Benchmarking Java Virtual Machines

Peter Amberg (pamberg@bigfoot.com) Patrik Stähli (patrik@staehli.cjb.net)

March 13, 2000

Abstract

This is a project for the lecture *Computer Systems Performance Analysis and Benchmarking* held by Prof. Thomas M. Stricker at the *Eidgenössische Technische Hochschule Zürich.*

1 Goal

The goal of this paper is basically to compare different Java Virtual Machines from different vendors and under different platforms against each-other. In particular, we are not going to compare the entire JDKs against each-other, that is we don't compile the benchmarks on each platform anew, but we are going to use the same byte code of the benchmarks in each JVM.

It seems possible that the byte code we're going to use can be interpreted by one JVM (probably the one from the same JDK as the compiler that was used to compile the benchmark) more efficiently than by another. Therefore, it *should* be made sure that the compiler that is used does not have any correlation to the performance of each JVM. However, we are *not* going to do this because the 'ReadMe's that come along with the Java SPEC benchmark says the byte code that comes together with SPEC *must* be used:-

> *Source code for some of the benchmarks is provided with so that people can better understand what the benchmarks are doing, and to facilitate academic research. However, no compilation of Java source code to class files is required to run the benchmarks, and no such compilation is allowed for reported results.*

The JVMs are to be compared in terms of speed only, *not* in terms of accuracy (ie. the JVM does what it should/what is specified by the Java API) and availability (ie. the JVM implements the Java API completely). Only the Java API 1.1 is to be used for the tests; the JVMs may or may not implement a newer version.

2 Before we start...

Before we start running the Java SPEC benchmarks, we state what the characteristics and properties of our SUT (System Under Test) are.

2.1 Services and Outcomes

A Java Virtual Machine provides one major service: running a Java program. This could be divided into a part 'running a Java applet' and a part 'running a Java application.' However, dividing these further into subservices for each Java package that is provided would probably get messy and not be very useful.

For this paper, we will use only the most abstract form of service provided by a JVM:

'running a Java program.' This service has the following outcomes:-

- The Java program does not run at all; an error that occurs in the loading process already prevents it from running.
- The Java program is started but unexpectedly terminates at a certain point of execution due to some error that is not caused by the program itself.
- The Java program runs until it correctly terminates, but it does not do what it is expected to do, ie. its outputs/results in calculation are wrong somehow.
- The Java program runs until it correctly terminates, and it does actually what it is expected to do.

2.2 Metrics

There are three basic kinds of metrics: speed, accuracy and availability. For a JVM, speed means the performance with which a certain Java program runs under that JVM. This is the kind of metric we are interested in this paper; it is expressed in a certain SPEC metric that is calculated from the time a benchmark takes for execution.

Accuracy would mean that the JVM does what it should do/what is specified by the Java API. Availability would mean that the JVM completely implements the Java API to be used for the test (we will use Version 1.1). In this paper, we're interested neither in accuracy nor availability—although it is required of course that the JVMs correctly and thoroughly implement the API that is going to be used by the benchmarks.

2.3 Parameters

The following system parameters may affect the performance of the benchmarks:-

- The Java Virtual Machine under test.
- The benchmark (workload).
- The underlying operating system, ie. the efficiency of the system calls and the load that is on the computer besides the benchmark.
- The underlying hardware, ie. the type of processor, the processor clock frequency, the bus clock frequency, the amount of RAM that is available, the speed of RAM, and so on.

In addition to these system parameters, there are workload parameters. The SPEC Java Benchmarks have *lots* of parameters, so we don't list them here. Figure 1 shows the essential settings of the workload parameters in order that the results are reportable. We've made sure these parameters were set as directed; the other parameters were left in their default settings.

(However, it must be pointed out that our results are not reportable anyway, since we are not going to run the benchmarks via a web server as it would be required. Therefore, this paper is not meant to be officially published...)

2.4 Factors

Of the parameters listed in the previous subsection, we pick the following ones as factors to study:-

- The Java Virtual Machine under test.
- The workload.
- The underlying operating system.

The table 1 shows what are the different JVMs and OSes we're going to study, and in what combinations. In case a combination is

A reportable result must:

- Select benchmark size 100.
- Select benchmark group "All".
- Run benchmarks in an "autorun" sequence.
- Leave benchmark harness cache turned off. E.g., `spec.initial.cache=false`.
- Set `spec.initial.autodelay` to no more than 1000 ms.
- Run all the benchmarks via a web server.

FIGURE 1: Essential workload parameters

	Windows	*Linux*
IBM	V. 1.1.8	V. 1.1.8
SUN	V. 1.1.8	V. 1.1.8
	V. 1.2.2	V. 1.2.2 RC3
M$	V. unknown	*n/a*

TABLE 1: The factors to study

to be studied, we've written the version of the JDK into the corresponding cell in the table. We could not find out what version the JVM by Microsoft under *Windows 98* is; we could not find it in any of the documentation files that came with the JVM. The Microsoft JVM is not available for Linux, so this combination cannot be studied.

The *Operating Systems* to study are:-

- *Linux*: SuSE Linux 6.3, Kernel version 2.2.14, glibc 2.1.2
- *Windows*: Micro$oft Windows 98SE (Second Edition), 4.10 Build 2222 A

The *Linux* version of *SUN V1.2.2* is only a release candidate (RC3).

