CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2003; **15**:459–484 (DOI: 10.1002/cpe.666)

Platform independent dynamic Java virtual machine analysis: the Java Grande Forum benchmark suite

David Gregg [1], James Power[2] and John Waldron[1,*,†]

[1]*Department of Computer Science, Trinity College, Dublin 2, Ireland*
[2]*Department of Computer Science, National University of Ireland, Maynooth, Co. Kildare, Ireland*

SUMMARY

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. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: Java Virtual Machine; Java Grande; instruction frequency; method profiling

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 [1]. 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 (e.g. [2]), 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 [3,4] was used.

*Correspondence to: John Waldron, Department of Computer Science, Trinity College, Dublin 2, Ireland
†E-mail: John.Waldron@cs.tcd.ie

Received 31 August 2001
Revised 21 January 2002

The remainder of this paper is organized 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 summarize 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. Sections 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 variations 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 used (e.g. *iconst_1*). 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 in two main areas:

- static analysis of Java class files, for purposes such as optimization [5] or compression [6];
- the performance of the bytecode interpreter, yielding techniques such as just-in-time (JIT) (e.g. [7,8]) and hotspot-centred compilation (see [9] for a survey).

The platform-independent bytecode analysis presented in this paper describes the bytecode as it is interpreted, without the interference of JIT compilation or any machine-specific issues. The virtual machine is instrumented and run in interpretative mode for the purpose of making dynamic measurements. The JIT compiler is not used during the gathering of this data, but we feel the resulting data will be of use to those involved in the design or implementation of JIT compilers and JVMs. 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 [10] and ML [11], 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 [12]. The compiler comparisons presented in this paper help to calibrate this and other such studies by showing the effect of compiler choice on the data collected.

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 [6]. 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 JVM. 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 the 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 JIT bytecode-to-native compilers. Also of particular interest 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 [3,4] and differences in execution times have been found. However, 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-size programs the chosen benchmark suite is.

In order to study dynamic method usage, it was necessary to modify the source code of a JVM. Kaffe [13] is an independent implementation of the JVM distributed under the GNU Public License. It comes with its own standard class libraries, including Beans and Abstract Window Toolkit (AWT), native libraries, and a configurable virtual machine with a JIT compiler for enhanced performance. Kaffe version 1.0.6 was used for these measurements. The results will be the same for any virtual machine using the platform independent dynamic technique, except where there are differences in the Java API library source code, the execution of which we record seperately. The changes to the Kaffe virtual machine, together with the raw data and the tools to analyse it, are available from the authors.

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 [3,4] is intended to be representative of such applications and thus provides 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 applications in the suite are as follows.

- 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 **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 **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).
- 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 **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 two.

Version 2.0 of the benchmark suite (Size A) was used. The default Kaffe maximum heap size of 64 M was sufficient for all programs except *mon*, which needed a maximum heap size of 128 M. The *ray* application failed its validation test when interpreted, but as the failure was by a small amount, it was included in the measurements. All of the applications discussed in Sections 3 and 4 were compiled using SUN's *javac* compiler, standard edition (JDK build 1.3.0-C). It should be noted that the JIT compiler was not used and the bytecodes were dynamically interpreted to generate the measurements.

3. DYNAMIC METHOD EXECUTION FREQUENCIES

In this section we present our dynamic profile of the Grande programs studied. Here we partition the execution profiles based on methods, since these provide both a logical source of modularity at source-code level, as well as a likely unit of granularity for hotspot analysis [14,15]. It should be noted that these figures are not the usual *time-based* analysis such as found in, e.g., [3,4] for the Java Grande suite, or [16] for the SPEC98 suite. Rather, the figures are based on the more platform-independent method frequency and bytecode usage analyses. We believe the platform-independent dynamic measurements capture different (and useful) information compared to time-based measurements. For example the trade-off in JIT compilation could be better based on the number of times a bytecode would be interpreted, rather than the underlying time its execution would take in the run time environment. It should be noted that all measurements in this paper were made with the Kaffe API library, which may differ from other Java API libraries.

Table I 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 I 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. For example, there is no guarantee that API methods have the same bytecode frequencies or execution times as non-API methods.

The figures on the left of Table II show measurements of the Java method calls excluding native calls. A more detailed view is given by the figures on the right of Table II which show the number of bytecodes executed for each application. While nearly 70% of method calls are directed to non-API methods, Java method *execution* is even more focused (92% on average) in the non-API bytecodes of the programs. This is a significant difference from traditional Java applications such as applets or

Table I. Measurements of total number of method calls including native calls by Grande applications, compiled using SUN's *javac* compiler. Also shown is the percentage of the total which are in the API, and percentage of total which are both in the API and are native methods.

Program	Total methods	API %	API native %
eul	3.34×10^7	58.0	12.6
mol	5.49×10^5	22.7	19.9
mon	8.07×10^7	98.7	37.4
ray	4.58×10^8	3.1	1.6
sea	7.12×10^7	0.0	0.0
ave	1.29×10^8	36.5	14.3

Table II. Measurements of Java method calls excluding native calls made by Grande applications compiled using SUN's *javac* compiler.

	Java method calls		bytecodes executed	
Program	Number	% in API	Number	% in API
eul	2.92×10^7	51.9	1.46×10^{10}	0.5
mol	4.40×10^5	3.4	7.60×10^9	0.0
mon	5.05×10^7	97.9	2.63×10^9	38.0
ray	4.50×10^8	1.5	1.18×10^{10}	0.1
sea	7.12×10^7	0.0	7.10×10^9	0.0
ave	1.20×10^8	30.9	8.75×10^9	7.7

compiler-type tools which spend most of the time in the API [17]. 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. It is interesting to observe that while 98% of Java methods are API for the *mon* benchmark, these account for only 38% of the bytecodes executed. Again, this point highlights the greater information provided by a bytecode-level analysis.

