Platform Independent Dynamic Java Virtual Machine Analysis: the Java Grande Forum Benchmark Suite

Charles Daly
Computer Applications, Dublin City University, Dublin 9, Ireland.
Jane Horgan
Computer Applications, Dublin City University, Dublin 9, Ireland.
James Power
Dept of Computer Science, National University of Ireland, Maynooth, Co. Kildare, Ireland.
John Waldron
Department of Computer Science, Trinity College, Dublin 2, Ireland.
email: John.Waldron@cs.tcd.ie

ABSTRACT

In this paper we present a platform independent analysis of the dynamic profiles of Java programs when executing on the Java Virtual Machine. The Java programs selected are taken from the Java Grande Forum benchmark suite, and five different Java-to-bytecode compilers are analysed. The results presented describe the dynamic instruction usage frequencies, as well as the sizes of the local variable, parameter and operand stacks during execution on the JVM.

These results, presenting a picture of the actual (rather than presumed) behaviour of the JVM, have implications both for the coverage aspects of the Java Grande benchmark suites, for the performance of the Java-to-bytecode compilers, and for the design of the JVM.

Keywords

Java Virtual Machine, Java Grande

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. The second stage of the process involves hardware specific conversions, perhaps by a JIT compiler for the particular hardware in question, followed by the execution of the code. The problem addressed by this research is that while there exist static tools such as class file viewers to look at this intermediate representation, there is currently no easy way of studying the dynamic behaviour at this point in the program. This research therefore sets out to perform dynamic analysis at the platform independent level and investigate whether or not useful results can be gained. In order to test the technique, the Java Grande Forum's Benchmark suite was used.

JAVA Grande/ ISCOPE 01 Palo Alto CA USA

The remainder of this paper 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. Sections 3 and 4 summarise the profiles of each of the Grande programs studied. In particular, section 3 presents a method-level view of the dynamic profile, while section 4 presents a more detailed bytecode-level view. Section 5 and 6 discuss some of the issues that can affect these figures. Section 5 discusses the influence of compiler choice on dynamic analysis, and describes the variances caused by five of the most common Java compilers. Section 6 profiles the method stack frame sizes, since the size and distribution of data on the stack has an influence on the position-specific bytecodes (e.g. *iconst_1*) used. Section 7 concludes the paper.

2. BACKGROUND

The increasing prominence of internet technology, and the widespread use of the Java programming language has given the Java Virtual Machine (JVM) a unique position in the study of compilers and related technologies. To date, much of this research has concentrated on the performance of the bytecode interpreter, yielding techniques such as Just-In-Time (JIT) and hotspot-centered compilation.

However, the production of bytecode for the JVM is no longer 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 and Scheme, all targeted on the JVM. In previous work we have studied the impact of the choice source language on the dynamic profiles of programs running on the JVM [2]. In this paper we examine the impact of the choice of Java compiler on the dynamic execution of JVM bytecodes, and analyse the degree to which the Java Grande [1] applications can fulfill the role as a standard test suite for these and other aspects of the JVM.

2.1 Dynamic Bytecode-Level Analysis

The *static* bytecode frequency, which is the number of times a bytecode appears in a class file or program has been studied in [3] where a wide difference was found between the bytecodes appearing in different class files, with each class file using on average 25 different bytecodes. The *dynamic* frequency of an instruction is the number of times it is executed during a program run. Dynamic bytecode analysis is a valuable technique for studying the behaviour of Java Programs and the design of the Java Virtual Machine. Even though the majority of Java code executed may now be using some form of JIT compiler, dynamic analysis of interpreted bytecode usage, and associated dynamic analysis of stack frame usages can provide valuable information for profiling of programs and for the design and implementation of virtual machines.

The output of a dynamic bytecode analysis will therefore be important for the design of both Java to bytecode and Just-In-Time bytecode to native compilers. Of particular interest also is the instruction set used by an intermediate representation to implement platform independence. By dynamically analysing the Java bytecodes, lessons may be drawn to facilitate construction of more efficient intermediate representations for both procedural object-oriented programming languages like Java and programming languages from different categories.

Speed comparisons of the Java Grande benchmark suite using different Java Platforms have been performed [1] and differences in execution times have been found, but it has not been known whether the resulting differences measured have been due to the Java compiler, the JIT compiler or the virtual machine implementation on the particular underlying operating system and hardware architecture. This paper shows, by means of the dynamic bytecode analysis technique, that the bytecodes executed by a particular Grande application are very similar for a wide variety of Java compilers, implying compiler choice is not the main explanation of execution speed variations for these programs. In addition, it is possible to study how representative of Grande programs the chosen benchmark suite is.

In order to study dynamic bytecode usage it was necessary to modify the source code of a Java Virtual Machine. Kaffe [4] is an independent implementation of the Java Virtual Machine which was written from scratch and is free from all third party royalties and license restrictions. It comes with its own standard class libraries, including Beans and Abstract Window Toolkit (AWT), native libraries, and a highly configurable virtual machine with a JIT compiler for enhanced performance. Kaffe is available under the Open Source Initiative and comes with complete source code, distributed under the GNU Public License. Versions 1.0.5 and 1.0.6 were used for these measurements.

2.2 Grande Programs Measured

A *Grande* application is one which uses large amounts of processing, I/O, network bandwidth or memory. The Java Grande Forum Benchmark Suite

(http://www.epcc.ed.ac.uk/javagrande/)

is intended to be representative of such applications, and thus to provide a basis for measuring and comparing alternative Java execution environments. It is intended that the suite should include not only applications in science and engineering but also, for example, corporate databases and financial simulations.

