A Dynamic Comparison of the SPEC98 and Java Grande Benchmark Suites.

Siobhán Byrne
Telecom Management Systems, Ericsson Telecom,
Adelphi Centre, Upper Georges Street,
Dun Laoghaire, Co.Dublin, Ireland.
James Power
Department of Computer Science, NUI Maynooth, Ireland.
John Waldron
Department of Computer Science, Trinity College, Dublin 2, Ireland.

May 9, 2001

Abstract

Two of the most commonly used benchmark suites for Java Programs are the SPEC98 and Grande Forum benchmark suites. This research uses a Platform Independent Dynamic Analysis Technique to study there suites and quantify the significant similarities and differences in behaviour between the suites. Dynamic frequencies adduced include method execution divided into program, API and native categories. The most informative basis for measurement is shown to be percentages of executed bytecodes charged to each method, and results are reported for the API packages.

Keywords: Java Bytecode, Intermediate Representation, Java Grande

1 Introduction

The Java programming language has evolved from a specialist programming language designed for embedded systems to a widely-used platform for mobile code, and a common choice as a general-purpose programming language. As such, it has become the focus of a variety of software tools, right from high-level design and analysis tools for software engineering down to low-level profiling tools for Virtual Machine development. The Java paradigm for executing programs is a two stage process. Firstly the source is converted into a platform independent intermediate representation, consisting of bytecode and other information stored in class files. The second stage of the process involves hardware specific conversions, perhaps by a JIT compiler for the particular hardware in question, followed by the execution of the code.

As the transformation from source to machine instructions is more complex in Java, than say C, it is more difficult to measure and analyse the dynamic execution of particular programs or benchmark suites. In [1] it was shown that Platform Independent Dynamic Analysis (PIDA) is a powerful methodology for characterizing the behavior of Java Grande Applications from the Java Grande Forums benchmark suite. In this paper, the PIDA technique is used to study the SPEC98 [2] benchmark suite, a standard suite of Java Programs, and the results compared with the Grande programs.

2 PIDA Instrumentation of Kaffe

In order to study dynamic method usage it was necessary to modify the source code of a Java Virtual Machine. Kaffe [3] is an independent implementation of the Java Virtual Machine which was written from scratch and is free from all third party royalties and license restrictions. It comes with its own standard class libraries, including Beans and Abstract Window Toolkit (AWT), native libraries, and a highly configurable virtual machine with a JIT compiler for enhanced performance. Kaffe is available under the Open Source Initiative and comes with complete source code, distributed under the GNU Public License. Versions 1.0.6 was used for these measurements.

In order to modify the Kaffe virtual machine to ac-

Accepted for the Workshop on Intermediate Representation Engineering for the Java Virtual Machine Orlando, Florida, USA, July 22-25, 2001

cumulate dynamic platform independent statistics, most of the alterations are made in the machine.c file. The simplest measurement, how many of the bytecodes are in the API, can be made in the interpreter loop in the runVirtualMachine() function. Of course since it is in the inner loop, it will impact execution speed when the measurement is being performed. In order to measure dynamic method call frequencies, it is necessary to use a hash table dictionary with method names as keys. This can be called once per method in the virtualMachine() function. The best metric, however, for estimating eventual running time is to use a cost center design pattern to "charge" each bytecode to the appropriate method. This would necessitates accessing an item in the hash table dictionary each time round the interpreter inner loop in the run runVirtualMachine() function and incrementing a counter, which would slow down execution significantly (about 8 times interpreted speed) while the measurement is being performed. To improve performance, when a method is invoked, a local variable in the virtual machines stack frame can be initialised to point to the counter for that method, requiring only one hash table access per method invocation and only doubling the running time relative to interpretation.

3 Grande Programs Measured

A *Grande* application is one which uses large amounts of processing, I/O, network bandwidth or memory. The Java Grande Forum Benchmark Suite (http://www.epcc.ed.ac.uk/javagrande/) [4] is intended to be representative of such applications, and thus to provide a basis for measuring and comparing alternative Java execution environments. It is intended that the suite should include not only applications in science and engineering but also, for example, corporate databases and financial simulations.