Table III shows dynamic measurements of the Java API package *method call* percentages and Table IV shows API *bytecode* percentages. The figures in these two tables are broadly similar, implying the API methods each execute the same number of bytecodes. As would be expected for the programs

Table III. Breakdown of Java (non-native) API method dynamic usage percentages by package for Grande applications compiled using SUN's *javac* compiler. The percentages show the number of non-native API method calls directed to methods in each package used.

Package	eul	mol	mon	ray	sea	ave
io	2.4	2.9	0.0	0.0	3.0	1.7
lang	97.6	82.3	2.3	100.0	80.2	72.5
net	0.0	0.8	0.0	0.0	1.1	0.4
text	0.0	0.3	0.0	0.0	0.0	0.1
util	0.0	13.7	97.6	0.0	15.7	25.4

Table IV. Breakdown of Java (non-native) API bytecode percentages by package for Grande applications compiled using SUN's *javac* compiler. The percentages show the proportion of (non-native) API bytecodes executed from each package.

Package	eul	mol	mon	ray	sea	ave
io	7.6	1.2	0.3	0.0	1.2	2.1
lang	92.2	69.5	2.0	99.3	69.6	66.5
net	0.0	1.1	0.0	0.0	1.3	0.5
text	0.0	0.6	0.0	0.0	0.0	0.1
util	0.1	27.6	97.7	0.7	28.0	30.8

considered, the applet and awt packages are not used at all as graphics have been removed from the benchmarks. 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.

Tables V and VI present two contrasting analyses of method usage. Table V ranks methods based on the frequency with which they are called at run-time. Table VI on the other hand ranks methods based on the proportion of total executed bytecodes that they account for. The figures in Table V are related to the *method reuse* factor as described in [16], proposed as an indication of the benefits obtained from JIT compilation. However, we suggest that the difference in rankings between Tables V and VI shows that the method-call figures do not give a full picture of where the program is spending its time. The difference is most striking in *mol*, where Table V seems to show an equal distribution of effort between four methods, yet Table VI clearly shows that just one method, particle.force(), accounts for the majority of the bytecodes executed. In fact, particle.force() contains a significant loop, while the other three methods do not contain any loop at all.

Table V. Dynamic method execution frequencies for the most frequently called methods for the Grande applications, compiled using SUN's *javac* compiler. The percentage represents the proportion of the total number of (non-native) method calls that were calls to this method during the program's execution.

Methods from	%
eul	
java.lang.Math.abs	28.1
java.lang.Object.<init>	22.4
Statevector.<init>	22.4
Statevector.svect	22.0
Vector2.dot	2.1
Vector2.magnitude	1.6
java.io.StreamTokenizer.lookup	0.3
java.io.StreamTokenizer.chrRead	0.2
mol	
particle.velavg	23.3
particle.mkekin	23.3
particle.force	23.3
particle.domove	23.3
random.update	1.8
java.lang.String.indexOf	1.2
random.seed	0.7
java.lang.Object.<init>	0.7
mon	
java.util.Random.next	50.4
java.util.Random.nextDouble	25.2
java.util.Random.nextGaussian	19.8
java.lang.StringBuffer.append	0.6
java.lang.Object.<init>	0.4
PathId.get_dTime	0.2
java.lang.Math.abs	0.2
java.lang.Character.forDigit	0.2
ray	
Vec.dot	47.7
Vec.sub2	23.6
Sphere.intersect	23.1
java.lang.Object.<init>	1.3
Vec.<init>	0.8
Vec.normalize	0.6
Isect.<init>	0.6
RayTracer.intersect	0.4
sea	
Game.wins	46.5
SearchGame.ab	10.3
Game.makemove	10.3
Game.backmove	10.3
TransGame.hash	9.3
TransGame.transpose	5.3
TransGame.transtore	4.0

Table VI. Dynamic method bytecode percentages for the Grande applications, compiled using SUN's *javac* compiler. The percentages in this table represent the proportion of the total number of bytecodes executed by the program that belonged to this method.

Methods from	%
eul	
Tunnel.calculateR	51.3
Tunnel.calculateDamping	16.0
Tunnel.doIteration	8.7
Tunnel.calculateG	6.6
Tunnel.calculateF	6.6
Tunnel.calculateStateVar	4.1
Tunnel.calculateDeltaT	3.3
Statevector.svect	1.5
mol	
particle.force	99.6
particle.mkekin	0.1
particle.domove	0.1
md.runiters	0.1
random.update	0.0
random.seed	0.0
random.<init>	0.0
particle.velavg	0.0
mon	
ReturnPath.computeVariance	19.0
java.util.Random.next	17.4
java.util.Random.nextGaussian	12.4
ReturnPath.computeMean	10.6
MonteCarloPath.computeFluctuationsGaussian	10.3
MonteCarloPath.computePathValue	8.0
RatePath.getReturnCompounded	7.6
java.util.Random.nextDouble	7.3
ray	
Vec.dot	32.8
Sphere.intersect	29.5
Vec.sub2	19.8
RayTracer.intersect	14.0
Vec.normalize	1.0
RayTracer.shade	1.0
Vec.<init>	0.3
Vec.comb	0.3
sea	
Game.wins	32.6
SearchGame.ab	30.7
TransGame.hash	8.2
Game.makemove	8.1
Game.backmove	7.9
TransGame.transpose	7.3
TransGame.transput	3.8
TransGame.transtore	0.6

