

Measurement and Analysis of Runtime Profiling Data for Java Programs

Jane Horgan

School of Computer Applications, Dublin City University, Dublin 9, Ireland.

James Power

Department of Computer Science, National University of Ireland, Maynooth, Ireland.

John Waldron

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

Abstract

In this paper we examine a procedure for the analysis of data produced by 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.

Keywords Java Virtual Machine, Bytecode Analysis, Contingency Measure

1. Introduction

The Java programming language [7] has gained widespread popularity as a general-purpose programming language and, as such, is increasingly the focus of studies in source code analysis and manipulation. In this paper we describe a technique for the analysis of data gained from the dynamic profiling of Java programs, and we present a contingency measure that has proved extremely useful in this analysis. We describe a case study showing this technique in practice, as applied to programs from the Java Grande Benchmark Suite.

The remainder of this section describes our reasons for studying dynamic profiling data of Java programs, motivates the method used, and defines the contingency measure. Sections 2 and 3 demonstrate the use of the method, first to measure the difference between programs in a test suite, and second to measure the impact of the choice of compiler on the results. Section 4 concludes the paper.

1.1. Analysing Java Programs

The process for generating an executable file from Java source code takes place in two stages. First, at compile-time, the source is converted into a platform independent intermediate representation [10], consisting of bytecode and other information stored in class files. Second, at run-time, hardware-specific conversions are performed, followed by the execution of the code on the Java Virtual Machine (JVM).

This process provides at least four levels at which Java programs may be analysed:

1. Statically, at the source code level; studies at this level are similar to those of programs written in other languages, where standard software metrics [6] can be applied.
2. Statically, at the bytecode level, where the usage of bytecode instructions can be analysed for the purposes such as optimisation [13] or compression [1], or even as a source of software metrics [4].
3. Dynamically, at the bytecode level in a platform-independent manner; this information can be used to determine potential for optimisation [11], or to estimate the coverage and effectiveness of programs commonly used in benchmarking [14].
4. Dynamically on a specific JVM and architecture; this is the basis for studies of performance optimisations such as Just-In-Time (JIT) [8] and hotspot-based [2] compilation, as well as comparative JVM performance [3].

The remainder of this paper focuses on the third of these levels.

1.2. Platform Independent Dynamic Analysis

Dynamic analysis can provide information on the characteristics of programs at the bytecode level, such as instruction usage frequencies, stack frame usage, method invocation and object creation. Given the increasing variety and sophistication of JVM implementations (see [9] for a survey) it is clearly useful to distinguish those features of a given Java program or suite of programs that are independent of the JVM implementation, and thus will be common across all platforms.

Our goal in performing such platform-independent dynamic analysis was twofold:

- To develop a technique for profiling benchmark suites, so that different suites may be combined, and omissions in existing suites may be addressed
- To examine the effect of compiler choice on such profiles, since this should be known when gathering results for a given JVM

In performing this analysis it was necessary to process dynamic profiling data from a number of different applications from the test suite, over a number of different compilers. For example, one study [5] examines the differences between seven different compilers over a test suite involving five applications. Each one of these 35 choices (of compiler and application) involved the execution of roughly 10^{10} instructions, presenting a formidable volume of data requiring analysis.

The overall contribution of this paper is to outline our method for analysing such large volumes of data, and, in particular, to define a difference measure that can be used to guide this analysis.

1.3. Normalised Mean Square Contingency Measure

The data most commonly collected as a result of dynamic profiling consists of counts of execution frequencies for particular operations, such as stack loads and stores, method invocation etc. When dealing with a number of different programs, compilers or environments, blunt measures such as totals and averages often do not capture subtle differences between test data, possibly varying on individual instruction counts. To this end we describe a contingency measure which, while providing a single overall figure for a given comparison, will also take into account differences in individual frequencies.

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 then form the $m \times 2$ matrix $(n_i \ n_j) = (n_{k\ell})$ whose columns are given by n_i and n_j . For bytecode instructions, m is always 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

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

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 applications i and j .

Clearly this measure provides summary information, and is not a substitute for a closer examination of the underlying data. In the next two sections we show how it can be used to guide the analysis of data collected from dynamic profiling of Java programs.

