Chapter 1

BENCHMARKING THE JAVA VIRTUAL ARCHITECTURE

The SPEC JVM98 Benchmark Suite

David Gregg
Department of Computer Science, Trinity College, Dublin 2, Ireland.
David.Gregg@cs.tcd.ie

James Power
Department of Computer Science, National University of Ireland, Maynooth, Ireland.
James.Power@may.ie

John Waldron
Department of Computer Science, Trinity College, Dublin 2, Ireland.
John.Waldron@cs.tcd.ie

Abstract In this chapter we present a study of the SPEC JVM98 benchmark suite at a platform-independent level. The results presented describe the influence of class library code, the relative importance of various methods in the suite, as well as the sizes of the local variable, parameter and operand stacks. We also examine the dynamic bytecode instruction usage frequencies, and discuss their relevance. The influence of the choice of Java source to bytecode compiler is shown to be relatively insignificant at present.

These results have implications for the coverage aspects of the SPEC JVM98 benchmark suites, for the performance of the Java-to-bytecode compilers, and for the design of the Java Virtual Machine.

Keywords: Java Virtual Machine, SPEC JVM98, benchmark suite, instruction usage frequency, method execution frequency

Chapter 1 of "Java Microarchitectures"
N. Vijaykrishnan and M. Wolczko (Eds.), Kluwer, 2002

1. Introduction

The Java paradigm for executing programs is a two stage process. Firstly the source is converted into a platform independent intermediate representation, consisting of bytecode and other information stored in class files [Lindholm and Yellin, 1996]. The second stage of the process involves hardware specific conversions, perhaps by a JIT or hotspot compiler for the particular hardware in question, followed by the execution of the code. This research sets out to perform dynamic analysis at the platform independent bytecode level, and investigate whether or not useful results can be gained. In order to test the technique, the SPEC JVM98 benchmark suite [SPEC, 1998] was used.

The remainder of this chapter is organised as follows. Section 2 discusses the background to this work, including the rationale behind bytecode-level dynamic analysis, and the test suite used. Section 3 presents a method-level view of the dynamic profile, while Section 4 profiles the method stack frame sizes. Section 5 presents a more detailed bytecode-level view of the applications, and Section 6 discusses the influence of compiler choice on these figures. Section 7 concludes the chapter.

2. Background and Related Work

The increasing prominence of internet technology, and the widespread use of the Java programming language has given the Java Virtual Machine (JVM) an important position in the study of compilers and related technologies. To date, much of this research has concentrated in two main areas:

- Static analysis of Java class files, for purposes such as optimisation [Vallee-Rai et al., 1999], compression [Antonioli and Pilz, 1988; Rayside et al., 1999], software metrics [Cohen and Gil, 2000], or the extraction of object models [Jackson and Waingold, 2001]
- The performance of the bytecode interpreter, yielding techniques such as Just-In-Time (JIT) compilation [Adl-Tabatabai et al., 1998; Ishizaki et al., 1999] and hotspot-centered compilation [Sun Microsystems, 2001]. See [Kazi et al., 2000] for a survey.

This chapter presents a platform-independent dynamic analysis of the SPEC JVM98 suite, including data related to bytecode instruction usage, method frequencies and stack frame profiles. This platform-independent bytecode analysis describes the bytecode as it is interpreted, without the interference of JIT compilation or any machine-specific issues. This type of analysis can help to clarify the potential impact of

the data gained from static analysis, can provide information on the scope and coverage of the test suite used, and can act as a basis for machine-dependent studies.

The production of bytecode for the JVM is, of course, not limited to a single Java-to-bytecode compiler. Not only is there a variety of different Java compilers available, but there are also compilers for extensions and variations of the Java programming language, as well as for other languages such as Eiffel [Colnet and Zendra, 1999] and ML [Benton et al., 1998], all targeted on the JVM. In previous work we have studied the impact of the choice of source language on the dynamic profiles of programs running on the JVM [Waldron, 1999], as well as the choice of compiler on the profiles of the Java Grande benchmark suite [Daly et al., 2001]. The compiler comparisons presented in this chapter help to calibrate this and other such studies.

2.1 The SPEC JVM98 Benchmark Suite

All the programs in this study were taken from the SPEC JVM98 benchmark suite [SPEC, 1998]. The SPEC JVM98 suite was designed as an industry-standard benchmark suite for measuring the performance of client-side Java applications, and we have used the seven main programs from this suite. These are:

cmprs	Modified Lempel-Ziv compression method (LZW)
db	Performs multiple database functions on memory resident database
jack	A Java parser generator that is based on PCCTS
javac	The Java compiler from SUN's JDK 1.0.2.
jess	An Expert Shell based on the CLIPS expert shell system
mpeg	Decompresses ISO MPEG Layer-3 audio files
mtrt	A ray-tracer with two threads each rendering a scene

There have been a number of studies of the SPEC JVM98 benchmark suite. [SPEC, 1998] provides speed comparisons of the suite using different Java Platforms, and [Ishizaki et al., 1999] examines the speed impact of various optimisations. [Driesen et al., 2000] uses the SPEC JVM98 suite in an examination the prediction rate achieved by invoke-target and other predictors. Both [Li et al., 2000] and [Radhakrishnan et al., 2001] discuss low-level timing and cache performance for the suite. [Shuf et al., 2001] also looks at cache misses, but from the perspective of the SPEC JVM98 programs' memory behaviour. This theme is investigated in depth in [Dieckmann and Hölzle, 1999], which studies the allocation behaviour of the SPEC JVM98 suite from the perspective of memory management. Both [Shuf et al., 2001] and [Zhang and Seltzer, 2000] note that the SPEC JVM98 suite may not be suitable for assessing all types of Java applications. Finally, [Bowers and Kaeli, 1998] analyses the SPEC JVM98 suite using dynamic bytecode level analysis similar to

our own, but does not present data related to method usage or compiler differences.