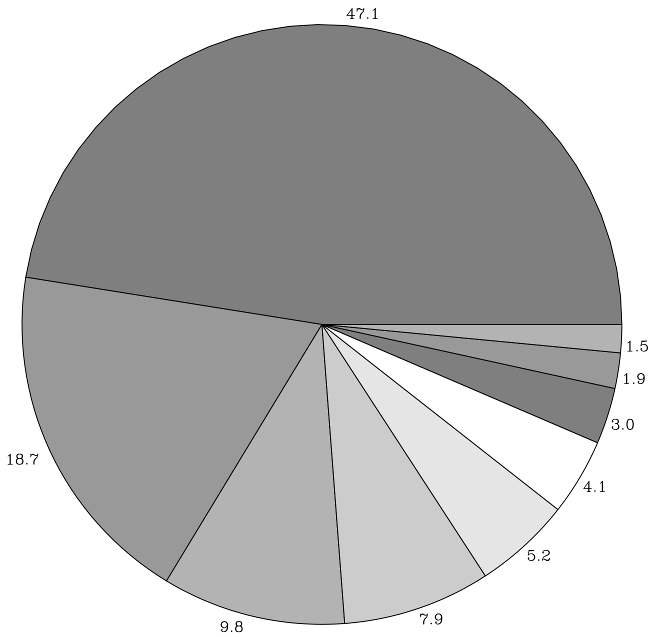

Figure 1. Average dynamic bytecode percentages for the 10 hottest methods for the Grande applications compiled using SUN's *javac* compiler. That is, 47.1% of bytecodes executed were in the hottest method, 18.7% in the second hottest, etc.

Figure 1 shows that on average, for the Grande programs studied, 66% as measured by bytecode use of the execution time is spent in the top two methods.

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 VII 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 VII 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.

Table VII. Total (API and non-API) dynamic bytecode usage frequencies by Grande applications compiled using SUN's *javac* compiler. The top 35 instructions are presented for each application.

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	8.6
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	5.4
dmul	4.1	dsub	4.7	dload	4.1	dadd	4.7	istore	5.3
dadd	4.0	dmul	4.3	ldc2w	4.1	dsub	3.7	ishl	4.3
putfield	3.3	getfield	4.3	dmul	3.4	putfield	3.1	bipush	3.8
iconst_1	3.2	getstatic	4.3	dadd	3.3	aload_2	2.8	iload_1	3.6
dload	2.8	aaload	4.2	if_icmplt	3.1	dload_2	1.9	iadd	3.5
daload	2.0	dcmpl	4.1	putfield	3.1	dreturn	1.9	iand	3.5
isub	2.0	dneg	4.1	iinc	3.0	invokestatic	1.9	iload_2	2.6
dup	1.7	ifge	4.1	iload_2	2.7	invokevirtual	1.9	iload_3	2.5
aload_3	1.5	ifle	4.1	bipush	2.4	iload	1.8	iconst_1	2.3
dsub	1.4	dadd	3.4	dsub	2.0	aload	1.3	ior	2.3
aload	1.3	ifgt	1.4	invokevirtual	1.9	dload	1.1	iconst_2	2.1
aload_2	1.3	if_icmplt	1.4	isub	1.7	dcmpg	1.0	dup	2.0
iadd	1.1	iinc	1.4	dstore	1.6	dconst_0	1.0	iinc	1.7
iload_3	1.1	dload_1	1.0	dastore	1.5	dstore	1.0	ifeq	1.6
ldc2w	1.1	aload_0	0.1	dup	1.5	ifge	1.0	iastore	1.5
dstore	1.0	putfield	0.1	iload_3	1.5	return	1.0	iconst_4	1.4
ddiv	0.6	aastore	0.0	ladd	1.5	aaload	0.9	iconst_5	1.4
aload_1	0.4	aconst_null	0.0	invokestatic	1.2	aconst_null	0.9	if_icmplt	1.4
dconst_0	0.4	aload	0.0	ddiv	1.1	areturn	0.9	if_icmple	1.3
dload_1	0.3	aload_1	0.0	i2l	1.0	arraylength	0.9	dup2	1.0
dload_3	0.3	aload_2	0.0	iconst_1	1.0	astore	0.9	invokevirtual	1.0
if_icmplt	0.3	aload_3	0.0	ireturn	1.0	dstore_2	0.9	if_icmpgt	0.9
iinc	0.3	anewarray	0.0	l2i	1.0	if_icmplt	0.9	isub	0.9
dastore	0.2	areturn	0.0	land	1.0	ifnull	0.9	istore_3	0.8
dstore_1	0.2	arraylength	0.0	lmul	1.0	iinc	0.9	ldc1	0.8
dstore_3	0.2	astore	0.0	lushr	1.0	dload_1	0.2	iconst_0	0.7
dcmpg	0.1	astore_0	0.0	dreturn	0.9	dcmpl	0.1	ifne	0.7
dload_0	0.1	astore_1	0.0	aload_1	0.8	ddiv	0.1	imul	0.7
dneg	0.1	astore_2	0.0	iload	0.8	dload_3	0.1	istore_1	0.7
dreturn	0.1	astore_3	0.0	dconst_1	0.7	dup	0.1	putfield	0.7
goto	0.1	athrow	0.0	dload_3	0.7	goto	0.1	aload	0.6

Table VIII. Categories of Java bytecodes.

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

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 variables into one of the efficient slots most of the time (see also Table XI). Alternatively, if the compiler simply 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 (see Table XVI).

The *mol* benchmark has the same number of `getfield` as `getstatic` instructions, uses a much smaller set of instructions 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 instructions for *eul* or *mol* and is under 2% for the other three applications, 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 programs, using `getfield` and `aload_0` to access the `this`-reference as their most frequent instructions.

Table IX. Dynamic percentages of category usages by the applications in the Java Grande suite compiled using SUN's *javac* compiler.

