presented. -s1 is the smallest, -s10 is a medium size data set and -s100 forms the largest data set

and *jack* `sunwjit_opt` executes a smaller number of instructions than the interpreted mode, but higher than when using the `sunwjit` JIT. For the other benchmarks (which also happen to be the smallest benchmarks in the suite) using `sunwjit_opt` results in a higher number of instructions as compared to the interpreter. `sunwjit_opt` performs more aggressive optimizations as compared to `sunwjit` and therefore the overhead associated is more pronounced for smaller programs, as can be seen from the number of instructions executed. However, for a large program like compress (which on profiling showed that it makes a lot of virtual calls to methods which are defined in the same compiled unit), compilation of methods would result in efficient code generation. The `sunwjit` compiler which performs more conservative optimizations executed lesser instructions than the interpreter for all benchmarks.

To find out if the benchmarks *jess*, *db* and *javac* would generate higher instructions across all data sets when used with the `sunwjit_opt` JIT compiler, we analyzed its instruction count for the remaining data sets. The results for this can be found in Table 3. It is seen that as the data set is increased the performance of both the JIT compilers improves. This could be due to the number of methods used in the bigger data sets not increasing linearly with the number of dynamic instructions. This would translate to better performance as there is more method reuse.

4.2 Instruction Mix

The rest of the columns in Table 2 breaks down the instructions into three main categories - control transfer instructions, computational instructions and memory instructions. We see pro-

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 4: Instruction Mix at the Bytecode level

nounced differences in the instruction type frequencies for the interpreter and the JIT compiler modes of execution. In the JIT execution mode the frequencies of branches and calls are higher. One sees a more significant difference in the percentage of jump instructions. A significant percentage of the jumps are register indirect instructions that are used to implement the switch statement in the interpreter. These jumps are also used in the SPARC architecture as a virtual function call. One sees a smaller number of such indirect jumps in the JIT mode of execution, since the JVM executes the compiled native code and does not use the interpreter loop for those instructions which are executed within the method. The JIT compiler also optimizes virtual calls by inlining those calls which it can prove to be non-virtual, thereby further lowering the number of indirect jump instructions. It is observed in [14] [15] indirect branch prediction accuracy is much lower than that of direct branches using current micro-architectural resources. Hence, the inlining in JIT compilers benefit not only from the reduction in method calls but also due to the decrease in branch prediction penalties. Dreisen et. al. mention that object oriented programs in C++ and Java use indirect branches with a much higher frequency than in SPECint95 programs. Here, we observe in the case of Java execution that the frequency also depends on the degree of optimizations. The performance tradeoffs due to decrease in indirect branches will be of more relevance to developers of aggressive HotSpot compilers. In such cases, de-optimization of inlined code is essential when new classes that are loaded invalidate the inlining. Hence, there would be a trade-off between the time saved due to the inlined execution and the de-optimization. Such optimizations are planned for future versions of the JDK1.2.

The next three columns gives the percentage of instructions that were used for arithmetic, logical and shift operations. The percentage of shift operations are more for the interpreter due to the concatenation of different stack words to form the operands. The last two columns in Table 2 show the percentage of loads and stores for the different execution modes. In the interpreter mode a large percentage of the instructions are used for stack operations. Most of these stack operations are loads and stores to local variables which result in a high number of loads and stores when translated to native code. In the JIT execution mode memory accesses are reduced by translating the stack operations into register operations and this is why one sees a significant reduction in the frequency of memory operations in the instruction mix.