The SPEC JVM98 suite is just one of many possible benchmarks suites for Java. A similar suite, the Java Grande Forum Benchmark Suite, [Bull et al., 1999; Bull et al., 2000] has been studied in [Daly et al., 2001]. Micro-benchmarks for Java include CaffeineMark [Corporation, 1999] Richards and DeltaBlue [Wolczko, 2001a]. As these measure small, repetitive operations, it was felt that their results would not be typical of Java applications. For the same reason larger suites, designed to test Java's threads or server-side applications, such as SPEC's Java Business Benchmarks, the Java Grande Forum's Multi-threaded Benchmarks, IBM's Java Server benchmarks [Baylor et al., 2000] or VolanoMark [Neffenger, 1999] have not been included here.

2.2 Methodology

The data presented in this chapter were gathered by running each of the SPEC JVM98 independently on a modified JVM. The JVM used was Kaffe [Wilkinson, 2000], an independent cleanroom implementation of the JVM, distributed under the GNU Public License. While Kaffe can be built to emit debugging information, we modified its source slightly to collect information more directly suited to our purposes. Version 1.0.6 of Kaffe was used, and it was built with debugging enabled but with JIT compilation disabled.

Other approaches to tracing the execution of Java programs include bytecode-level instrumentation [Lee, 1997], and special-purpose JVMs such as SUN's Tracing JVM [Wolczko, 2001b] and IBM's Jikes Research Virtual Machine, a development of the Jalapeño Virtual Machine [Alpern et al., 2000].

It should be noted that all measurements in this chapter were made with the Kaffe class library. This library is not 100% compliant with SUN's JDK, and may, of course, differ from other Java class libraries. In subsequent sections we will distinguish between code from the (Kaffe) class library and "SPEC-code" i.e. those bytecodes from the SPEC JVM98 benchmark suite itself.

3. Dynamic Method Execution Frequencies

In this section we present a profile of the SPEC JVM98 based on methods, since these provide both a logical source of modularity at source-code level, as well as a possible unit of granularity for hotspot analysis [Sun Microsystems, 2001; Armstrong, 1998]. It should be noted that these figures are not the usual *time-based* analysis such as found in

Program	Total methods	CL %	CL native %
cmprs	2.26e+08	0.0	0.0
jess	1.35e+08	32.5	1.9
db	1.24e+08	98.7	0.1
javac	1.53e+08	62.0	2.8
mpeg	1.10e+08	1.3	1.1
mtrt	2.88e+08	3.2	0.1
jack	1.16e+08	92.3	4.2
average	1.65e+08	41.4	1.5

Table 1.1. Measurements of total number of method calls by SPEC JVM98 applications. Also shown is the percentage of the total which are in the class library, and percentage of total which are in class library and are native methods. The figures include calls of native methods, but excludes calls within native methods.

Program	Java method calls		bytecodes executed	
	number	% in CL	number	% in CL
cmprs	2.26e+08	0.0	1.25e+10	0.0
jess	1.32e+08	31.2	1.91e+09	18.8
db	1.24e+08	98.7	3.77e+09	70.4
javac	1.48e+08	60.9	2.43e+09	58.3
mpeg	1.08e+08	0.1	1.15e+10	0.0
mtrt	2.88e+08	3.1	2.20e+09	3.5
jack	1.11e+08	92.0	1.50e+09	82.3
ave	1.62e+08	40.9	5.12e+09	33.3

Table 1.2. Measurements of Java method calls made and bytecodes executed by SPEC JVM98 applications. The percentage of calls and bytecode instructions in the class library is also shown. In all cases, native methods have been excluded.

	cmprs	jess	db	javac	mpeg	mtrt	jack
io	8.5	0.4	0.1	33.5	14.0	24.6	3.1
lang	52.9	49.8	40.8	21.9	71.0	75.3	24.0
net	0.4	0.0	0.0	0.0	0.1	0.0	0.0
util	38.2	49.8	59.2	44.7	15.0	0.1	72.9

Table 1.3. Calls to non-native methods in the class library. This table shows the percentage of class library methods called in each of the API packages used.

	cmprs	jess	db	javac	mpeg	mtrt	jack
io	3.9	0.5	0.0	31.0	8.9	56.6	2.7
lang	52.4	37.9	73.4	32.4	59.4	43.1	19.5
net	0.7	0.0	0.0	0.0	0.3	0.0	0.0
util	43.0	61.5	26.6	36.6	31.3	0.3	77.8

Table 1.4. Bytecode instructions executed in the class library. This table shows the percentage of class library bytecode instructions executed in each of the API packages used.

e.g. [Radhakrishnan et al., 2001] for the SPEC JVM98 suite, or [Bull et al., 1999; Bull et al., 2000] for the Java Grande suite. Also, since our analysis is carried out at the bytecode level, we do not track method calls or other activities within native methods.

Table 1.1 shows measurements of the total number of method calls including native calls by SPEC JVM98 applications. For the programs studied, on average 1.5% of methods are class library methods which are implemented by native code. As the benchmark suite is written in Java it is possible to conclude that any native methods are in the class library. Table 1.1 must be interpreted carefully as it is a method frequency table, without reference to bytecode usage, and so may not correlate with eventual running times.

The figures on the left part of Table 1.2 show measurements of the Java method calls excluding native calls. The right part of Table 1.2 shows the number of bytecodes executed for each application. Over 40% of method calls are directed to methods in the SPEC suite, and 33% of bytecodes executed are in the class library. This is a significant difference from Java Grande applications [Bull et al., 2000] which spend almost all of the time outside the class library. This suggests that mixed mode compiled-interpreted systems, which pre-compile the class library methods to some native format, could be effective for improving the running time of the SPEC JVM98 programs.