Category	eul	mol	mon	ray	sea	ave
local_load	37.1	41.4	33.2	36.2	31.4	35.9
object_fields	19.5	8.7	16.8	29.2	8.3	16.5
arithmetic	13.3	16.5	14.0	15.0	5.8	12.9
array_load	20.2	4.2	4.6	0.9	5.8	7.1
push_const	4.7	0.0	8.4	2.1	13.7	5.8
conditional_branch	0.6	11.0	3.8	2.9	6.9	5.0
local_store	1.4	6.8	2.0	2.8	7.5	4.1
comparison	0.1	9.6	0.2	1.1	0.1	2.2
method_invoke	0.3	0.0	3.1	3.9	1.0	1.7
misc	0.3	1.4	3.0	0.9	1.7	1.5
stack	1.7	0.0	1.9	0.1	3.5	1.4
method_return	0.2	0.0	1.9	3.8	1.0	1.4
logical_boolean	0.0	0.0	1.0	0.0	6.1	1.4
logical_shift	0.0	0.0	1.5	0.0	4.7	1.2
conversion	0.0	0.0	2.5	0.0	0.4	0.6
array_store	0.2	0.0	1.5	0.0	1.5	0.6
array_manage	0.0	0.0	0.4	0.9	0.1	0.3
unconditional_branch	0.1	0.0	0.0	0.1	0.5	0.1
table_jump	0.0	0.0	0.0	0.0	0.0	0.0
subroutine	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
monitor	0.0	0.0	0.0	0.0	0.0	0.0

For the purposes of this study, the 202 bytecodes can be split into 22 categories as shown in Table VIII. By assigning those instructions that behave similarly into groups it is possible to describe clearly what is happening. Table IX is summarized in Figure 2. As is the case for the programs studied in [12] *local_load*, *push_const* and *local_store* instruction categories account for very close to 40% of instructions executed, a property of the JVM instruction set, irrespective of source language, compiler or compiler optimisations used. As can be seen in Figure 2, *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 noting that, in practice, loads are dynamically executed roughly ten times as often as stores.

We have shown above that for Grande applications Java method execution time is shown to be predominantly in the non-API bytecodes of the programs (92% 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. We have shown 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 Tables V and VI, where the bytecode counts help to present a different picture of where the interpreter is spending its time.

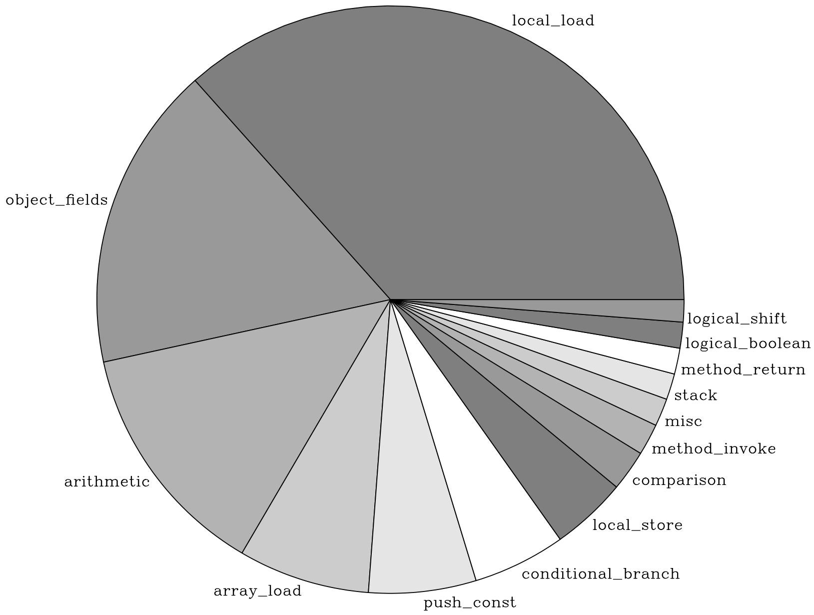

Figure 2. A summary of dynamic percentages of category usages by the applications in the Java Grande suite compiled using SUN's *javac* compiler.

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 (compared to, say, C or C++) in that optimizations 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. In C, platform-independent and platform-dependent optimizations can of course both be performed. The difference is that a Java program is distributed and stored in a platform-independent form, giving rise to a distinct division of labour, whereas in C the ultimate hardware is known at compile time so there is no reason to ever divide the phases of optimization. We consider here those optimizations 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.

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/

Table X. Percentage differences for total non-API dynamic bytecode usage, relative to SUN's *javac* compiler, for Grande Applications. In each case the figures represent the per cent increase in the total number of bytecodes executed compared with *jdk13*. For *gcj*, a minor alteration to the sea program source was needed to get it to compile.

Compiler	eul	mol	mon	ray	sea	ave
jdk13	0.0	0.0	0.0	0.0	0.0	0.0
pizza	0.3	1.4	4.9	1.8	2.9	2.3
borl	0.3	1.4	4.9	1.8	3.1	2.3
kopi	8.1	0.0	1.2	0.0	3.6	2.6
gcj	8.7	1.4	6.1	0.9	6.0	4.6

- **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 http://www.borland.com/

The API was not recompiled and those bytecodes have been excluded from the dynamic comparisons in this section.

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 X shows the percentage differences in total non-API dynamic bytecode counts for the Grande programs using different compilers, as compared to the JDK. While it is difficult to draw direct conclusions based on these figures, two facts are at least apparent. First, examining each column of Table X, it can be seen that there are differences between total number of bytecodes executed for a single application between the different compilers. Second, this variance is not consistent through all five applications and it is clear that a more detailed analysis, which we now present, is necessary to account for these differences.