The *Linux* version of both *SUN* JVMs are not from *SUN* directly, but from *Blackdown* (`www.blackdown.org`), because *SUN* only provides JVMs for Win32 and Solaris.

The levels of the factor 'Workload' will be the workloads of the SPEC Java Benchmark.

We are not going to vary the underlying hardware, but we state our configuration here all the same for completeness:-

- *Processor:* Intel Celeron`(r)`, 450MHz.
- *Bus Speed:* 75MHz.
- *RAM:* 128MB, SDRAM, 7ns.
- *HD:* IDE, IBM, 10GB.

2.5 Evaluation

The technique of evaluation is measurement and analysis. We use the SPEC Java Benchmarks as the workload. The table 2 shows the names and a short description of the various workloads of the SPEC Java Benchmark.

3 Running the Benchmarks

We have three factors: the operating system, the JVM (vendor and version), and the workload. We study two operating systems and three JVMs under seven workloads (from the SPEC Java Benchmark). This makes $2 * 3 * 7 = 42$ combinations. If each combination is done with three replications, this yields

Workload	*Description*
_200_check	test the functionality of the JVM (performance not measured)
_227_mtrt	raytracer; two threads
_202_jess	if-thens, calls
_201_compress	string operations
_209_db	database stuff (adding, deleting, searching, sorting addresses)
_222_mpegaudio	decompressing some audio data
_228_jack	Java parser generator
_213_javac	Java compiler

Table 2: The workloads of the SPEC Java Benchmark

$42 * 3 = 126$ experiments that have to be done for a full-factorial design. This is a reasonable amount, and so we decided to do a full-factorial design.

(Actually, we have one more JVM in the test: the Microsoft JVM under *Windows*. However, this JVM is not available for *Linux*, so it would make things only complicated if we included it in the calculations we're going to make. We present the SPEC results for this JVM here, but later it is dropped.)

The SPEC Java Benchmarks do replications themselves, but the number of replications is variable (depending on different criterias such as the time taken by previous replications). These replications then are packed into two values for each benchmark: the slowest time taken by any of the replications and the fastest one are picked. We've decided to do three replications on top of SPEC's replications because we found that SPEC is doing too few replications by itself. Also, doing these replications will be useful when we calculate the variation.

The tables 3, 5, and 7 present the *slowest* results of the benchmarks running under *Linux* and under the JVMs *IBM V1.1.8*, *SUN V1.1.8* and *SUN V1.2.2*, respectively. The tables 4, 6, and 8 present the *fastest* results for the same combinations of the factors. The tables 9, 11, 13, and 15 present the *slowest* results of the benchmarks running under *Window 98* and under the JVMs *IBM V1.1.8*, *SUN V1.1.8*, *SUN V1.2.2* and *M$ JVM*, respectively. Finally, the tables 10, 12, 14, and 16 show the *fastest* results for the same combinations of the factors.

Benchmark	1st	2nd	3rd
_227_mtrt	13.20	14.60	12.90
_202_jess	20.30	20.10	20.00
_201_compress	34.90	35.10	35.10
_209_db	9.45	9.48	9.47
_222_mpegaudio	50.40	49.40	49.90
_228_jack	27.00	26.90	26.70
_213_javac	8.61	8.31	8.44
Geometric Mean	19.40	19.50	19.20

Table 3: Slowest results with the JVM *IBM 1.1.8* under Linux

The figure 2 shows the results in a more comprehensive form. Both the slowest and fastest results are displayed. The standard deviation is indicated by a mark on top of each bar.

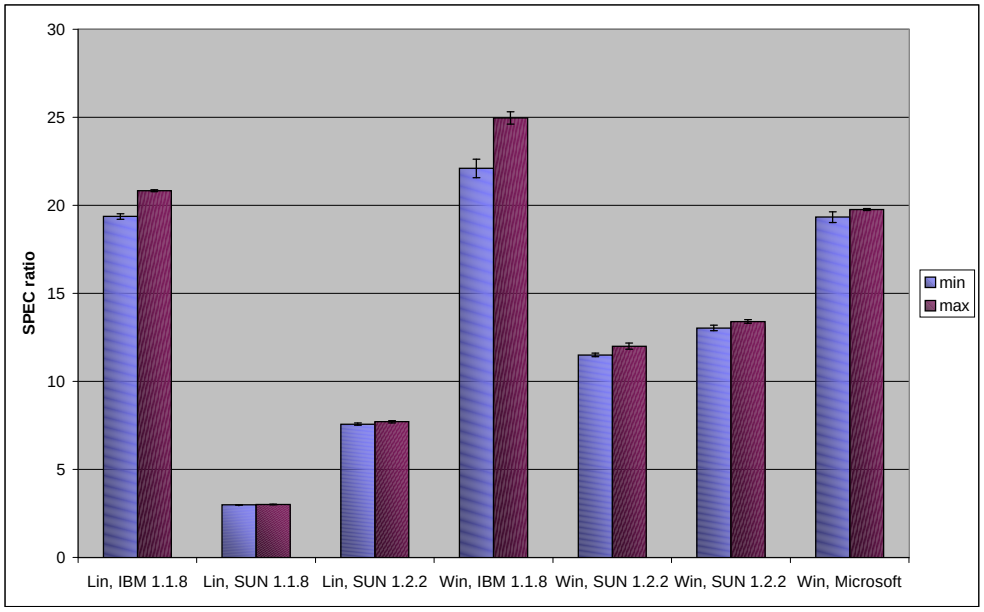

FIGURE 2: A more comprehensive representation of the results

Benchmark	1st	2nd	3rd
_227_mtrt	16.80	17.50	17.60
_202_jess	21.10	21.00	21.00
_201_compress	34.90	35.10	35.10
_209_db	9.83	9.72	9.72
_222_mpegaudio	51.20	51.30	50.80
_228_jack	28.30	28.50	28.10
_213_javac	9.56	9.45	9.37
Geometric Mean	20.80	20.90	20.80

