EPCC-SSP00-03

Benchmarking, Profiling and Performance Analysis of the Java Grande Benchmarking Suite

Robin Freeman

September 8, 2000

Abstract

In previous papers a number of issues concerning the Java Grande Benchmarking suite have been raised. This paper hopes to examine these issues by conducting a new set of benchmarks and analysing the results. It then investigates the relationship between benchmarks within the suite, while reviewing the various profiling utilities available for Java. Finally, it discusses the design and implementation of a user interface for the suite.

Contents

1 Introduction 3

2 Background 3
2.1 A Benchmark Suite . . . 3
2.2 Organisation of the Suite . . . 4

3 Benchmarking and results 5
3.1 New Results . . . 5
3.2 New Results Vs Previous Results . . . 6

4 Profiling and Performance Analysis 8
4.1 Profilers . . . 8
4.2 Analysing Section III . . . 8
4.3 The Relationship between Section I and Section III . . . 10

5 Graphical user interface 10

6 Conclusions 11

7 Further Work 12

A Appendix 13
A.1 Further Results from New Benchmarks . . . 13
A.2 Correlation results . . . 14
A.3 CPU Profiling Information . . . 15
A.4 Program code for moldyn.particle.force() . . . 16

List of Figures

1 JGF Numbers for various new Virtual Machines . . . 6
2 Comparison of JGF Numbers from Previous and New JDKs . . . 7
3 Heap Allocation graph from JGFSearchBench . . . 9
4 Heap Allocation Graph from JGFMonteCarloBench . . . 9
5 Organisation of currently implemented GUI classes . . . 12
6 Further Results: Section 2, new Java Virtual Machines (data size B) . . . 13
7 Further Results: Section 3, new Java Virtual Machines (data size A) . . . 13

List of Tables

1 JGF Numbers for various execution environments . . . 6
2 Overall JGF Numbers for Previous and New Versions of JDKs . . . 7
3 Comparison of Sun JDK Versions . . . 7
4 Simple correlation of Section I to Section III - With IBM . . . 14
5 Simple correlation of Section I to Section III - Without IBM . . . 14

1 Introduction

The Edinburgh Parallel Computing Centre (EPCC) has constructed a benchmarking suite for Java, which examines Java execution environments for their use for Grand scale applications. A number of results from the suite have been obtained and analysed. A number of problems can be highlighted concerning these results and the suite in general. Firstly, these results are already obsolete: a new set of Virtual Machines has already been released. The suite also relies on file I/O, so comparisons between benchmarks can be a long and tedious task. There is also a lack of correspondence between some of the low-level operations (Section I) and some of the application style benchmarks (Section III) in the suite. Finally, these application style benchmarks are mostly third party codes which have been submitted and converted for use in the suite, so it would also be beneficial to gain any further information on them.

This paper will initially conduct a series of benchmarks on some of the newly released execution environments, and then examine those results with reference to the problems noted above. Then profiling information from the execution of some of the Section III benchmarks will be collected, initially with the hope of explaining why there is such a low correspondence between Section I and Section III and also to lend further insight into the section III benchmarks. Finally, this paper will examine the construction of a graphical user interface for the suite, which should simplify the analysis of the benchmark results.

2 Background

2.1 A Benchmark Suite

Over the past few years there has been an increasing interest in the use of Java for Grande scale applications (large-scale, potentially requiring large amount of processing power, network bandwidth, I/O, and memory). Sun and the Northeast Parallel Architectures Centre have led a community initiative under the title of the Java Grande Forum (JGF) to address this issue with the hope promoting the use of Java for Grande scale codes. The EPCC at the University of Edinburgh is highly involved in this initiative, and has created a benchmark suite (see [3] and [2]) aimed at testing aspects of Java execution environments (JVMs, Java compilers, etc.) for their use with Grande scale applications.

The methodology for creating this benchmark [2] highlights a number of requirements for any successful benchmark, the suite must be:

- Representative: The nature of the computation in the benchmark suite should reflect the types of computation which might be expected in the Java Grande applications.
- Interpretable: As far as possible, the suite as a whole should not merely report performance of a Java environment, but also lend some insight into why a particular level of performance was achieved.
- Robust: The performance of the suite should not be sensitive to factors which are of little interest (for example, the size of cache memory, or the effectiveness of dead code elimination).

- Portable: The benchmark suite should run on as wide a variety of Java environments as possible.
- Standardised: The elements of the benchmark should have a common structure and a common ’look and feel’. Performance metrics should have the same meaning across the suite.
- Transparent: It should be clear to anyone running the suite exactly what is being tested.