Benchmarks	Instructions per Bytecode	CTI per Bytecode	Loads per Bytecode	Stores per Bytecode
compress(intr)	10.91	1.60	3.64	1.15
(jit)	1.451	0.121	0.589	0.098
(opt)	1.375	0.133	0.480	0.102
jess(intr)	31.93	6.24	7.32	2.66
(jit)	23.20	4.999	3.856	1.762
(opt)	34.46	7.163	5.716	2.336
db(intr)	42.64	8.71	8.65	3.26
(jit)	37.31	8.064	5.988	2.843
(opt)	63.46	13.28	10.30	4.050
javac(intr)	33.37	6.58	7.32	2.64
(jit)	28.12	5.997	4.654	2.068
(opt)	47.05	9.567	7.801	3.029
mpeg(intr)	11.35	1.65	3.61	1.09
(jit)	2.288	0.211	0.730	0.149
(opt)	2.687	0.363	0.645	0.204
mtrt(intr)	30.22	5.63	7.61	2.67
(jit)	18.59	4.007	3.406	1.148
(opt)	18.81	4.087	3.413	1.033
jack(intr)	15.18	2.63	4.42	1.46
(jit)	5.614	1.117	1.253	1.253
(opt)	5.905	1.201	1.144	0.469

Table 5: Number of Instructions Executed per Bytecode

The dynamic instruction count per bytecode is presented in this table. The number of control transfer instructions, loads and store instructions generated for each bytecode is also shown.

4.3 Instruction Mix at Bytecode Level

We instrument the JVM to gather information at the bytecode level. The instruction mix for the bytecodes are presented in Table 4. This information is useful for research in java specific hardware optimizations. Most benchmarks show similar distributions for the different instruction types. Most of the instructions are load instructions, which account for 35% of the total bytecodes on an average. The next most frequent instructions are constant pool and method calls. From an architectural point of view, this means that in the Java run-time environment most of the operations consist of transfer of data elements to and from the memory space allocated for local variables and stack, thus placing heavy stress on the memory system.

We then use the total count of bytecodes in each benchmark and calculate how many SPARC machine code instructions are used to execute a bytecode on an average. This data is presented in Table 5 and shows on the average how many SPARC instructions are needed in each benchmark to execute one bytecode. It is seen that the instructions per bytecode go down to as low as 1.37 for *compress*. Benchmarks *compress* and *mpeg* had the least numbers for the interpreter mode, and for the JIT mode of execution it goes down further. This shows that using a JIT removes almost all overhead in terms of instructions generated. For the other benchmarks the numbers are higher, varying from 5.6 for *jack* to 63.4 for *db*.

Further analysis of the traces show that only a few unique bytecodes constitute most of the dynamic bytecode stream. Fewer than 45 distinct bytecodes in most benchmarks constitute 90% of the executed bytecodes. Table 7 shows the number of distinct bytecodes that account for 90% of the dynamic bytecode trace. Table 6 lists the top 15 bytecodes that dominate the dynamic bytecode stream. It may be observed that memory access and memory allocation related byte codes dominate the byte code stream of all the benchmarks. This also hints that if the instruction cache can hold the JVM interpreter code corresponding to these byte codes, the instruction cache performance will be good.

4.4 Using profile information as a heuristic for using JIT

Further, we profile the benchmarks and obtain information about the unique number of methods and the total method calls made in each benchmark. The data obtained from profiling is presented in Table 8. The number of methods for each benchmark and the dynamic calls to the methods are listed in the table, for all three data sets. From this data, by looking at the counts for -s1 we can find correlation to the instruction counts presented in Table 2. When we divide the number of dynamic method calls by the static number of methods, we get the average number of times a method was reused. This number was $\approx$ 30,000 for compress, which shows the least reduction in instruction count when using the JITs. *Mtrt*, *jack* and *mpeg* which also shows a decrease in the instructions generated for the JIT mode of execution, has a method reuse from $\approx$ 1100 to 2440. For the benchmarks *jess*, *db* and *javac* which showed an increase