Ideally, the optimizations 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. A static analysis would involve looking at the bytecodes for every method in every application and seeing where they differed. However, one extra *goto*, for example, might never be executed, might be executed only once, or might produce millions of extra bytecodes. Therefore dynamic analysis is superior to static analysis in highlighting significant compiler differences.

Instead, a detailed analysis of the dynamic bytecode executed frequencies was carried out. The raw statistics are presented in Tables XI–XV which show the top 35 most executed instructions for each

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

Instruction	borl	gcj	kopi	pizza	jdk13	ave
aaload	18.2	19.8	19.9	18.2	18.3	18.9
iload	19.8	7.8	21.3	19.8	19.8	17.7
getfield	16.2	16.6	16.5	16.2	16.2	16.3
aload_0	8.3	9.4	9.2	8.3	8.3	8.7
dmul	4.1	3.8	3.8	4.1	4.1	4.0
dadd	4.0	3.7	3.7	4.0	4.0	3.9
putfield	3.3	3.0	3.0	3.3	3.3	3.2
iconst_1	3.2	3.0	3.0	3.2	3.2	3.1
dload	2.8	3.0	2.6	2.8	2.8	2.8
iload_3	1.1	6.1	1.0	1.1	1.1	2.1
isub	2.0	1.9	1.9	2.0	2.0	2.0
daload	2.0	1.8	1.8	2.0	2.0	1.9
aload_3	1.5	1.6	1.3	1.5	1.5	1.5
iload_2	0.0	7.2	0.0	0.0	0.0	1.4
dsub	1.4	1.3	1.3	1.4	1.5	1.4
aload	1.4	1.0	1.3	1.4	1.4	1.3
aload_2	1.3	1.2	1.2	1.3	1.3	1.3
ldc2w	1.1	0.9	1.1	1.1	1.2	1.1
dstore	1.0	1.3	1.0	1.0	1.0	1.1
iadd	1.1	1.0	1.1	1.1	1.1	1.1
dup	1.7	0.0	0.0	1.7	1.7	1.0
ddiv	0.6	0.6	0.6	0.6	0.6	0.6
dconst_0	0.4	0.3	0.3	0.4	0.4	0.4
aload_1	0.4	0.4	0.4	0.4	0.4	0.4
iinc	0.3	0.3	0.3	0.3	0.3	0.3
if_icmpge	0.4	0.4	0.1	0.4	0.1	0.3
goto	0.4	0.4	0.1	0.4	0.1	0.3
iload_1	0.0	1.1	0.0	0.0	0.0	0.2
dload_1	0.3	0.0	0.2	0.3	0.3	0.2
dload_3	0.3	0.0	0.3	0.3	0.3	0.2
dstore_1	0.2	0.0	0.2	0.2	0.2	0.2
dstore_3	0.2	0.0	0.2	0.2	0.2	0.2
dastore	0.2	0.2	0.2	0.2	0.2	0.2
dneg	0.1	0.1	0.1	0.1	0.1	0.1
if_icmplt	0.0	0.0	0.3	0.0	0.3	0.1

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

Instruction	borl	gcj	kopi	pizza	jdk13	ave
dload	32.8	32.8	33.3	32.8	33.3	33.0
iload	6.9	6.9	7.0	6.9	7.0	6.9
dstore	6.7	6.7	6.8	6.7	6.8	6.7
dcmpl	4.1	4.1	9.7	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	5.4	5.4	0.0	5.4	5.5	4.3
aaload	4.2	4.2	4.2	4.2	4.2	4.2
getstatic	4.2	4.2	4.3	4.2	4.3	4.2
getfield	4.2	4.2	4.3	4.2	4.3	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	1.4	1.4	0.0	1.4	0.0	0.8
goto	1.4	1.4	0.0	1.4	0.0	0.8
if_icmplt	0.0	0.0	1.4	0.0	1.4	0.6
aload_0	0.1	0.1	0.1	0.1	0.1	0.1
putfield	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
aconst_null	0.0	0.0	0.0	0.0	0.0	0.0
iconst_m1	0.0	0.0	0.0	0.0	0.0	0.0
iconst_0	0.0	0.0	0.0	0.0	0.0	0.0
iconst_1	0.0	0.0	0.0	0.0	0.0	0.0
iconst_2	0.0	0.0	0.0	0.0	0.0	0.0
iconst_3	0.0	0.0	0.0	0.0	0.0	0.0
iconst_4	0.0	0.0	0.0	0.0	0.0	0.0
iconst_5	0.0	0.0	0.0	0.0	0.0	0.0
lconst_0	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
fconst_0	0.0	0.0	0.0	0.0	0.0	0.0
fconst_1	0.0	0.0	0.0	0.0	0.0	0.0