TABLE 4: Fastest results with the JVM *IBM 1.1.8* under Linux

Benchmark	1st	2nd	3rd
_227_mtrt	3.17	3.14	3.12
_202_jess	3.69	3.71	3.72
_201_compress	2.76	2.76	2.76
_209_db	2.29	2.30	2.29
_222_mpegaudio	3.14	3.13	3.13
_228_jack	3.32	3.31	3.30
_213_javac	2.98	2.81	3.02
Geometric Mean	3.02	2.99	3.02

TABLE 6: Fastest results with the JVM *SUN 1.1.8* under Linux

Benchmark	1st	2nd	3rd
_227_mtrt	3.03	3.03	2.99
_202_jess	3.68	3.69	3.71
_201_compress	2.76	2.75	2.75
_209_db	2.29	2.29	2.29
_222_mpegaudio	3.14	3.13	3.12
_228_jack	3.31	3.31	3.30
_213_javac	2.95	2.80	3.00
Geometric Mean	2.99	2.97	2.99

TABLE 5: Slowest results with the JVM *SUN 1.1.8* under Linux

Benchmark	1st	2nd	3rd
_227_mtrt	6.94	7.02	7.14
_202_jess	6.63	6.50	6.95
_201_compress	15.60	15.70	15.80
_209_db	3.58	3.57	3.59
_222_mpegaudio	19.20	18.90	19.10
_228_jack	6.76	6.67	6.64
_213_javac	4.15	4.25	4.30
Geometric Mean	7.53	7.53	7.65

TABLE 7: Slowest results with the JVM *SUN 1.2.2 RC3* under Linux

Benchmark	1st	2nd	3rd
_227_mtrt	7.18	7.25	7.26
_202_jess	6.95	6.63	7.20
_201_compress	15.90	15.90	15.80
_209_db	3.61	3.59	3.62
_222_mpegaudio	19.50	19.30	19.40
_228_jack	6.83	6.72	6.64
_213_javac	4.16	4.31	4.45
Geometric Mean	7.69	7.65	7.77

TABLE 8: Fastest results with the JVM *SUN 1.2.2 RC3* under Linux

Benchmark	1st	2nd	3rd
_227_mtrt	30.00	31.00	29.90
_202_jess	23.30	24.00	22.40
_201_compress	39.10	38.50	37.30
_209_db	13.40	13.40	13.20
_222_mpegaudio	53.40	52.00	51.10
_228_jack	8.89	9.10	8.78
_213_javac	15.60	16.10	14.40
Geometric Mean	22.30	22.50	21.50

TABLE 9: Slowest results with the JVM *IBM 1.1.8* under Windows

Benchmark	1st	2nd	3rd
_227_mtrt	45.50	45.50	45.50
_202_jess	25.70	26.40	25.40
_201_compress	39.10	38.80	39.10
_209_db	13.60	13.60	13.50
_222_mpegaudio	55.50	54.40	52.90
_228_jack	9.20	9.42	9.13
_213_javac	19.10	20.20	18.70
Geometric Mean	25.00	25.30	24.60

TABLE 10: Fastest results with the JVM *IBM 1.1.8* under Windows

Benchmark	1st	2nd	3rd
_227_mtrt	9.57	9.53	9.52
_202_jess	12.30	12.80	12.40
_201_compress	33.70	33.90	33.90
_209_db	5.19	5.59	5.31
_222_mpegaudio	43.30	43.40	43.80
_228_jack	6.34	6.21	6.45
_213_javac	4.56	4.45	4.46
Geometric Mean	11.40	11.60	11.50

TABLE 11: Slowest results with the JVM *SUN 1.1.8* under Windows

Benchmark	1st	2nd	3rd
_227_mtrt	10.40	11.50	10.80
_202_jess	12.60	13.10	12.70
_201_compress	34.20	34.30	34.30
_209_db	5.21	5.64	5.34
_222_mpegaudio	44.50	43.70	44.80
_228_jack	6.41	6.25	6.51
_213_javac	5.10	5.15	4.69
Geometric Mean	11.90	12.20	11.90

TABLE 12: Fastest results with the JVM *SUN 1.1.8* under Windows

Benchmark	1st	2nd	3rd
_227_mtrt	12.10	12.10	12.40
_202_jess	12.40	12.40	10.80
_201_compress	31.50	32.70	32.70
_209_db	6.66	6.69	6.76
_222_mpegaudio	42.10	42.50	43.10
_228_jack	6.69	7.10	7.18
_213_javac	6.59	6.95	6.84
Geometric Mean	12.90	13.20	13.00

TABLE 13: Slowest results with the JVM *SUN 1.2.2* under Windows

Benchmark	1st	2nd	3rd
_227_mtrt	12.90	13.10	13.00
_202_jess	12.90	12.60	11.90
_201_compress	32.90	32.80	32.70
_209_db	6.83	6.90	6.95
_222_mpegaudio	43.80	43.40	44.40
_228_jack	6.77	7.24	7.19
_213_javac	6.62	7.01	6.88
Geometric Mean	13.30	13.50	13.40

