Relating Static and Dynamic Measurements for the Java Virtual Machine Instruction Set

TOM DOWLING[1], JAMES POWER[1] and JOHN WALDRON[2]

[1] Dept. of Computer Science,
National University of Ireland,
Maynooth, Co. Kildare,
IRELAND.
{tdowling,jpower}@cs.may.ie

[2] Dept. of Computer Science,
Trinity College Dublin,
Dublin 2,
IRELAND.
john.waldron@cs.tcd.ie

Abstract: - It has previously been noted that, for conventional machine code, there is a strong relationship between static and dynamic code measurements. One of the goals of this paper is to examine whether this same relationship is true of Java programs at the bytecode level. To this end, the hypothesis of a linear correlation between static and dynamic frequencies was investigated using Pearson's correlation coefficient. Programs from the Java Grande and SPEC benchmarks suites were used in the analysis.

Key-Words: - Java Virtual Machine, bytecode frequency analysis, instruction set design

1 Introduction

The Java programming language [10], and its related technology, the Java Virtual Machine (JVM) [14], have become increasingly popular as the language and technology of choice for object-oriented software construction. The JVM, as a stack-based virtual machine, is an example of a paradigm of compiler implementation stretching back to (at least) UCSD Pascal [4]. This is a two stage process, where Java programs are first compiled to an intermediate language consisting of Java *bytecodes*. These platform-independent bytecode instructions are then interpreted or compiled by a platform-specific JVM.

This mode of compilation and execution provides interesting opportunities for the study of Java programs, since there are now at least three levels at which such study can occur: at the Java source-code level, at the bytecode level, or ultimately, at the platform-specific machine-code level. As yet, Java compilers perform minimal optimisations to programs when translating from Java source code to bytecode [12]. The theme of this paper is to explore the relationship between the static bytecodes produced by a Java compiler, and the dynamic bytecode usage as executed by a JVM.

It has previously been noted that, for conventional machine code, there is a strong linear relationship between static and dynamic code measurements [8]. That research posited that it was thus possible to use easily-obtained static measurements to predict their dynamic counterparts. One of the goals of this paper is to examine whether this same relationship is true of Java programs at the bytecode level; that is:

- Can the dynamic performance and operation of Java programs be accurately predicted by static studies of Java bytecode?

The remainder of this paper is organised as follows. Section 2 discusses the background to this work, describes the test suite used, and summarises related work. Section 3 describes the context of the experiments and measurements carried out, while Section 4 presents the results of these measurements. Section 5 concludes the paper.

The Java Grande Forum Benchmark Suite Section 2: Kernels	
k-crypt	IDEA encryption
k-fft	Fast Fourier Transform
k-heapsort	Integer sorting
k-lufact	LU Factorisation
k-series	Fourier coefficient analysis
k-sor	Successive over-relaxation
k-sparse	Sparse Matrix multiplication

The Java Grande Forum Benchmark Suite Section 3: Large Scale Applications	
g-eul	Computational Fluid Dynamics
g-mol	Molecular Dynamics simulation
g-mon	Monte Carlo simulation
g-ray	3D Ray Tracer
g-sea	Alpha-beta pruned search

Standard Performance Evaluation Corporation JVM98 Benchmarks	
s-compress	Modified Lempel-Ziv method (LZW)
s-db	Performs multiple database functions on memory resident database
s-jack	A Java parser generator that is based on PCCTS
s-javac	This is the Java compiler from the JDK 1.0.2.
s-jess	Java Expert Shell System is based on NASA's CLIPS expert shell system
s-mpeg	Decompresses ISO MPEG Layer-3 audio files
s-mtrt	A raytracer with two threads each rendering a scene

Figure 1: *The programs used in this analysis.* There were 19 programs in total, taken from two separate suites of the Java Grande Forum sequential benchmarks, and from the SPEC JVM98 benchmarks.

2 Background and Related Work

In this section we present an overview of the benchmark programs used in the experiment, as well as the method of data collection.

2.1 The Benchmark Suites

A total of 19 programs have been analysed for this study, taken from both the SPEC JVM98 benchmark suite [18] and the Java Grande Forum Benchmark Suite [5, 6]. The SPEC suite was designed as an industry-standard benchmark suite for measuring the performance of client-side Java applications, and we have used the seven main programs from this suite. We have chosen two sets of benchmarks from version 2.0 of the Grande suite's sequential benchmarks. The first set of seven programs comprises section 2 of the suite, "kernels", designed to measure frequently used operations. The second set of programs, section 3 of the Grande suite, contains five large-scale applications, designed as examples of "real" applications.

The benchmarks included in this work are shown in Figure 1.