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

Instruction	borl	gcj	kopi	pizza	jdk13	ave
aload_0	17.1	19.8	20.7	17.1	17.9	18.5
getfield	17.0	17.4	18.2	17.0	17.8	17.5
daload	7.0	6.9	7.3	7.0	7.3	7.1
iload_1	5.9	5.8	6.1	5.9	6.1	6.0
dload	4.7	4.6	4.8	4.7	4.9	4.7
dadd	4.7	4.6	4.8	4.7	4.9	4.7
iinc	4.7	4.6	4.8	4.7	4.9	4.7
iload_2	4.1	4.6	4.8	4.1	4.3	4.4
dmul	2.9	2.9	3.0	2.9	3.1	3.0
if_icmpge	4.7	4.6	0.0	4.7	0.0	2.8
goto	4.7	4.6	0.0	4.7	0.0	2.8
iload_3	2.3	2.3	2.4	2.3	2.5	2.4
dastore	2.3	2.3	2.4	2.3	2.5	2.4
dsub	2.3	2.3	2.4	2.3	2.5	2.4
putfield	2.4	2.4	2.5	2.4	2.5	2.4
if_icmplt	0.0	0.0	4.8	0.0	4.9	1.9
dstore	1.8	1.7	1.8	1.8	1.8	1.8
dup	2.3	0.0	0.0	2.3	2.5	1.4
iconst_1	1.2	1.2	1.2	1.2	1.2	1.2
iload	1.2	1.2	1.2	1.2	1.2	1.2
aload_1	1.2	1.2	1.2	1.2	1.2	1.2
isub	1.2	1.2	1.2	1.2	1.2	1.2
invokestatic	1.2	1.2	1.2	1.2	1.2	1.2
aload_3	0.6	0.6	0.6	0.6	0.6	0.6
ddiv	0.6	0.6	0.6	0.6	0.6	0.6
invokevirtual	0.6	0.6	0.7	0.6	0.7	0.6
arraylength	0.6	0.6	0.6	0.6	0.6	0.6
dup2	0.6	0.0	0.0	0.6	0.6	0.4
nop	0.0	0.0	0.0	0.0	0.0	0.0
aconst_null	0.0	0.0	0.0	0.0	0.0	0.0
iconst_m1	0.0	0.0	0.0	0.0	0.0	0.0
iconst_0	0.0	0.0	0.0	0.0	0.0	0.0
iconst_2	0.0	0.0	0.0	0.0	0.0	0.0
iconst_3	0.0	0.0	0.0	0.0	0.0	0.0
iconst_4	0.0	0.0	0.0	0.0	0.0	0.0

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

Instruction	borl	gcj	kopi	pizza	jdk13	ave
getfield	25.7	25.9	26.1	25.7	26.1	25.9
aload_0	15.8	16.1	16.2	15.8	16.1	16.0
aload_1	10.7	10.8	10.9	10.7	10.9	10.8
dmul	6.4	6.5	6.6	6.4	6.6	6.5
dadd	4.6	4.7	4.7	4.6	4.7	4.7
dsub	3.6	3.6	3.7	3.6	3.7	3.6
putfield	3.0	3.1	3.1	3.0	3.1	3.1
aload_2	2.7	2.8	2.8	2.7	2.8	2.8
invokevirtual	1.8	1.9	1.9	1.8	1.9	1.9
invokestatic	1.9	1.9	1.9	1.9	1.9	1.9
iload	1.8	1.8	1.8	1.8	1.8	1.8
dreturn	1.8	1.8	1.8	1.8	1.8	1.8
dload	1.1	2.9	1.1	1.1	1.1	1.5
dload_2	1.8	0.0	1.9	1.8	1.9	1.5
aload	1.3	1.2	1.3	1.3	1.3	1.3
aconst_null	1.7	0.9	0.9	1.7	0.9	1.2
dstore	1.0	1.8	1.0	1.0	1.0	1.2
dconst_0	0.9	1.0	1.0	0.9	1.0	1.0
ifge	1.0	1.0	1.0	1.0	1.0	1.0
return	0.9	1.0	1.0	0.9	1.0	1.0
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
iinc	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
arraylength	0.9	0.9	0.9	0.9	0.9	0.9
dcmpg	1.0	1.0	0.0	1.0	1.0	0.8
dstore_2	0.9	0.0	0.9	0.9	0.9	0.7
goto	0.9	0.9	0.1	0.9	0.1	0.6
if_icmpge	0.9	0.9	0.0	0.9	0.0	0.5
ifnull	0.0	0.9	0.9	0.0	0.9	0.5
if_icmplt	0.0	0.0	0.9	0.0	0.9	0.4
if_acmpeq	0.9	0.0	0.0	0.9	0.0	0.4
dcmpl	0.1	0.1	1.1	0.1	0.1	0.3
dload_1	0.2	0.2	0.2	0.2	0.2	0.2
iconst_0	0.1	0.1	0.1	0.1	0.1	0.1

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

Instruction	borl	gcj	kopi	pizza	jdk13	ave
iload	12.8	12.4	13.5	12.9	13.2	13.0
aload_0	8.3	8.9	9.6	8.3	8.6	8.7
getfield	7.1	7.6	8.0	7.1	7.3	7.4
iaload	5.2	5.1	5.2	5.2	5.4	5.2
istore	5.2	5.2	5.2	5.2	5.4	5.2
ishl	4.2	4.1	4.2	4.2	4.3	4.2
bipush	3.6	4.3	3.6	3.7	3.8	3.8
iadd	3.4	4.1	4.2	3.4	3.5	3.7
iand	3.4	4.1	3.4	3.4	3.5	3.6
iload_1	3.5	2.8	3.8	3.5	3.6	3.4
iload_2	2.5	3.3	2.5	2.6	2.6	2.7
iload_3	2.5	3.3	2.7	2.5	2.5	2.7
ior	2.3	2.2	2.2	2.3	2.3	2.3
iconst_1	2.2	2.0	2.0	2.2	2.3	2.1
iconst_2	2.0	2.0	2.0	2.0	2.1	2.0
dup	1.9	1.8	1.5	1.9	2.0	1.8
iinc	1.7	1.6	1.7	1.7	1.7	1.7
iconst_5	1.4	1.7	1.8	1.4	1.4	1.5
iconst_0	2.6	0.7	0.7	2.5	0.7	1.4
iconst_4	1.4	1.4	1.4	1.4	1.4	1.4
iastore	1.4	1.4	1.4	1.4	1.5	1.4
if_icmpgt	1.7	1.4	0.9	1.7	0.9	1.3
goto	1.5	1.5	0.4	1.5	0.5	1.1
ifeq	0.1	1.9	1.2	0.1	1.6	1.0
invokevirtual	1.0	0.9	1.0	1.0	1.0	1.0
isub	0.9	0.8	0.9	0.9	0.9	0.9
if_icmple	0.6	0.8	1.3	0.6	1.3	0.9
ldc1	0.9	0.8	0.8	0.8	0.8	0.8
istore_3	0.8	0.8	0.8	0.8	0.8	0.8
if_icmpeq	1.7	0.2	0.2	1.7	0.2	0.8
if_icmplt	0.5	0.5	1.3	0.5	1.4	0.8
dup2	1.0	0.3	0.1	1.0	1.0	0.7
imul	0.7	0.6	0.6	0.7	0.7	0.7
if_icmpge	1.1	0.9	0.1	1.1	0.1	0.7
putfield	0.7	0.7	0.7	0.7	0.7	0.7