- The **moldyn** benchmark is a translation of a Fortran program designed to model the interaction of molecular particles. Its origin as non object-oriented code probably explains its relatively unusual profile, with a few methods which make intensive use of fields within the class, even for temporary and loop-control variables. This program may still represent a large number of Grande type applications that will initially run on the JVM
- The **search** benchmark solves a game of connect-4 on a 6×7 board using alpha-beta pruning. Intended to be memory and numerically intensive, this is also the only application to demonstrate an inheritance hierarchy of depth greater than 2.
- The **euler** benchmark solves a set of equations using a fourth order Runge-Kutta method. This suite demonstrates a considerable clustering of functionality in the `Tunnel` class, as well as a comparatively high percentage of methods with very large local variable requirements.
- The **raytracer** measures the performance of a 3D ray tracer rendering a scene containing 64 spheres. It is represented using a fairly shallow inheritance tree, with functionality (as measured in methods) fairly well distributed throughout the classes.
- The **montecarlo** benchmark is a financial simulation using Monte Carlo techniques to price products derived from the price of an underlying asset. Its use of classical object-oriented get and set methods accounts for the relatively high proportion of methods with no temporary variables and 1 or 2 parameters (including the `this`-reference).

Version 2.0 of the suite (Size A) was used. The default Kaffe maximum heap size of 64M was sufficient for all programs except *mon* which needed a maximum heap size of 128M. The *ray* application failed its validation test when interpreted, but as the failure was by a small amount, it was included in the measurements.

3. DYNAMIC METHOD EXECUTION FREQUENCIES

In this section we present our first dynamic profile of the Grande programs studied. Here we partition the execution profiles based on methods, since these provide both a logical level of modularity at source-code level, as well as a likely unit of granularity for hotspot analysis. It should be noted that these figures are not the usual *time-based* analysis, which will vary considerably between different computer configurations and architectures, but are based on the more platform-independent *bytecode frequency* analysis.

Table 1 shows dynamic method execution frequencies for the most heavily used methods for the Grande applications

Euler	
Method	Frequency
java/lang/Math.abs	24.5
java/lang/Object.<init>	19.6
euler/Statevector.<init>	19.6
euler/Statevector.svect	19.2
java/lang/Math.sqrt†	11.5
euler/Vector2.dot	1.8
euler/Vector2.magnitude	1.4
java/lang/Math.pow†	0.3
Moldyn	
java/lang/Math.sqrt†	19.2
moldyn/particle.velavg	18.6
moldyn/particle.mkekin	18.6
moldyn/particle.force	18.6
moldyn/particle.domove	18.6
moldyn/random.update	1.4
java/lang/String.indexOf	1.0
moldyn/random.seed	0.6
Montecarlo	
java/util/Random.next	31.6
java/lang/Math.log†	18.6
java/util/Random.nextDouble	15.8
java/util/Random.nextGaussian	12.4
java/lang/Math.exp†	12.4
java/lang/Math.sqrt†	6.2
java/lang/StringBuffer.append	0.3
java/lang/System.arraycopy†	0.2
Raytracer	
raytracer/Vec.dot	47.0
raytracer/Vec.sub2	23.2
raytracer/Sphere.intersect	22.8
java/lang/Math.sqrt†	1.6
java/lang/Object.<init>	1.3
raytracer/Vec.<init>	0.7
raytracer/Vec.normalize	0.6
raytracer/Isect.<init>	0.6
Search	
search/Game.wins	46.5
search/SearchGame.ab	10.3
search/Game.makemove	10.3
search/Game.backmove	10.3
search/TransGame.hash	9.3
search/TransGame.transpose	5.3
search/TransGame.transtore	4.0
search/TransGame.transput	4.0

Table 1: ***Dynamic method execution frequencies for the most heavily used methods for the Grande application including native methods, indicated by †.***

Program	Total methods	API %	API native %
eul	3.34e+07	58.0	12.6
mol	5.49e+05	22.7	19.9
mon	8.07e+07	98.7	37.4
ray	4.58e+08	3.1	1.6
sea	7.12e+07	0.0	0.0
average	1.29e+08	36.5	14.3

Table 2: ***Measurements of total number of method calls including native calls by Grande applications.*** **Also shown is the percentage of the total which are in the API, and percentage of total which are in API and are native methods.**

Program	Java method calls		bytecodes executed	
	number	% in API	number	% in API
eul	2.92e+07	51.9	1.46e+10	21.9
mol	4.40e+05	3.4	7.60e+09	0.0
mon	5.05e+07	97.9	2.63e+09	48.3
ray	4.50e+08	1.5	1.18e+10	0.8
sea	7.12e+07	0.0	7.13e+09	0.0
average	1.20e+08	31.0	8.75e+09	14.2

Table 3: ***Measurements of Java method calls excluding native calls made by Grande applications.***

including native methods. It can be seen that virtually all method invocations are to the top 5 methods.

Table 2 shows measurements of the total number of method calls including native calls by Grande applications. For the programs studied, on average 14.3% of methods are API 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 API. This paper is confined to studying how the Java methods execute.

Table 3 shows measurements of the Java method calls excluding native calls. Java method bytecode execution is mostly (86% on average) in the non-API bytecodes of the programs. This is a significant difference from traditional Java applications such as applets or compiler type tools which spend most of the time in the API [5]. Mixed compiled interpreted systems which precompile the API methods to some native format will therefore not be as effective at speeding up Grande applications like these. The finding that API usage is very low may imply that the benchmark suite may not be fully representative of a broad range of Grande applications (see Table 4). It is interesting to observe that while 98% of Java methods are API for the *mon* benchmark, only 48% of the bytecodes executed. All measurements in this paper were made with the Kaffe API library, which may differ from other Java API libraries.

Table 4 shows dynamic measurements of the Java API package method percentages. As would be expected for the programs considered, the applet and awt packages are not used at all as graphics has been removed from the benchmarks. Of interest is that the math package is not used by the benchmarks which simply use the java.lang.Math class. java.math contains only the two classes BigDecimal and BigInteger, which are not that common in Grande applications.

Program	io	lang	net	text	util
eul	2.4	97.6	0.0	0.0	0.0
mol	2.9	82.3	0.8	0.3	13.7
mon	0.0	2.3	0.0	0.0	97.7
ray	0.0	100.0	0.0	0.0	0.0
sea	3.0	80.3	1.1	0.0	15.6

Table 4: *Breakdown of Java (non-native) API method dynamic usage percentages by package for Grande applications.* None of the applications used methods from the applet, awt, beans, math, security or sql packages.

