

Measurement and Analysis of Runtime Profiling Data for Java Programs

Jane Horgan¹, James Power and John Waldron²

Department of Computer Science,
National University of Ireland, Maynooth

Date: February 2001

Technical Report: NUIM-CS-TR-2001-04

Key words: Java bytecode, dynamic analysis, software metrics

Abstract. In this paper we examine a procedure for the analysis of data based on the dynamic profiling of Java programs. In particular, we describe the issues involved in dynamic analysis, propose a metric for discrimination between the resulting data sets, and examine its application over different test suites and compilers.

¹ Computer Applications, Dublin City University, Dublin 9, Ireland.

² Dept. of Computer Science, Trinity College, Dublin 2, Ireland.

1 Introduction

The Java paradigm for executing programs is a two stage process. First the source is converted into a platform independent intermediate representation [5], consisting of bytecode and other information stored in class files. Second, hardware-specific conversions are performed, followed by the execution of the code on the Java Virtual Machine (JVM).

The problem addressed by this research is that while there exist static tools such as class file viewers to look at this intermediate representation, this does not provide full information on the dynamic profile of the program. For large programs the dynamic data can easily involve the execution of billions of bytecodes, so ad hoc approaches to analysis can quickly run into difficulty. This paper seeks to contribute to the study of Java and the JVM by outlining a process, and a related metric, for the investigation of such data.

1.1 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 [2].

However, the production of bytecode for the JVM is no longer limited to a single Java-to-bytecode compiler, but can involve any of a number of different compilers, working from source languages as diverse as Eiffel, Scheme or Prolog. In previous work we have informally studied the impact of the choice of source language on the dynamic profiles of programs running on the JVM [6]. In this paper we seek to provide a scientific framework within which such analysis can take place, and to use it to examine the importance of the choice of Java compiler on dynamic execution data.

1.2 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 [1] and [4] where a wide difference was found between the bytecodes appearing in different class files. Speed comparisons of benchmark suites using different Java Platforms have been performed [3] and differences in execution times have been found, but it is difficult to apportion responsibility for these between the compiler, virtual machine or underlying architecture.

The *dynamic* frequency of an instruction is the number of times it is executed during a program run and can provide valuable information for profiling of programs and for the design and implementation of virtual machines. However, such study involves large quantities of data, and is not readily assessed using standard software metrics. To this end we propose a metric designed to measure the dissimilarity between two sets of dynamic frequency data.

1.3 Normalised Mean Square Contingency Measure

Suppose $n_i = (n_{ki})$ and $n_j = (n_{kj})$ are variables ($k = 1, 2, \dots, m$) describing the instruction count for two applications i and j (or for one application under different circumstances, e.g. under a change of compiler). We can form the $m \times 2$ matrix $(n_i \ n_j) = (n_{k\ell})$ whose columns are given by n_i and n_j ; in all cases m is less than or equal to 202, the number of usable bytecode instructions.

As a measure of the similarity of the two applications we could write

$$\|(n_i \ n_j) - e(n_i \ n_j)\|^2 = \sum_{k\ell} (n_{k\ell} - e(n_i \ n_j)_{k\ell})^2, \quad (1)$$

where

$$e(n_i \ n_j) = \frac{(n_{k\bullet} n_{\bullet i} \quad n_{k\bullet} n_{\bullet j})}{n_{\bullet\bullet}}, \quad (2)$$

the $m \times 2$ matrix whose columns are multiples of the sum of the two columns of $(n_i \ n_j) = (n_{k\ell})$ by the sum of the column elements. We can think of $e(n_i \ n_j)$ as the expected values $n_{k\ell}$ under the assumption of statistical independence between n_i and n_j . As a measure of the association between the instruction count of the two applications we consider the chi-square coefficient

$$\chi_{ij}^2 = \sum_{\ell=i,j} \sum_{k=1}^m \frac{(n_{k\ell} - e(n_i \ n_j)_{k\ell})^2}{e(n_i \ n_j)_{k\ell}}. \quad (3)$$

If this is small, then the count distributions of the two applications are similar, and if it is large, the distributions differ. We observe that, after division of the expression (3) by $n_{\bullet\bullet}$, the result lies between 0 and 1. Thus we define a normalised mean-square contingency measure

$$\Phi_{ij} = \sqrt{\frac{\chi_{ij}^2}{n_{\bullet i} + n_{\bullet j}}},$$

where $n_{\bullet i}$ is the total number of bytecodes executed for program i and $n_{\bullet j}$ is the total number of bytecodes executed for program j , as a measure of the relationship between instruction usage of compilers i and j .