TABLE 14: Fastest results with the JVM *SUN 1.2.2* under Windows

Benchmark	1st	2nd	3rd
_227_mtrt	26.70	27.10	26.90
_202_jess	19.20	19.70	19.80
_201_compress	31.80	33.20	33.30
_209_db	12.10	12.20	12.20
_222_mpegaudio	47.90	50.10	49.60
_228_jack	8.62	8.73	8.59
_213_javac	11.00	11.50	11.40
Geometric Mean	19.00	19.60	19.40

TABLE 15: Slowest results with the JVM by Microsoft under Windows

Benchmark	1st	2nd	3rd
_227_mtrt	27.00	27.30	26.90
_202_jess	20.20	20.40	20.30
_201_compress	33.60	33.40	33.40
_209_db	12.20	12.30	12.30
_222_mpegaudio	51.10	50.40	50.80
_228_jack	8.72	8.80	8.67
_213_javac	11.80	11.70	11.70
Geometric Mean	19.80	19.80	19.70

TABLE 16: Slowest results with the JVM by Microsoft under Windows

4 Analyzing the Results

4.1 Intuitive Analysis

After a quick look at the results, it appears that the *IBM V1.1.8* be the fastest JVM of the selected ones.

Having read the documentation, we learned that *SUN* does not provide a *JIT* (Just-in-time) compiler with the *Linux* version—which explains why the resulting performance for the *SUN V1.1.8* are much slower under *Linux* than under *Windows*. The lack of a *JIT* compiler may lead to a high percentage of variation that is not explained by the chosen factors, for we did not include a factor *JIT* in our model. In our further analysis, we will assume that the corresponding JVMs are actually the same under different operating systems, which is not quite right if one of them lacks a *JIT* compiler. The analysis of variance will prove whether or not the *JIT* was really an important factor.

4.2 Preparing the Raw Data

For the analysis in a more mathematical approach, we only use the *fastest* results of the benchmarks. We could have taken the average between the fastest and the slowest results, but if there are outliers, this would lead to distorted values. We picked the fastest results since they are less likely to contain outliers—it is much easier for a benchmark to take *much more* time than previous replications (for example if the swapper or a background process like `cron` gets active) than to take *less*.

The table 17 presents the *fastest* results again in a single table.

For many calculations that follow, we'll take the mean value here and there. Since the results of the various workloads are not really comparable (they do different things), we often would rather use the *geometric* mean instead of the *arithmetic* mean when the workloads are

	— *Linux* —			— *Windows* —		
Workload	*IBM V1.1.8*	*SUN V1.1.8*	*SUN V1.2.2 RC3*	*IBM V1.1.8*	*SUN V1.1.8*	*SUN V1.2.2*
_227_mtrt	16.80	3.17	7.18	45.50	10.40	12.90
	17.50	3.14	7.25	45.50	11.50	13.10
	17.60	3.12	7.26	45.50	10.80	13.00
_202_jess	21.10	3.69	6.95	25.70	12.60	12.90
	21.00	3.71	6.63	26.40	13.10	12.60
	21.00	3.72	7.20	25.40	12.70	11.90
_201_compress	34.90	2.76	15.90	39.10	34.20	32.90
	35.10	2.76	15.90	38.80	34.30	32.80
	35.10	2.76	15.80	39.10	34.30	32.70
_209_db	9.83	2.29	3.61	13.60	5.21	6.83
	9.72	2.30	3.59	13.60	5.64	6.90
	9.72	2.29	3.62	13.50	5.34	6.95
_222_mpegaudio	51.20	3.14	19.50	55.50	44.50	43.80
	51.30	3.13	19.30	54.40	43.70	43.40
	50.80	3.13	19.40	52.90	44.80	44.40
_228_jack	28.30	3.32	6.83	9.20	6.41	6.77
	28.50	3.31	6.72	9.42	6.25	7.24
	28.10	3.30	6.64	9.13	6.51	7.19
_213_javac	9.56	2.98	4.16	19.10	5.10	6.62
	9.45	2.81	4.31	20.20	5.15	7.01
	9.37	3.02	4.45	18.70	4.69	6.88

Table 17: Raw Data

involved. However, the technique for calculating the effects (which follows) is not designed to be used with geometric mean values. Therefore, we don't use the geometric mean, but we take the logarithm from the results in order that they behave in a multiplicative way, similar to when using the geometric mean. The table 18 shows the results after taking the logarithm to a base of 10.

4.3 Means and Errors

The table 19 shows the same data, but for each of the three replication of each workload, the results have been averaged. Since the results do not seem to vary much with the replications, we've used the arithmetic mean here.

The overall mean is 1.048. The overall mean is simply the arithmetic mean over all the cells of the table 19.

The mean value itself is not very expressive without the variance, so we calculate the variance or the standard deviation, too; see the table 20. However, the *errors* will be slightly more useful for what we're going to calculate. The errors is simply the difference between the result of one of the replications (ie. a cell of the table 18) and the mean of the replications (ie. a cell of the table 19). The errors are presented in the table 21.

4.4 Calculating the Effects

We now want to know how much of the variance is caused by each of the factors, ie. how heavily does the performance depend on each of the factors. The *effects* express this.