For these requirements to be fulfilled, a number of steps were taken in the suites design. Firstly, in order to make the suite representative the structure of the GENISIS Benchmark [1] suite is adopted, providing three types of benchmark; low-level operations (Section I), simple kernels (Section II), and applications (Section III). Thus, the results from Section I should determine the performance of real applications (corresponding to Section III) running under the Java environment, as noted in [2]

> ...we hope to observe the behaviour of the most complex applications and interpret that behaviour through the behaviour of the simpler codes.

To achieve a robust suite, a range of data sizes are provided for sections II and III. Care is also taken to prevent possible compiler optimisations of strictly dead code. Maximum portability is achieved by relying on simple file I/O throughout the suite, since some systems may not provide interactive execution environments (although, see below 5). A JGF instrumentor class is used throughout the suite to achieve standardisation. Finally, transparency is achieved by releasing the source code for all the benchmarks, thus removing any ambiguity in what’s being tested.

The design is comprehensive, but in order to assess whether these goals have been achieved, actual results must be examined.

These results are provided in [3] with a comparison of various execution environments on various systems. The results seem to show that most of the requirements have been achieved: comparisons between execution environments on the same hardware platform show similar trends in performance across the suite. However, it appears that predicting the performance of the Section III codes by analysing the Section I results is difficult, this is complicated further by the inability to examine the resulting assembly code produced by the JIT compilers.

2.2 Organisation of the Suite

To understand the results from the suite, an explanation of the suite and its performance metrics is required. The suite consists of three sections (following the GENESIS benchmark suite structure [1]): low-level operations (Section I), computational kernels (Section II), and applications (Section III). Within each section there are a number of benchmarks, which in the case of section I are further broken down into micro-benchmarks. Section II and III then contain differing data set sizes, three for section II and two for section III.

Each benchmark (Section II and III) produces a number of results. Firstly, execution time: the wall-clock time required to execute the ’interesting’ portion of the benchmark code. Then temporal performance: defined in units of a operations per second, where the operation is chosen to be the most appropriate for each individual benchmark. Finally, relative performance: the ration of temporal performance to that obtained for a reference system (a chosen Virtual Machine, Operating System and Hardware configuration. The reference system is Sun 1.2.1_002

JDK running on a Sun Ultra Enterprise 3000 250Mhz under Solaris 2.6). The section I micro-benchmarks do not report execution time.

Within section I the results from the micro-benchmarks are built up to give a JGF Number for the benchmark. This is done by taking the geometric mean (See eqn. 1) of the relative performances (against the reference set). Similarly this is done for each data size in Section II and III. The same process is then used to produce a JGF Number for each section, and finally for the entire suite. The overall JGF Number, however, is only the geometric mean of the Section II medium data size and the Section III small data size. This means that possible optimisation of the section I benchmarks will not effect the overall JGF Number.

Geometric Mean

$$\sqrt[n]{x_1 x_2 \ldots x_n} \tag{1}$$

n = number of results,
x = relative performance.

3 Benchmarking and results

3.1 New Results

A new series of Virtual Machines have been released over the past few months. Therefore, to make sure the arguments throughout this paper are current, it must be the results from these VMs which are analysed.

All benchmarks were performed on a Pentium III 700Mhz system with 256MB of RAM available, the Virtual Machine configurations tested were:

- Sun JDK Version 1.3.0 Standard Edition
- Sun JDK Version 1.3.0 with Hotspot Client Build 2.0
- Sun JDK Version 1.3.0 with Hotspot Server Build 2.0
- IBM JDK Version 1.2.0
- Microsoft SDK for Java Version 4.0

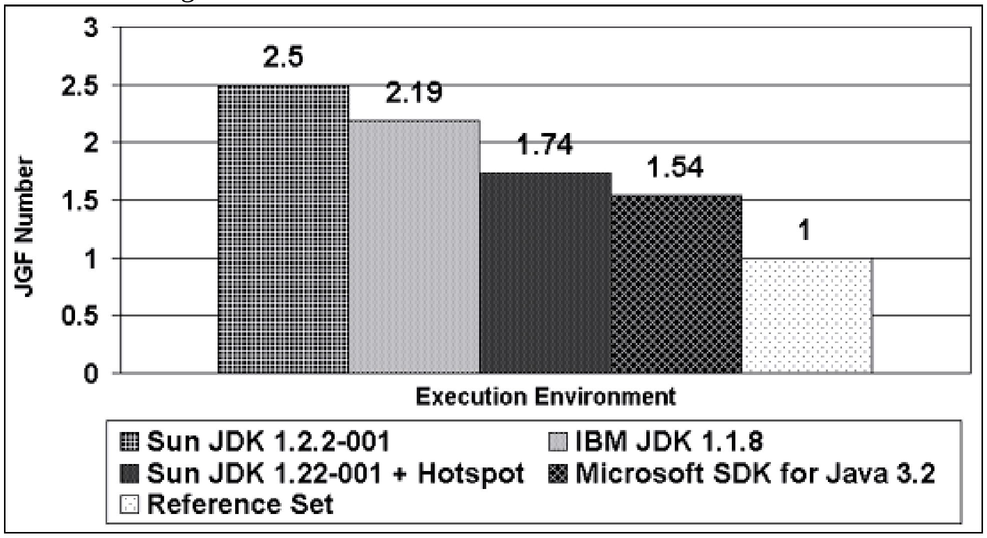
It should be noted that since the last set of benchmarks were completed Sun have altered the structure of their JDK. The previous JDK consisted of the ’classic’ Virtual Machine (VM), and then the Hotspot Engine. The new JDK retains the ’classic’ VM, but the Hotspot Engine has been broken into ’Client’ and ’Server’ versions. Therefore, when comparing the results from previous benchmarks to the new results the faster of the two new Hotspot versions is used to compare to the old Hotspot result.