1.4 The Test Suite

In order to study dynamic bytecode usage it was necessary to modify the source code of a Java Virtual Machine. Kaffe [8] is an independent implementation of the Java Virtual Machine which comes with its own standard class libraries; version 1.0.5 was used for these measurements.

A *Grande* application is one which uses large amounts of processing, I/O, network bandwidth or memory. The Java Grande Forum Benchmark Suite [3] is intended to be representative of such applications, and thus to provide a basis for measuring and comparing alternative Java execution environments.

Four Grande applications were used in our study:

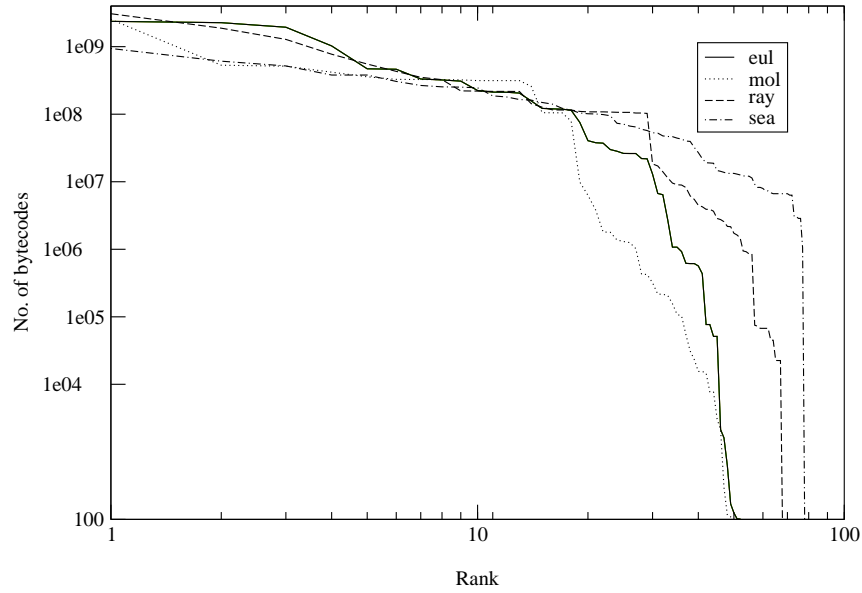


Fig. 1. *Distribution of the Dynamic Data.* This graph shows the bytecode count for each instruction plotted against its corresponding rank on a log-log scale.

- **eul**, a benchmark that solves a set of equations using a fourth order Runge-Kutta method
- **mol**, a translation of a Fortran program designed to model the interaction of molecular particles
- **ray**, which measures the performance of a 3D ray tracer rendering a scene containing 64 spheres
- **sea**, which solves a game of connect-4 on a 6×7 board using alpha-beta pruning.

A fifth application from this suite dealing with a *montecarlo* benchmark failed by a large amount when interpreted on Kaffe, and hence does not form part of our study.

The Kaffe JVM was instrumented to count each bytecode executed, and the standard test suites were run for each application.

2 Dynamic Bytecode Execution Frequencies

In this section we present a more detailed view of the dynamic profiles of the Grande programs studied by examining the size and distribution of the data. In this section, all the programs were compiled using Sun’s *javac* compiler, from version 1.3 of the JDK, and the data reflects only the non-API bytecodes executed.

Table 1 summarises the data collected. As can be seen, all data sets are of the order of 10^{10} , spread over roughly 100 bytecodes in each case. This spread is far from even, however, with a relatively high standard deviation. This is demonstrated by Figure 1,

which shows a high concentration of instruction usage in a few instructions, with a sharp tailing off of use among the remaining instructions. There is a slight variance between application here, with *mol* showing the greatest concentration of usage in high-ranking bytecodes, and *sea* showing a slightly less uneven distribution.