	db		jack		javac	
1	iload	10.67%	aload_0	20.34%	iload	9.21%
2	aload_0	8.38%	getfield_quick	15.62%	aload_0	7.19%
3	getfield_quick	7.37%	putfield_quick	9.04%	getfield_quick	6.61%
4	invokevirtual_quick	3.99%	dup_x1	8.52%	iinc	4.46%
5	aload	3.64%	iconst_1	4.77%	invokevirtual_quick	4.03%
6	bipush	3.54%	dup	4.62%	iload_1	3.82%
7	iadd	3.45%	aaload	4.41%	iadd	3.34%
8	iload_1	3.22%	ifgt	4.35%	aload	3.17%
9	istore	2.29%	ifnull	4.33%	iload_3	2.96%
10	if_icmplt	2.27%	isub	4.33%	if_icmplt	2.58%
11	iload_3	2.27%	iload	1.55%	iload_2	2.56%
12	iload_2	2.27%	invokevirtual_quick	1.26%	caload	2.55%
13	iinc	2.09%	aload_1	1.01%	ireturn	2.33%
14	iconst_1	2.07%	iload_1	0.86%	aload_1	2.13%
15	dup	1.90%	if_icmplt	0.81%	istore	2.05%
	Total	59.41%	Total	85.83%	Total	58.97%
	jess		**mtrt**		**mpeg**	
1	getfield_quick	10.66%	invokevirtual_quick	18.00%	iload	10.85%
2	aload_0	9.59%	getfield_quick	12.83%	faload	8.59%
3	invokevirtual_quick	6.45%	aload_0	6.42%	getfield_quick	6.90%
4	iload	4.78%	freturn	4.31%	aload_0	6.41%
5	aload_1	4.05%	iload	3.83%	bipush	5.44%
6	iload_2	2.99%	areturn	3.46%	fmul	3.82%
7	aload	2.90%	aload	3.23%	fadd	3.69%
8	iload_1	2.65%	aaload	3.00%	fload	3.58%
9	iload_3	2.11%	putfield_quick	2.58%	aload_1	3.16%
10	iinc	1.91%	aload_1	2.41%	aaload	3.10%
11	if_icmplt	1.85%	iload_1	2.35%	aload	2.95%
12	aaload	1.79%	iinc	2.14%	fstore	2.84%
13	getfield_quick_w	1.73%	if_icmplt	1.76%	iadd	2.81%
14	iconst_1	1.70%	dup	1.76%	iconst_1	2.41%
15	ireturn	1.70%	iconst_0	1.47%	istore	1.75%
	Total	56.86%	Total	69.54%	Total	68.31%

Table 6: The 15 most frequently used bytecodes and their frequencies.
We show the dynamic frequencies of occurrence for the most frequently executed 15 bytecodes in each benchmark. The first column in each benchmark shows the bytecode and the second column gives the frequency count.

Benchmark	Number of distinct byte codes
jess	48
db	45
javac	45
mpegaudio	36
mtrt	39
jack	22

Table 7: Number of distinct bytecodes that account for 90% of the dynamic count

Benchmarks	-s1 calls	-s1 methods	-s10 calls	-s10 methods	-s100 calls	-s100 methods
compress	17330744	577	18170275	578	14566857	449
db	65379	642	1610941	645	91753107	658
jack	2318110	1230	4621508	1233	39172145	1240
mpeg	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 8: Total number of methods and dynamic method calls for the three data sets

This table gives the count of methods and method calls for the different benchmarks. method calls/ # methods would give the method reuse factor, which is the an average measure of how many times each method was called. The performance benefits that can be obtained from using a JIT is directly propotional to the method reuse factor.

in the number of instructions executed when using the JITs, the instruction reuse ratio was low ranging from ≈ 101 to 339. This proves that even an approximate analysis of the reuse in methods would tell us if a JIT compiler would prove fruitful. This profile information can be generated when running the program for the first time, and depending on the method reuse seen one can decide on whether to use a JIT compiler to speed up execution.