Table 1.3 shows dynamic measurements of the Java API package *method call* percentages and Table 1.4 shows API package *bytecode* percentages. Some care should be taken when considering these tables, since, as shown in Table 1.2, the total number of calls and bytecodes represented by these percentages varies considerably across applications. The percentages in Table 1.4 and Table 1.4 are broadly similar, implying the class library methods each execute similar numbers of bytecodes. As would be expected for the programs considered, the *applet* and *awt* packages are not used at all as graphics have been removed from the benchmarks.

Table 1.5 presents two contrasting analyses of method usage. The left part of Table 1.5 ranks methods based on the frequency with which they are called at run-time. The right part of Table 1.5 on the other hand ranks methods based on the proportion of total executed bytecodes that they account for. The figures on the left are related to the *method reuse* factor as described in [Radhakrishnan et al., 2001], proposed as an indication of the benefits obtained from JIT compilation. However, we suggest that the difference in rankings between frequency of invocation and proportion of bytecodes executed shows that the method-call figures do not give an accurate picture of where the program is spending its time. The difference is most striking in *cmprs*, where the left part seems to show a similar distribution of effort between the top three methods, yet the right part clearly shows that two completely different methods (compressor.compress, decrompressor.decompress), account for the majority of the bytecodes executed.

This result highlights a danger of a naive approach to determining "hot" methods in a Java program in terms of frequently-called methods. The danger is that the most expensive method will be, for example, a large matrix multiplication which is invoked only once, but dominates the running time. These figures show that there may be little correlation between the frequency of invocation and the running time spent in a method.

Figure 1.1 summarises the information from the right-hand side of Table 1.5. On average, the top two methods account for more than 40% of bytecodes executed. Thus it is vital that these methods are optimised, even if they are invoked only a handful of times.

4. Dynamic Stack Frame Usage Analysis

Each Java method that executes is allocated a stack frame which contains (at least) an array holding the actual parameters and the variables declared in that method. Instance methods also have a slot for the `this`-pointer in the first position of the array. This array is referred to as the *local variable array*, and those variables declared inside a method are called *temporary variables*. In this section we examine the dynamic size of this array, its division into parameters and temporary variables, along with the maximum size of the operand stack during the method's execution. As well as having an impact on the overall memory usage of a Java program, this size also has implications for the possible usage of specialised `load` and `store` instructions, which exist for the first four slots of the array.

cmprs method % by call		**cmprs** method % by instruction count	
compress/Code_Table.of	32.9	compress/Compressor.compress	34.2
compress/Output_Buffer.putbyte	29.0	compress/Decompressor.decompress	23.8
compress/Input_Buffer.getbyte	20.8	compress/Compressor.output	9.3
compress/Code_Table.set	7.9	compress/Input_Buffer.getbyte	8.3
compress/Decompressor.getcode	4.4	compress/Decompressor.getcode	7.3
jess method % by call		**jess** method % by instruction count	
jess/ValueVector.get	16.5	jess/Node2.runTests	11.6
jess/Value.equals	10.7	jess/ValueVector.equals	11.2
jess/ValueVector.equals	7.1	jess/Value.equals	10.1
jess/ValueVector.size	5.4	jess/Token.data_equals	8.0
java/util/HashMap.find	5.3	jess/ValueVector.get	5.7
db method % by call		**db** method % by instruction count	
java/util/Vector.elementAt	36.4	java/lang/String.compareTo	47.0
java/lang/String.compareTo	18.2	db/Database.shell_sort	25.9
java/lang/Math.min	18.2	java/util/Vector.elementAt	10.7
java/util/Vector$1.nextElement	6.6	java/util/Vector$1.nextElement	3.9
java/util/Vector$1.hasMoreElements	5.5	java/lang/Math.min	3.2
javac method % by call		**javac** method % by instruction count	
java/io/BufferedInputStream.read	11.1	java/io/BufferedInputStream.read	12.9
javac/ScannerInputStream.read	5.4	javac/ScannerInputStream.read	8.3
java/util/HashMap.find	3.7	java/lang/String.hashCode	5.0
java/lang/Object.equals	3.1	java/lang/String.replace	4.1
java/lang/Object.<init>	2.7	java/lang/String.equals	4.1
mpeg method % by call		**mpeg** method % by instruction count	
mpegaudio/q.j	16.6	mpegaudio/q.l	43.4
mpegaudio/l.read	15.9	mpegaudio/q.m	7.5
mpegaudio/l.V	10.2	mpegaudio/lb.read	6.1
mpegaudio/cb.M-DM-#	6.3	mpegaudio/cb.M-DM-#	4.9
mpegaudio/cb.M-CM-^Z	5.1	mpegaudio/tb.M-DM-^U	3.9
mtrt method % by call		**mtrt** method % by instruction count	
raytrace/Point.GetX	19.5	raytrace/OctNode.Intersect	17.5
raytrace/Point.GetY	17.3	raytrace/OctNode.FindTreeNode	10.6
raytrace/Point.GetZ	16.3	raytrace/Point.Combine	9.6
raytrace/Face.GetVert	11.1	raytrace/Point.GetX	7.7
raytrace/Ray.GetDirection	6.0	raytrace/Face.GetVert	7.3
jack method % by call		**jack** method % by instruction count	
java/lang/Object.<init>	8.8	java/util/HashMap$EntryIterator.nextBucket	21.0
java/util/Vector.size	5.2	jack/RunTimeNfaState.Move	5.5
java/util/Vector.<init>	3.8	java/util/Vector.insertElementAt	5.1
java/util/HashMap.access$1	3.7	java/util/Vector.indexOf	4.2
java/util/HashMap.find	3.2	jack/TokenEngine.getNextTokenFromStream	3.8

Table 1.5. Dynamic method execution frequencies for the SPEC JVM98 programs, excluding native methods. The figures on the right show the percentage of total method calls for each method. The figures on the left show the percentage of total bytecodes executed that were in this method. The names of some methods in the **mpeg** use non-alphanumeric characters.