From Table 1 and Figure 1, it can be seen that there is a general trend across the sections (a better performance at any given section usually indicates better performance in the other sections). However, there still seems to be a lack of correspondence between Section I and Section III. In comparing the Sun + Hotpsot Server against the Microsoft result it can be seen that although the Sun VM outperforms the Microsoft by almost 50% on Section I, this affect is not seen on

Table 1: JGF Numbers for various execution environments

	JGF Number			
Virtual Machine	*Section 1*	*Section 2*	*Section 3*	*Overall*
Sun JDK 1.3.0	0.69	0.57	0.36	0.45
Sun JDK 1.3.0 + Hotspot Client 2.0	2.46	2.28	2.88	2.56
Sun JDK 1.3.0 + Hotspot Server 2.0	3.12	2.32	2.61	2.46
IBM JDK 2.0	6.84	3.53	3.04	3.28
Microsoft SDK for Java 4.0	2.08	3.22	2.60	2.89

Figure 1: JGF Numbers for various new Virtual Machines



Section III where the results are almost the same, and indeed the Microsoft VM outperforms the Sun on Section II. This effect could be put down to the Sun VM optimising away some of the Section I benchmarks, and is worthy of further investigation.

Also worthy of note is the high performance of the IBM VM. Given that the benchmarks were conducted on the Microsoft Windows NT 4.0 platform and that Sun are the foremost Java developers it is interesting to see IBM give the best performance. The extremely high results for Section I from the IBM could again be due to optimisation effect, yet this does not affect the overall score for the benchmark since only the Section II and Section III results are used to calculate the overall result (see Section 2.2).

3.2 New Results Vs Previous Results

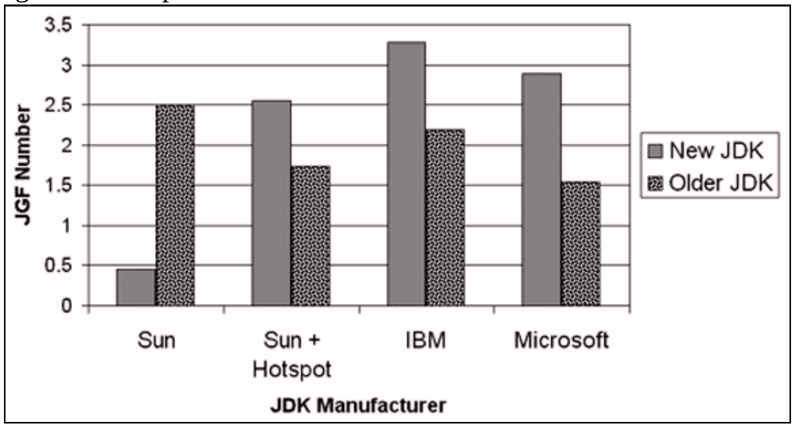
When the previous results are presented against the new results (Table 2) it can be seen that the new JDKs are generally scoring higher that their previous counterparts. This is true for all of the results except the Sun JDK (classic). The new JDK seems to show a dramatic drop in performance.

When the Sun results were examined more closely, this was initially put down to the fact that the new benchmark was compiled without using the '- O' flag for compile time optimisation. Another run of the benchmark was completed using this flag, yet this proved to have no effect

Table 2: Overall JGF Numbers for Previous and New Versions of JDKs

	JGF Number	
Manufacturer	*Old JDK*	*New JDK*
Sun	2.5	0.45
Sun + Hotspot	1.74	2.56
IBM	2.19	3.28
Microsoft	1.54	2.89

Figure 2: Comparison of JGF Numbers from Previous and New JDKs



on the result. Eventually it was decided to run the previous results again on precisely the same system as the new results (previous results were run on a Dual 500Mhz Pentium PC running Windows NT). The suite was compiled using the '- O' optimisation flag and run using the '-classic' flag. It should be noted here, however, that the previous run had used the Sun JDK Version 1.2.2_001. This was, however, unavailable so the Sun JDK Version 1.2.2_006 was used. The results are included in Table 3. As can be seen, it seems that the new JDK performs significantly worse than the previous version. Whether this is due to initial release problems, platform issues, or is in fact intended is not known, but is worthy of further research.