A Grande application should use large amounts of processing, I/O, network bandwidth or memory, yet it is interesting to note how little of the API packages are dynamically used by this benchmark suite.

4. DYNAMIC BYTECODE EXECUTION FREQUENCIES

In this section we present a more detailed view of the dynamic profiles of the Grande 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. This also provides an alternative to typical time-based analysis, which, while useful for efficiency analysis, can be considerably influenced by the underlying architecture's proficiency in dealing with different types of bytecode instructions.

Table 5 shows total (API and non-API) dynamic bytecode usage frequencies by Grande applications. The JVM instruction set has special efficient load and store instructions for the first four local variable array entries, and less efficient generic instructions for higher local variable array positions. The first thing that stands out from Table 5 is that for *mol, sea* and *eul* the highest frequency instruction is a generic load, rather than an efficient load from one of the first four elements of the local variable array. For *mol* one third of instructions are a single load of this type.

Although the Java to bytecode compiler does not have access to dynamic execution data, it should be able to put the most heavily used local variable into one of the efficient slots most of the time (see also Table 10). Alternatively, if the compiler just assigns the local variables in the order they are declared, the application programmer might be able to alter the sequence to increase efficiency in some cases, but not if the compiler always puts the parameters first and there are a large number of these. This is further highlighted later in this paper under dynamic stack frame analysis Table 14.

The *mol* benchmark has the same number of `getfield` as `getstatic` instructions, uses a much smaller set of instruction than the other benchmarks, and does not have method invocations in its high frequency instructions, suggesting it may not have been designed in an object-oriented fashion. The comparison instruction `dcmpg` is also at very high frequency in *mol* relative to the other benchmarks, suggesting something different is happening in the structure of the code involving a high number of dynamic decisions. `invokevirtual` does not appear at all in the high frequency instruction for *eul* or *mol*, and is at 1% for *sea* and 1.9% for *ray* suggesting that worries about the inefficiencies of virtual method invocation in the Java language may have been overstated for Grande applications. Of course, the execution time for the `invokevirtual` instruction will be much higher than for ordinary instructions on any hardware platform. *ray* and *mon* seem to be the most object-oriented program, using `getfield` and `aload_0` to access the `this`-reference as their most frequent instructions.

Category	Number	Bytecodes
misc	5	nop, iinc, athrow, wide, breakpoint
push_const	20	1-20
local_load	25	21-45
array_load	8	46-53
local_store	25	54-78
array_store	8	79-86
stack	9	87-95
arithmetic	24	96-119
logical_shift	6	120-125
logical_boolean	6	126-131
conversion	15	133-147
comparison	5	148-152
conditional_branch	16	153-166, 198, 199
unconditional_branch	2	goto, goto_w
subroutine	3	jsr, ret, jsr_w
table_jump	2	tableswitch, lookupswitch
method_return	6	172-177
object_fields	4	178-181
method_invoke	4	182-185
object_manage	3	new, checkcast, instanceof
array_manage	4	188-190, 197
monitor	2	monitorenter, monitorexit

Table 6: *Categories of Java bytecodes.*

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

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

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 Grande program. For the purposes of this study, the 202 bytecodes can be split into 22 categories as shown in Table 6. By assigning those instructions that behave similarly into groups it is possible to describe clearly what is happening. Table 5 and Table 6 are summarised in Figure 1. As has been noted in [2] *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 optimizations used. As can be seen in Figure 1, *local_load* = 35.9%, *push_const* = 5.8% and *local_store* = 4.2%, giving a total of 45.9% of instructions moving data between operand stack and local variable array. It is also worth not-

eul		mol		mon		ray		sea	
iload	19.7	dload	33.3	aload_0	16.8	getfield	26.1	iload	13.2
aaload	18.2	iload	7.0	getfield	13.7	aload_0	16.1	aload_0	8.5
getfield	16.2	dstore	6.8	iload_1	4.8	aload_1	10.9	getfield	7.3
aload_0	8.3	dcmpg	5.5	daload	4.6	dmul	6.5	iaload	5.3
dmul	4.1	dsub	4.7	ldc2w	4.1	dadd	4.7	istore	5.3
dadd	4.0	dmul	4.3	dload	4.1	dsub	3.7	ishl	4.3
putfield	3.3	getstatic	4.3	dmul	3.4	putfield	3.1	bipush	3.7
iconst_1	3.2	getfield	4.3	dadd	3.3	aload_2	2.8	iload_1	3.6
dload	2.8	aaload	4.2	if_icmplt	3.1	dreturn	1.9	iand	3.5
isub	2.0	dneg	4.1	putfield	3.1	invokevirtual	1.9	iadd	3.5
daload	2.0	dcmpl	4.1	iinc	3.0	invokestatic	1.9	iload_2	2.6
dup	1.7	ifge	4.1	iload_2	2.7	dload_2	1.9	iload_3	2.5
aload_3	1.5	ifle	4.1	bipush	2.4	iload	1.8	ior	2.3
dsub	1.4	dadd	3.4	dsub	2.0	aload	1.3	iconst_1	2.3
aload	1.3	iinc	1.4	invokevirtual	1.9	dload	1.1	iconst_2	2.1
aload_2	1.3	ifgt	1.4	isub	1.7	dconst_0	1.0	dup	2.0
ldc2w	1.1	if_icmplt	1.4	dstore	1.6	dcmpg	1.0	iinc	1.7
iload_3	1.1	dload_1	1.0	iload_3	1.5	ifge	1.0	iastore	1.5
iadd	1.1	putfield	0.1	dastore	1.5	return	1.0	if_icmplt	1.4
dstore	1.0	aload_0	0.1	dup	1.5	dstore	1.0	iconst_4	1.4
ddiv	0.6	nop	0.0	ladd	1.5	iinc	0.9	iconst_5	1.4
dconst_0	0.4	isub	0.0	invokestatic	1.2	if_icmplt	0.9	if_icmple	1.3
aload_1	0.4	lsub	0.0	ddiv	1.1	areturn	0.9	ifeq	1.2
iinc	0.3	fsub	0.0	lmul	1.0	arraylength	0.9	ifne	1.1
if_icmplt	0.3	imul	0.0	lushr	1.0	ifnull	0.9	invokevirtual	1.0
dload_1	0.3	lmul	0.0	land	1.0	aconst_null	0.9	dup2	1.0
dload_3	0.3	fmul	0.0	i2l	1.0	aaload	0.9	isub	0.9
dstore_1	0.2	idiv	0.0	l2i	1.0	astore	0.9	if_icmpgt	0.9
dstore_3	0.2	ldiv	0.0	ireturn	1.0	dstore_2	0.9	goto	0.9
dastore	0.2	lconst_1	0.0	iconst_1	1.0	dload_1	0.2	ldc1	0.9
dneg	0.1	fdiv	0.0	dreturn	0.9	ddiv	0.1	istore_3	0.8
dcmpg	0.1	ddiv	0.0	iload	0.8	dcmpl	0.1	imul	0.7
ifge	0.1	irem	0.0	aload_1	0.8	ifle	0.1	putfield	0.7
if_icmpge	0.1	lrem	0.0	dconst_1	0.7	goto	0.1	iconst_0	0.7
if_icmple	0.1	frem	0.0	dload_3	0.7	invokespecial	0.1	istore_1	0.7