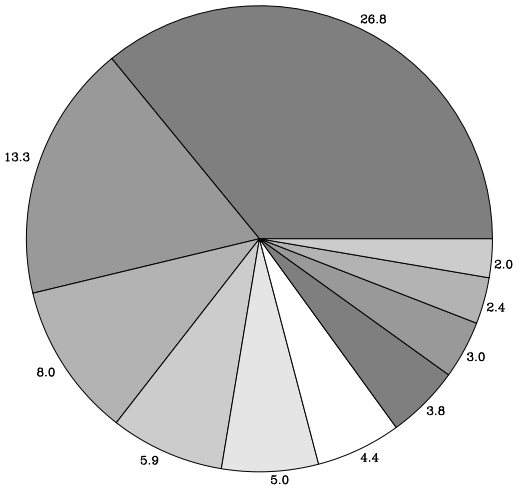

Figure 1.1. Average Dynamic bytecode percentages for the top 10 methods in terms of bytecodes executed. Here, the averages have been taken across applications for the methods with the most bytecodes for each application, the second most bytecodes, and so on.

Table 1.6 shows dynamic percentages of local variable array sizes, and further divides this into parameter sizes and temporary variable array sizes. One finding that stands out is the absence of zero parameter size methods across all applications. All the SPEC JVM98 applications have some zero parameter methods, but these appear as zero in the percentages as they are swamped by those methods with high bytecode counts in the applications which have non-zero parameter sizes.

5. Dynamic Bytecode Execution Frequencies

In this section we present a more detailed view of the dynamic profiles of the SPEC JVM98 programs studied by considering the frequencies of the different bytecodes used. These figures help to provide a detailed description of the nature of the operations being performed by each program, and thus give a picture of the aspects of the JVM actually being tested by the suite.

In order to study overall bytecode usage across the programs, it is possible to calculate the average bytecode frequency

$$f_i = \frac{100}{n} \sum_{k=1}^{n} \frac{c_{ik}}{\sum_{j=1}^{256} c_{jk}}$$

Local variable array size								
Size	cmprs	jess	db	javac	mpeg	mtrt	jack	ave
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	8.3	6.1	6.2	14.8	0.8	29.5	39.2	15.0
2	12.5	15.7	15.0	13.2	6.9	13.6	16.0	13.3
3	1.0	18.9	0.0	10.9	0.5	0.7	9.6	5.9
4	3.6	27.0	2.3	25.5	8.2	16.3	17.0	14.3
5	16.6	0.3	0.7	1.4	4.4	11.7	12.4	6.8
6	0.0	7.8	47.0	16.8	7.0	1.6	0.2	11.5
7	23.8	3.2	1.5	8.6	8.5	4.1	0.4	7.2
8	0.0	4.0	0.9	4.8	7.1	0.0	1.2	2.6
>8	34.2	17.0	26.5	3.9	56.6	22.5	4.0	23.5
Parameter size								
0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0
1	74.5	7.9	7.8	38.4	3.6	30.8	52.0	30.7
2	21.0	62.9	91.4	23.8	14.6	24.2	24.2	37.4
3	4.6	17.6	0.8	18.6	59.9	6.9	16.9	17.9
4	0.0	11.7	0.0	16.4	13.5	25.5	6.0	10.4
5	0.0	0.0	0.0	1.0	3.6	10.3	0.6	2.2
6	0.0	0.0	0.0	1.1	4.1	1.8	0.0	1.0
7	0.0	0.0	0.0	0.6	0.0	0.5	0.1	0.2
8	0.0	0.0	0.0	0.1	0.1	0.0	0.0	0.0
>8	0.0	0.0	0.0	0.0	0.6	0.0	0.0	0.1
Temporary variable size								
0	21.0	23.3	21.2	29.6	11.6	58.2	62.5	32.5
1	4.4	24.4	0.2	18.8	5.1	1.3	16.1	10.0
2	0.0	19.5	1.2	16.4	5.6	13.3	4.8	8.7
3	9.3	8.5	1.5	16.4	9.9	0.0	10.6	8.0
4	7.3	0.1	47.0	7.4	9.8	4.2	0.5	10.9
5	0.0	3.1	1.5	1.8	0.0	19.7	0.1	3.7
6	23.8	4.1	0.9	8.1	6.5	1.7	1.0	6.6
7	0.0	2.0	25.9	0.4	0.0	0.0	0.5	4.1
8	0.0	3.4	0.6	0.7	0.1	0.0	0.0	0.7
>8	34.2	11.6	0.0	0.2	51.4	1.6	3.9	14.7

Table 1.6. Bytecode based dynamic percentages of local variable array sizes, as well as temporary and parameter sizes for SPEC JVM98 programs. The local variable array and parameter sizes include the this-reference for non-static methods.

where c_{ik} is the number of times bytecode i is executed during the execution of program k and n is the number of programs averaged over. f_i is an approximation of that bytecode's usage for a typical SPEC JVM98 program.

For the purposes of this study, the 202 bytecodes can be split into the 22 categories used by the Java Virtual Machine Specification [Lindholm and Yellin, 1996]. By assigning those instructions that behave similarly

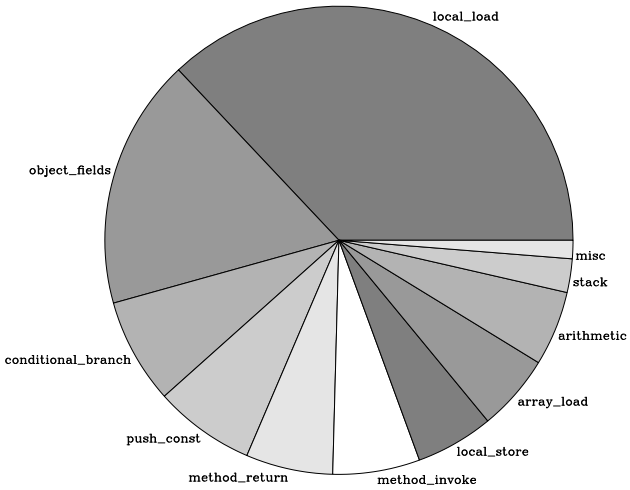

Figure 1.2. A summary of dynamic percentages of category usage by the applications in the SPEC JVM98 suite.