Table 1. *Summary of the Dynamic Data.* Note that the average and standard deviation are for *used* bytecodes only.

	eul	mol	ray	sea
Total bytecode count	11,394,409,844	7,599,606,435	11,706,547,247	7,103,719,939
No. of bytecodes used	97	95	107	112
Average count	44,509,414	29,685,963	45,728,700	27,748,906
Std. Deviation	3.6%	3.8%	3.3%	2.0%

Table 2. *Static and Dynamic Dissimilarity between Grande Applications.* This table shows the values of Φ , giving the differences between instruction usage in the four Grande applications.

Static Differences					Dynamic Differences				
	eul	mol	ray	sea		eul	mol	ray	sea
eul	0.000	0.357	0.430	0.665	eul	0.000	0.751	0.650	0.735
mol	0.357	0.000	0.399	0.699	mol	0.751	0.000	0.783	0.896
ray	0.430	0.399	0.000	0.623	ray	0.650	0.783	0.000	0.821
sea	0.665	0.699	0.623	0.000	sea	0.735	0.896	0.821	0.000

The differences between applications are further demonstrated by Table 2, which shows the results of applying the mean square contingency measure to both the static and dynamic profiles of bytecode usage. It is interesting to note that while *sea* demonstrates a higher degree of static dissimilarity with the other programs, this is reduced in the dynamic profile. Presumably, high dynamic dissimilarity figures are desirable in test suites designed to exercise different aspects of the JVM, and the Grande suite has succeeded in this respect at least.

Table 3. *Comparing the static and dynamic profiles of the Grande Applications.* This table shows the values of Φ , reflecting the dissimilarity between the static and dynamic bytecode counts for each application.

Static vs. Dynamic			
eul	mol	ray	sea
0.203	0.464	0.212	0.175

Table 3 shows the comparison between the static and dynamic profiles for each application, which measures the degree to which a static analysis can reflect the dynamic

profile of the applications. As can be seen from the tables, *sea* has the lowest dissimilarity, whereas the static figures for *mol* are the least useful for predicting its dynamic behaviour.

A consideration of the instruction usage, ranked by frequencies give a good overall view of the nature of the operations being tested by each application³. As has been noted for other programs in [7], load and store instructions, which data between the operand stack and the local variable array, account for a significant proportion of the instructions used in all cases. Some differences can also be noted

- The use of arrays of objects in *eul* is reflected in its high usage of object-load and field-access instructions
- *mol*'s dependence on double-precision floating-point values is reflected in the prominence of `dload` and `dstore` instructions over their integer counterparts
- *ray*'s profile reflects its usage of objects through the high frequency of field access, as well as the `aload_0` instruction, which retrieves the `this`-pointer in a method.
- *sea* shows the widest distribution of instruction usage, although this is chiefly based around integer-related operations.

In all cases method calls, as reflected by the `invoke` instructions, are relatively infrequent compared to stack and field accesses.

3 Comparison across different compilers

In this section we consider the impact of the choice of Java compiler on the dynamic bytecode frequency data. For the purposes of this study we used seven different Java compilers:

- **borland** Borland Compiler for Java, version 1.2.006
- **gcj** The GNU Compiler for Java, version 2.95.2
- **jdk12** Blackdown Java for Linux version pre-release 2
- **jdk13** SUN's javac compiler, (JDK build 1.3.0-C)
- **jikes** IBM Jikes Compiler, version 1.06 (17 Sep 99)
- **kopi** KOPI Java Compiler Version 1.3C
- **pizza** Pizza version 0.39g, 15-August-98

Table 4. *Dynamic bytecode usage count differences for Grande Applications using different compilers.* The figures show the difference in bytecode counts between each of the six compilers and *jdk13*, expressed as a percentage increase over the *jdk13* figures.

	borland	gcj	jdk12	jikes	kopi	pizza
eul	0.3	10.1	0.0	0.0	9.5	0.3
mol	1.4	1.4	0.0	0.0	0.0	1.4
ray	1.8	0.9	0.0	0.0	0.0	1.8
sea	3.1	6.0	0.4	0.4	4.0	2.9

³ These are listed in Table 6 in the appendices

The four Grande applications were compiled using each of the seven compilers, and data collected for the dynamic behaviour of each. The first indications of differences can be gleaned from Table 4, which shows the difference between the total dynamic bytecode count for each compiler, compared with that for the *jdk13*. Both *jikes* and *jdk12* are very similar to *jdk13*, with *gcj* reflecting the greatest increase, although this is unevenly distributed through the applications.