First, we calculate the *main effect* ('primary effect') of each factor. It is calculated by taking the average of all the samples that fall under a given category of the factor of which the main effect is to be calculated; then we subtract the overall mean from it. For example, if we want to calculate the main effect of the category *Linux* of the factor *OS*, we take the average of all the cells in the three columns below the label *Linux* in the table 19 and subtract the overall mean from it. The tables 22, 23 and 24 show the main effects for all the categories of the factors *OS*, *JVM* and *Workload*, respectively.

OS	*Effect*
Linux	−0.153
Windows	0.153

TABLE 22: Main effects, factor 'OS'

JVM	*Effect*
IBM 1.1.8	0.310
SUN V1.1.8	−0.269
SUN V1.2.2	−0.041

TABLE 23: Main effects, factor 'JVM'

(Note that the main effects for all the categories of a given factor sum up to 0.)

Further, we are interested in the interactions between all combinations of two factors ('secondary effects'). We obtain these values in a similar way to how we obtained the values of the main effects: for two given categories of two factors, we take the average of all the cells in the table 19 that correspond to them, subtract the overall mean (as before), and then subtract the two effects of each factor–category combination. The tables 25, 26, and 27 show the interactions between the factors *OS* and *JVM*,

Workload	*Effect*
_227_mtrt	0.019
_202_jess	0.010
_201_compress	0.257
_209_db	−0.280
_222_mpegaudio	0.371
_228_jack	−0.145
_213_javac	−0.232

TABLE 24: Main effects, factor 'Workload'

	— *Linux* —			— *Windows* —		
Workload	*IBM V1.1.8*	*SUN V1.1.8*	*SUN V1.2.2 RC3*	*IBM V1.1.8*	*SUN V1.1.8*	*SUN V1.2.2*
_227_mtrt	1.225	0.501	0.856	1.658	1.017	1.111
	1.243	0.497	0.860	1.658	1.061	1.117
	1.246	0.494	0.861	1.658	1.033	1.114
_202_jess	1.324	0.567	0.842	1.410	1.100	1.111
	1.322	0.569	0.822	1.422	1.117	1.100
	1.322	0.571	0.857	1.405	1.104	1.076
_201_compress	1.543	0.441	1.201	1.592	1.534	1.517
	1.545	0.441	1.201	1.589	1.535	1.516
	1.545	0.441	1.199	1.592	1.535	1.515
_209_db	0.993	0.360	0.558	1.134	0.717	0.834
	0.988	0.362	0.555	1.134	0.751	0.839
	0.988	0.360	0.559	1.130	0.728	0.842
_222_mpegaudio	1.709	0.497	1.290	1.744	1.648	1.641
	1.710	0.496	1.286	1.736	1.640	1.637
	1.706	0.496	1.288	1.723	1.651	1.647
_228_jack	1.452	0.521	0.834	0.964	0.807	0.831
	1.455	0.520	0.827	0.974	0.796	0.860
	1.449	0.519	0.822	0.960	0.814	0.857
_213_javac	0.980	0.474	0.619	1.281	0.708	0.821
	0.975	0.449	0.634	1.305	0.712	0.846
	0.972	0.480	0.648	1.272	0.671	0.838

Table 18: log10 Data

	— *Linux* —			— *Windows* —		
Workload	*IBM V1.1.8*	*SUN V1.1.8*	*SUN V1.2.2 RC3*	*IBM V1.1.8*	*SUN V1.1.8*	*SUN V1.2.2*
_227_mtrt	1.238	0.497	0.859	1.658	1.037	1.114
_202_jess	1.323	0.569	0.840	1.412	1.107	1.096
_201_compress	1.544	0.441	1.200	1.591	1.535	1.516
_209_db	0.989	0.360	0.557	1.132	0.732	0.838
_222_mpegaudio	1.708	0.496	1.288	1.734	1.647	1.642
_228_jack	1.452	0.520	0.828	0.966	0.805	0.849
_213_javac	0.976	0.468	0.634	1.286	0.697	0.835

Table 19: Mean Data

	— Linux —			— Windows —		
Workload	*IBM V1.1.8*	*SUN V1.1.8*	*SUN V1.2.2 RC3*	*IBM V1.1.8*	*SUN V1.1.8*	*SUN V1.2.2*
_227_mtrt	0.011	0.003	0.003	0.000	0.022	0.003
_202_jess	0.001	0.002	0.018	0.009	0.009	0.018
_201_compress	0.001	0.000	0.002	0.002	0.001	0.001
_209_db	0.003	0.001	0.002	0.002	0.018	0.004
_222_mpegaudio	0.002	0.001	0.002	0.010	0.006	0.005
_228_jack	0.003	0.001	0.006	0.007	0.009	0.016
_213_javac	0.004	0.017	0.015	0.017	0.022	0.013