We believe the suites chosen are as close as possible to "standard" based on published results. Many other suites of benchmark programs for Java exist, including micro-benchmarks such as CaffeineMark [16] Richards and DeltaBlue [20]. As these measure small, repetitive operations, it was felt that their results would not be typical of Java applications. For the same reason larger suites, designed to test Java's threads or server-side applications, such as SPEC's Java Business Benchmarks (SPECjbb2000), the Java Grande Forum's Multi-threaded Benchmarks, IBM's Java Server benchmarks [2] or VolanoMark [15] have not been included at this point.

Studies of the SPEC and Grande suites have typically concentrated on performance issues for various JVMs. Studies of the Grande suite include performance-related measures such as [5, 6], as well as dynamic byte-code level views [7]. For the SPEC suite, [9] presents a study of the allocation and heap behaviour , whereas [13] and [17] discuss low-level architectural issues. [3] provides a higher-level dynamic bytecode view of the SPEC suite, but does not

provide any comparison with static usage frequencies.

Statistical analyses of benchmark suites have been carried out in [12] and [11], but neither of these directly address the existence of a linear correlation between static and dynamic bytecode frequencies.

2.2   Technical Details

The Java Grande Forum Benchmark suite is distributed in source code format. For this study, the programs were compiled using the Java compiler from SUN's JDK, version 1.3, and the static counts are based on its output. The SPEC suite is distributed in bytecode format, and the static counts were compiled directly from these files. In order to perform the dynamic measurements it was necessary to instrument a JVM to produce the relevant data. Kaffe [19], an independent implementation of the JVM distributed under the GNU Public License, was used to collect data on the dynamic operation of all 19 programs. Kaffe version 1.0.6 was used for this study.

3   Measurement and Analysis Model

Of the 256 possible bytecodes, only 201 bytecode instructions are defined for use in a class file (a further three are reserved for "internal use" by a JVM). While it is clearly possible to collect data for each individual bytecode (see e.g. [11]), it will suit our purposes to divide these instructions into categories. A number of possible sub-divisions could be used. We have chosen a relatively fine-grained approach, selecting 31 categories to roughly parallel those used for machine instruction sets in [8]. The bytecodes in each of these categories are listed in Figure 3.

For each of these 31 categories, information was first compiled on their frequency of static occurrence in the bytecode files for each of the 19 applications in the three test suites. The 19 applications were then run on the instrumented Kaffe Virtual Machine, and information was collected on the dynamic usage frequency for each of the categories. In all cases, the results collected excluded bytecodes executed in API routines, since these are specific to the Kaffe class library implementation. It should be noted however, that the percentage of executed bytecodes in the API varies considerably across applications [7]. While lack of space prohibits presenting the full frequency

Grande Kernels		
	Static	Dynamic
k-crypt	1416	905,263,342
k-fft	1007	4,221,501,308
k-heapsort	405	1,103,190,209
k-lufact	1511	833,455,892
k-series	508	560,524,605
k-sor	368	3,413,988,850
k-sparse	433	1,063,559,470

Grande Applications		
	Static	Dynamic
g-eul	10580	14,514,096,409
g-mol	2514	7,599,820,106
g-mon	4181	1,632,874,942
g-ray	2518	11,792,255,694
g-sea	3060	7,103,726,472

SPEC JVM98		
	Static	Dynamic
s-compress	2403	12,472,779,077
s-db	1924	1,116,130,852
s-jack	26317	264,885,522
s-javac	58301	1,014,147,162
s-jess	24193	1,554,389,306
s-mpeg	44788	11,488,951,023
s-mtrt	8647	2,121,501,462

Figure 2: *Summary of the experimental environment.* This table summarises the test environment by giving the number of (static) bytecode instructions in each application program, along with the total number of dynamically-executed bytecode instructions.

counts here, the overall size of the measurement is summarised in Figure 2.

In order to test for a linear relationship between the static and dynamic instruction frequencies, we use Pearson's coefficient. Pearson's sample correlation coefficient r is given by:

$$ r = \frac{\Sigma(x - \bar{x})(y - \bar{y})}{(n-1)s_x s_y} $$

Here, the two sets of data of size n are represented by the variables x and y, with means $\bar{x}$ and $\bar{y}$ and standard deviations s_x and s_y

Since the value of r does not depend on the unit of measurement, the static and dynamic figures for each bytecode category were expressed as *percentages* of the total. This facilitated comparison across different applications, since the total number of bytecode