- The **moldyn** benchmark is a translation of a Fortran program designed to model the interaction of molecular particles. Its origin as non object-oriented code probably explains its relatively unusual profile, with 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 **search** benchmark solves a game of connect-4 on a 6×7 board using alpha-beta pruning. Intended to be memory and numerically intensive, this is also the only application to demonstrate an inheritance hierarchy of depth greater than 2.
- The **euler** benchmark solves a set of equations using a fourth order Runge-Kutta method. This suite demonstrates a considerable clustering of functionality in the `Tunnel` class, as well as a comparatively high percentage of methods with very large local variable requirements.
- The **raytracer** measures the performance of a 3D ray tracer rendering a scene containing 64 spheres. It is represented using a fairly shallow inheritance tree, with functionality (as measured in methods) fairly well distributed throughout the classes.
- 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).

Version 2.0 of the suite (Size A) was used. The default Kaffe maximum heap size of 64M was sufficient for all programs except *mon* which needed a maximum heap size of 128M. The *ray* application failed its validation test when interpreted, but as the failure was by a small amount, it was included in the measurements.

4 SPEC98 Programs Measured

- The **compress** benchmark uses modified Lempel-Ziv method (LZW) which finds common substrings and replaces them with a variable size code. This is deterministic, and can be done on the fly.
- The **JESS** benchmark is the Java Expert Shell System is based on NASA's CLIPS expert shell system. In simplest terms, an expert shell system continuously applies a set of if-then statements, called rules, to a set of data, called the fact list. The benchmark workload solves a set of puzzles commonly used with CLIPS. To increase run time the benchmark problem interactively asserts a new set of facts representing the same puzzle but with different literals. The older sets of facts are not retracted. Thus the inference engine must search through progressively larger rule sets as execution proceeds.

Program	Total methods	API %	API native %
eul	3.34e+07	58.0	12.6
mol	5.49e+05	22.7	19.9
mon	8.07e+07	98.7	37.4
ray	4.58e+08	3.1	1.6
sea	7.12e+07	0.0	0.0
ave	1.29e+08	36.5	14.3

Table 1: *Measurements of total number of method calls including native calls by Grande applications compiled using SUNs javac compiler, Standard Edition (JDK build 1.3.0-C). Also shown is the percentage of the total which are in the API, and percentage of total which are in API and are native methods.*

- The **db** benchmark performs multiple database functions on memory resident database. reads in a 1 MB file which contains records with names, addresses and phone numbers of entities and a 19KB file called scr6 which contains a stream of operations to perform on the records in the file.
- The **javac** benchmark is the Java compiler from the JDK 1.0.2. As this is a commercial application, no source code is provided.
- The **mpegaudio** benchmark is an application that decompresses audio files that conform to the ISO MPEG Layer-3 audio specification. As this is a commercial application only obfuscated class files are available. The workload consists of about 4MB of audio data.
- The **mtrt** benchmark is a raytracer that works on a scene depicting a dinosaur, where two threads each renders the scene in the input file time-test model, which is 340KB in size.
- The **jack** benchmark is a Java parser generator that is based on the Purdue Compiler Construction Tool Set (PCCTS). This is an early version of what is now called JavaCC. The workload consists of a file named jack.jack, which contains instructions for the generation of jack itself. This is fed to jack so that the parser generates itself multiple times.

5 PIDA Comparison

Table 1 [1] and Table 2 show dynamic method frequencies and native frequencies for Grande and SPEC98 applications. It is interesting to note that the SPEC benchmarks have higher frequencies than the so-called Grande applications. The higher native frequencies shown by the Grande Applications

Program	Total methods	API %	API native %
Compress	2.26e+08	0.0	0.0
JESS	1.35e+08	32.5	1.9
Database	1.24e+08	98.7	0.1
javac	1.53e+08	62.0	2.8
mpegaudio	1.10e+08	1.3	1.1
mtrt	2.88e+08	3.2	0.1
jack	1.16e+08	92.3	4.2
ave	1.65e+08	41.4	1.5

Table 2: *Measurements of total number of method calls including native calls by SPEC JVM98 applications Also shown is the percentage of the total which are in the API, and percentage of total which are in API and are native methods.*

Program	Java method calls		bytecodes executed	
	number	% in API	number	% in API
eul	2.92e+07	51.9	1.46e+10	0.5
mol	4.40e+05	3.4	7.60e+09	0.0
mon	5.05e+07	97.9	2.63e+09	38.0
ray	4.50e+08	1.5	1.18e+10	0.1
sea	7.12e+07	0.0	7.10e+09	0.0
ave	1.20e+08	30.9	8.75e+09	7.7

Table 3: *Measurements of Java method calls excluding native calls made by Grande applications compiled using SUNs javac compiler, Standard Edition (JDK build 1.3.0-C).*

(14.3% against 1.5%) are due to the more mathematical nature of these programs, which tend to call native methods such as java/lang/Math.sqrt and java/lang/Math.log with high frequency.