TABLE 20: Standard deviations of the logarithmized data

	— Linux —			— Windows —		
Workload	*IBM V1.1.8*	*SUN V1.1.8*	*SUN V1.2.2 RC3*	*IBM V1.1.8*	*SUN V1.1.8*	*SUN V1.2.2*
_227_mtrt	−0.013	0.004	−0.003	0.000	−0.020	−0.003
	0.005	0.000	0.001	0.000	0.024	0.003
	0.008	−0.003	0.002	0.000	−0.004	0.000
_202_jess	0.001	−0.002	0.002	−0.002	−0.007	0.015
	−0.001	0.000	−0.019	0.009	0.010	0.005
	−0.001	0.002	0.017	−0.007	−0.003	−0.020
_201_compress	−0.002	0.000	0.001	0.001	−0.001	0.001
	0.001	0.000	0.001	−0.002	0.000	0.000
	0.001	0.000	−0.002	0.001	0.000	−0.001
_209_db	0.003	−0.001	0.000	0.001	−0.015	−0.004
	−0.002	0.001	−0.002	0.001	0.019	0.000
	−0.002	−0.001	0.002	−0.002	−0.004	0.004
_222_mpegaudio	0.001	0.001	0.002	0.010	0.002	−0.001
	0.002	0.000	−0.002	0.001	−0.006	−0.005
	−0.003	0.000	0.000	−0.011	0.005	0.005
_228_jack	0.000	0.001	0.006	−0.002	0.001	−0.018
	0.003	0.000	−0.001	0.008	−0.010	0.011
	−0.003	−0.001	−0.006	−0.006	0.008	0.008
_213_javac	0.005	0.007	−0.015	−0.005	0.011	−0.014
	0.000	−0.019	0.001	0.019	0.015	0.011
	−0.004	0.012	0.014	−0.014	−0.026	0.003

TABLE 21: Errors

	Linux	*Win 98*
IBM 1.1.8	0.114	−0.114
SUN 1.1.8	−0.147	0.147
SUN 1.2.2	0.033	−0.033

TABLE 25: Interactions between the factors 'OS' and 'JVM'

	Linux	*Win 98*
_227_mtrt	−0.049	0.049
_202_jess	0.006	−0.006
_201_compress	−0.089	0.089
_209_db	0.021	−0.021
_222_mpegaudio	−0.102	0.102
_228_jack	0.183	−0.183
_213_javac	0.030	−0.030

TABLE 26: Interactions between the factors 'OS' and 'Workload'

OS and *Workload*, and *JVM* and *Workload*, respectively.

Finally, we are interested in the interaction between all three factors ('tertiary effects'). The calculation is analogous to the calculation of the main effect and secondary effects. The table 28 shows the results.

4.5 Confidence Intervals

In order to calculate the confidence intervals for the effects, we need to do an analysis of variance (ANOVA). We first calculate the sum of squares of our (logarithmized) data from table 18. Then we divide them each by their corresponding degrees of freedom to get the mean squares. The ratio of the mean squares of a factor to the mean squares of errors approximately has an F distribution. The F distribution has a parameter α, which represents the 'significance level'. We choose α as 0.05.

	IBM V1.1.8	*SUN V1.1.8*	*SUN V1.2.2*
_227_mtrt	0.071	−0.031	−0.040
_202_jess	0.000	0.049	−0.049
_201_cmpr.	−0.047	−0.048	0.095
_209_db	−0.017	0.047	−0.029
_222_mpga.	−0.008	−0.079	0.087
_228_jack	−0.004	0.028	−0.024
_213_javac	0.005	0.035	−0.040

TABLE 27: Interactions between the factors 'JVM' and 'Workload'

The table 29 shows the results of these calculations. The factors *OS*, *JVM* and *Workload* have been abbreviated as A, B and C, respectively. The column *F–Computed* shows the ratio of the mean squares of a factor to the mean squares of errors. The column *F–Table* shows the corresponding value in the table for the F distribution. It can easily be seen that the entries in the table are generally smaller than the computed values. This means that the factor is statistically significant. There are exceptions, however: the interactions BC (ie. between *JVM* and *Workload*) and ABC (ie. between all three factors) are too small to be statistically significant.

The table 29 also shows the *percentage of variation*, which explains the percentage with which a factor contributes to the total variation. The remaining percentage of variation—which is rather large in our case—cannot be explained by any of the chosen factors.

The figure 3 shows the percentage of variation as a pie chart.

Using the results of the analysis of variance, we now can calculate the confidence intervals for the effects and interactions. The table 30 shows the confidence intervals for the main effects, while the tables 31 and 32 show the confidence intervals for the interactions of each combination of two factors, and finally the table 34 shows the confidence intervals of the interactions between all three factors (sort of; we dropped the actual intervals in favour of simplicity and replaced them with signs indicating whether zero is included in the interval, or if

	— *Linux* —			— *Windows* —		
	IBM V1.1.8	*SUN V1.1.8*	*SUN V1.2.2 RC3*	*IBM V1.1.8*	*SUN V1.1.8*	*SUN V1.2.2*
_227_mtrt	−0.122	0.080	0.042	0.122	−0.080	−0.042
_202_jess	−0.012	0.025	−0.014	0.012	−0.025	0.014
_201_compress	0.105	−0.157	0.052	−0.105	0.157	−0.052
_209_db	−0.053	0.094	−0.041	0.053	−0.094	0.041
_222_mpegaudio	0.128	−0.173	0.045	−0.128	0.173	−0.045
_228_jack	0.099	−0.025	−0.074	−0.099	0.025	0.074
_213_javac	−0.146	0.156	−0.010	0.146	−0.156	0.010

TABLE 28: Interactions between all three factors 'OS', 'JVM' and 'Workload'