Table 3: Comparison of Sun JDK Versions

	JGF Number			
Virtual Machine	*Section I*	*Section II*	*Section III*	*Overall*
Sun 1.2.2_001 (Previous Run)	3.01	2.78	2.24	2.50
Sun 1.2.2_006	–	3.14	3.12	3.13
Sun 1.3.0	0.69	0.57	0.36	0.45
Sun 1.3.0 using '- O'	0.69	0.57	0.36	0.45

4 Profiling and Performance Analysis

4.1 Profilers

The benchmarking group at EPCC have been interested in examining the various profilers available for Java with a hope of both gaining further insight into the Section III benchmarks and perhaps their relationship to Section I.

A web survey was performed to discover the various profiling utilities which were available for Java, the most prominent of which were:

- OptimizeIt
- JProbe Profiler
- HProf

Each of the profilers is very powerful in its own right, yet each also has various downfalls. The OptimizeIt and JProbe profilers are the main commercially available profilers and should therefore be examined separately from HProf, which is freely distributed with many JDKs.

Both OptimizeIt and JProbe present the data they record in an easy to use format, showing heap allocation, garbage collection, active threads, and other information graphically while displaying CPU sampling information in a ’tree’ structure’. Where they differ most is on their output. While OptimizeIt outputs its heap/garbage collection/thread/etc. information as image files, JProbe outputs it as formatted text (HTML/ascii). Both techniques are valid, and contain the same data, it simply depends on what the user requires/prefers. Also, both OptimizeIt and JProbe only work with a selected number of JDKs, which in this case (comparison of runs on different JDKs) limited their usefulness.

HProf is a simpler utility which connects to the Java Virtual Machine Profiling Interface (JVMPI) and should therefore work with any VM which implements the JVMPI. It therefore allowed us to make comparisons between profiling information from runs on the Sun and IBM JDKs. Although HProf only produces binary or ascii output, which makes it harder to analyse the data, there do exist a number of utilities which can reformat this data so its easier to understand (PerfAnal, for example).

The choice of profiler therefore seems dependent on what is required. While OptimizeIt and JProbe are very powerful and implement simpler data formatting, they are commercial and only work with a limited number of JDKs. HProf, however, is free and works with numerous JDKs, yet produces data which can be harder to analyse.

4.2 Analysing Section III

When it came to analysing the Section III codes, OptimizeIt was used due to its output of graphical images to file. A number of things can be seen from the profiling data. Firstly when the heap allocation patterns were examined, almost all of the Section III benchmarks produced a standard allocation/garbage-collection pattern, as can be seen in Figure 3. This highlighted the irregularity of the allocation pattern from the JGFMonteCarlo benchmark, shown in Figure 4.

Figure 3: Heap Allocation graph from JGFSearchBench

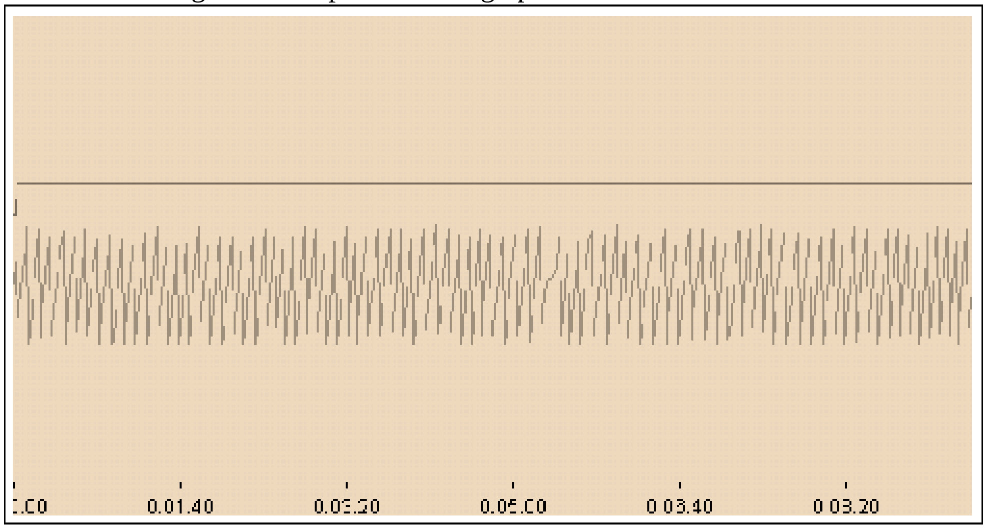
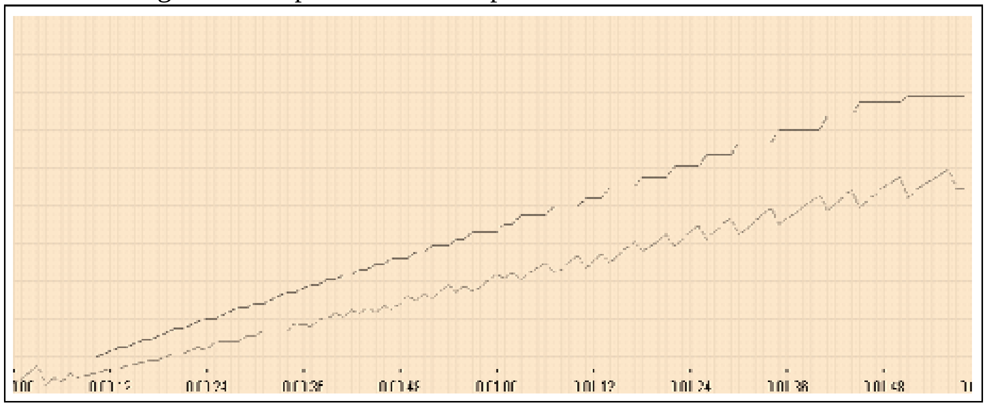