We can use the information about method calls to explain why the performance for both the JITs improves when the data set is increased, as seen in Table 3. It is observed that the number of methods in *db* increase from 642 to 658, but the dynamic method calls increase from 65379 to 91753107. Thus, the increase in the instructions when run for the large data sets is not obtained by changing the static structure of the program. This is analogous to increasing the loop indices in a program to get a bigger number of dynamic instructions.

Looking at all three data sets, we see that there is very little difference in the number of methods across data sets. The higher number of method calls indicate that the method reuse in the benchmarks for higher data sets would be substantially more. This would then lead to better performance for the JITs, as seen in Table 3. Since there is a lot of method reuse in all benchmarks for the larger data sets, using any JIT would perform better.

5 Cache Behavior

After looking at the instruction usage in the different execution modes for Java, we now discuss the impact of execution mode on two important hardware features that affect performance.

Firstly we discuss the cache performance in this section. The details about the cache configurations are given in Section 2. It is seen that there is a significant difference in the cache locality for both Instruction and Data caches between the interpreter and JIT mode of execution.

5.1 Instruction Cache Performance

In the case of Instruction caches, it is seen from the first three columns of Table 9 that there is an increase in the number of misses in the Instruction cache for the JIT mode of execution. This is the case in spite of the instruction references being much higher in some instances for the interpreter. Therefore we see good instruction cache performance for the interpreter with the miss rates varying from 0.001% to 0.016%. We see such good instruction locality in the interpreter mode since the interpreter is basically a large switch statement that has around 250 cases. However, the number of distinct bytecodes that constitute 90% of the dynamic execution is small as shown in Table 7. Hence the execution of the interpreter loop exhibits high locality, leading to very few capacity misses. This result shows that the instruction cache behavior of the *go* benchmark indicated as an exceptional case [3] is more of the norm in the SPEC JVM98 benchmarks.

In the case of the JIT mode, the compilation code may not have any locality with the interpreter code and could lead to large misses. Also the compiled code for the methods are small, and do not have large basic block sizes there would be frequent breaks in the instruction run. Since there is no connection between the compiled code for different methods, as they can be called from anywhere it leads to a lot of misses. In [6], the C Compiler benchmark, gcc is shown to have much higher instruction cache miss rates as compared to other C benchmarks. The results presented in Table 9 are based on the '-s1' small data sets. Hence, the compilation process dominates the instruction cache behavior of the JIT. Thus, the JIT tends to behave similar to the *gcc* compiler. In order to verify whether the compilation behavior is the cause for higher miss rates, the data set size was increased to '-s10'. It can be observed from Table 10 the instruction cache miss rates decrease as compared to the '-s1' case. Here, the execution of the application code after translation into the native form starts dominating the behavior and also there is more re-use of the compiled methods. In [3], it was observed that the interpreter incurs a large number of misses as compared to the execution of an equivalent native C code. Here, we find that JIT code has an even more misses than the interpreter execution model.