2. Case Study I: Variances between programs in a benchmark suite

In order to demonstrate the use of our approach to the analysis of dynamic bytecode data, we will outline the results of a case study using the Java Grande Forum Benchmark Suite [3]. This suite is intended to be representative of applications that use large amounts of processing, I/O, network bandwidth or memory.

Five applications from the Java Grande Suite (Version 2.0, size A) were used in our study:

Table 1. *Summary of the Dynamic Data.* This table gives some overall figures for the dynamic profile of the Java Grande Benchmark Suite, including the total count of bytecode instructions executed, the number of different instructions used, along with the average and standard deviation, calculated over the 202 usable bytecodes.

	euler	moldyn	montecarlo	raytracer	search
Total bytecode count	14,514,096,409	7,599,820,106	1,632,874,942	11,792,255,694	7,103,726,472
No. of bytecodes used	107	105	111	112	114
Average count	71,851,962	37,622,872	8,083,539	58,377,503	35,166,963
Std. Deviation	341,107,381	199,113,676	33,832,479	282,282,446	106,439,932

- **eul** an array-based maths-intensive program
- **mol** a translation of a Fortran program designed to model molecular particles
- **mon** a financial simulation using Monte Carlo techniques
- **ray** which measures the performance of a 3D ray tracer
- **sea** which implements a search algorithm for a game of connect-4

All of these programs were compiled using Sun’s *javac* compiler, from version 1.3 of the JDK. The Kaffe JVM [16] (version 1.0.5) was instrumented to count each bytecode executed, and the standard test suites were run for each application. In order to ensure platform independence for the bytecode counts, all optimisations (such as JIT compilation) were disabled. Also, all bytecode information relating to the Kaffe class library has been excluded from the figures, since this ensures independence from the Kaffe implementation, and was essential to highlight compiler differences in our second case study.

2.1. Dynamic Bytecode Execution Frequencies

In order to gain a rough idea of the nature of the data, Table 1 presents some outline figures that summarise the data collected. As can be seen, all data sets are of the order of 10^{10} instructions executed, spread over roughly 100 different bytecodes in each case. The range is roughly a factor of 10, between the smallest application *mol* and the largest *eul*. The relatively high standard deviation in each case, however, indicates that the instruction usage is unevenly spread throughout the different bytecodes.

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

	eul	mol	mon	ray	sea
eul	0.000	0.741	0.487	0.607	0.731
mol	0.741	0.000	0.776	0.783	0.896
mon	0.487	0.776	0.000	0.561	0.653
ray	0.607	0.783	0.561	0.000	0.821
sea	0.731	0.896	0.653	0.821	0.000

In order to further examine the usage distribution over bytecodes, Figure 1 plots the number of times each instruction was used against its rank (ranging from 1, the most frequently used, down towards 100, the least frequently used), on a log-log scale. As can be seen from this graph, there is a high concentration of usage in a few instructions, with a sharp tailing off of use among the remaining instructions. Such a distribution is familiar from case studies involving other programming languages [12, §3.2.5].

This distribution is important in the context of analysing a benchmark suite. Frequently, studies are interested in specific types of instructions representing important operations (e.g. representing object creation, virtual method calls, exception handling). However a benchmark suite with such a concentration of usage among relatively few instructions risks representing certain possibly significant instructions hardly at all.

There is a slight variance between applications here, with *mol* showing the greatest concentration of usage in high-ranking bytecodes, and *sea* showing a slightly less uneven distribution. However, more information is clearly required in order to distinguish between the applications.

2.2. Applying the Contingency Measure

The differences between applications are further demonstrated by Table 2, which shows the results of applying the