Component	Sum of Squares	%age of Variation	Degr. of Freedom	Mean Square	F–Computed	F–Table $\alpha = 0.05$
y	158.84	126				
$y_{..}$	132.06	1				
$y - y_{..}$	26.79	125				
A	2.96	11.06	1	2.96	39.18	$F[1-\alpha; 1, 84] = 4.06$
B	7.14	26.64	2	3.57	47.17	$F[1-\alpha; 2, 84] = 3.21$
C	6.43	24	6	1.07	14.17	$F[1-\alpha; 6, 84] = 2.32$
AB	1.5	5.61	2	0.75	9.94	$F[1-\alpha; 2, 84] = 3.21$
AC	1	3.74	6	0.17	2.21	$F[1-\alpha; 6, 84] = 2.32$
BC	0.28	1.06	12	0.02	0.31	$F[1-\alpha; 12, 84] = 1.98$
ABC	1.11	4.16	12	0.09	1.23	$F[1-\alpha; 12, 84] = 1.98$
e	6.35	23.72	84	0.08		

TABLE 29: ANOVA Table

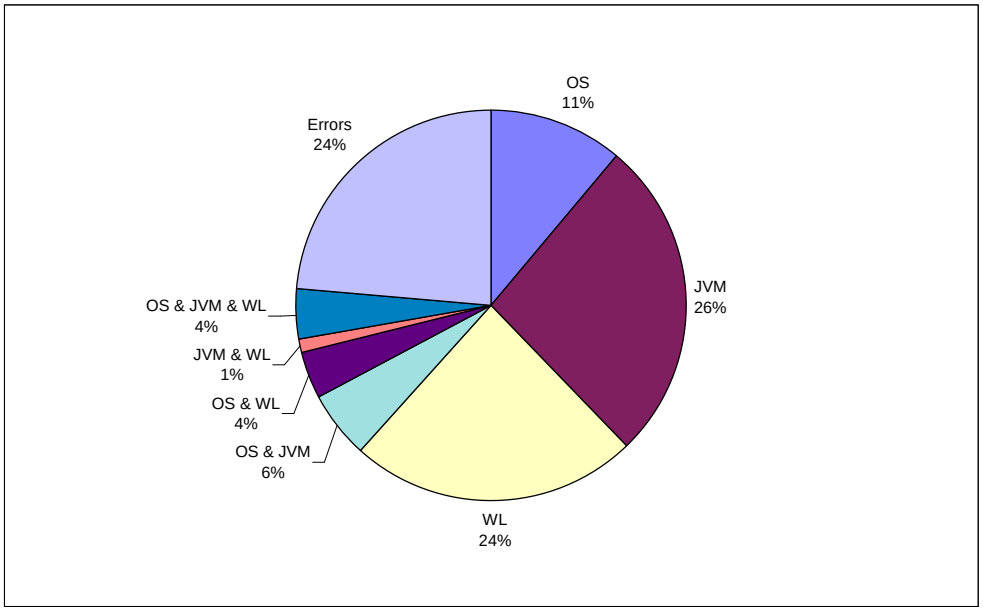

FIGURE 3: Percentage of variation. (The label 'WL' stands for 'Workload'.)

the intervals is positive, or negative.) The figure 4 shows the confidence interval for the main effects graphically. We will discuss these tables and figures later when we draw the conclusions.

4.6 Final Diagnostic Check

As a final diagnostic check, we plot a scatter diagram of residual versus predicted responses. If there is no visible trend in the magnitude or spread in the residuals, then it is verified that the assumption of independence of residuals and of their having an identical distribution holds. The figure 5 shows that this is true for our case.

5 Conclusions

The figure 4 shows quite clearly that the JVMs are generally running faster under *Windows* than under *Linux*. This may be explained by the fact that the JVMs are typically heavily optimized for *Windows*. Although the *SUN* JVMs are expected to be optimized for *Solaris*, either *Solaris* behaves rather different than *Linux* in these respects, or the JVM by *SUN* is not quite fast even under *Solaris*.

Looking at the confidence intervals of the JVMs, the two *SUN* JVMs seem to be slower than the *IBM* one. However, it must be pointed out that zero is included in the confidence interval for *SUN V1.2.2*, which means that we can't tell whether it is slower or faster than the average. All the same, *SUN V1.2.2* is faster than *SUN V1.1.8*, because the corresponding confidence intervals for these two JVMs do not overlap.

The confidence intervals of the factor 'Workload' are not that expressive.

Looking at the table 31, it can be seen that zero is included in all the confidence intervals except the one for the workload `_228_jack` are not significant. This should be so if the workloads are properly chosen. The workload `_228_jack` being much faster under *Linux* than under *Windows* is rather surprising. If we check table 34, we can see that this phenomenon is mainly caused by the *IBM* virtual

[a] Not significant.

Parameter	*Mean Effect*	*Standard Deviation*	*Confidence Interval*
μ	1.048	0.025	(0.999, 1.097)
Operating Systems			
Linux	−0.153	0.025	(−0.202, −0.105)
Windows	0.153	0.025	(0.105, 0.202)
JVMs			
IBM 1.1.8	0.310	0.035	(0.241, 0.379)
SUN 1.1.8	−0.269	0.035	(−0.338, −0.200)
SUN 1.2.2	−0.041	0.035	(−0.110, 0.028)[a]
Workloads			
`_227_mtrt`	0.019	0.060	(−0.100, 0.139)[a]
`_202_jess`	0.010	0.060	(−0.110, 0.129)[a]
`_201_compress`	0.257	0.060	(0.137, 0.376)
`_209_db`	−0.280	0.060	(−0.399, −0.160)
`_222_mpegaudio`	0.371	0.060	(0.252, 0.491)
`_228_jack`	−0.145	0.060	(−0.264, −0.025)
`_213_javac`	−0.232	0.060	(−0.352, −0.113)