Benchmark	I-Cache			D-Cache			L2-Cache		
	Refs	Misses	% miss	Refs	Misses	% miss	Refs	Misses	% miss
compress(intr)	10425M	84398	0.001	5365M	21M	0.394	1287M	43M	3.4
(jit)	1385M	218101	0.013	751M	43M	5.828	115M	23M	20.3
(opt)	1313M	353226	0.023	631M	58M	9.342	120M	27M	23.1
jess(intr)	259M	179545	0.068	81M	2.3M	2.910	22M	2.5M	11.3
(jit)	188M	616962	0.321	45M	3.7M	8.298	15M	2.1M	13.4
(opt)	280M	1.07M	0.376	65M	4.3M	6.599	21M	1.9M	9.1
db(intr)	86M	82873	0.094	24M	751592	3.097	6.9M	313673	4.52
(jit)	75M	232046	0.300	17M	1.2M	6.764	6.2M	546219	8.69
(opt)	129M	395351	0.301	29M	1.7M	6.009	9.3M	867402	9.6
javac(intr)	199M	140239	0.069	59M	1.8M	3.099	16M	612916	3.72
(jit)	167M	469143	0.274	40M	3.3M	8.335	13M	1.5M	11.2
(opt)	280M	851480	0.298	64M	4.4M	6.865	19M	1.6M	8.14
mpeg(intr)	1314M	92439	0.007	544M	1.9M	0.356	127M	703161	0.55
(jit)	264M	355896	0.134	101M	3.2M	3.232	17M	1.9M	10.9
(opt)	310M	553374	0.177	98M	4.3M	4.481	24M	1.4M	6.04
mtrt(intr)	1531M	252370	0.016	521M	8.6M	1.654	141M	3.9M	2.82
(jit)	942M	522692	0.054	230M	16M	7.239	63M	9.0M	14.2
(opt)	953M	894243	0.091	225M	13M	6.121	58M	5.9M	10.1
jack(intr)	2668	124563	0.005	1033M	11M	1.069	261M	8M	3.42
(jit)	986	1.0M	0.102	298M	15M	5.341	83M	11M	14.1
(opt)	1037	2.4M	0.227	283M	16M	5.795	89M	10M	11.3

Table 9: Cache Performance for the SPEC JVM98

This table shows the requests, misses and % misses for instruction, data and the L2 cache. M - indicates million.

5.2 Data Cache Performance

In the interpreter execution mode the benchmark data and benchmark bytecodes will be allocated and accessed from the data cache [3]. Whenever a method is executed, the bytecodes are accessed from the data cache and decoded by the interpreter. In contrast, the JIT translates the bytecodes fetched from the data cache into native code before the first execution of the method. Thus, the subsequent invocations of the method do not access the data cache for bytecodes. Columns 4 to 6 in Table 9 gives a measure of the data locality available in the interpreter and JIT execution modes. The first observation is that the data references come down in the JIT execution mode for almost all the benchmarks. This is due to the elimination of the repeated access to the bytecodes of the same method in the JIT mode. Next, we observe that the interpreter has a better data locality than the JITs. This is similar to their relative instruction cache performance. However, the miss rates are higher for data caches as compared to the instruction caches. It can be seen that though *compress* and *mpeg* executed a lot more instructions than the other benchmarks, their count of unique methods was small (data obtained from profiling, presented in Table 8). Since the bytecode are data to the interpreter, a smaller footprint at the bytecode level leads to better data cache performance. For these two benchmarks, the most commonly used methods would remain in the cache most of the time leading to the lower miss rates as compared to other benchmarks. It must be noted that the data cache behavior of JIT mode is not influenced by the method re-use behavior. For the JIT execution we see that the data cache misses are higher than that of the interpreted execution. This is because of the inherent locality in stack based operations of the interpreter as compared to the register based operations of the JIT. Thus JIT has more random accesses to memory, resulting in poor locality.

It can also be observed that the data cache miss rates data improve for interpreter and deteriorate for JIT when we move from the smaller '-s1' data set to '-s10' data set. This can again be ascribed to the greater re-use of the methods in the larger data set for the interpreter. Thus, the accessing of benchmarks bytecodes still dominate the accessing of benchmark data. In contrast, the larger data set increases the conflicts and capacity misses in the JIT mode.

5.3 L2 Cache Performance

The last three columns in Table 9 show the performance of the L2 cache. The number of references to the L2 cache is higher in the interpreter mode. As seen in the L1 caches, we see a higher number of misses for the JIT execution.