Table 5: ***Total (API and non-API) dynamic bytecode usage frequencies by Grande applications compiled using SUN's javac compiler, Standard Edition (JDK build 1.3.0-C)*** **The top 35 instructions are presented.**

ing that, in practice, loads are dynamically executed roughly ten times as often as stores. There are an equal number of loads and stores in the instruction set, although this seems to be unnecessary dynamically.

5. COMPARISONS OF DYNAMIC BYTECODE USAGES 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 (as compared to, say, C or C++) in that optimisations can be implemented in two separate phases: first when the source program is compiled into bytecode, and again when this bytecode is executed on a specific JVM. We consider here those optimisation which 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.

For the purposes of this study we used five different Java compilers, from the following development environments:

kopi KOPI Java Compiler Version 1.3C
`http://www.dms.at/kopi`

pizza Pizza version 0.39g, 15-August-98

Category	eul	mol	mon	sea	ray	f_i
local_load	37.1	41.5	33.3	36.1	31.4	35.9
object_fields	19.5	8.7	16.8	29.2	8.3	16.5
arithmetic	13.4	16.6	14.0	15.0	5.8	13.0
array_load	20.2	4.2	4.6	0.9	5.7	7.1
push_const	4.9	0.1	8.4	2.1	13.6	5.8
con_bra	0.7	11.1	3.8	3.0	7.7	5.3
local_store	1.6	6.8	2.1	2.9	7.5	4.2
comparison	0.1	9.7	0.3	1.1	0.1	2.3
method_invoke	0.2	0.0	3.1	3.9	1.0	1.6
misc	0.3	1.4	3.0	0.9	1.7	1.5
stack	1.7	0.0	1.9	0.2	3.5	1.5
logical_boolean	0.0	0.0	1.0	0.0	6.1	1.4
method_return	0.2	0.0	1.9	3.8	1.0	1.4
logical_shift	0.0	0.0	1.5	0.0	4.7	1.2
array_store	0.2	0.0	1.5	0.0	1.5	0.6
conversion	0.0	0.0	2.4	0.0	0.4	0.6
array_manage	0.0	0.0	0.4	0.9	0.1	0.3
uncon_bra	0.1	0.0	0.0	0.1	0.9	0.2
monitor	0.0	0.0	0.0	0.0	0.0	0.0
object_manage	0.0	0.0	0.0	0.1	0.0	0.0
subroutine	0.0	0.0	0.0	0.0	0.0	0.0
table_jump	0.0	0.0	0.0	0.0	0.0	0.0

Table 7: ***Dynamic percentages of category usages by the applications in the Java Grande suite.***

Compiler	mol	eul	sea	ray
kopi	7599606497	12475753926	7388409738	11706547525
pizza	7704747144	11431095142	7311241755	11919084828
gcj	7704740202	12540807644	7527673585	11810849733
jdk13	7599606435	11394409844	7103719939	11706547247
borland	7705054344	11431120742	7324210788	11919084856

Table 8: ***Total Non-API dynamic bytecode usage counts for Grande Applications using different compilers. For* gcj*, a minor alteration to the* sea *program source was needed to get it to compile.***

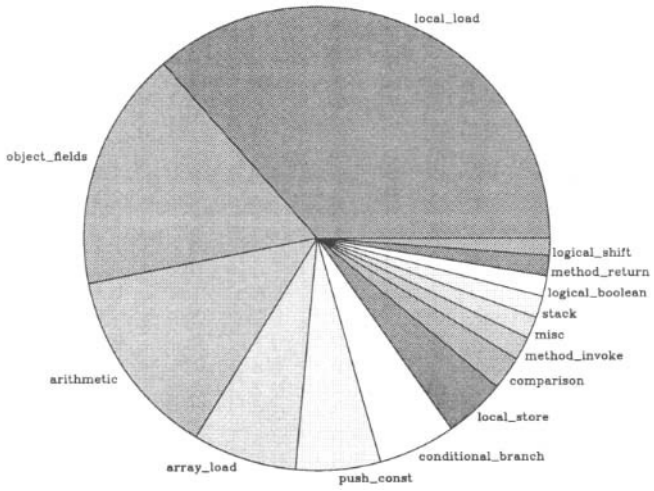

Figure 1: ***Total (API and Non-API) dynamic bytecode execution frequencies by category.***

`http://www.cis.unisa.edu.au/~pizza/`

gcj The GNU Compiler for the Java Programming Language version 2.95.2
`http://sources.redhat.com/java/`

jdk13 SUN's javac compiler, Standard Edition (JDK build 1.3.0-C)

borl Borland Compiler 1.2.006 for Java

The figures for the Java compiler from 1.2 of SUN's JDK, as well as version 1.06 of the IBM Jikes Compiler were also computed, but since the code produced was almost identical to that produced by the compiler from version 1.3 of the JDK we do not consider them further here. As *mon* spends a significant amount of its time in the API, it was not used in this comparison.