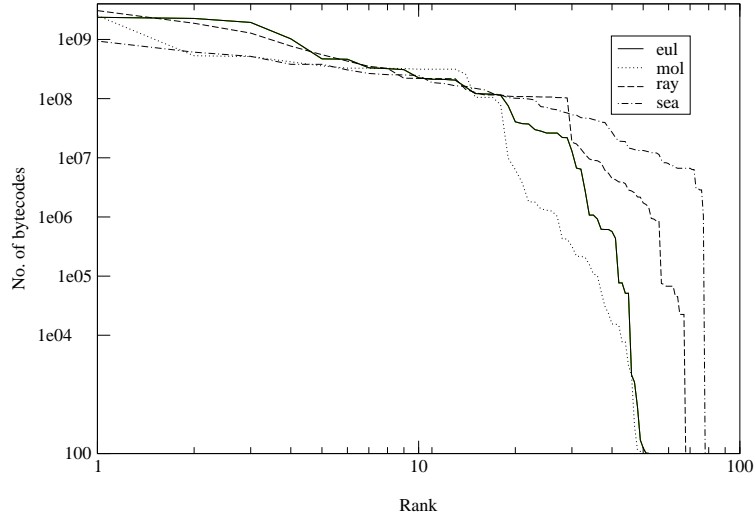


Figure 1. *Distribution of the Dynamic Data.* This graph shows the bytecode count (i.e. number of types the instruction was executed) for each instruction plotted against its corresponding rank (where 1 is the most frequently executed) on a log-log scale.

Table 3. *Static Dissimilarity between Grande Applications.* This table shows the values of Φ , giving the differences between **static** instruction usage in the five Grande applications.

	eul	mol	mon	ray	sea
eul	0.000	0.355	0.627	0.427	0.666
mol	0.355	0.000	0.523	0.397	0.701
mon	0.627	0.523	0.000	0.420	0.663
ray	0.427	0.397	0.420	0.000	0.627
sea	0.666	0.701	0.663	0.627	0.000

mean square contingency measure to the bytecode instruction usage frequencies. In all cases the dissimilarity is high, presumably a desirable feature of test suite applications designed to exercise different aspects of the JVM. This table can be used as a basis for the extension of the benchmark suite: desirable additions are (at least) those applications exhibiting a significant difference to any of the existing applications in the suite.

An interesting side-issue here relates to the difference between instruction usage measured statically and dynamically. Table 3 presents the contingency measure for the applications where the static frequency of bytecodes is used (i.e. the number of times they appear in the bytecode files). Comparing Table 2 with Table 3, we note that applications appearing similar based on a static analysis (e.g. *mol*, *ray* and *eul*) appear quite different when dynamically analysed. Presumably this reflects the “characteristic” aspects of the

applications being present inside frequently executed loops, and indicates the importance of dynamic execution frequencies.

A consideration of the instruction usage, ranked by frequencies give a more detailed view of the nature of the operations being tested by each application, and is presented in Appendix B. As has been noted for other programs in [15], load and store instructions, which move data between the operand stack and the local variable array, account for a significant proportion of the instructions used in all cases.

While data such as that presented in Appendix B provides the ultimate detail in relation to instruction usage, the summary data collected using the contingency measure presents a useful overall picture of the differences.

3. Case Study II: Variances across different compilers

In this section we examine another application of the contingency measure - to determine the impact of the choice of Java compiler on the dynamic bytecode frequency data.

For the purposes of this study we used five different Java compilers:

- **borland**, the Borland compiler for Java, version 1.2.006
- **gcj**, the GNU Compiler for Java, version 2.95.2
- **jdk13**, SUN’s javac compiler from JDK build 1.3.0-C

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 four compilers and *jdk13*, expressed as a percentage increase over the *jdk13* figures.

	borland	gcj	kopi	pizza
	%	%	%	%
euler	0.3	8.7	8.1	0.3
moldyn	1.4	1.4	0.0	1.4
montecarlo	4.9	6.1	1.2	4.9
raytracer	1.8	0.9	0.0	1.8
search	3.1	6.0	3.6	2.9
<i>average</i>	<i>1.6</i>	<i>4.7</i>	<i>3.4</i>	<i>1.5</i>

- **kopi**, KOPI Java Compiler Version 1.3C
- **pizza**, Pizza version 0.39g, 15-August-98

The five Grande applications were compiled using each of the five compilers, and, as in the previous section, data was collected for the dynamic behaviour of each.

3.1. Overall Differences

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*. These figures show the percentage increase in the number of bytecodes executed for each compiler against the *jdk13* figures previously presented in Table 1.

The average figures in Table 1 suggest that *gcj* and *kopi* show the greatest increase, although this is not consistent across all the applications. These figures do however give some insight as to where the main discrepancies may be found: in *eul* for *gcj* and *kopi*, and in *mon* for *borland* and *pizza*. It also suggest that *borland* and *pizza* exhibit similar divergences from the *jdk13*, whereas the increases for *mon* show that *gcj* and *kopi* do not always differ in the same way.

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 summarised in Table 5; full details are shown in Appendix C.