application. In order to analyse these tables, the differences in each row were selected and the relevant sections of the corresponding source code were then examined.

It is notable that the different applications, in exercising different areas of the instruction set, reflected compiler differences to varying degrees. In particular, the figures for *mol* are virtually identical across all compilers, and *gcj* seems to exhibit the greatest variations across applications. Below we summarize the main differences exhibited in these tables.

5.1. Main compiler differences

There were three main differences between the optimizations implemented by the compilers, which we now give.

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*:

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

and *pizza*/*gcj*/*borland*:

```
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. Specialized `load` *instructions*

Tables XI and XV highlight an important difference between the compilers in their treatment of specialized `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 optimize the programs for integer usage.

However, it is interesting to note that this is not significant for the other three applications. This can be explained directly by the nature of the programs involved—*mol*, *mon* and *ray* make greater use of `doubles` and objects and *gcj* does not appear to optimize the stack positions for these types.

5.1.3. Usage of the `dup` *instruction*

There is a dramatic difference in the use of `dup` instructions shown in Table XI and, to a lesser extent, in Table XV, 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 optimize 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 optimization 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 specialized instructions for comparison with zero: `ifeq` and `ifne`. As can be seen from Table XV, 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 four programs rate this difference as significant. However, Java bytecode also has a specialized instruction for comparing object references with null, `ifnull`. The object-intensive program *ray* (Table XIV) 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 XV 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 XI. 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 XII 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 reorganized 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 specialized `load` and `store` instructions, which exist for the first four slots of the array.

Table XVI 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 parameter methods, but these appear as zero in the percentages as they are swamped by those methods with high bytecode counts 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 specialized versions of `load` and `store` operations dealing with these array locations. These figures are: eul, 2.6%; mol, 0.2%; mon, 62.5%; ray, 54.0%; sea, 8.0%. Indeed, these figures are an under-estimation of the possibility of using specialized `load` and `store` operations, since register allocation techniques can reduce these stack sizes further. As already noted, the overall figures for specialized `load` instructions *eul* presented in Table XI do not seem to reflect the high proportion (97.5%) of the methods which would facilitate this.

Table XVII presents two perspectives on 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. Looking at the figures based on numbers of method calls, we see that a significant number of methods called have low operand stack sizes, reflecting the number of trivial constructors, as well as simple *get* and *set* methods. However, the figures based on the number of bytecodes executed show that while calls to methods with low operand stack sizes may be common, they typically involve very little internal computation. We suggest that both method-call *and* bytecode-level analyses are necessary in order to present a complete picture of operand stack usage.

Table XVI. Bytecode based dynamic percentages of local variable array sizes, as well as temporary and parameter sizes for Grande programs compiled using SUN's *javac* compiler. The local variable array and parameter sizes include the this-reference for non-static methods.

Size	eul	mol	mon	ray	sea	ave
Local variable array size						
0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.7	0.1	7.5	0.2	0.0	1.7
2	0.4	0.0	47.2	32.8	0.0	16.1
3	1.5	0.1	7.8	21.0	8.0	7.7
4	0.1	0.0	14.5	0.4	0.8	3.2
5	8.7	0.1	0.1	0.0	12.2	4.2
6	0.0	0.0	0.1	0.3	48.5	9.8
7	0.0	0.0	0.1	14.3	0.0	2.9
8	0.0	0.0	0.0	0.2	0.0	0.0
>8	88.6	99.7	22.8	30.7	30.5	54.5
Parameter size						
0	0.0	0.0	0.0	0.0	0.0	0.0
1	64.2	0.1	57.3	1.3	24.4	29.5
2	2.0	0.0	17.7	62.4	8.1	18.0
3	16.0	0.2	24.5	20.0	34.9	19.1
4	17.8	0.0	0.3	14.3	32.5	13.0
5	0.0	0.0	0.0	0.4	0.0	0.1
6	0.0	0.0	0.0	0.3	0.0	0.1
7	0.0	99.6	0.0	0.3	0.0	20.0
8	0.0	0.0	0.0	1.0	0.0	0.2
>8	0.0	0.0	0.0	0.0	0.0	0.0
Temporary variable size						
0	1.1	0.3	25.4	54.0	0.6	16.3
1	1.5	0.0	43.8	0.2	0.0	9.1
2	0.1	0.1	7.7	1.0	43.6	10.5
3	0.0	0.0	0.1	14.2	0.8	3.0
4	8.7	0.0	0.1	0.0	16.5	5.1
5	0.0	0.0	0.0	0.0	7.9	1.6
6	4.1	0.0	0.0	0.0	0.0	0.8
7	0.0	0.0	0.0	29.5	0.0	5.9
8	0.0	0.0	12.5	0.0	0.0	2.5
>8	84.4	99.6	10.3	1.2	30.5	45.2

Table XVII. Dynamic percentages of maximum operand stack sizes for the methods in the Java Grande programs, compiled using SUN's *javac* compiler. (a) presents percentages calculated based on proportions of methods called, while (b) measures the proportion of total bytecodes executed.

Size	eul	mol	mon	ray	sea	ave
(a)						
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	1.0	0.8	46.5	14.6