Table 8 shows total Non-API dynamic bytecode counts for the Grande programs using different compilers. The API was not recompiled and those bytecodes were excluded from the dynamic comparisons. While it is difficult to draw direct conclusions based on these figures, two facts are at least apparent. First, examining each column of Table 8, it can be seen that there are differences between total number of bytecodes executed for a single application between the different compilers (up to 6% for *sea*). Second, this variance is not consistent through all four applications, and it is clear that a more detailed analysis is necessary to account for these differences.

Instruction	kopi	pizza	gcj	jdk13	borl	f_i
dload	33.3	32.8	32.8	33.3	32.8	33.0
iload	7.0	6.9	6.9	7.0	6.9	6.9
dstore	6.8	6.7	6.7	6.8	6.7	6.7
dcmpl	9.7	4.1	4.1	4.1	4.1	5.2
dsub	4.7	4.7	4.7	4.7	4.7	4.7
dmul	4.3	4.3	4.3	4.3	4.3	4.3
dcmpg	0.0	5.4	5.4	5.5	5.4	4.3
getstatic	4.3	4.2	4.2	4.3	4.2	4.2
getfield	4.3	4.2	4.2	4.3	4.2	4.2
aaload	4.2	4.2	4.2	4.2	4.2	4.2
dneg	4.1	4.1	4.1	4.1	4.1	4.1
ifge	4.1	4.1	4.1	4.1	4.1	4.1
ifle	4.1	4.1	4.1	4.1	4.1	4.1
dadd	3.4	3.4	3.4	3.4	3.4	3.4
iinc	1.4	1.4	1.4	1.4	1.4	1.4
ifgt	1.4	1.4	1.4	1.4	1.4	1.4
dload_1	1.0	1.0	1.0	1.0	1.0	1.0
if_icmpge	0.0	1.4	1.4	0.0	1.4	0.8
goto	0.0	1.4	1.4	0.0	1.4	0.8
if_icmplt	1.4	0.0	0.0	1.4	0.0	0.6
putfield	0.1	0.1	0.1	0.1	0.1	0.1
aload_0	0.1	0.1	0.1	0.1	0.1	0.1
nop	0.0	0.0	0.0	0.0	0.0	0.0
isub	0.0	0.0	0.0	0.0	0.0	0.0
lsub	0.0	0.0	0.0	0.0	0.0	0.0
fsub	0.0	0.0	0.0	0.0	0.0	0.0
imul	0.0	0.0	0.0	0.0	0.0	0.0
lmul	0.0	0.0	0.0	0.0	0.0	0.0
fmul	0.0	0.0	0.0	0.0	0.0	0.0
idiv	0.0	0.0	0.0	0.0	0.0	0.0
ldiv	0.0	0.0	0.0	0.0	0.0	0.0
lconst_1	0.0	0.0	0.0	0.0	0.0	0.0
fdiv	0.0	0.0	0.0	0.0	0.0	0.0
ddiv	0.0	0.0	0.0	0.0	0.0	0.0
irem	0.0	0.0	0.0	0.0	0.0	0.0

Table 9: ***Non-API dynamic bytecode usage frequencies for* mol *using different compilers.* The top 35 instructions are presented.**

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 executed frequencies was carried out. The raw statistics are presented in Table 9, Table 10, Table 11 and Table 12, which

Instruction	kopi	pizza	gcj	jdk13	borl	f_i
aaload	21.6	19.8	21.5	19.9	19.8	20.5
iload	22.4	20.7	5.4	20.8	20.7	18.0
getfield	17.3	17.0	17.4	17.0	17.0	17.1
aload_0	10.0	9.0	10.1	9.0	9.0	9.4
dadd	3.8	4.1	3.8	4.1	4.1	4.0
dmul	3.7	4.1	3.7	4.1	4.1	3.9
iconst_1	2.7	2.9	2.7	2.9	2.9	2.8
putfield	2.6	2.8	2.6	2.8	2.8	2.7
dload	2.5	2.7	2.9	2.7	2.7	2.7
iload_3	1.3	1.4	7.7	1.4	1.4	2.6
isub	1.7	1.9	1.7	1.9	1.9	1.8
iload_2	0.0	0.0	9.1	0.0	0.0	1.8
aload_3	1.7	1.9	1.7	1.9	1.9	1.8
daload	1.6	1.8	1.6	1.8	1.8	1.7
dup	0.1	2.0	0.1	2.0	2.0	1.2
dstore	1.0	1.1	1.4	1.1	1.1	1.1
dsub	0.9	1.0	0.9	1.0	1.0	1.0
ldc2w	1.0	1.1	0.7	1.1	1.1	1.0
iadd	1.0	1.0	0.9	1.0	1.0	1.0
ddiv	0.6	0.7	0.6	0.7	0.7	0.7
aload_2	0.3	0.4	0.3	0.4	0.4	0.4
iinc	0.3	0.3	0.3	0.3	0.3	0.3
iload_1	0.0	0.0	1.4	0.0	0.0	0.3
dconst_0	0.2	0.2	0.2	0.2	0.2	0.2
if_icmpge	0.0	0.3	0.3	0.0	0.3	0.2
goto	0.0	0.3	0.3	0.0	0.3	0.2
dload_1	0.2	0.3	0.0	0.3	0.3	0.2
dload_3	0.2	0.2	0.0	0.2	0.2	0.2
aload_1	0.2	0.2	0.2	0.2	0.2	0.2
dstore_1	0.2	0.2	0.0	0.2	0.2	0.2
dstore_3	0.2	0.2	0.0	0.2	0.2	0.2
dastore	0.2	0.2	0.2	0.2	0.2	0.2
if_icmplt	0.3	0.0	0.0	0.3	0.0	0.1
invokespecial	0.1	0.1	0.1	0.1	0.1	0.1
new	0.1	0.1	0.1	0.1	0.1	0.1

Table 10: *Non-API dynamic bytecode usage frequencies for* eul *using different compilers.* The top 35 instructions are presented.