	cmprs	jess	db	javac	mpeg	mtrt	jack	ave
local_load	32.4	37.8	41.2	35.3	32.8	33.6	32.5	35.1
object_fields	18.9	14.8	17.0	17.9	9.6	15.7	20.7	16.4
conditional_branch	6.2	10.8	7.8	8.0	3.3	3.7	8.4	6.9
push_const	7.1	6.0	1.3	7.2	13.4	5.8	5.7	6.6
method_return	1.8	6.9	3.3	6.1	1.0	13.1	7.4	5.7
method_invoke	1.9	7.0	3.2	6.3	1.0	13.0	7.7	5.7
local_store	9.1	5.1	6.9	3.5	6.3	1.8	2.7	5.1
array_load	3.7	4.5	6.7	2.3	12.2	3.5	1.9	5.0
arithmetic	5.5	0.4	6.0	2.8	11.7	5.1	2.7	4.9
stack	5.8	0.5	0.6	2.6	1.1	1.1	4.0	2.2
misc	1.0	1.4	1.8	1.4	1.7	0.2	0.6	1.2
unconditional_branch	0.4	1.0	1.2	1.4	0.4	0.2	1.4	0.9
array_store	1.6	0.2	0.7	0.9	2.1	0.3	0.3	0.9
object_manage	0.0	1.8	1.7	0.8	0.0	0.3	1.0	0.8
logical_boolean	1.6	0.6	0.0	0.6	1.5	0.0	0.1	0.6
array_manage	0.0	0.5	0.0	1.2	0.0	0.0	2.2	0.6
logical_shift	2.6	0.0	0.0	0.0	0.7	0.0	0.0	0.5
comparison	0.0	0.2	0.0	0.0	0.4	2.2	0.1	0.4
conversion	0.4	0.0	0.0	0.4	1.0	0.0	0.0	0.3
table_jump	0.0	0.2	0.0	1.0	0.0	0.0	0.0	0.2
subroutine	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
monitor	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 1.7. Dynamic percentages of category usage by the applications in the Java SPEC JVM98 suite.

	jdk13	jikes	bcj	gcj
cmprs	4.2	4.5	4.8	9.2
jess	0.5	1.1	0.5	1.5
db	1.1	1.1	2.5	6.6
mtrt	0.0	0.1	0.2	1.0
avg	1.5	1.7	2.0	4.6

Table 1.8. Total SPEC dynamic bytecode usage increases. For each compiler, these figures show the percentage increase in the total number of bytecode instructions executed, as compared to the distributed SPEC bytecodes.

into groups it is possible to describe clearly what is happening. Table 1.7 is summarised in Figure 1.2. As has been noted in [Waldron, 1999] *local_load*, *push_const* and *local_store* instruction categories always account for very close to 40% of instructions executed, a property of the Java Virtual Machine, irrespective of compiler or compiler optimisations used. As can be seen in Table 1.7, *local_load* = 35.1%, *push_const* = 6.6% and *local_store* = 5.1%, giving a total of 46.8% of instructions moving data between the local variable array and constant pool and the operand stack. It is also worth noting that, in practice, loads are dynamically executed roughly ten times as often as stores.

6. Comparisons of dynamic bytecode usage across different compilers

In this section we consider the impact of the choice of Java compiler on the dynamic bytecode frequency figures. Java is relatively unusual (compared to, say, C or C++) in that optimisations can be implemented either when the source program is compiled into bytecode, or when this bytecode is executed on a specific JVM. We consider here those optimisations that are implemented at the compiler level, and thus may be considered to be platform independent, and which must be taken into account in any study of the bytecode frequencies.

The programs in the SPEC JVM98 suite are supplied in bytecode format and we refer to these programs in this section as **spec**. The Java source code for *javac*, *jack* and *mpeg* was not supplied as part of the SPEC JVM98 suite, but the remaining four programs were compiled using the following four compilers:

jdk13 SUN's javac compiler, Standard Edition (JDK build 1.3.0-C)
jikes IBM's Jikes compiler, version 1.06
bcj Borland Compiler for Java 1.2.006
gen Generic Java, version 0.6m of 5-Aug-99

compress						
inst	**Σ spec**	**Δ bcj**	**Δ gen**	**Δ jdk13**	**Δ jikes**	**Δ %**
aload_0	1.49e+09	479543210	479543285	479543285	479543285	15.4
aload	4.80e+08	-479541245	-479541245	-479541245	-479541245	15.4
astore	3.38e+08	-338382920	-338382920	-338382920	-338382920	10.9
invokevirtual	2.06e+08	338384974	338384974	338385026	338385025	10.9
ireturn	1.33e+08	273400040	273400040	273400040	273400040	8.8
iload_1	4.02e+08	244277320	244277320	244277320	244277320	7.8
iconst_0	6.19e+07	46	361718297	46	46	2.9
goto	4.74e+07	84616469	141529534	9863300	56901750	2.3
return	9.34e+07	64984960	64984960	64984960	64984960	2.1
istore	3.02e+08	-56018830	-56018880	-56018880	-56018880	1.8
iload	6.95e+08	-56018830	-56018880	-56018880	-56018880	1.8
ifeq	9.41e+07	-37191210	-47054840	-37191210	9861565	1.1
ifne	3.43e+07	47052775	-34341021	47052775	0	1.0
if_icmpeq	1.02e+02	47052800	56916430	0	0	0.8
if_icmple	1.11e+07	9861590	77870930	0	0	0.7
if_icmpgt	1.97e+07	-9861590	77665160	0	0	0.7
if_icmpge	7.66e+07	-16845589	68354806	0	0	0.7
if_icmpne	1.99e+08	-47052800	34340996	0	0	0.7
ifgt	8.75e+07	-50	-87526800	0	0	0.7
ifge	8.52e+07	0	-85200395	0	0	0.7
iload_2	7.29e+08	17932185	26633015	17932185	17932185	0.6
iconst_1	3.67e+08	9861565	28423015	9861565	9861565	0.5
if_icmplt	6.07e+07	16845589	46569929	0	0	0.5
ifle	6.80e+07	50	-68009290	0	0	0.5