	I-Cache			**D-Cache**			**L2-Cache**		
Benchmark	Refs	Misses	% miss	Refs	Misses	% miss	Refs	Misses	% miss
jess(intr)	1883M	181229	0.010	760M	6M	0.822	184M	18M	9.78
(jit)	596M	814293	0.134	175M	15M	8.818	39M	6M	15.3
(opt)	661M	1060831	0.158	186M	13M	7.459	42M	4M	9.52
db(intr)	2563M	85720	0.003	705M	20M	2.903	162M	10M	6.17
(jit)	1961M	253464	0.012	411M	31M	7.533	98M	16M	16.32
(opt)	1960M	2190386	0.107	402M	27M	6.825	95M	14M	14.73
javac(intr)	1686M	221260	0.013	532M	14M	2.766	128M	6M	4.68
(jit)	1092M	1588420	0.140	253M	18M	7.422	65M	8M	12.3
(opt)	1239M	2500610	0.195	281M	19M	7.097	73M	7M	9.58

Table 10: Cache Performance for *jess*, *db* and *javac* using -s10

This table shows the requests, misses and % misses for instruction, data and the L2 cache. M - indicates million.

6 Branch Behavior

In this section we compare the branch characteristics for the interpreter and JIT execution modes. In modern superscalar processors it is necessary to speculate the outcome of a branch to fetch instruction from further down the instruction stream to keep the processor pipeline filled. In the event of a misprediction, it becomes necessary to squash the speculated instructions, or flush the pipeline which would result in costly stalls in the processor. Therefore accurate branch prediction is extremely important for good performance on modern processors. For an object-oriented language like Java which has a higher number of indirect jumps and virtual function calls, the task of predicting the outcome of these control transfer instructions is more difficult. In Table 2 we see that the interpreter mode has a higher number of indirect jumps which would make the task of the branch predictor tougher as seen in the results presented in this section.

Benchmark	BTF!T	2-BIT	BHT	Gshare	2-level
compress(intr)	69.78	60.23	34.69	34.90	35.41
(jit)	45.48	28.31	9.26	8.97	8.93
(opt)	44.14	29.03	9.77	9.35	9.66
jess(intr)	51.70	46.39	19.13	19.37	18.66
(jit)	52.67	38.16	13.05	12.74	12.88
(opt)	47.23	38.59	14.42	13.64	13.72
db(intr)	52.07	43.97	17.12	16.82	16.69
(jit)	52.35	39.64	12.82	12.70	12.81
(opt)	45.34	39.66	14.16	13.36	13.53
javac(intr)	50.77	44.74	18.04	17.90	17.39
(jit)	49.91	39.06	12.92	12.18	12.47
(opt)	44.79	39.23	14.80	13.50	13.81
mpeg(intr)	65.23	52.54	31.61	33.29	31.59
(jit)	49.39	37.47	12.24	11.88	12.16
(opt)	42.71	37.96	13.40	12.70	12.84
mtrt(intr)	43.37	48.78	14.54	13.17	13.42
(jit)	42.99	41.43	11.62	9.20	10.44
(opt)	42.50	41.52	12.03	9.74	10.75
jack(intr)	65.46	56.26	28.31	28.78	27.92
(jit)	52.65	35.68	12.78	12.26	12.65
(opt)	50.74	36.08	12.97	12.31	12.33

Table 11: Branch misprediction rates for different predictors

In this table we show the misprediction rates for simple backward Taken Forward Not Taken (BTF!T), 2-bit counter, 1 level BHT, Gshare and 2 level predictor which is indexed by the PC.

The first column in Table 11 shows the performance for the simplest predictor, a static prediction scheme which assumes that all backward branches are taken, and forward branches are not taken. We see misprediction rates of 65% and higher for *compress*, *mpeg* and *jack*. On obtaining numbers for the taken branches for all the programs, it was seen that these three benchmarks only took 22.4%, 39.4% and 41.5% of their branches as opposed to the other benchmarks which took more than 50% of all the branches. The JIT execution mode seems to optimize these branches and we see a higher percentage of taken branches for both the JITs as compared to the interpreter. This translates to better performance for the BTF!T prediction scheme during JIT execution.