Instruction	kopi	pizza	gcj	jdk13	borl	f_i
iload	13.4	12.9	12.4	13.2	12.8	12.9
aload_0	9.6	8.3	8.9	8.6	8.3	8.7
getfield	7.9	7.1	7.6	7.3	7.1	7.4
iaload	5.2	5.2	5.1	5.4	5.2	5.2
istore	5.1	5.2	5.2	5.4	5.2	5.2
ishl	4.1	4.2	4.1	4.3	4.2	4.2
bipush	3.6	3.7	4.3	3.8	3.6	3.8
iadd	4.2	3.4	4.1	3.5	3.4	3.7
iand	3.3	3.4	4.1	3.5	3.4	3.5
iload_1	3.8	3.5	2.8	3.6	3.5	3.4
iload_2	2.5	2.6	3.3	2.6	2.5	2.7
iload_3	2.7	2.5	3.3	2.5	2.5	2.7
ior	2.2	2.3	2.2	2.3	2.3	2.3
iconst_1	2.0	2.2	2.0	2.3	2.2	2.1
iconst_2	2.0	2.0	2.0	2.1	2.0	2.0
dup	1.5	1.9	1.8	2.0	1.9	1.8
iinc	1.7	1.7	1.6	1.7	1.7	1.7
iconst_5	1.8	1.4	1.7	1.4	1.4	1.5
iconst_0	0.7	2.5	0.7	0.7	2.6	1.4
iastore	1.4	1.4	1.4	1.5	1.4	1.4
iconst_4	1.4	1.4	1.4	1.4	1.4	1.4
if_icmpgt	0.9	1.7	1.4	0.9	1.7	1.3
goto	0.8	1.5	1.5	0.5	1.5	1.2
ifeq	1.2	0.1	1.9	1.6	0.1	1.0
invokevirtual	1.0	1.0	0.9	1.0	1.0	1.0
isub	0.9	0.9	0.8	0.9	0.9	0.9
if_icmple	1.3	0.6	0.8	1.3	0.6	0.9
if_icmpeq	0.2	1.7	0.2	0.2	1.7	0.8
if_icmplt	1.3	0.5	0.5	1.4	0.5	0.8
ldc1	0.8	0.8	0.8	0.8	0.9	0.8
istore_3	0.8	0.8	0.8	0.8	0.8	0.8
imul	0.6	0.7	0.6	0.7	0.7	0.7
if_icmpge	0.1	1.1	0.9	0.1	1.1	0.7
putfield	0.7	0.7	0.7	0.7	0.7	0.7
dup2	0.1	1.0	0.3	1.0	1.0	0.7

Table 11: *Non-API dynamic bytecode usage frequencies for* sea *using different compilers.* The top 35 instructions are presented. For *gcj*, a minor alteration to the program source was needed to get it to compile.

show the top 35 most executed instructions for each application. In order to analyse these tables, the differences in each row were selected, and the relevant sections of the corresponding source code was then examined. Below we summarise the main differences exhibited in these tables.

5.1 Main Compiler Differences

There were three main differences between the optimisations implemented by the compilers:

5.1.1 Loop Structure

The figures show a difference in the use of comparison and jump instructions between the compilers. For each usage of the `if_cmplt` instruction by *kopi* and *jdk13* there is a corresponding usage of `goto` and `if_cmpge` by *pizza*, *gcj* and *borland*. This can be explained by the implementation of loop structures. for example, a loop of the form:

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

is implemented by the different compilers as follows:

kopi/jdk13	*pizza/gcj/borland*

`goto end`
`beg:` *stats*
`end:` *expr*
`if_cmplt beg`
	`beg:` *expr*
	`if_cmpge end`
	stats
	`goto beg`
	`end:`

A simple static analysis would regard these as similar implementations, but the dynamic analysis clearly shows the savings resulting from the *kopi/jdk13* approach.

5.1.2 Specialised `load` *Instructions*

Table 10 and Table 11 highlight an important difference between the compilers in their treatment of specialised `iload` instructions. *gcj* gives a significantly lower usage of the generic `iload` instruction relative to all other compilers, and a corresponding increase in the more specific `iload_2` and `iload_3` instructions showing that this compiler is attempting to optimise the programs for integer usage.

However, it is interesting to note the failure of this approach as demonstrated by Table 9 and Table 12, where the differences in `iload` instructions are not significant. This can be explained directly by the nature of the programs involved - *mol* and *ray* make greater use of `doubles` and objects respectively, and *gcj* makes no attempt to optimise the stack positions for these types.