Category	Instruction No.
acnst	1
aload	25, 42, 43, 44, 45
array	188, 189, 190, 197
astor	58, 75, 76, 77, 78
cjump	153-166, 170, 171, 198, 199
compr	148, 149, 150, 151, 152
f_add	98, 99
fcall	182, 183, 184, 185
fcnst	11, 12, 13, 14, 15
f_div	110, 111, 114, 115
field	178, 179, 180, 181
fload	20, 23, 24, 34-41
f_mul	106, 107
fstor	56, 57, 67-74
f_sub	102, 103, 118, 119
i_add	96, 97
icnst	2, 3, 4, 5, 6, 7, 8, 9, 10
i_div	108, 109, 112, 113
iload	16-19, 21, 22, 26-33
i_mul	104, 105
istor	54, 55, 59-66
i_sub	100, 101, 116, 117
logic	126-132
miscl	0, 186, 191, 194, 195, 202-255
objct	187, 192, 193
retrn	169, 172, 173, 174, 175, 176, 177
shift	120, 121, 122, 123, 124, 125
stack	87-95, 133-147, 196
ujump	167, 168, 200, 201
yload	46-53
ystor	79-86

Figure 3: *The bytecode categories used in this paper*. This table lists the 31 bytecode categories used in this paper, along with the set of bytecode instructions assigned to each category.

instruction	$\bar{x}$	s_x	$\bar{y}$	s_y	r
i_mul	0.247	0.422	0.367	0.969	0.880
logic	1.816	2.006	3.372	3.982	0.873
iload	14.511	6.332	18.063	9.185	0.847
shift	0.269	0.550	0.465	1.196	0.844
fload	3.461	3.392	6.328	10.089	0.798
i_add	0.981	0.837	2.535	2.899	0.788
i_div	0.223	0.326	0.186	0.640	0.781
astor	1.293	1.271	0.936	1.524	0.750
fstor	0.870	0.985	1.486	2.333	0.743
f_mul	1.095	1.362	2.714	2.595	0.723
f_add	0.755	0.849	3.042	3.532	0.697
fcall	10.233	5.845	3.627	4.580	0.694
istor	2.763	2.045	2.505	3.111	0.691
objct	1.461	1.254	0.554	1.267	0.665
fcnst	0.667	0.841	0.332	0.848	0.665
yload	3.717	3.343	7.525	6.502	0.651
f_sub	0.523	0.611	1.145	2.212	0.628
icnst	7.126	3.382	3.206	3.037	0.588
aload	17.030	6.243	16.233	8.433	0.554
cjump	3.082	1.356	5.665	3.092	0.546
field	11.777	6.632	11.122	8.813	0.503
ujump	1.697	0.815	0.508	0.683	0.468
compr	0.375	0.441	0.745	2.229	0.443
f_div	0.315	0.395	0.093	0.203	0.419
i_sub	0.768	0.787	0.687	0.968	0.360
array	1.034	0.896	0.224	0.380	-0.355
acnst	0.291	0.346	0.067	0.205	0.289
retrn	3.182	1.539	2.074	3.336	0.258
ystor	3.303	3.841	1.790	1.747	0.183
stack	5.001	3.526	2.400	2.583	0.109
miscl	0.134	0.304	0.005	0.021	0.087

Figure 4: *Summary of results obtained*. Here, variables x and y range over the static and dynamic data respectively and r is the linear correlation coefficient.

instructions varied considerably from one application to another.

The values for this coefficient range between -1 and 1, where $|r| \geq 0.8$ denotes a strong linear correlation, $0.5 \leq |r| \leq 0.8$ denotes a moderate linear correlation, and values of $|r| \leq 0.5$ denote a weak linear correlation.

4 Results and Observations

A summary of the results obtained from calculating Pearson's correlation coefficient is presented in Figure 4.

As explained in the previous section, the bivariate data was calculated as:

x The frequencies of static occurrence of instructions from each of the 30 categories, expressed as a percentage of the static instruction total

y The frequency of dynamic usage of instructions from each of the 30 categories, expressed as a percentage of the dynamic instruction total

In Figure 4 we show the summary data for each instruction category, consisting of the averages $\bar{x}$ and $\bar{y}$, the standard deviations s_x and s_y for both sets of data,

and the linear correlation coefficient r. The results are sorted in decreasing order of linear correlation.

It is immediately apparent that there is overall no strong linear correlation between the static and dynamic figures. As can be seen in Figure 4, only 4 of the categories have a linear correlation coefficient greater than 0.8, and none greater than 0.9. The remaining 27 categories exhibit only a moderate or weak linear correlation. Indeed, even among the 4 categories exhibiting a strong linear correlation, three of these, *i_mul*, *shift* and *logic* have a very small average percentage use. It is easily verified that the main reason for the strong linear correlation for these three categories is the large number of applications that make little or no use, statically or dynamically, of these instructions.