Table 1.9. SPEC bytecode usage for **compress** *using the different compilers.*

db						
inst	**Σ spec**	**Δ bcj**	**Δ gen**	**Δ jdk13**	**Δ jikes**	**Δ %**
ifle	2.25e+07	-22506612	-22526825	-22507552	-22507552	8.1
goto	1.39e+07	27516782	27525732	11735983	11889333	7.0
ifgt	1.99e+04	22506612	-19918	22507552	22507552	6.1
iconst_0	1.58e+06	19	45596126	19	19	4.1
ifge	2.30e+07	-23048135	-23048448	0	0	4.1
if_icmpge	1.72e+06	12412825	12413138	0	0	2.2
if_icmplt	1.24e+07	-12412825	10635321	0	0	2.1
iflt	1.10e+01	23048135	-11	0	0	2.1
if_icmpgt	8.62e+02	0	22526530	0	0	2.0
ifne	9.64e+06	-6733734	-6734639	0	0	1.2
ifeq	7.58e+04	6733734	6733734	0	0	1.2
iload_3	1.24e+08	-1860473	-1860473	-1860473	-1860473	0.7
iload	8.46e+07	1879355	1860476	1860476	1860476	0.7

Table 1.10. SPEC bytecode usage for **db** *using the different compilers.*

jess						
inst	**Σ spec**	**Δ bcj**	**Δ gen**	**Δ jdk13**	**Δ jikes**	**Δ %**
if_icmpge	3.69e+02	37676280	37676280	0	0	4.8
if_icmplt	3.97e+07	-37676280	-37676280	0	0	4.8
goto	1.32e+07	15985302	18575266	378754	9710283	2.9
iconst_0	3.86e+07	5668339	13878309	2601752	2601452	1.6
ifeq	2.28e+07	-4387217	-12593144	-4383693	-993983	1.4
if_icmpeq	2.17e+07	838796	9044723	837199	4199008	1.0
ifnull	5.19e+06	-3066429	-5192017	-3067583	-3067583	0.9
aload_0	1.95e+08	-3753013	-3755172	2603623	2601459	0.8
putfield	2.50e+07	-3755176	-3755176	2601460	2601459	0.8
ifnonnull	2.39e+04	3066429	-22349	3067583	3067583	0.6
lookupswitch	9.67e+04	3167506	3167506	3167506	-94	0.6
tableswitch	3.20e+06	-3167506	-3167506	-3167506	94	0.6
if_icmpne	2.14e+07	2227791	2228124	-837199	-4199008	0.6
iload	6.72e+07	1960135	1960255	1960037	1960036	0.5
ifne	2.73e+07	1320636	1320303	4383699	993983	0.5

Table 1.11. SPEC bytecode usage for **jess** *using the different compilers.*

mtrt						
inst	**Σ spec**	**Δ bcj**	**Δ gen**	**Δ jdk13**	**Δ jikes**	**Δ %**
goto	3.79e+06	4663508	6636867	40596	1425663	0.6
aconst_null	4.89e+06	18	10657545	-4	-16	0.5
if_icmpge	8.62e+04	5328906	5328881	0	0	0.5

Table 1.12. SPEC bytecode usage for **mtrt** *using the different compilers.*

We used the -O (optimise) option wherever it was available. We also compiled and ran the benchmarks using the Java compiler from JDK 1.2, but the results were identical those for the JDK 1.3 compiler. The class library was not recompiled and those bytecodes have been excluded from the dynamic comparisons in this section.

Table 1.8 shows the percentage differences in total SPEC dynamic bytecode counts for each recompiled program using different compilers, as compared to the compiled version distributed by SPEC. Perhaps the most remarkable result is that SPEC's version executes fewer bytecodes than the code produced by any of the widely used Java compilers. In particular, the code for *cmprs* is substantially better in all cases (4.2% – 9.2%).

The bytecode programs distributed as part of the SPEC JVM98 suite were originally compiled using the compiler from SUN's JDK 1.1.4. Such early Java compilers contained some very aggressive optimisations which were later discovered to be unsafe, such as method inlining. In more recent Java systems (since Java 1.2), the inlining abilities of *javac* have been vastly reduced. Those originally compiling the SPEC benchmarks also modified some of the source code to eliminate inefficiencies in the outputted bytecode (at least in the case of *cmprs*). Given that the source code was tuned for a particular compiler, it is not surprising that other compilers produce slightly worse bytecode.

Ideally, the optimisations implemented by each compiler should be described in the corresponding documentation; regrettably this is not the case in reality. Also, since each of the applications produces significantly large bytecode files, a static analysis of the differences between these files is not practical. Further, a bytecode-level static analysis would not be sufficient for determining those differences which resulted in a significant variance in the dynamic profiles.

Instead, a detailed analysis of the dynamic bytecode execution frequencies was carried out. The raw statistics are presented in Table 1.9, Table 1.10, Table 1.11 and Table 1.12. Each row of these four tables corresponds to a single bytecode instruction and shows:

- Σ **spec**: the total number of times this instruction was executed in the original **spec** suite
- Δ **bcj**, **gen**, **jdk13**, **jikes**: the increase/decrease in the count of dynamic bytecodes executed between each compiler and the original **spec** suite
- Δ %: The average of the absolute values of the four Δs for each compiler, expressed as a percentage of the total number of bytecodes executed for the application in the original **spec** suite

The tables are sorted in descending order of Δ %, and only average differences greater than 0.5% are shown.

