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 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. 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-
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 be-
ing 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 in-
terpreter 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 ac-
cess 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 com-
paring alternative Java execution environments. It
is intended that the suite should include not only
applications in science and engineering but also, for
elementary, corporate databases and financial simu-
lations.

- The moldyn benchmark is a translation of a
  Fortran program designed to model the inter-
  action of molecular particles. Its origin as non
  object-oriented code probably explains its rela-
  tively 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 x 7 board using alpha-beta
  pruning. Intended to be memory and numer-

Version 2.0 of the suite (Size A) was used. The de-
fault Kaffe maximum heap size of 64M was sufficient
for all programs except mon which needed a max-
imum heap size of 128M. The ray application failed
its validation test when interpreted, but as the fail-
ure 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 sys-
  tem continuously applies a set of if-then state-
  ments, 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.
Table 1: Measurements of total number of method calls including native calls by Grande applications compiled using SUN's 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.

<table>
<thead>
<tr>
<th>Program</th>
<th>Total methods</th>
<th>API %</th>
<th>API native %</th>
</tr>
</thead>
<tbody>
<tr>
<td>eul</td>
<td>3.34e+07</td>
<td>58.0</td>
<td>12.6</td>
</tr>
<tr>
<td>mol</td>
<td>5.49e+05</td>
<td>22.7</td>
<td>19.9</td>
</tr>
<tr>
<td>mon</td>
<td>8.07e+07</td>
<td>98.7</td>
<td>37.4</td>
</tr>
<tr>
<td>ray</td>
<td>4.58e+08</td>
<td>3.1</td>
<td>1.6</td>
</tr>
<tr>
<td>sea</td>
<td>7.12e+07</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>ave</td>
<td>1.29e+08</td>
<td>36.5</td>
<td>14.3</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Program</th>
<th>Total methods</th>
<th>API %</th>
<th>API native %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compress</td>
<td>2.26e+08</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>JESS</td>
<td>1.35e+08</td>
<td>32.5</td>
<td>1.9</td>
</tr>
<tr>
<td>Database</td>
<td>1.24e+08</td>
<td>98.7</td>
<td>0.1</td>
</tr>
<tr>
<td>javac</td>
<td>1.53e+08</td>
<td>62.0</td>
<td>2.8</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>1.10e+08</td>
<td>1.3</td>
<td>1.1</td>
</tr>
<tr>
<td>mtr</td>
<td>2.88e+08</td>
<td>3.2</td>
<td>0.1</td>
</tr>
<tr>
<td>jack</td>
<td>1.16e+08</td>
<td>92.3</td>
<td>4.2</td>
</tr>
<tr>
<td>ave</td>
<td>1.60e+08</td>
<td>41.4</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Table 3: Measurements of Java method calls excluding native calls made by Grande applications compiled using SUN's 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 4 [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 their time in the API methods.

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


![Image of a page from a document with text]

**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.

<table>
<thead>
<tr>
<th></th>
<th>eul</th>
<th>mol</th>
<th>mon</th>
<th>ray</th>
<th>sea</th>
<th>ave</th>
</tr>
</thead>
<tbody>
<tr>
<td>io</td>
<td>7.6</td>
<td>1.2</td>
<td>0.3</td>
<td>0.0</td>
<td>1.2</td>
<td>2.1</td>
</tr>
<tr>
<td>lang</td>
<td>92.2</td>
<td>69.5</td>
<td>2.0</td>
<td>99.3</td>
<td>69.6</td>
<td>66.5</td>
</tr>
<tr>
<td>net</td>
<td>0.0</td>
<td>1.1</td>
<td>0.0</td>
<td>0.0</td>
<td>1.3</td>
<td>0.5</td>
</tr>
<tr>
<td>text</td>
<td>0.0</td>
<td>0.6</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
</tr>
<tr>
<td>util</td>
<td>0.1</td>
<td>27.5</td>
<td>97.7</td>
<td>0.7</td>
<td>28.0</td>
<td>30.8</td>
</tr>
</tbody>
</table>

Table 5: Breakdown of Java (non-native) API bytecode percentages by package for Grande applications compiled using SUNs javac compiler, Standard Edition (JDK build 1.3.6-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 behavior between the suites. Dynamic frequencies added 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