>3	54.3	96.6	96.9	97.3	48.2	78.6
(b)						
0	0.0	0.0	0.0	0.1	0.0	0.0
1	0.0	0.0	0.0	0.1	0.0	0.0
2	0.0	0.0	0.2	0.1	7.3	1.5
3	0.7	0.0	0.3	0.4	32.5	6.8
>3	99.8	100	99.5	99.3	60.2	91.7

7. CONCLUSIONS

This paper set out to investigate platform-independent dynamic JVM 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 used partly to understand their ultimate behaviour on a specific hardware platform.

For Grande applications, Java method execution time is shown to be predominantly in the non-API bytecodes of the programs (92% 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.

A constant theme of this paper 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 V and Table VI, where it is shown that method execution frequency and a count of bytecodes executed by a method provide different pictures of where the interpreter is actually spending its time. Table XVII also demonstrates the additional perspective gained from a bytecode-level analysis.

Overall, this study raises questions about the balance of optimization work between Java compilers and the execution 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 run-time bytecode verifier or platform specific JIT compilers. If this is so, it may explain why various standard efficiency improvements have not been used by different 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 following algorithms such as those in [18,19], yet only the *gcj* compiler seems to make a significant attempt at this. A more common optimization 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.

Clearly, run-time optimization 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 possible that the JVM may be bearing an unreasonable part of the burden of performing these optimizations. If Java compilers performed various standard optimizations such as constant propagation, branch folding, dead assignment elimination etc, then the code to do this could be eliminated from the virtual machine, making its design less complex. Further research would be necessary to determine to what extent (if any) running time and memory usages could be improved by moving as much work as possible into the platform-independent phase of compilation.

REFERENCES

1. Lindholm T, Yellin F. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
2. Dahm M. Byte code engineering with the JavaClass API. *Technical Report B-17-98*, Freie Universität Berlin, July 1998.
3. Bull M, Smith L, Westhead M, Henty D, Davey R. A methodology for benchmarking Java Grande applications. *Proceedings of the ACM 1999 Java Grande Conference*. ACM Press: New York, NY, 1999; 81–88.
4. Bull M, Smith L, Westhead M, Henty D, Davey R. Benchmarking Java Grande applications. *Proceedings of the Second International Conference and Exhibition on the Practical Application of Java*. PAJava: Blackpool, U.K., 2000.
5. Vallee-Rai R, Hendren L, Sundaresan V, Lam P, Gagnon E, Co P. Soot—a Java optimization framework. *Proceedings of the 9th NRC/IBM Center for Advanced Studies Conference*. IBM Canada: Ontario, 1999; 125–135.
6. Antonioli D, Pilz M. Analysis of the Java class file format. *Technical Report 98.4*, Department of Computer Science, University of Zurich, Switzerland, April 1988.
7. Adl-Tabatabai A-R, Cierniak M, Lueh G-Y, Parikh VM, Stichnoth JM. Fast, effective code generation in a Just-In-Time Java compiler. *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press: New York, NY, 1998; 280–290.
8. Ishizaki K, Kawahito M, Yasue T, Takeuchi M, Ogasawara T, Suganuma T, Onodera T, Komatsu H, Nakatani T. Design, implementation and evaluation of optimisations in a Just-In-Time compiler. *Proceedings of the ACM 1999 Java Grande Conference*. ACM Press: New York, NY, 1999; 119–128.
9. Kazi IH, Chan HH, Stanley B, Lilja DJ. Techniques for obtaining high perfromance in Java programs. *ACM Computing Surveys* 2000; **32**(3):213–240.
10. Colnet D, Zendra O. Optimizations of Eiffel programs: SmallEiffel, the GNU Eiffel compiler. *Proceedings Conference on Technology of Object-Oriented Languages and Systems*. IEEE Computer Society Press: Los Alamitos, CA, 1999; 341–350.
11. Benton N, Kennedy A, Russell G. Compiling standard ML to Java bytecodes. *Proceedings of the 3rd ACM SIGPLAN Conference on Functional Programming*. ACM Press: New York, NY, 1998; 129–140.
12. Waldron J. Object oriented programs and a stack based virtual machine. *Journal of South African Computer Society* 2000; **25**:45–55.
13. Wilkinson TJ. *KAFFE, A Virtual Machine to run Java Code*. http://www.kaffe.org [July 2000].
14. Armstrong E. Hotspot: A new breed of virtual machine. *Java World*. http://www.javaworld.com [March 1998].

15. Sun Microsystems. The Java HotSpot virtual machine. *Technical White Paper*. http://java.sun.com/-products/hotspot/ [7 November 2001].
16. Radhakrishnan R, Vijaykrishnan N, John LK, Sivasubramaniam A, Rubio J, Sabarinathan J. Java runtime systems: Characterization and architectural implications. *IEEE Transactions on Computers* 2001; **50**(2):131–146.
17. Waldron JT, Daly C, Gray D, Horgan J. Comparison of factors influencing bytecode usage in the Java virtual machine. *Proceedings of the Second International Conference and Exhibition on the Practical Application of Java*. PAJava: Blackpool, U.K., 2000.
18. Koopman P. A preliminary exploration of optimized stack code generation. *Rochester Forth Conference*. Institute of Applied Forth Research: VA, 1992.
19. Maierhofer M, Ertl MA. Local stack allocation. *Proceedings of the 7th International Conference on Compiler Construction*. Springer: Berlin, 1998; 189–203.