It is notable that the different applications, in exercising different areas of the instruction set, reflected compiler differences to varying degrees. For example, there are significant differences in the bytecode for *cmprs*, less so for *db* and *jess*, and almost no differences for *mtrt*. Below we summarise the main differences exhibited by the different compilers.

6.1 Method Inlining

Table 1.9 shows that the bytecode distributed by SPEC executes far fewer `invokevirtual` instructions than the bytecodes produced by other compilers. Correspondingly, each of the other compilers shows an increase in `ireturn` and `return` instructions, as these methods return control to the caller.

The reason for the reduction in calls is that SPEC used the JDK 1.1.4 compiler, which inlines virtual method calls very aggressively. This was subsequently found to be unsafe — it can lead to bytecodes that fail verification. More recent compilers have much reduced the inlining ability. This raises the question of whether *cmprs* is representative of real bytecodes, given that much of its efficiency depends on an unsafe optimisation.

With less virtual methods being called, there is a corresponding drop in `aload_0` instructions, which corresponding to loading an object from that method's class. However, the spec programs do show one peculiarity here, in that they redundantly store an object reference to a local variable and immediately load it again. This results in a corresponding increase in `astore` and `aload` instructions.

6.2 Use of `goto`

The figures show a difference in the use of comparison and `goto` instructions between the compilers. The *jdk13* and *jikes* compilers use significantly fewer `goto` instruction than the other compilers due to a better loop structure, and the **spec** suite makes further improvements on this. For example, a naive translation loop of the form:

`while (`*expr*`)` { *stats* }

would involve a test and jump after the *expr*, and an unconditional backwards jump after *stats*.

This is the approach taken by *gen* and *bcj*, whereas *jdk13* and *jikes* rearrange the code so there is just one jump on each loop iteration. The approach taken by *gen* and *bcj* is simpler to implement, but results in an unnecessary `goto` at the end of the loop. It is remarkable that these

compilers do not implement what is really a very basic optmisation. In addition to this, *jdk13* optimises `goto` instructions whose target is also a `goto` into one jump, which gives it a slight edge over *jikes*.

6.3 Comparisons

In addition to the differences in gotos resulting from loop structure, there are also smaller differences resulting from the type of comparison used. For the compilers other than *gen*, we can see a reversal in order between "not equal" and "equal" comparisons, as well as "less-equal" and "greater than" comparisons, witnessed by instructions ending in `-ne`, `-eq`, `-le` and `-gt` respectively.

As well as generic comparison instructions for each type, Java bytecode has a specialised `ifeq` instruction for comparison with zero. As can be seen from Table 1.11, the frequencies for these instructions for the *gen* compiler is lower than the other compilers, and a price is paid in a correspondingly higher use of `iconst_0`, `if_icmpeq` and `if_icmpne` instructions.

7. Conclusions

This chapter investigates platform independent dynamic Java Virtual Machine analysis using the SPEC JVM98 benchmark suite as a test case. It has been shown that useful information about a Java program can be extracted at the intermediate representation level, which can be used to understand its ultimate behaviour on a specific hardware platform.

For SPEC JVM98 applications, a significant proportion of the methods called and bytecodes executed are in the class library. This is a significant difference from the Java Grande applications and small microbenchmarks which spend little time in the class library. Pre-compiling the class library to some native representation, or running the programs with a different class library, may yield a significant speed up.

A constant theme of this chapter is that useful information can be gained from a platform-independent study of bytecode level data. We believe that this is borne out in particular in the analysis of methods presented in Table 1.5, where the bytecode counts help to present a different picture of where the JVM engine is spending its time.

This study raises questions about the balance of optimisation work between Java compilers and the JVM. SPEC compiled the SPEC JVM98 benchmarks with a compiler which includes some potentially unsafe optimisations. Since it is now more common to defer this work to run-time, perhaps this should be taken into account in the SPEC JVM98 suite.

Furthermore, some compilers do not implement even the simplest of optimisations, such as producing code for `while` loops which do not end in a `goto`. Most of the optimisations described in the previous section are relatively trivial, and impact little on the overall bytecode usage frequencies.

Clearly, run-time optimisation techniques will always be essential within the JVM, because of the extra information about the run-time architecture available to the JVM. However, it is not obvious that Java compilers are putting much effort into generating efficient bytecode, and it is arguable that the JVM may be bearing an unreasonable part of the burden of performing these optimisations. Further work is required to determine exactly which statically-applied optimisations, if any, can yield a significant performance improvement over a range of possible JVMs.

References

Adl-Tabatabai, A.-R., Cierniak, M., Lueh, G.-Y., Parikh, V. M., and Stichnoth, J. M. (1998). Fast, effective code generation in a Just-In-Time Java compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 280–290, Montreal, Canada.

Alpern, B., Attanasio, C. R., Barton, J. J., Burke, M. G., P.Cheng, Choi, J.-D., Cocchi, A., Fink, S. J., Grove, D., Hind, M., Hummel, S. F., Lieber, D., Litvinov, V., Mergen, M. F., Ngo, T., Russell, J. R., Sarkar, V., Serrano, M. J., Shepherd, J. C., Smith, S. E., Sreedhar, V. C., Srinivasan, H., and Whaley, J. (2000). The jalapeño virtual machine. *IBM System Journal*, 39(1):211–238.

Antonioli, D. and Pilz, M. (1988). Analysis of the Java class file format. Technical Report 98.4, Dept. of Computer Science, University of Zurich.

Armstrong, E. (1998). Hotspot: A new breed of virtual machine. *Java World*.