Table 5 demonstrates a number of aspects relating to the compiler differences. First, the variation is small compared to that between the applications (as shown previously in table Table 2). Clearly, at the present stage of development of Java compilers, a change in the application being studied is more significant than a change in the compiler being used.

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	kopi	pizza
euler	0.063	0.308	0.098	0.063
moldyn	0.147	0.147	0.202	0.147
montecarlo	0.267	0.294	0.129	0.267
raytracer	0.159	0.187	0.101	0.159
search	0.179	0.166	0.084	0.174
<i>average</i>	<i>0.163</i>	<i>0.220</i>	<i>0.123</i>	<i>0.162</i>

Second, the compilers are not completely independent - indeed, there appears to be a strong similarity to the approach taken by the *pizza* and *borland* compilers. On the other hand, there is a strong difference between the *gcj* compiler and all the others, perhaps reflecting a deliberate design choice on the part of the GNU project.

Third, we can see that not all applications are affected equally by varying the compiler. In particular *mon* consistently exhibits one of the highest variances, and this gives some indication of which parts of the benchmark suite should be examined in order to formulate an explanation.

Even though Java compilers are at a relatively early stage of development, it is reasonable to assume that the differences between them will increase, rather than decrease over time. Data along the lines of that presented in Table 5 provides a useful starting point for measuring the impact of compiler evolution on the type of code produced.

3.2. Detailed Compiler Differences

Since our intention here is to describe the method of our investigation we will not consider the various explanations for each of the compiler differences in detail. However, it is possible to make one more use of the mean square contingency measure in order to bring these differences into focus.

As an example, Appendix D shows the differences in bytecode usage between the *gcj* compiler and *jdk13*, itemised by bytecode instruction. To aid analysis the table is sorted in decreasing order of dissimilarity, calculated on a per-instruction basis. This ranking is useful here since it allows us to distinguish between dissimilarities based on their significance in terms of the overall program.

Below we summarise the main differences exhibited in these tables.

- *Loop Structure*

For each usage of the `if_cmplt` instruction by *jdk13* there is a corresponding usage of `goto` and

`if_cmpge` by `gcj`. This can be explained by a more efficient implementation of loop structures, ensuring that each iteration involves just a single test. A simple static analysis would regard these as similar implementations, but the dynamic analysis clearly shows the savings resulting from the *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 to make use of lower-numbered local variable array slots.

- *Common subexpression elimination*

There is a dramatic difference in the use of `dup` instructions. The *jdk13* exploits the usage of operators such as `+=` by duplicating the operands on the stack; `gcj` does not, and shows a corresponding increase in the usages of `aload`, `aaload` and `getfield` instructions as the expression is re-evaluated.

Our purpose in reviewing these compiler differences here is to demonstrate the use of the contingency measure in collecting the data and guiding the search for differences. A fuller account of the details of these and other compiler differences can be found in [5].

4. Conclusion

This paper defines and demonstrates a process, and associated metric, for the investigation of data collected from the dynamic analysis of Java bytecode. It has been shown above that useful information about a Java programs can be extracted at the intermediate representation level, which can then be used to understand their ultimate behaviour on a specific hardware platform.

One of the problems with this approach is the large quantity of data collected, and a major goal of this paper is to provide a procedure for dealing with this data. Two case studies have been presented as examples of this approach - a comparison of programs in a benchmark suite, and a comparison of the effects of various Java compilers on the generated bytecode.

We see this work as being useful in three main areas:

- As a foundation for the study of the performance of Java programs on a given JVM. The procedure for data collection outlined above establishes the nature and composition of the platform-independent aspects of a test suite, and this can then be used to set the parameters for performance measurement on a given JVM.
- As a method for determining the coverage and mutual independence of test suite applications. The difference measures presented in Table 2 above can be used to evaluate the suitability of some other application for inclusion in a benchmark suite - a minimum requirement should be a high dissimilarity to those applications already in the suite.
- As a method for determining and tracking the effect of compiler transformations on generated bytecode. Java compilers are still in an early stage of development, but are likely to grow increasingly diverse. The process presented here will help to measure the impact of the compiler on the generated bytecode, and thus on any data collected using bytecode generated by a given compiler.

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. We believe however that it is useful as an auxiliary to such information, and that useful information about Java programs, test suites and Java compilers can be collected by following the strategy outlined in this paper.