Figure 4: Heap Allocation Graph from JGFMonteCarloBench



It appears that the JGFMonteCarlo benchmark does not allow the garbage collector to dispose of any of the objects it creates. This leads to huge memory usage (>250MB for the large data size) which prevents the benchmark from being run to completion on systems with less memory.

Also available from the profilers was the use of CPU sampling to measure the amount of CPU time taken executing each method of the benchmark. The Section III codes varied greatly in this respect, with some spending large portions of their time in a single method (see Appendix A.3), and some spreading the CPU time over various methods (see Appendix A.3).

Although the heap allocation and CPU sampling information was all that was examined for the serial benchmark suite, a parallel version is in development and these profiling utilities may prove considerably more useful when examining the multi-threaded benchmarks.

4.3 The Relationship between Section I and Section III

As noted above in Section 3.1, it seems hard to find a relationship between the performance of the Section I benchmarks and the Section III benchmarks. Using a simple correlation, it can be seen that such a relationship may exist (see Appendix A.2, Tables 4 and 5), although this does not give us any insight into the nature of such a relationship. These results must, therefore, be analysed in more detail.

It was thought that through the use of the newly acquired profiling information it would be possible to discover the most frequently used method within a Section III benchmark. The program code for that benchmark could then be examined and the low-level operations it used recorded. This would hopefully result in a relationship between Section I (the low level benchmark) and Section III being evident.

Given the CPU sampling information for the JGFMolDyn benchmark (Appendix A.3) it can be seen that around 99% of its execution time is spent in the moldyn.particle.force() method. When the moldyn.particle.force() method was examined it could be seem that all the calls within it were to low-level or primitive methods (See Appendix A.4).

By collating the results for the Section I benchmarks concerned, and comparing this to the result of the JGFMolDyn benchmark a relationship should therefore be seen. This poses a problem: how are the Section I results to be collated? For any given Section III code, each of the low-level operations are performed only a limited number of times, while the Section I benchmarks perform them many times.

On initial examination it seemed as though a relationship had been found (using the Arith Benchmark result as the total score for the combined Section I), however further results did not conform with this. A number of different methods of collating the Section I results were made yet none of them produced consistent results.

5 Graphical user interface

A final problem concerning the benchmark suite was the method of compiling results from the suite. Currently the suite produces a text output file comprising of all of the benchmark results (around 160 in total). An HTML generating utility is currently in use, which compiles the results into appropriate sections, computes the JGF Numbers and generates HTML tables of the results.

Although this significantly simplifies the analysis of results, it still proves irritating to have to compare different runs of the benchmark suite by hand.

It was therefore decided that a graphical user interface to the suite should be produced, with the aim of simplifying the analysis process. It is worth noting here that , due to time constraints, this has not been completed (although a significant amount of work has been put into it).

After initial discussions, a set of requirements which the interface should fulfil were drawn up:

- Allow presentation of multiple results graphically.
- Allow sets of results to be loaded/saved at runtime.
- Also allow users to view the numerical results.
- Allow users to insert H/W and S/W information into a results set.
- Allow users to select which results they wished to view.
- Provide some rudimentary statistical analysis of results.
- Allow simple navigation of results.
- Output graphical/statistical data in appropriate formats.
- Be written using the Java Swing packages.