To gain a greater insight into the nature of the compiler differences, the mean square contingency measure between the compilers was calculated for each application, and the results are summarised⁴ in Table 5. Three of the compilers, *jikes*, *jdk12* and *jdk13* are very similar, showing only a minor dissimilarity for the *sea* application. Also, the *pizza* and *borland* compilers appear to be quite similar to each other for all applications. Both the *kopi* and *gcj* compilers exhibit the highest dissimilarities, with *eul* highlighting the differences for *gcj*, and *mol* highlighting *kopi*'s differences.

Table 5. Comparing compilers against *jdk13*. This table summarises the compiler differences, by showing the value of Φ for each when compared against the *jdk 1.3*

	borland	gcj	jdk12	jikes	kopi	pizza
eul	0.071	0.367	0.000	0.000	0.103	0.071
mol	0.147	0.147	0.000	0.000	0.202	0.147
ray	0.159	0.187	0.000	0.000	0.101	0.159
sea	0.179	0.166	0.038	0.045	0.086	0.174
ave	0.134	0.210	0.016	0.019	0.118	0.134

3.1 Compiler Differences

Having gained some insight into the overall compiler differences, it is possible to make one more use of the mean square contingency measure. Tables 8 through 11 in the appendices show the differences in bytecode usage between each compiler and *jdk13*, itemised by bytecode instruction. To aid analysis each table is sorted in decreasing order of dissimilarity, calculated on a per-instruction basis. Below we summarise the main differences exhibited in these tables.

- *Eliminating Unnecessary Jumps* It is notable that for *sea* the *jdk13* has fewer unconditional `gotos` than other compilers. This results from a small optimisation for nested `if`-statements where the target of one `goto` statement is another `goto`. This difference appears insignificant from a static analysis of the code, but shows up clearly when the dynamic figures are studied.
- *Loop Structure* For each usage of the `if_cmlt` 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 a more efficient implementation of loop structures by *kopi* and *jdk13*, ensuring that each iteration involves just a single test. A simple static

⁴ full details are shown in Table 7 in the appendices

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

- *Specialised load 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 variables. However *mol* and *ray* make greater use of doubles and objects respectively, and the differences in `iload` instructions are not significant here.
- *Common subexpression elimination*
 There is a dramatic difference in the use of `dup` instructions between *pizza*, *jdk13* and *borland* versus *kopi* and *gcj*. The former exploit the usage of operators such as `+=` by duplicating the operands on the stack; the latter do not, and show a corresponding increase in the usages of `aload`, `aaload` and `getfield` instructions as the expression is re-evaluated.
- *Comparisons with 0 and null*
 Java bytecode has specialised instructions for comparison with zero and null, including `ifeq`, `ifne` and `ifnull`. *borl* and *pizza* do not use these instructions, and the counts for the corresponding constant-load and comparison instructions are thus higher.
- *Constant Propagation*
 The *gcj* compiler does not do as much constant propagation as the other compilers and this is evidenced particularly in *eul*, which uses a number of constant fields. Thus there is a drop in `ldc2w` instructions, and a corresponding increase in the number of `getfield` instructions.

4 Conclusions

This paper defines and demonstrates a process, and associated metric, for the investigation of data collected from dynamic Java Virtual Machine analysis. 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.

References

1. D. Antonioli and M. Pilz. Analysis of the Java class file format. Technical Report 98.4, Dept. of Computer Science, University of Zurich, April 1988.
2. E. Armstrong. Hotspot: A new breed of virtual machine. *Java World*, March 1998.
3. M. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. Benchmarking Java Grande applications. In *Second International Conference and Exhibition on the Practical Application of Java*, Manchester, UK, April 2000.

4. T. Cohen and Y. Gil. Self-calibration of metrics of java methods. In *Technology of Object-Oriented Languages and Systems*, pages 94–106, Sydney, Australia, November 2000.
5. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.
6. J. Waldron. Dynamic bytecode usage by object oriented Java programs. In *Technology of Object-Oriented Languages and Systems*, Nancy, France, June 1999.
7. J. Waldron and O. Harrison. Analysis of virtual machine stack frame usage by Java methods. In *Third IASTED Conference on Internet and Multimedia Systems and Applications*, Nassau, Grand Bahamas, Oct 1999.
8. T.J. Wilkinson. *KAFFE, A Virtual Machine to run Java Code*. <www.kaffe.org>, 2000.