5.1.3 Usage of the `dup` *Instruction*

Instruction	kopi	pizza	gcj	jdk13	borl	f_i
getfield	26.3	25.8	26.0	26.3	25.8	26.0
aload_0	16.2	15.8	16.0	16.1	15.8	16.0
aload_1	10.9	10.7	10.8	10.9	10.7	10.8
dmul	6.6	6.5	6.5	6.6	6.5	6.5
dadd	4.7	4.6	4.7	4.7	4.6	4.7
dsub	3.7	3.6	3.7	3.7	3.6	3.7
putfield	3.0	3.0	3.0	3.0	3.0	3.0
aload_2	2.8	2.7	2.8	2.8	2.7	2.8
invokestatic	1.9	1.9	1.9	1.9	1.9	1.9
dreturn	1.9	1.8	1.8	1.9	1.8	1.8
invokevirtual	1.9	1.8	1.8	1.9	1.8	1.8
iload	1.9	1.8	1.8	1.9	1.8	1.8
dload	1.1	1.1	2.9	1.1	1.1	1.5
dload_2	1.9	1.8	0.0	1.9	1.8	1.5
aconst_null	0.9	1.7	0.9	0.9	1.7	1.2
aload	1.2	1.2	1.2	1.2	1.2	1.2
dstore	1.0	1.0	1.8	1.0	1.0	1.2
ifge	1.0	1.0	1.0	1.0	1.0	1.0
iinc	0.9	0.9	0.9	0.9	0.9	0.9
dconst_0	0.9	0.9	0.9	0.9	0.9	0.9
areturn	0.9	0.9	0.9	0.9	0.9	0.9
return	0.9	0.9	0.9	0.9	0.9	0.9
arraylength	0.9	0.9	0.9	0.9	0.9	0.9
aaload	0.9	0.9	0.9	0.9	0.9	0.9
astore	0.9	0.9	0.9	0.9	0.9	0.9
dcmpg	0.0	1.0	1.0	1.0	1.0	0.8
dstore_2	0.9	0.9	0.0	0.9	0.9	0.7
goto	0.1	0.9	1.0	0.1	0.9	0.6
if_icmpge	0.0	0.9	0.9	0.0	0.9	0.5
ifnull	0.9	0.0	0.9	0.9	0.0	0.5
if_icmplt	0.9	0.0	0.0	0.9	0.0	0.4
if_acmpeq	0.0	0.9	0.0	0.0	0.9	0.4
dcmpl	1.1	0.1	0.1	0.1	0.1	0.3
dload_1	0.2	0.2	0.2	0.2	0.2	0.2
ddiv	0.1	0.1	0.1	0.1	0.1	0.1

Table 12: *Non-API dynamic bytecode usage frequencies for* ray *using different compilers.* The top 35 instructions are presented.

There is a dramatic difference in the use of dup instructions show in Table 10 and, to a lesser extent, in Table 11, with *kopi* and *gcj* having a much lower usage than the other compilers. (dup instructions do not account for a significant proportion of bytecode usage in the other applications). This can be explained by the usage of the shorthand arithmetic instructions (such as +=) in the source Java code. For example, the *eul* suite contains lines of the form:

```
r[i][j].a += ...
```

A simple translation of this line to the longer form

```
r[i][j].a = r[i][j].a + ...
```

results in code which references the expression `r[i][j].a` twice.

The *pizza*, *jdk13* and *borland* compilers optimise for the first form by duplicating the value of the expressions. The other two compilers do not, and show a corresponding increase in the usages of `aload`, `aaload` and `getfield` instructions.

The presence of the line in what is evidently a program hotspot gives particular relevance to this compiler optimisation in this case.

5.2 Minor compiler differences

Some minor differences between the frequencies can also be noted as follows:

5.2.1 *Comparisons with 0 and* `null`

As well as generic comparison instructions for each type, Java bytecode has two specialised instructions for comparison with zero: `ifeq` and `ifne`. As can be seen from Table 11, the frequencies for these instructions for both the *pizza* and *borland* compilers is lower than the other compilers, and a price is paid in a correspondingly higher use of `iconst_0` and `if_icmpeq` instructions.

As before, this variance is shown to differing degrees dependent on the application: none of the other three programs rate this difference as significant. However, Java bytecode also has a specialised instruction for comparing object references with null, `ifnull`. The object-intensive program *ray* (Table 12) exhibits the results of the *pizza* and *borland* compilers not using this instruction, with a corresponding increase in `aconst_null` and `if_acmpeq` instructions.

5.2.2 *The Decrement Instruction*

There are two approaches to decrementing an integer value. Either you can push minus 1 and add (`iconst_m1`, `iadd`), or push 1 and subtract (`iconst_1`, `isub`). Only the *kopi* and *gcj* compilers choose the former, and so Table 11 shows an increase in the use of `iadd` instructions, along with a corresponding drop in the use of `iconst_1` instructions.

5.2.3 *Constant Propagation*

The *gcj* compiler does not do as much constant propagation as the other compilers and this is evidenced in Table 10. The *eul* application has a number of constant fields, and this is reflected by a drop in `ldc2w` instructions, and a corresponding increase in the number of `getfield` instructions.

5.2.4 *Comparison operations*

A minor variation is shown in Table 9 for the usages of `dcmpl` and `dcmpg` instructions, with the *kopi* compiler showing a strong preference for the former; the dependent statement blocks in the corresponding if-statements are reorganised accordingly.

6. 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 will 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 dynamically examine the 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.

Table 13 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 Grande applications have some zero param-

Local variable array size						
size	eul	mol	mon	ray	sea	f_i
0	0.0	0.0	0.0	0.0	0.0	0.0
1	45.4	0.9	27.2	2.2	0.0	15.1
2	28.5	1.0	51.9	48.0	0.0	25.9
3	23.6	23.5	0.5	24.4	9.3	16.3
4	2.4	0.8	0.5	0.6	0.0	0.9
5	0.0	23.9	0.0	0.0	13.4	7.5
6	0.0	0.0	0.1	0.3	67.0	13.5
7	0.0	0.0	0.0	0.8	0.0	0.2
8	0.0	0.1	0.0	0.2	0.0	0.1
> 8	0.0	49.5	19.8	23.3	10.3	20.6
Parameter size						
size	eul	mol	mon	ray	sea	f_i
0	0.0	0.1	0.0	0.0	0.0	0.0
1	47.3	3.7	47.3	2.9	24.9	25.2
2	52.7	1.2	52.1	71.3	10.3	37.5
3	0.0	47.4	0.4	23.9	18.3	18.0
4	0.0	0.7	0.2	0.8	46.5	9.6
5	0.0	23.3	0.0	0.3	0.0	4.7
6	0.0	0.0	0.0	0.3	0.0	0.1
7	0.0	23.3	0.0	0.5	0.0	4.8
8	0.0	0.0	0.0	0.1	0.0	0.0
> 8	0.0	0.5	0.0	0.0	0.0	0.1
Temporary variable size						
size	eul	mol	mon	ray	sea	f_i
0	73.9	26.5	79.4	75.2	4.0	51.8
1	22.0	0.3	0.3	0.3	0.0	4.6
2	3.7	24.0	0.3	0.6	55.8	16.9
3	0.3	0.1	0.0	0.6	0.0	0.2
4	0.0	23.3	0.1	0.0	19.6	8.6
5	0.0	0.0	0.0	0.0	10.3	2.1
6	0.0	0.1	0.0	0.0	0.0	0.0
7	0.0	0.0	0.0	23.1	0.0	4.6
8	0.0	0.7	19.8	0.0	0.0	4.1
> 8	0.0	25.1	0.0	0.2	10.3	7.1

