Some Measurements of Java-to-bytecode Compiler Performance in the Java Virtual Machine

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.

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.

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 Interpreter

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.

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 discusses the influence of compiler choice on dynamic analysis, and describes the variances caused by five of the most common Java compilers. Section 7 concludes the paper.

E-mail address for correspondence:
John.Waldron@cs.tcd.ie

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 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 dynami-

Accepted for the Workshop on Java Optimization Strategies for Embedded Systems
University of Genova, Italy, 1 April, 2001

mol methods	freq
java/lang/Math.sqrt†	19.4
moldyn/particle.velavg	18.8
moldyn/particle.mkekin	18.8
moldyn/particle.force	18.8
moldyn/particle.domove	18.8
moldyn/random.update	1.4
moldyn/random.seed	0.6
java/lang/Math.log†	0.6

eul methods	frequency
java/lang/Math.abs	24.5
java/lang/Object.<init>	19.6
euler/Statevector.<init>	19.5
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

sea methods	freq
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

ray methods	frequency
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

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

cally 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 those chosen for the benchmark suite are.

In order to study dynamic bytecode usage it was necessary to modify the source code of a Java Virtual Machine. Kaffe [3] 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. Version: 1.0.5 was 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 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.

Accepted for the Workshop on Java Optimization Strategies for Embedded Systems
University of Genova, Italy, 1 April, 2001

Program	Total methods	API %	API native %
mol	5.45e+05	22.0	20.0
sea	7.12e+07	0.0	0.0
eul	3.34e+07	58.0	12.6
ray	4.58e+08	3.1	1.6
average	1.41e+08	20.8	8.6

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.

- 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).

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.

As Kaffe and the JVM are not yet mature technologies for Grande applications, some programs in the suite fail to compute the correct result. It was decided to exclude the *montecarlo* benchmark from this study as it failed by a large amount when interpreted, but *raytracer* was included as the error in the result was very small.

Table 1 shows dynamic method execution frequencies for the most heavily used methods for the Grande applications including native methods. It can be seen that virtually all execution time is spent in at most five methods for these applications.

Table 2 shows measurements of the total number of method calls including native calls by Grande applications. For the programs studied, on average 8.6% 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. With the exception of the *eul* benchmark, Java method execution time is virtually all 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 [4]. Mixed compiled interpreted systems which precompile the API methods to some native format will therefore

Program	Java method calls		bytecodes executed	
	number	% in API	number	% in API
mol	4.36e+05	2.5	7.60e+09	0.0
sea	7.12e+07	0.0	7.39e+09	0.0
eul	2.92e+07	51.9	1.58e+10	20.9
ray	4.50e+08	1.5	1.18e+10	0.8
average	1.38e+08	14.0	1.06e+10	5.4

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

Program	io	lang	net	text	util
mol	4.0	74.7	1.1	0.4	19.8
sea	4.9	68.4	1.5	0.0	25.2
eul	2.4	97.5	0.0	0.0	0.0
ray	0.0	100.0	0.0	0.0	0.0

Table 4: *Breakdown of Java 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.

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 also possible to observe that since 51% of Java methods are API for the *eul* benchmark, but only 21% of the bytecodes executed, that the API methods are smaller in size than the Grande program's methods. 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 major interest is that the math package is not used by the benchmarks which implies either the benchmarks are not representative of numerical programs or the math package is not in fact of much use to such programs which simply use the java.lang.Math class. 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