The misprediction rates for a simple 2-bit counter scheme is presented in column 2 of Table 11. This scheme uses a 2-bit saturating counter which is the most common implementation for elements in the Branch History Table (BHT). Using the 2-bit counter, we see a decrease in the misprediction rates for all the benchmarks. Comparing between the interpreter and JIT modes, one sees that the JIT has lower misprediction rates compared to the interpreter mode. The improvement is consistent with that seen for the previous BTF!T scheme, since the predictability of the branches remain the same in both cases.

The third column in Table 11 gives misprediction rates for a prediction scheme using a BHT. The basic idea for the BHT is that it separates the stream of instructions into substreams that contain history of the individual branch instructions. This method provides fairly good prediction accuracy as seen for all benchmarks. The prediction accuracy improves by a factor of two for all the benchmarks compared to the previous schemes and this is also observed for the different execution modes. Columns 4 and 5 give the misprediction rates for Gshare (indexed by PC and Global history bits) [19] and for a two-level branch predictor (indexed by PC), which is described as the GAp by Yeh and Patt [18]. One sees the same trends for these predictors, as for the simple predictors. The interpreter performs worse than the JIT. We still incur high misprediction rates for the JIT execution because of a high frequency of indirect branches. A study on indirect branches by Driesen et. al. [14] showed that an object oriented program like C++ showed higher misprediction rates compared to the SPEC95 suite of programs. Whether specialized predictors for indirect branches will help in achieving better prediction accuracies for the interpreter mode is an area we are working on currently.

7 Future Work

The results presented here can be extended in many ways. In addition to the JITs we used, other JITs can be used with other interpreters to strengthen our observations. Using a dynamic compiler like the HotSpot can show if selective compilation of the methods will always give better performance than interpreters. Further work is needed on instruction level parallelism to determine the impact on superscalar processors and dynamically scheduled processors. Comparing the branch performance of more specialized branch predictors for indirect branches is also an important area of work.

8 Conclusions

We have compared the execution characteristics of Java programs when executed using different Just-In-Time compilers. We have seen significant differences in the execution characteristics, especially its impact on the microarchitectural features. To strengthen our simulation results we compare the execution time observed for the different benchmarks under the interpreter and JIT mode of execution in Table 12. The time was measured using the system *time* utility. Since the accuracy of such a measurement is always questionable, it should be taken with a pinch of salt and we only present it here for comparison with the data we present in our paper. The time is divided into system, user and real time. The user time is time spent in the user level, which include the methods and program execution time spent in user code. System time is the time spent in system code, for example garbage collection, and real time represents the time the user observed time which includes time waiting for I/O.

	Interpreter			**sunwjit**			**sunwjit_opt**		
Benchmark	user	sys	real	user	sys	real	usr	sys	real
jess	1.98	0.46	4.15	1.43	0.63	2.19	1.94	0.57	2.86
db	0.66	0.30	1.02	0.64	0.30	1.08	0.91	0.31	1.34
javac	1.52	0.32	3.09	1.34	0.34	1.98	1.90	0.33	2.75
mpeg	9.54	0.32	10.27	1.94	0.30	2.46	2.03	0.31	2.44
mtrt	8.96	0.30	10.20	6.08	0.34	6.56	6.01	0.27	7.47
jack	17.54	0.71	18.64	7.06	0.73	8.15	7.22	0.75	8.25
compress	78.10	0.34	79.76	11.08	0.28	12.74	10.29	0.31	10.86

Table 12: Execution Time

It is seen that the time spent in user mode forms a major percentage for benchmarks like *compress*, *mpeg* and *mtrt*, and therefore the speedup obtained by the JIT compilers result in good overall performance. For other benchmarks the percentage of user code is smaller and speedup by JIT execution will have a smaller impact, according to Amdahl's law. The general trend seen here correlates with the data presented in the previous sections, especially the instruction counts. It is also seen that the number of instructions executed does not give an indication of performance, when comparing the different modes of execution. For example, *javac* and *jess* execute more instructions in the JIT mode. The data in Table 12 shows that they execute faster in the JIT mode, this is because the code generated in the different modes have different characteristics.