Table 3 [1] and Table 4 compare dynamic measurements of Java method call frequencies with *bytecode usage frequencies* which should provide a more accurate measure of execution time spent in areas of the programs. It can be seen that 92% of the Grande programs execution time on average is spend in the program methods, whereas in the SPEC suite the compiler like tools and also the database application spend most of there time in the API methods,

Program	Java method calls		bytecodes executed	
	number	% in API	number	% in API
Compress	2.26e+08	0.0	1.25e+10	0.0
JESS	1.32e+08	31.2	1.91e+09	18.8
Database	1.24e+08	98.7	3.77e+09	70.4
javac	1.48e+08	60.9	2.43e+09	58.3
mpegaudio	1.08e+08	0.1	1.15e+10	0.0
mtrt	2.88e+08	3.1	2.20e+09	3.5
jack	1.11e+08	92.0	1.50e+09	82.3
ave	1.62e+08	40.9	5.12e+09	33.3

Table 4: *Measurements of Java method calls excluding native calls made by SPEC JVM98 applications.*

	Compress	JESS	Database	javac	mpegaudio	mtrt	jack	ave
io	3.9	0.5	0.0	31.0	8.9	56.6	2.7	14.8
lang	52.4	37.9	73.4	32.4	59.4	43.1	19.5	45.4
net	0.7	0.0	0.0	0.0	0.3	0.0	0.0	0.1
util	43.0	61.5	26.6	36.6	31.3	0.3	77.8	39.6

Table 6: *Breakdown of Java (non-native) API bytecode percentages by package for SPEC JVM98 applications.. None of the applications used methods from the applet, awt, beans, math, security or sql packages.*

	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

Table 5: *Breakdown of Java (non-native) API bytecode percentages by package for Grande applications compiled using SUNs javac compiler, Standard Edition (JDK build 1.3.0-C). None of the applications used methods from the applet, awt, beans, math, security or sql packages.*

giving an average figure of 67% of time in the program bytecodes. The total bytecodes executed by the Grande applications are only slightly higher than the SPEC98 figure.

Table 5 and Table 6 compare dynamic measurements of *bytecode usage frequencies* for the different API packages, which should provide a measure of execution time spent in areas of those parts of the libraries. In both case all the time is concentrated in *lang, util* and *io*, with the Grande applications having high *lang* and lower *io* usage. It is surprising that the standard benchmarking programs do not exercise a greater variety of API packages.

6 Conclusions

Two of the most commonly used benchmark suites for Java Programs are the SPEC98 and Grande Forum benchmark suites. This research set out to use the Platform Independent Dynamic Analysis Technique [1] to study these suites and quantify the significant similarities and differences in behaviour between the suites. Dynamic frequencies adduced include method execution divided into program, API and native categories. It is interesting to note that the SPEC benchmarks have higher frequencies than the so-called Grande applications.

The most informative measurement is shown to be percentages of executed bytecodes charged to each method, and it has been shown that 92% of the Grande programs execution time on average is spend in the program methods, whereas in the SPEC suite the compiler like tools and also the database application spend most of their time in the API methods, giving an SPEC98 average figure of 67% of time in the program bytecodes. It is surprising that the standard benchmarking programs do not exercise a greater variety of API packages.

References

[1] C. Daly, J Horgan, J Power and J. T. Waldron, *Platform Independent Dynamic Java Virtual Machine Analysis: the Java Grande Forum Benchmark Suite*, Proceedings of Joint ACM Java Grande - ISCOPE (International Symposium on Computing in Object-oriented Parallel Environments) 2001 Conference, Stanford University, June 2-4, 2001

[2] *SPEC JVM98 Benchmarks*
<http://www.spec.org/osg/jvm98>
URL last accessed on 2/4/2001

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

[4] Bull M, Smith L, Westhead M, Henty D and Davey R. *Benchmarking Java Grande Applications*, Second International Conference and Exhibition on the Practical Application of Java, Manchester, UK, April 12-14, 2000.

[5] J. Waldron, C. Daly, D. Gray and J. Horgan, *Comparison of Factors Influencing Bytecode Usage in the Java Virtual Machine*, Second International Conference and Exhibition on the Practical Application of Java, Manchester, UK, April 12-14, 2000.