A Proof that the Mean Square Contingency Measure is Normalised

Theorem If $n_i = (n_{ki})$ and $n_j = (n_{kj})$ are m -tuples of positive numbers and $n = (n_i \ n_j) \in \mathbf{R}^{m \times 2}$ then

$$\sum_{k\ell} \frac{((n_i \ n_j)_{k\ell} - e(n_i \ n_j)_{k\ell})^2}{e(n_i \ n_j)_{k\ell}} \leq \sum_{k\ell} n_{k\ell} = n_{\bullet i} + n_{\bullet j} = n_{\bullet\bullet}.$$

Proof. We compute

$$\begin{aligned} & n_{\bullet\bullet} \left(\sum_k \frac{(n_{ki} - \frac{n_{k\bullet}n_{\bullet i}}{n_{\bullet\bullet}})^2}{\frac{n_{k\bullet}n_{\bullet i}}{n_{\bullet\bullet}}} + \sum_k \frac{(n_{kj} - \frac{n_{k\bullet}n_{\bullet j}}{n_{\bullet\bullet}})^2}{\frac{n_{k\bullet}n_{\bullet j}}{n_{\bullet\bullet}}} \right) \\ &= \sum_k \frac{(n_{\bullet\bullet}n_{ki} - n_{k\bullet}n_{\bullet i})^2}{n_{k\bullet}n_{\bullet i}} + \sum_k \frac{(n_{\bullet\bullet}n_{kj} - n_{k\bullet}n_{\bullet j})^2}{n_{k\bullet}n_{\bullet j}} \\ &= \sum_k \frac{(n_{\bullet j}n_{ki} - n_{kj}n_{\bullet i})^2}{n_{k\bullet}n_{\bullet i}} + \sum_k \frac{(n_{\bullet i}n_{kj} - n_{ki}n_{\bullet j})^2}{n_{k\bullet}n_{\bullet j}} \\ &= \left(\frac{1}{n_{\bullet i}} + \frac{1}{n_{\bullet j}} \right) \sum_k \frac{(n_{\bullet j}n_{ki} - n_{kj}n_{\bullet i})^2}{n_{k\bullet}} \\ &= \frac{n_{\bullet\bullet}}{n_{\bullet i}n_{\bullet j}} \sum_k \frac{(n_{\bullet j}n_{ki} - n_{kj}n_{\bullet i})^2}{n_{k\bullet}}, \end{aligned}$$

which we claim is $\leq n_{\bullet\bullet}^2$: for we have

$$\sum_k (n_{\bullet j}^2 n_{ki}^2 + n_{\bullet i}^2 n_{kj}^2) \prod_{k' \neq k} n_{k'\bullet} \leq n_{\bullet i} n_{\bullet j} n_{\bullet\bullet} \prod_k n_{k\bullet}.$$

Indeed we count $m \cdot 2m^2 \cdot 2^{m-1} = m^3 2^m$ terms on the left and $m \cdot m \cdot 2m \cdot 2^m = m^3 2^{m+1}$ terms on the right, and can pick off a match on the right for each term on the left •

B Bytecode Usage Frequencies for each of the Grande Applications

Table 6. Dynamic bytecode usage frequencies by Grande applications compiled using SUN's javac compiler. The top 35 instructions are presented.