Table 13: ***Dynamic percentages of local variable array sizes, as well as temporary and parameter sizes for Grande programs compiled with KOPI Java Compiler Version 1.3C.*** **As in previous tables, the local variable array and parameter sizes include the `this`-reference for non-static methods.**

Operand stack size						
size	eul	mol	mon	ray	sea	f_i
0	22.4	0.7	0.4	1.3	0.0	5.0
1	0.3	0.2	0.4	0.6	0.0	0.3
2	0.5	0.4	1.3	0.1	5.3	1.5
3	22.6	2.1	0.8	0.8	46.5	14.6
4	28.1	2.0	25.8	0.8	0.0	11.3
5	22.2	24.0	51.1	23.6	38.9	32.0
6	4.0	70.6	20.0	48.7	9.3	30.5
7	0.0	0.0	0.1	23.4	0.0	4.7
8	0.0	0.0	0.1	0.2	0.0	0.1
> 8	0.0	0.0	0.2	0.6	0.0	0.2

Table 14: ***Dynamic percentages of maximum operand stack sizes for the methods in the Java Grande programs, compiled with KOPI Java Compiler Version 1.3C.***

eter methods, but these appear as zero in the percentages as they are swamped by the frequently used methods in the Grande applications which have non zero parameter sizes.

An interesting point here is the percentages of methods with local variable array sizes of less than 4, since these methods should be able to exclusively use the specialised versions of `load` and `store` operations dealing with these array locations. These figures are:

eul	mol	mon	ray	sea
97.5%	25.4%	79.6%	74.6%	9.3%

Indeed, these figures are an under-estimation of the possibility of using specialised `load` and `store` operations, since dataflow analysis techniques can reduce these stack sizes further. As already noted, the overall figures for specialised `load` instructions *eul* presented in Table 10 do not seem to reflect the high proportion (97.5%) of the methods which would facilitate this.

Table 14 shows the dynamic percentages for the operand stack sizes; these figures are determined by the complexity of expressions evaluated at run-time, as well as the need to push parameters onto the operand stack before calling a method. Both these factors are reflected in the high operand stack size for both *mol* and *ray*, with the former being clearly influenced by the relatively high percentage of methods called with large numbers of parameters. The high percentage of Java methods with zero operand stack size, particularly in the *eul* application, gives an indication of the proportion of simple constructor calls in the dynamic profile of the program [6].

7. CONCLUSIONS

This paper set out to investigate platform independent dynamic Java Virtual Machine analysis using the Java Grande Forum benchmark suite as a test case. This type of analysis, of course, does not look in any way at hardware specific issues, such as JIT compilers, interpreter design, memory effects or garbage collection which may all have significant impacts on the eventual running time of a Java program, and is limited in this respect. It has been shown above however that useful information about a Java program can be extracted at the intermediate representation level, which can be partly used to understand their ultimate behaviour on a specific hardware platform. The technique has also been shown to help in the design of Java to bytecode compilers.

Although the Java to bytecode compiler does not have access to dynamic execution data, it should be able to put the most heavily used local variable into one of the efficient slots most of the time, yet only the *gcj* compiler seems to make a significant attempt at this. A more common optimisation was in the translation of loop constructs, where each successful iteration involves executing two branching instructions, a potential branch if the condition is false and a backward goto (unconditional branch) at the end of the loop for the *pizza*, *gcj* and *borland* compilers, whereas the other compilers combine both of these into a single conditional branch at the end of the loop.

Overall, this study raises questions about the balance of optimisation work between Java compilers and the interpreter component of the JVM. One possibility is that com-

piler writers are trying to produce as closely as possible the bytecodes produced by the original SUN compiler so as to avoid incompatibility with the runtime bytecode verifier, or platform specific JIT compilers. If this is so, it may explain why various other efficiency improvements have not been used by different compilers.

Clearly, run-time optimisation techniques will always be essential within the JVM, because of both the potential inefficiency of the compiler, and 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.

Platform independent dynamic analysis has been shown to be a useful tool for studying the Grande benchmark suite. For Grande applications Java method execution time is shown to be predominantly in the non-API bytecodes of the programs (86% average). This is a significant difference from traditional Java applications such as applets or compiler type tools which spend most of the time in the API. Since a Grande application should use large amounts of processing, I/O, network bandwidth or memory, it is interesting to note how little of the API packages are dynamically used by this benchmark suite. Precompiling the API to some native representation therefore will not yield significant speedup.

8. REFERENCES

[1] Bull M, Smith L, Westhead M, Henty D and Davey R. *Benchmarking Java Grande Applications*, Second International Conference and Exhibition on the Practical Application of Java, Manchester, UK, April 12-14, 2000.

[2] J. Waldron, *Object Oriented Programs and a Stack Based Virtual Machine*, Journal of South African Computer Society, In press.

[3] D. Antonioli and M. Pilz, *Analysis of the Java Class File Format*, Dept. of Computer Science, University of Zurich, Technical Report 98.4, April 1998.

[4] T.J. Wilkinson, *KAFFE, A Virtual Machine to run Java Code*, <www.kaffe.org> URL last accessed on 20/10/2000

[5] J. Waldron, C. Daly, D. Gray and J. Horgan, *Comparison of Factors Influencing Bytecode Usage in the Java Virtual Machine*, Second International Conference and Exhibition on the Practical Application of Java, Manchester, UK, April 12-14, 2000.

[6] J. Waldron and O. Harrison, *Analysis of Virtual Machine Stack Frame Usage by Java Methods*, Third IASTED Conference on Internet and Multimedia Systems and Applications (IMSA), Nassau, Grand Bahamas, Oct 18-21, 1999.