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 J. Gil. Self-calibration of metrics of Java methods. In *Technology of Object-Oriented Languages and Systems*, pages 94–106, Sydney, Australia, November 2000.
- [5] C. Daly, J. Horgan, J. Power, and J. Waldron. Platform independent dynamic Java virtual machine analysis: the Java Grande Forum Benchmark Suite. In *Joint ACM Java Grande - ISCOPE 2001 Conference*, pages 106–115, Stanford, CA, USA, June 2001.
- [6] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. Thomson Computer Press, first edition, 1996.
- [7] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.
- [8] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. Design, implementation and evaluation of optimisations in a just-in-time compiler. In *ACM 1999 Java Grande Conference*, pages 119–128, San Francisco, CA, USA, June 1999.

- [9] I. Kazi, H. Chan, B. Stanley, and D. Lilja. Techniques for obtaining high performance in Java programs. *ACM Computing Surveys*, 32(3):213–240, September 2000.
- [10] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.
- [11] R. Radhakrishnan, N. Vijaykrishnan, L. John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan. Java runtime systems: Characterization and architectural implications. *IEEE Transactions on Computers*, 50(2):131–146, February 2001.
- [12] M. Shooman. *Software Engineering: design, reliability and management*. McGraw-Hill, 1983.
- [13] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.
- [14] J. Waldron. Dynamic bytecode usage by object oriented Java programs. In *Technology of Object-Oriented Languages and Systems*, Nancy, France, June 1999.
- [15] 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.
- [16] T. 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

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

We claim this is $\leq n_{\bullet \bullet}^2$ as 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 Dynamic Bytecode Instruction Frequencies

These tables show the top 20 most frequently executed bytecode instructions for each application. For each instruction, the percentage figure represents the percentage of the total bytecodes executed when this program was run that were of this type.

euler %		moldyn %		montecarlo %	
iload	19.8	dload	33.3	aload_0	17.9
aaload	18.3	iload	7.0	getfield	17.8
getfield	16.2	dstore	6.8	daload	7.3
aload_0	8.3	dcmprg	5.5	iload_1	6.1
dmul	4.1	dsub	4.7	dadd	4.9
dadd	4.0	getfield	4.3	dload	4.9
putfield	3.3	getstatic	4.3	if_icmplt	4.9
iconst_1	3.2	dmul	4.3	iinc	4.9
dload	2.8	aaload	4.2	iload_2	4.3
daload	2.0	ifle	4.1	dmul	3.1
isub	2.0	ifge	4.1	dup	2.5
dup	1.7	dcmpl	4.1	dastore	2.5
aload_3	1.5	dneg	4.1	iload_3	2.5
dsub	1.5	dadd	3.4	putfield	2.5
aload	1.4	if_icmplt	1.4	dsub	2.5
aload_2	1.3	ifgt	1.4	dstore	1.8
ldc2w	1.2	iinc	1.4	iconst_1	1.2
iadd	1.1	dload_1	1.0	aload_1	1.2
iload_3	1.1	aload_0	0.1	iload	1.2
dstore	1.0	putfield	0.1	invokestatic	1.2

raytracer %		search %	
getfield	26.1	iload	13.2
aload_0	16.1	aload_0	8.6
aload_1	10.9	getfield	7.3
dmul	6.6	istore	5.4
dadd	4.7	iaload	5.4
dsub	3.7	ishl	4.3
putfield	3.1	bipush	3.8
aload_2	2.8	iload_1	3.6
dload_2	1.9	iadd	3.5
invokestatic	1.9	iand	3.5
invokevirtual	1.9	iload_2	2.6
iload	1.8	iload_3	2.5
dreturn	1.8	iconst_1	2.3
aload	1.3	ior	2.3
dload	1.1	iconst_2	2.1
dstore	1.0	dup	2.0
return	1.0	iinc	1.7
ifge	1.0	ifeq	1.6
dcmprg	1.0	iastore	1.5
dconst_0	1.0	iconst_5	1.4

C Mean Square Contingency Measure for Each Compiler

Here the value of Φ is shown for each pair of compilers, for each of the five applications. By the definition of the difference measure, a low value indicates similar dynamic profiles, while a high value indicates dissimilar profiles.