eul		mol		ray		sea	
iload	20.8	dload	33.3	getfield	26.3	iload	13.2
aaload	19.9	iload	7.0	aload_0	16.1	aload_0	8.6
getfield	17.0	dstore	6.8	aload_1	10.9	getfield	7.3
aload_0	9.0	dcmpl	5.5	dmul	6.6	istore	5.4
dadd	4.1	dsub	4.7	dadd	4.7	iaload	5.4
dmul	4.1	getfield	4.3	dsub	3.7	ishl	4.3
iconst_1	2.9	getstatic	4.3	putfield	3.0	bipush	3.8
putfield	2.8	dmul	4.3	aload_2	2.8	iload_1	3.6
dload	2.7	aaload	4.2	dload_2	1.9	iadd	3.5
dup	2.0	ifle	4.1	iload	1.9	iand	3.5
aload_3	1.9	ifge	4.1	invokestatic	1.9	iload_2	2.6
isub	1.9	dcmpl	4.1	invokevirtual	1.9	iload_3	2.5
daload	1.8	dneg	4.1	dreturn	1.9	iconst_1	2.3
iload_3	1.4	dadd	3.4	aload	1.2	ior	2.3
dstore	1.1	if_icmpl	1.4	dload	1.1	iconst_2	2.1
ldc2w	1.1	ifgt	1.4	dstore	1.0	dup	2.0
iadd	1.0	iinc	1.4	ifge	1.0	iinc	1.7
dsub	1.0	dload_1	1.0	dcmpl	1.0	ifeq	1.6
ddiv	0.7	aload_0	0.1	dstore_2	0.9	iastore	1.5
aload_2	0.4	putfield	0.1	astore	0.9	iconst_5	1.4
dload_1	0.3	lconst_0	0.0	aaload	0.9	iconst_4	1.4
if_icmpl	0.3	fadd	0.0	aconst_null	0.9	if_icmpl	1.4
iinc	0.3	ladd	0.0	ifnull	0.9	if_icmple	1.3
dastore	0.2	iadd	0.0	arraylength	0.9	dup2	1.0
dstore_3	0.2	swap	0.0	return	0.9	invokevirtual	1.0
dstore_1	0.2	dup2_x2	0.0	areturn	0.9	if_icmpgt	0.9
aload_1	0.2	dup2_x1	0.0	if_icmpl	0.9	isub	0.9
dload_3	0.2	dup2	0.0	dconst_0	0.9	istore_3	0.8
dconst_0	0.2	dup_x2	0.0	iinc	0.9	ldc1	0.8
aload	0.1	dup_x1	0.0	dload_1	0.2	istore_1	0.7
new	0.1	iconst_5	0.0	dup	0.1	iconst_0	0.7
invokespecial	0.1	dup	0.0	iconst_0	0.1	putfield	0.7
lconst_0	0.0	pop2	0.0	ldc2w	0.1	ifne	0.7
fadd	0.0	pop	0.0	invokespecial	0.1	imul	0.7
ladd	0.0	sastore	0.0	goto	0.1	iconst_3	0.6

C Mean Square Contingency Measure for Each Compiler

Table 7. Here the value of Φ is shown for each pair of compilers, for each of the four applications. Since the relation is symmetric, the upper-left half of each table has been included for reference purposes only.

	eul						
	borland	gcj	jdk12	jdk13	jikes	kopi	pizza
borland	0.000	0.361	0.071	0.071	0.071	0.125	0.001
gcj	0.361	0.000	0.367	0.367	0.367	0.351	0.361
jdk12	0.071	0.367	0.000	0.000	0.000	0.103	0.071
jdk13	0.071	0.367	0.000	0.000	0.000	0.103	0.071
jikes	0.071	0.367	0.000	0.000	0.000	0.103	0.071
kopi	0.125	0.351	0.103	0.103	0.103	0.000	0.125
pizza	0.001	0.361	0.071	0.071	0.071	0.125	0.000
	mol						
	borland	gcj	jdk12	jdk13	jikes	kopi	pizza
borland	0.000	0.007	0.147	0.147	0.147	0.249	0.007
gcj	0.007	0.000	0.147	0.147	0.147	0.249	0.001
jdk12	0.147	0.147	0.000	0.000	0.000	0.202	0.147
jdk13	0.147	0.147	0.000	0.000	0.000	0.202	0.147
jikes	0.147	0.147	0.000	0.000	0.000	0.202	0.147
kopi	0.249	0.249	0.202	0.202	0.202	0.000	0.249
pizza	0.007	0.001	0.147	0.147	0.147	0.249	0.000
	ray						
	borland	gcj	jdk12	jdk13	jikes	kopi	pizza
borland	0.000	0.179	0.159	0.159	0.159	0.189	0.000
gcj	0.179	0.000	0.187	0.187	0.187	0.212	0.179
jdk12	0.159	0.187	0.000	0.000	0.000	0.101	0.159
jdk13	0.159	0.187	0.000	0.000	0.000	0.101	0.159
jikes	0.159	0.187	0.000	0.000	0.000	0.101	0.159
kopi	0.189	0.212	0.101	0.101	0.101	0.000	0.189
pizza	0.000	0.179	0.159	0.159	0.159	0.189	0.000
	sea						
	borland	gcj	jdk12	jdk13	jikes	kopi	pizza
borland	0.000	0.198	0.171	0.179	0.172	0.187	0.035
gcj	0.198	0.000	0.167	0.166	0.169	0.160	0.194
jdk12	0.171	0.167	0.000	0.038	0.024	0.078	0.166
jdk13	0.179	0.166	0.038	0.000	0.045	0.086	0.174
jikes	0.172	0.169	0.024	0.045	0.000	0.082	0.168
kopi	0.187	0.160	0.078	0.086	0.082	0.000	0.183
pizza	0.035	0.194	0.166	0.174	0.168	0.183	0.000