Work initially began on creating the component which would display the results graphically. This proved complex due to the use of the Swing Model/View/Controller (MVC) architecture, and a lack of documentation on the subject. However such a component is partially complete, it displays any results which are presented to it in a TableModel object. (*This component only requires completion of axis and labels*).

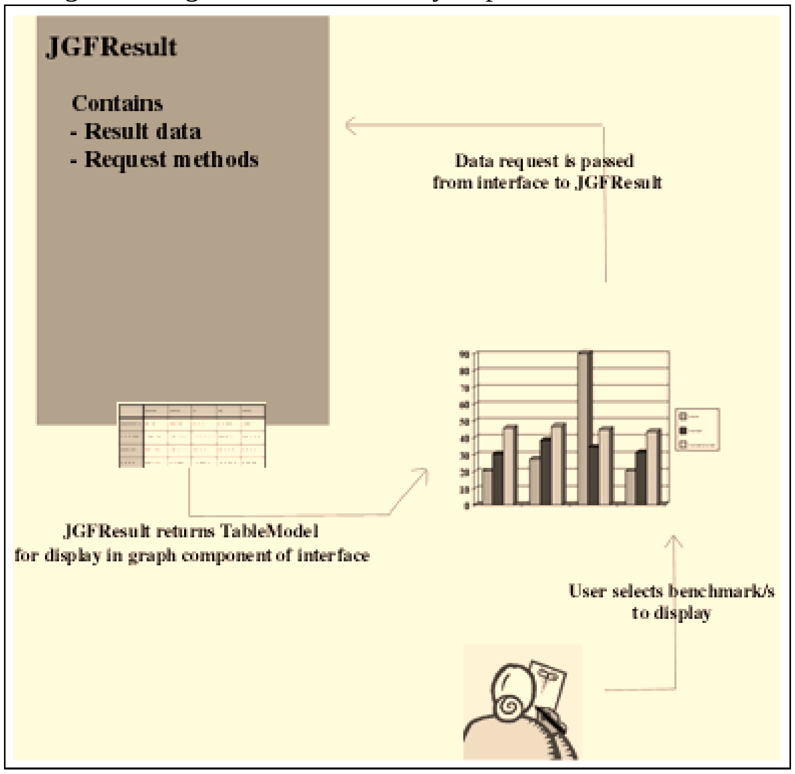
To allow results to be loaded and saved at runtime a JGFResult object was created, this object is created from a results file (which can be incomplete), and stores the results independently of the source file. Methods were added to this object to allow it to return a TableModel object of the results requested. For example, if the Arith benchmark results were requested, the JGFResult class would return a TableModel containing the Arith benchmark results, which could then be passed to the graph component for display. This process is shown more clearly in Figure 5.(*Again, this is near completion, only requiring minor modifications*).

The remaining components are still to be introduced. The graphical user interface, therefore, still requires considerable work.

6 Conclusions

A new series of benchmarks has been completed on the most recent set of Virtual Machines (for the Windows NT platform), this shows a general trend of increased performance over the previous versions. A series of profiling tools has been examined, lending further insight into the operation of the Section III benchmarks. The relationship between Section I and Section III has been further examined, but still requires work. Finally, work has begun on a graphical user interface which will eventually simplify the analysis of benchmark results.

Figure 5: Organisation of currently implemented GUI classes


JGFResult
Contains
- Result data
- Request methods
Data request is passed from interface to JGFResult
JGFResult returns TableModel for display in graph component of interface
User selects benchmark/s to display

7 Further Work

There are a number of areas in which further work could be focused:

- The completion of the graphical user interface
- Further analysis of the relationship between Section I and Section III
- Further examination of the low performance of the Sun VM.
- Investigation into the use of profiling tools for the multithreaded benchmark suite
- Revision of the JGFMonteCarlo benchmark to minimise its memory usage.

Figure 6: Further Results: Section 2, new Java Virtual Machines (data size B)