The one reasonably frequently-used instruction category that does exhibit a strong linear correlation is *iload*, which loads an integer from the local variable array onto the stack. Indeed, the coefficient for the floating-point equivalent *fload* is 0.798, suggesting a reasonably strong linear correlation here too. This contrasts with the weaker linear correlation of *aload* instructions at just 0.554. One explanation for this is that the *aload* category is made up chiefly of *aload_0* instructions - *aload_0* instructions count for 11.4/17.0 of the static total and 9.2/16.2 of the dynamic total of *aload* instructions. This instruction typically denotes the load of the `this`-pointer before a method invocation or field reference, and its usage varies considerably across applications. It is particularly unusual for the *g-mol* application, a translation of a FORTRAN program into Java.

5   Conclusion

In this paper we have presented a methodology and results for determining the presence of a linear correlation between static and dynamic bytecode-level data for Java programs. We have discussed the background and merits to such an approach, and have presented some results for programs taken from the SPEC and Java Grande benchmarks suites. Despite the suggestion of [8] that static frequency analysis may be used to predict dynamic performance for some assembly languages, our figures show that this result is not directly transferable to Java bytecodes.

In future work we propose to extend this study further, in particular addressing the following issues:

- Current JVM optimisation techniques concentrate on the determination of program *hotspots* [1], where dynamic execution is concentrated. We propose to investigate if there is a relationship between the importance of hotspots in a program's execution, and the lack of linear correlation between the static and dynamic frequency data.

- The 31 categories chosen in this study were selected to parallel those in [8]; other choices are possible, and we propose to investigate if any other choices produce better linear correlation.

- While the test suites involved are reasonably standard, we foresee incorporating more of the benchmarks suites discussed in Section 2 into our research. Ultimately, we would like to use this technique as an aid to determine the suitability of individual programs within a benchmark suite in terms of their overall bytecode profile.

We believe that the work presented here provide a solid basis for this further research.

References:[1]

[1] E. Armstrong. Hotspot: A new breed of virtual machine. *Java World*, March 1998.

[2] S. Baylor, M. Devarakonda, S. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S. J. Munroe. Java server benchmarks. *IBM Systems Journal*, 39(1):57–81, 2000.

[3] K. Bowers and D. Kaeli. Characterising the SPEC JVM98 benchmarks on the Java virtual machine. Technical report, Northeastern University Computer Architecture Research Group, Dept. of Electrical and Computer Engineering, Boston Massachusetts 02115, USA, 1998.

[4] K. Bowles. UCSD Pascal. *Byte*, 3:170–173, May 1978.

[1]All URLs last checked on November 27, 2002

[5] M. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. A methodology for benchmarking Java Grande applications. In *ACM 1999 Java Grande Conference*, pages 81–88, Palo Alto, CA, USA, June 1999.

[6] 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.

[7] 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 - IS-COPE 2001 Conference*, pages 106–115, Stanford, CA, USA, June 2001.

[8] J. Davidson, J. Rabung, and D. Whalley. Relating static and dynamic machine code measurements. *IEEE Transactions on Computers*, 41(4):444–454, April 1992.

[9] S. Dieckmann and U. Hölzle. A study of the allocation behaviour of the SPEC JVM98 Java benchmarks. In *13th European Conference on Object Oriented Programming*, pages 92–115, Lisbon, Portugal, June 1999.

[10] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[11] C. Herder and J. Dujmović. Frequency analysis and timing of Java bytecodes. Technical Report SFSU-CS-TR-00.02, San Francisco State University, Department of Computer Science, January 15 2000.

[12] J. Horgan, J. Power, and J. Waldron. Measurement and analysis of runtime profiling data for Java programs. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 122–130, Florence, Italy, November 10 2001.

[13] T. Li, L. John, V. Narayanan, A. Sivasubramaniam, J. Sabarinathan, and A. Murthy. Using complete system simulation to characterize SPEC JVM98 benchmarks. In *International Conference on Supercomputing*, pages 22–33, Santa Fe, NM, USA, May 2000.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.

[15] J. Neffenger. The Volano report: Which Java platform is fastest, most scalable? *JavaWorld*, March 1999.

[16] Pendragon. Caffeinemark 3.0, May 13 1999. http://www.pendragon-software.com/-pendragon/cm3/.

[17] 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.

[18] SPEC. SPEC releases SPEC JVM98, first industry-standard benchmark for measuring Java virtual machine performance. Press Release, August 19 1998. http://www.specbench.org/osg/jvm98/-press.html.

[19] T. Wilkinson. KAFFE, a virtual machine to run Java code, July 2000. http://www.kaffe.org.

[20] M. Wolczko. Benchmarking java with richards and deltablue, 2001. http://www.sun.com/-research/people/mario/java_benchmarking/.