D Detailed Compiler differences

Table 8. Bytecode count differences against jdk13 for the four Grande applications compiled with the **kopi** compiler.

Bytecode	kopi				
	eul	mol	ray	sea	χ
dcmpl	76800	419532800	115462653	0	45289
dcmpg	-76800	-419532800	-115462653	0	23131
dup	-217907207	-429110	-11647666	-32584709	15085
aaload	432537600	0	0	114084	9084
iload	425984000	0	0	51487721	8911
aload_0	217907231	429130	11480702	99752692	7917
dup2	0	0	0	-67053779	7827
getfield	216268800	0	-6	67167863	5726
iconst_m1	3	2	3	14705226	5121
goto	0	0	0	27761330	4888
ifne	0	0	0	30050302	4391
iadd	0	0	0	58631678	3695
iconst_5	0	0	0	29284324	2917
ifeq	0	0	0	-30050302	2801
iload_3	6553603	2	3	22065279	1725
iload_1	0	0	0	23013320	1443
iconst_1	2	1	0	-14705178	1156
aload_2	3	2	0	1050012	1024
baload	0	0	0	1050011	282
iand	0	0	0	1050011	66
bipush	4	6	0	1050151	64
aload	0	0	167106	228168	37
iconst_3	0	0	0	114094	18
iaload	0	0	0	342252	17
istore_0	0	0	0	22	15
ior	0	0	0	114084	8
isub	0	0	0	-63044	7
ishl	0	0	0	114084	6
iload_0	0	0	0	22	2
lookupswitch	-1	-1	-1	-1	2
tableswitch	1	1	1	1	2
dconst_1	2	0	0	0	2
iastore	2	2	0	112	1
iconst_2	1	1	0	19	1
newarray	1	1	0	2	1

Table 9. Bytecode count differences for **borland** and **pizza** compared against jdk13

borland					
Bytecode	eul	mol	ray	sea	χ
if_acmpeq	0	0	104200128	0	52100064
if_icmpge	37820501	105338117	108412568	70869631	14897083
goto	36685299	105132895	104302178	75573244	333656
if_icmpeq	0	0	0	113780021	33762
iconst_0	25600	315015	3181466	138273519	26414
if_icmplt	-37820501	-105338117	-108412568	-64225544	17125
aconst_null	0	0	105053966	0	10353
ifnull	0	0	-104200128	0	10207
ifeq	0	0	0	-107455075	10017
if_icmpgt	0	7746	0	54584526	6686
if_icmple	0	0	3181465	-51720439	6235
i2d	25600	307200	0	0	4310
i2l	0	0	1	12969033	3557
ldc2w	0	0	0	-6644087	2577
ifge	0	0	0	-6644087	2577
lconst_0	0	0	-1	-6324946	2514
ifne	0	-69	0	-14985324	2190
ifle	0	0	-3181465	-2864087	1980
if_icmpne	0	69	0	8660378	1368
ifnonnull	0	0	-853838	0	924
ldc1	0	0	0	6644087	877
dconst_0	-25600	-307200	0	0	303

pizza					
Bytecode	eul	mol	ray	sea	χ
if_acmpeq	0	0	104200128	0	52100064
if_icmpge	37820501	105338117	108412568	70869631	14897083
goto	36685299	105132895	104302178	75573244	333656
if_icmpeq	0	0	0	113780021	33762
iconst_0	0	7815	3181465	131948573	18149
if_icmplt	-37820501	-105338117	-108412568	-64225544	17125
aconst_null	0	0	105053966	0	10353
ifnull	0	0	-104200128	0	10207
ifeq	0	0	0	-107455075	10017
if_icmpgt	0	7746	0	54584526	6686
if_icmple	0	0	3181465	-51720439	6235
ifge	0	0	0	-6644087	2577
ifne	0	-69	0	-14985324	2190
ifle	0	0	-3181465	-2864087	1980
if_icmpne	0	69	0	8660378	1368
ifnonnull	0	0	-853838	0	924