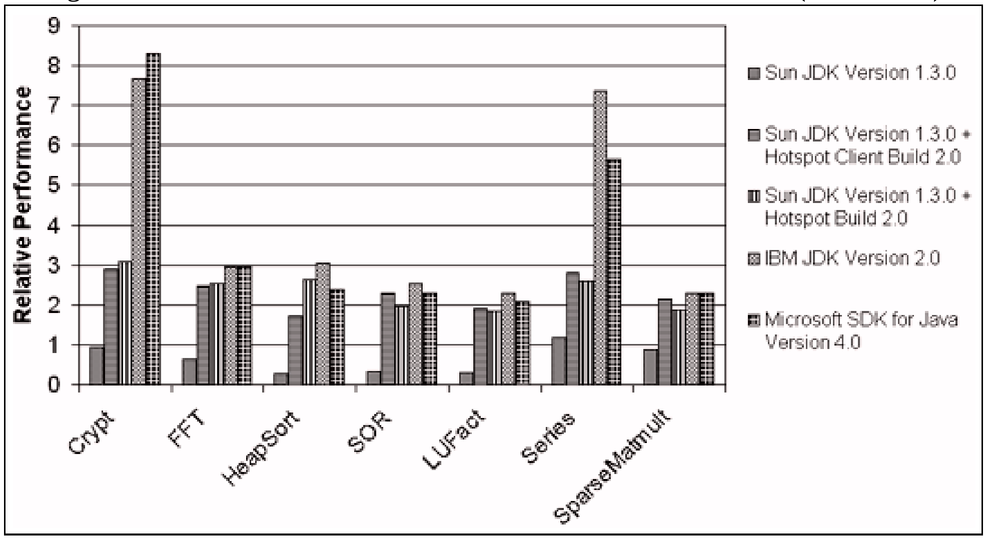
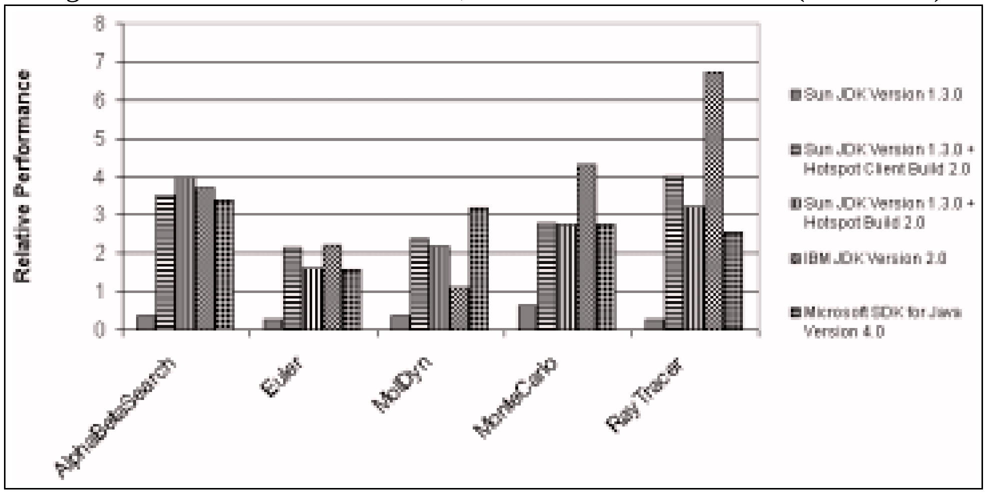


Figure 7: Further Results: Section 3, new Java Virtual Machines (data size A)



A Appendix

A.1 Further Results from New Benchmarks

Figure 6 shows the results for Section 2, data size B (medium).

Figure 7 shows the results for Section 3, data size A (small).

Table 4: Simple correlation of Section I to Section III - With IBM

Virtual Machine	*Section I*	*Section III*	*Correlation Coefficient*
Sun JDK 1.3.0	0.69	0.36	0.67
Sun JDK 1.3.0 + Hotspot Client 2.0	2.46	2.88	–
Sun JDK 1.3.0 + Hotspot Server 2.0	3.12	2.61	–
IBM JDK 2.0	6.84	3.04	–
Microsoft SDK for Java 4.0	2.08	2.60	–

Table 5: Simple correlation of Section I to Section III - Without IBM

Virtual Machine	*Section I*	*Section III*	*Correlation Coefficient*
Sun JDK 1.3.0	0.69	0.36	0.90
Sun JDK 1.3.0 + Hotspot Client 2.0	2.46	2.88	–
Sun JDK 1.3.0 + Hotspot Server 2.0	3.12	2.61	–
Microsoft SDK for Java 4.0	2.08	2.60	–

A.2 Correlation results

Tables 4 and 5 show that when the extremely high result from the IBM JDK is removed, a reasonable correlation between the Section I results and the Section III results can be found

A.3 CPU Profiling Information

Below is the CPU profiling data for the JGFMolDynBenchSizeA benchmark. It is easy to see that it spends most of the CPU time executing the moldyn.particle.force() method

```
Profiler output for thread main .
application JGFMolDynBenchSizeA
(CPU profiler output - Sampler / Methods)
--------------------------------------------------------------

Description of CPU usage for thread main
   99.51\% - 17495 ms - moldyn.particle.force()
   0.17\% - 30 ms - java.lang.Math.sqrt()
   0.12\% - 22 ms - moldyn.particle.domove()
   0.09\% - 16 ms - moldyn.particle.mkekin()
   0.03\% - 7 ms - moldyn.particle.dscal()
   0.03\% - 6 ms - moldyn.particle.velavg()
   0.02\% - 5 ms - moldyn.md.runiters()
```