mol		sea		eul		ray	
dload	33.3	iload	13.2	iload	19.7	getfield	26.1
iload	7.0	aload_0	8.6	aaload	18.2	aload_0	16.1
dstore	6.8	getfield	7.3	getfield	16.2	aload_1	10.9
dcmpg	5.5	iaload	5.4	aload_0	8.3	dmul	6.5
dsub	4.7	istore	5.3	dmul	4.1	dadd	4.7
dmul	4.3	ishl	4.3	dadd	4.0	dsub	3.7
getstatic	4.3	bipush	3.8	putfield	3.3	putfield	3.1
getfield	4.3	iload_1	3.6	iconst_1	3.2	aload_2	2.8
aaload	4.2	iand	3.5	dload	2.8	dreturn	1.9
dneg	4.1	iadd	3.5	isub	2.0	invokevirtual	1.9
dcmpl	4.1	iload_2	2.6	daload	2.0	invokestatic	1.9
ifge	4.1	iload_3	2.5	dup	1.7	dload_2	1.9
ifle	4.1	ior	2.3	aload_3	1.5	iload	1.8
dadd	3.4	iconst_1	2.3	dsub	1.4	aload	1.3
iinc	1.4	iconst_2	2.1	aload	1.3	dload	1.1
ifgt	1.4	dup	2.0	aload_2	1.3	dconst_0	1.0
if_icmplt	1.4	iinc	1.7	ldc2w	1.1	dcmpg	1.0
dload_1	1.0	ifeq	1.6	iload_3	1.1	ifge	1.0
putfield	0.1	iastore	1.5	iadd	1.1	return	1.0
aload_0	0.1	if_icmplt	1.4	dstore	1.0	dstore	1.0
nop	0.0	iconst_4	1.4	ddiv	0.6	iinc	0.9
isub	0.0	iconst_5	1.4	dconst_0	0.4	if_icmplt	0.9
lsub	0.0	if_icmple	1.3	aload_1	0.4	areturn	0.9
fsub	0.0	invokevirtual	1.0	iinc	0.3	arraylength	0.9
imul	0.0	dup2	1.0	if_icmplt	0.3	ifnull	0.9
lmul	0.0	isub	0.9	dload_1	0.3	aconst_null	0.9
fmul	0.0	if_icmpgt	0.9	dload_3	0.3	aaload	0.9
idiv	0.0	ldc1	0.8	dstore_1	0.2	astore	0.9
ldiv	0.0	istore_3	0.8	dstore_3	0.2	dstore_2	0.9
lconst_1	0.0	imul	0.7	dastore	0.2	dload_1	0.2
fdiv	0.0	ifne	0.7	dneg	0.1	ddiv	0.1
ddiv	0.0	putfield	0.7	dcmpg	0.1	dcmpl	0.1
irem	0.0	iconst_0	0.7	ifge	0.1	ifle	0.1
lrem	0.0	istore_1	0.7	if_icmpge	0.1	goto	0.1
frem	0.0	if_icmpne	0.6	if_icmple	0.1	invokespecial	0.1

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.

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 8). 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.

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 `dcmpl` 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* seems to be the most object-oriented program, using `getfield` as its most frequent instruction, followed by `aload_0` to access the `this`-reference.

In order to study overall bytecode usages across the pro-

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 6: *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.*

grams, 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.

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
`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.

Table 6 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 6, it can be seen that there are significant differences between the bytecodes executed for a single application between the different compilers. 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.

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 7, Table 8, Table 9 and Table 10, which 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:

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.

Specialised `load` Instructions Table 8 and Table 9 highlight an important difference between the compilers in their

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

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

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 7 and Table 10, 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.

Usage of the `dup` Instruction There is a dramatic difference in the use of `dup` instructions show in Table 8 and, to a lesser extent, in Table 9, 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.

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 9: *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.

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

5.2 Minor compiler differences

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

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 9, 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 10) exhibits the results of the *pizza* and *borland* compilers not using this instruction, with a corresponding increase in `aconst_null` and `if_acmpeq` instructions.

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 9 shows an increase in the use of `iadd` instructions, along with a corresponding drop in the use of `iconst_1` instructions.

Constant Propagation The *gcj* compiler does not do as much constant propagation as the other compilers and this is evidenced in Table 8. 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.

Accepted for the Workshop on Java Optimization Strategies for Embedded Systems
University of Genova, Italy, 1 April, 2001

Comparison operations A minor variation is shown in Table 7 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 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 programs 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 compiler 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. 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 unreliability 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 the studying the Grande benchmark suite. For Grande applications Java method execution time is shown to be virtually all 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. 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.

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 major interest is that the math package is not used by the benchmarks which implies either the benchmarks are not representative of numerical programs or the math package is not in fact of much use to such programs which simply use the java.lang.Math class.

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

[4] 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.