Table 10. Bytecode count differences against jdk13 for the four Grande applications compiled with the **gcj** compiler. These figures reflect the greatest dissimilarity among all the compilers, both in size and spread

Bytecode	gcj				
	eul	mol	ray	sea	χ
iload_2	1140583000	0	0	61682701	304833629
if_icmpge	37820501	105338117	108412568	64225544	14897079
iload_1	173824000	0	0	-41211794	13371077
aload_3	0	0	0	42830635	7138439
astore_3	0	0	0	7321072	2767105
goto	36685299	105133696	104302178	82484994	333709
dconst_1	256002	0	0	0	256002
istore_1	153600	0	0	-32299878	108714
iload_3	812595003	2	3	66241340	64949
iload	-1694464400	0	0	-7077327	34815
istore_2	103000	0	0	27382781	28236
dload	52428900	6945	212557630	0	19091
if_icmplt	-37820501	-105338117	-108412568	-64225544	17125
dup	-217907204	-3074	-11647662	-5759734	14826
dload_2	0	0	-216715004	0	14611
dstore	52428800	3873	100042754	0	10432
dstore_2	0	0	-104200128	0	10207
aaload	432537600	0	0	51031	9084
aload_0	245456234	3094	11480699	57109447	8008
aload	0	0	-7395465	-42728605	6565
dup2	0	0	0	-51298562	5988
getfield	243817800	0	0	51349593	5975
dload_1	-29491200	-6945	0	0	5388
dstore_1	-27852800	-3873	0	0	5234
iconst_m1	3	2	3	14705226	5121
dload_3	-22937700	0	4157374	0	4948
dstore_3	-24576000	0	4157374	0	4799
if_icmpgt	0	0	0	37677442	4615
ifne	0	0	0	-26621698	3890
if_icmple	0	0	0	-37677442	3865
iand	0	0	0	60586387	3859
iadd	0	0	0	58631656	3695
lload_3	0	0	0	-13288174	3645
bipush	4	6	0	56127412	3435
iconst_5	0	0	0	29284324	2917
astore	0	0	0	-7321073	2705
lstore_3	0	0	0	-6644087	2577
ldc2w	-27395293	3	0	0	2500
ifeq	0	0	0	26621698	2481
iconst_1	2	1	0	-14705178	1156
int2byte	0	0	0	1997661	1144
getstatic	0	0	0	4459094	1029
istore	-256400	0	0	7631872	518
aload_2	4	3	7562578	2	419
istore_3	-200	0	0	-2714775	350
isub	0	0	0	-1113055	138
ddiv	153600	0	0	0	17
dsub	153600	0	0	0	14
iconst_3	0	0	0	51041	8
iaload	0	0	0	153093	7

Table 11. Bytecode count differences against jdk13 for the four Grande applications compiled with the **jikes** and **jdk12** compiler. As can be seen from these figures, the only significant difference is in the number of extra `gotos` executed by each.

jikes					
Bytecode	eul	mol	ray	sea	χ
goto	0	0	0	29398178	5176
ifne	0	0	0	30050302	4391
sipush	0	0	0	-7944839	2818
ifeq	0	0	0	-30050302	2801
ldc1	0	0	0	7944839	1049
lookupswitch	-1	-1	-1	-1	2
tableswitch	1	1	1	1	2

jdk12					
Bytecode	eul	mol	ray	sea	χ
goto	0	0	0	27761330	4888
ifne	0	0	0	30050302	4391
sipush	0	0	0	0	0
ifeq	0	0	0	-30050302	2801
ldc1	0	0	0	0	0
lookupswitch	-1	-1	-1	-1	2
tableswitch	1	1	1	1	2