Our major observations in this study can be summarized as

- The execution characteristics of the interpreter and JIT mode of execution is very different. On comparing interpreted and JIT modes of execution, one finds a marked difference in the count of instructions, the instruction mix, the instruction and data locality, and branch predictability for the executed code.
- A JIT compiler will have better performance than an interpreter only if the method reuse factor in the program is high. This can be calculated easily by profiling the program, and should be used to decide the mode of execution for the particular program.
- The larger data sets for the SPEC JVM98 benchmarks show a significant increase in the number of dynamic instructions executed, but the method calls remain constant across the data sets. Using larger data sets of the SPEC JVM98 for JIT or compiler studies will not represent speed-ups that one would see in a real world program.
- The interpreter execution mode exhibited good instruction locality, since it runs in a tight loop. In the case of the JIT, a lack of locality was observed between the compiled code and the interpreter. We also showed how a JIT would have poor instruction locality when used with small programs in which the compilation phase would dominate.
- For interpreters it was found that a small footprint at the bytecode level will lead to better data cache performance. The performance for JITs will be worse due to more random accesses to memory caused by the register based operations of the JIT compiler.
- Running Java in the interpreter mode will incur costly stalls due to branch mispredictions. The higher percentage of indirect jump statements in the interpreter mode, causes the misprediction rates to be really high. In the JIT mode one sees some improvement in the prediction accuracy, due to elimination of a lot of the indirect jumps.

References

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

[2] B. Calder, D. Grunwald, and B. Zorn, *Quantifying Behavioral Differences Between C and C++ Programs*, Journal of Programming Languages, Vol. 2, No. 4, pp. 313-351, 1994.

[3] Cheng-Hsueh A. Hsieh, Marie T. Conte, Teresa L. Johnson, John C. Gyllenhaal and Win-mei W. Hwu, *A Study of the Cache and Branch Performance Issues with Running Java on Current Hardware Platforms*, Proceedings of COMPCON, February 1997, pp. 211-216.

[4] Cheng-Hsueh A. Hsieh, John C. Gyllenhaal and Win-mei W. Hwu, *Java bytecode to native code translation: The Caffine prototype and initial results*, Micro '29, 1996.

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

[6] 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*, Proceedings of ASPLOS VII, 1996, pp. 150-159.

[7] Java JIT Compiler, `http://www.sun.com/solaris/jit/`

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

[9] Standard Performance Evaluation Consortium, *http://www.spec.org/*

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

[11] J. O'Connor and M. Tremblay, *PicoJava-I: The Java Virtual Machine in Hardware*, *IEEE Micro*, March 1997.

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

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

[14] K. Dreisen and U. Holzle, Accurate indirect branch prediction, Proc. of the 25th Annual Intl Symposium on Computer Architecture, pp. 167-178, June 1998.

[15] N. Vijaykrishnan and N. Ranganathan, *Tuning Branch Predictors to Support Virtual Method Invocation in Java*, to appear in Proc. of COOTS'99, May 1999. http://www.usenix.org/events/coots99/brochure/tech02.html.

[16] J.D. Gee, M.D Hill, D. N. Pnevmatikatos and A. J. Smith, *Cache Performance of the SPEC92 Benchmark suite*, IEEE Micro, August 1993.

[17] G. Muller, B. Moura, F. Bellard and C. Consel, *Harissa: a flexible and efficient Java environment mixing bytecode and compiled code*, COOTS '97, June 1997.

[18] T. Yeh and Y. Patt, *A Comparison of dynamic branch predictors that use two levels of branch history*, 20th Annual Intl Symposium on Computer Architecture, pp. 257-266, 1995.

[19] Scott McFarling, *Combining branch predictors*, WRL Technical Note TN-36, June 1993.