Baylor, S., Devarakonda, M., Fink, S., Gluzberg, E., Kalantar, M., Muttineni, P., Barsness, E., Arora, R., Dimpsey, R., and Munroe, S. J. (2000). Java server benchmarks. *IBM Systems Journal*, 39(1):57–81.

Benton, N., Kennedy, A., and Russell, G. (1998). Compiling Standard ML to Java bytecodes. In *3rd ACM SIGPLAN Conference on Functional Programming*, pages 129–140, Baltimore, Maryland, USA.

Bowers, K. R. and Kaeli, D. (1998). Characterising the SPEC JVM98 benchmarks on the Java virtual machine. Technical report, Northeastern University Computer Architecture Research Group, Dept. of

Electrical and Computer Engineering, Boston Massachusetts 02115, USA.

Bull, M., Smith, L., Westhead, M., Henty, D., and Davey, R. (1999). A methodology for benchmarking Java Grande applications. In *ACM 1999 Java Grande Conference*, pages 81–88, Palo Alto, CA, USA.

Bull, M., Smith, L., Westhead, M., Henty, D., and Davey, R. (2000). Benchmarking Java Grande applications. In *Second International Conference and Exhibition on the Practical Application of Java*, Manchester, UK.

Cohen, T. and Gil, J. (2000). Self-calibration of metrics of Java methods. In *Technology of Object-Oriented Languages and Systems*, pages 94–106, Sydney, Australia.

Colnet, D. and Zendra, O. (1999). Optimizations of Eiffel programs: SmallEiffel, the GNU Eiffel compiler. In *Technology of Object-Oriented Languages and Systems*, pages 341–350, Nancy, France.

Corporation, P. S. (1999). *CaffeineMark 3.0*. http://www.pendragon-software.com/pendragon/cm3/.

Daly, C., Horgan, J., Power, J., and Waldron, J. (2001). Platform independent dynamic Java virtual machine analysis: the Java Grande Forum Benchmark Suite. In *Joint ACM Java Grande - ISCOPE 2001 Conference*, pages 106–115, Stanford, CA, USA.

Dieckmann, S. and Hölzle, U. (1999). A study of the allocation behaviour of the SPECjvm98 Java benchmarks. In *13th European Conference on Object Oriented Programming*, pages 92–115, Lisbon, Portugal.

Driesen, K., Lam, P., Miecznikowski, J., Qian, F., and Rayside, D. (2000). On the predictability of invoke targets in Java byte code. In *2nd Workshop on Hardware Support for Objects and Microarchitecture for JAva*, Austin, Texas.

Ishizaki, K., Kawahito, M., Yasue, T., Takeuchi, M., Ogasawara, T., Suganuma, T., Onodera, T., Komatsu, H., and Nakatani, T. (1999). Design, implementation and evaluation of optimisations in a Just-In-Time compiler. In *ACM 1999 Java Grande Conference*, pages 119–128, San Francisco, CA, USA.

Jackson, D. and Waingold, A. (2001). Lightweight extraction of object models from bytecode. *IEEE Transactions on Software Engineering*, 27(2):194–202.

Kazi, I., Chan, H., Stanley, B., and Lilja, D. (2000). Techniques for obtaining high perfromance in Java programs. *ACM Computing Surveys*, 32(3):213–240.

Lee, H. B. (1997). BIT: A tool for instrumenting Java bytecodes. In *USENIX Symposium on Internet Technologies and Systems*, pages 73–82, Monterey, California, U.S.A.

Li, T., John, L. K., Narayanan, V., Sivasubramaniam, A., Sabarinathan, J., and Murthy, A. (2000). Using complete system simulation to characterize SPECjvm98 benchmarks. In *International Conference on Supercomputing*, pages 22–33, Santa Fe, NM, USA.

Lindholm, T. and Yellin, F. (1996). *The Java Virtual Machine Specification*. Addison Wesley.

Neffenger, J. (1999). The Volano report: Which Java platform is fastest, most scalable? *JavaWorld*.

Radhakrishnan, R., Vijaykrishnan, N., John, L., Sivasubramaniam, A., Rubio, J., and Sabarinathan, J. (2001). Java runtime systems: Characterization and architectural implications. *IEEE Transactions on Computers*, 50(2):131–146.

Rayside, D., Mamas, E., and Hons, E. (1999). Compact Java binaries for embedded systems. In *9th NRC/IBM Center for Advanced Studies Conference*, pages 1–14, Toronto, Canada.

Shuf, Y., Serrano, M. J., Gupta, M., and Singh, J. P. (2001). Characterizing the memory behavior of Java workloads: a structured view and opportunities for optimizations. In *Joint International Conference on Measurements and Modeling of Computer Systems*, pages 194–205, Cambridge, MA, USA.

SPEC (1998). SPEC releases SPECjvm98, first industry-standard benchmark for measuring Java virtual machine performance. Press Release. http://www.specbench.org/osg/jvm98/press.html.

Sun Microsystems (2001). The Java HotSpot virtual machine. Technical White Paper, http://java.sun.com/products/hotspot/.

Vallee-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., and Co, P. (1999). Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135.

Waldron, J. (1999). Dynamic bytecode usage by object oriented Java programs. In *Technology of Object-Oriented Languages and Systems*, Nancy, France.

Wilkinson, T. (2000). *KAFFE, A Virtual Machine to run Java Code*. http://www.kaffe.org.

Wolczko, M. (2001a). *Benchmarking Java with Richards and DeltaBlue*. Sun Microsystems Laboratories, http://www.sun.com/research/people/-mario/java_benchmarking/.

Wolczko, M. (2001b). *The Tracing JVM*. Sun Microsystems Laboratories, http://www.experimentalstuff.com/Technologies/TracingJVM.

Zhang, X. and Seltzer, M. I. (2000). HBench: Java: an application-specific benchmarking framework for Java virtual machines. In *ACM Java Grande Conference*, pages 62–70, San Francisco, CA, USA.