Below is the CPU profiling data for the JGFMonteCarloBenchSizeA benchmark. Unlike the Monte Carlo benchmark above, here the CPU time is spread throughout different methods

```
Profiler output for thread main .
application JGFMonteCarloBenchSizeA
(CPU profiler output - Sampler / Methods)
-------------------------------------------------------------

Description of CPU usage for thread main
   13.58\% - 1896 ms - java.lang.Math.sqrt()
   13.39\% - 1870 ms - java.util.Random.nextGaussian()
   13.22\% - 1847 ms - java.util.Random.next()
   11.71\% - 1635 ms - montecarlo.ReturnPath.computeVariance()
   10.12\% - 1414 ms - montecarlo.MonteCarloPath.\
computeFluctuationsGaussian()
   9.59\% - 1340 ms - java.util.Random.nextDouble()
   6.99\% - 976 ms - montecarlo.RatePath.getReturnCompounded()
   5.78\% - 808 ms - java.lang.Math.log()
   4.62\% - 646 ms - montecarlo.MonteCarloPath.\
computePathValue()
   3.95\% - 552 ms - montecarlo.ReturnPath.computeMean()
   2.78\% - 389 ms - montecarlo.PriceStock.setInitAllTasks()
   1.93\% - 270 ms - java.lang.Math.exp()
   0.48\% - 68 ms - montecarlo.RatePath.<init>()
   0.21\% - 30 ms - java.lang.Long.toString()
   0.17\% - 24 ms - java.lang.String.<init>()
   0.17\% - 24 ms - java.lang.FloatingDecimal.developLongDigits()
   0.14\% - 20 ms - montecarlo.PriceStock.getResult()
   0.14\% - 20 ms - java.lang.StringBuffer.expandCapacity()
   ...
```

A.4 Program code for moldyn.particle.force()

As can be seen below, the force() method consists of mainly low-level operations (sum, product, difference, loops, etc.)

```
public void force(double side, double rcoff,int mdsize,int x) {

    double sideh;
    double rcoffs;

    double xx,yy,zz,xi,yi,zi,fxi,fyi,fzi;
    double rd,rrd,rrd2,rrd3,rrd4,rrd6,rrd7,r148;
    double forcex,forcey,forcez;

    int i;

    sideh = 0.5*side;
    rcoffs = rcoff*rcoff;

     xi = xcoord;
     yi = ycoord;
     zi = zcoord;
     fxi = 0.0;
     fyi = 0.0;
     fzi = 0.0;

       for (i=x+1;i<mdsize;i++) {
        xx = xi - md.one[i].xcoord;
        yy = yi - md.one[i].ycoord;
        zz = zi - md.one[i].zcoord;

        if(xx < (-sideh)) { xx = xx + side; }
        if(xx > (sideh))  { xx = xx - side; }
        if(yy < (-sideh)) { yy = yy + side; }
        if(yy > (sideh))  { yy = yy - side; }
        if(zz < (-sideh)) { zz = zz + side; }
        if(zz > (sideh))  { zz = zz - side; }

        rd = xx*xx + yy*yy + zz*zz;

        if(rd <= rcoffs) {
           rrd = 1.0/rd;
           rrd2 = rrd*rrd;
           rrd3 = rrd2*rrd;
           rrd4 = rrd2*rrd2;
           rrd6 = rrd2*rrd4;
           rrd7 = rrd6*rrd;
           md.epot = md.epot + (rrd6 - rrd3);
           r148 = rrd7 - 0.5*rrd4;
           md.vir = md.vir - rd*r148;
           forcex = xx * r148;
           fxi = fxi + forcex;
           md.one[i].xforce = md.one[i].xforce - forcex;
           forcey = yy * r148;
           fyi = fyi + forcey;
           md.one[i].yforce = md.one[i].yforce - forcey;
           forcez = zz * r148;
           fzi = fzi + forcez;
           md.one[i].zforce = md.one[i].zforce - forcez;
           md.interactions++;
        }

       }

     xforce = xforce + fxi;
     yforce = yforce + fyi;
     zforce = zforce + fzi;

}
```

References

[1] C.A. Addison, V.S Getov, R.W. Hockney, and I.C. Walton. The GENESIS Distributed-memory Benchmarks. *Advances in Parallel Computing*, 8:257–271, 1991.

[2] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A Benchmark Suite for High Performance Java. EPCC, University of Edinburgh, September 2000. available from http://www.epcc.ed.ac.uk/javagrande.

[3] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. Benchmarking Java Grande Applications. EPCC, University of Edinburgh, September 2000. available from http://www.epcc.ed.ac.uk/javagrande.

Robin is originally from Edinburgh, but is currently completing his degree in Computer Science(AI) at the University of Aberdeen. He comes back to Edinburgh to play and to see his cat.

Project Supervisor: Dr. Lorna Smith