Since the relation is symmetric, the upper-right corner of each table has been included for reference purposes only.

euler					
	borland	gcj	jdk13	kopi	pizza
borland	0.000	0.302	0.063	0.116	0.000
gcj	0.302	0.000	0.308	0.293	0.302
jdk13	0.063	0.308	0.000	0.098	0.063
kopi	0.116	0.293	0.098	0.000	0.116
pizza	0.000	0.302	0.063	0.116	0.000

moldyn					
	borland	gcj	jdk13	kopi	pizza
borland	0.000	0.007	0.147	0.249	0.007
gcj	0.007	0.000	0.147	0.249	0.001
jdk13	0.147	0.147	0.000	0.202	0.147
kopi	0.249	0.249	0.202	0.000	0.249
pizza	0.007	0.001	0.147	0.249	0.000

montecarlo					
	borland	gcj	jdk13	kopi	pizza
borland	0.000	0.126	0.267	0.296	0.003
gcj	0.126	0.000	0.294	0.265	0.126
jdk13	0.267	0.294	0.000	0.129	0.267
kopi	0.296	0.265	0.129	0.000	0.296
pizza	0.003	0.126	0.267	0.296	0.000

raytracer					
	borland	gcj	jdk13	kopi	pizza
borland	0.000	0.178	0.159	0.188	0.000
gcj	0.178	0.000	0.187	0.211	0.178
jdk13	0.159	0.187	0.000	0.101	0.159
kopi	0.188	0.211	0.101	0.000	0.188
pizza	0.000	0.178	0.159	0.188	0.000

search					
	borland	gcj	jdk13	kopi	pizza
borland	0.000	0.198	0.179	0.193	0.035
gcj	0.198	0.000	0.166	0.167	0.194
jdk13	0.179	0.166	0.000	0.084	0.174
kopi	0.193	0.167	0.084	0.000	0.189
pizza	0.035	0.194	0.174	0.189	0.000

D Detailed Compiler Differences: gcj vs. jdk13

This table shows the difference in dynamic bytecode usage between Grande programs compiled with the Java compiler from the JDK 1.3 and the GNU compiler for Java. For each of the five applications the difference between GNU and the JDK is given in numbers of bytecode instructions, where a positive figure means an increase going from the JDK to GCJ. The final column gives the value for χ , an estimate of the significance of this difference over the five applications, and this is used to rank the table.

Bytecode	eul	mol	mon	ray	sea	χ
if_icmpge	46426954	105340746	80063949	108412746	64225554	15006823
aload_3	33894400	0	-20776	0	42830635	6608910
astore_3	6425701	1	-2596	1	7321073	3812935
iload_2	1139474718	0	10000000	0	61682701	1082343
goto	45029048	105135741	79945133	104302306	82484994	386845
iload_1	172866677	0	0	0	-41211794	176642
iload_3	812089979	0	21756	0	66241339	64805
istore_1	153590	0	0	0	-32299878	44589
iload	-1652572189	0	-20856	0	-7077327	30815
if_icmplt	-46426960	-105340746	-80063989	-108412746	-64225554	19543
dload	75171580	6945	0	210036772	0	18677
dup	-237567985	-3060	-39970396	-11647704	-5759840	16702
dload_2	0	0	0	-216715004	0	14611
dstore	60587290	3873	0	100042754	0	10525
dstore_2	0	0	0	-104200128	0	10207
aaload	471859200	0	0	0	51031	9167
aload_0	268428007	3079	50091302	11480687	57109443	8582
aload	-33894400	0	20776	-12437181	-42728605	7041
dup2	0	0	-10001000	0	-51298562	6772
dload_3	-40798590	0	0	6678232	0	6613
istore_2	100653	0	0	0	27382781	6609
getfield	266789597	0	10000900	0	51349593	5967
dload_1	-34372990	-6945	0	0	0	5689
dstore_1	-29474395	-3873	0	0	0	5385
dstore_3	-31112895	0	0	4157374	0	5304
iconst_m1	0	0	20000	0	14705197	5122
if_icmpgt	0	6	-33761	0	37677442	4618
ifne	0	0	17404	0	-26621698	3890
if_icmple	0	-6	33761	0	-37677442	3877
iand	0	0	0	0	60586387	3859
astore_2	-1	-1	-1	2520857	-1	3858
astore	-6425700	0	2597	-2520858	-7321073	3715
iadd	0	0	0	0	58631656	3695
bipush	0	0	10000	0	56127294	3685
lload_3	0	0	0	0	-13288174	3645