TABLE 30: Confidence Intervals for the Effects

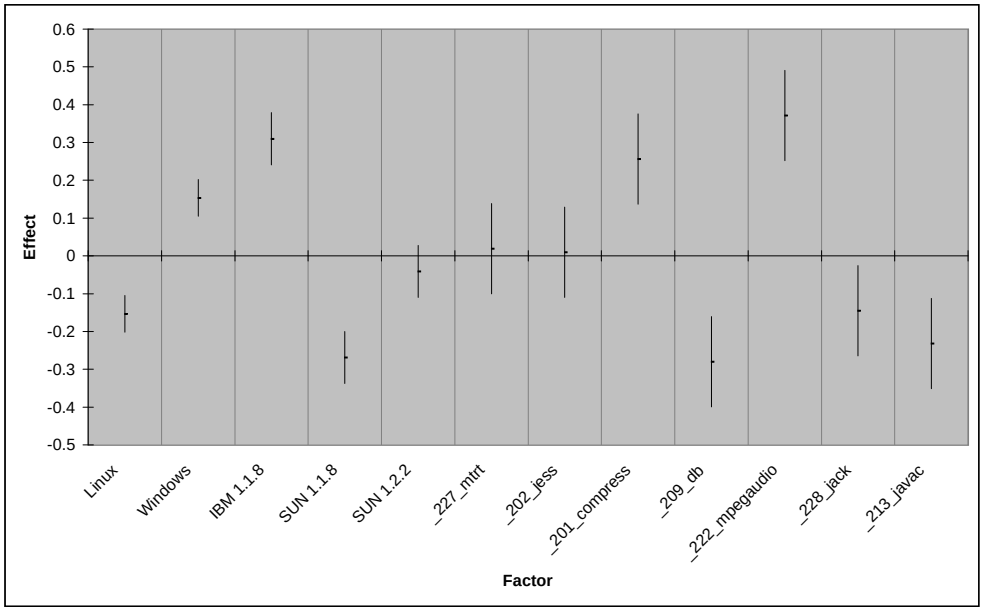

FIGURE 4: Confidence Intervals for the Effects

[a] Not significant.

	Linux	*Windows*
_227_mtrt	$(-0.168, 0.070)^a$	$(-0.070, 0.168)^a$
_202_jess	$(-0.113, 0.126)^a$	$(-0.126, 0.113)^a$
_201_compress	$(-0.209, 0.030)^a$	$(-0.030, 0.209)^a$
_209_db	$(-0.099, 0.140)^a$	$(-0.140, 0.099)^a$
_222_mpegaudio	$(-0.221, 0.018)^a$	$(-0.018, 0.221)^a$
_228_jack	$(0.064, 0.303)$	$(-0.303, -0.064)$
_213_javac	$(-0.089, 0.149)^a$	$(-0.149, 0.089)^a$

TABLE 31: Confidence Intervals for the Interaction between the factors 'OS' and 'Workload'

[a] Not significant.

	IBM 1.1.8	*SUN 1.1.8*	*SUN 1.2.2*
_227_mtrt	$(-0.098, 0.240)^a$	$(-0.200, 0.138)^a$	$(-0.208, 0.129)^a$
_202_jess	$(-0.169, 0.169)^a$	$(-0.120, 0.218)^a$	$(-0.218, 0.120)^a$
_201_compress	$(-0.216, 0.122)^a$	$(-0.217, 0.121)^a$	$(-0.074, 0.264)^a$
_209_db	$(-0.186, 0.152)^a$	$(-0.122, 0.216)^a$	$(-0.198, 0.140)^a$
_222_mpegaudio	$(-0.177, 0.161)^a$	$(-0.248, 0.090)^a$	$(-0.082, 0.256)^a$
_228_jack	$(-0.173, 0.165)^a$	$(-0.141, 0.197)^a$	$(-0.193, 0.145)^a$
_213_javac	$(-0.164, 0.174)^a$	$(-0.134, 0.204)^a$	$(-0.209, 0.129)^a$

TABLE 32: Confidence Intervals for the Interaction between the factors 'JVM' and 'Workload'

[a] Not significant.

	Linux	*Windows*
IBM V1.1.8	$(0.045, 0.183)$	$(-0.183, -0.045)$
SUN V1.1.8	$(-0.216, -0.078)$	$(0.078, 0.216)$
SUN V1.2.2	$(-0.036, 0.102)^a$	$(-0.102, 0.036)^a$

TABLE 33: Confidence Intervals for the Interaction between the factors 'OS' and 'JVM'

[a] Not significant.

	— *Linux* —			— *Windows* —		
	IBM V1.1.8	*SUN V1.1.8*	*SUN V1.2.2*	*IBM V1.1.8*	*SUN V1.1.8*	*SUN V1.2.2*
_227_mtrt	a	a	a	a	a	a
_202_jess	a	a	a	a	a	a
_201_compress	a	a	a	a	a	a
_209_db	a	a	a	a	a	a
_222_mpegaudio	a	$-$	a	a	$+$	a
_228_jack	a	a	a	a	a	a
_213_javac	a	a	a	a	a	a

TABLE 34: Confidence Intervals for the Interaction between all three factors, showing only if the interval is positive, negative, or not significant.

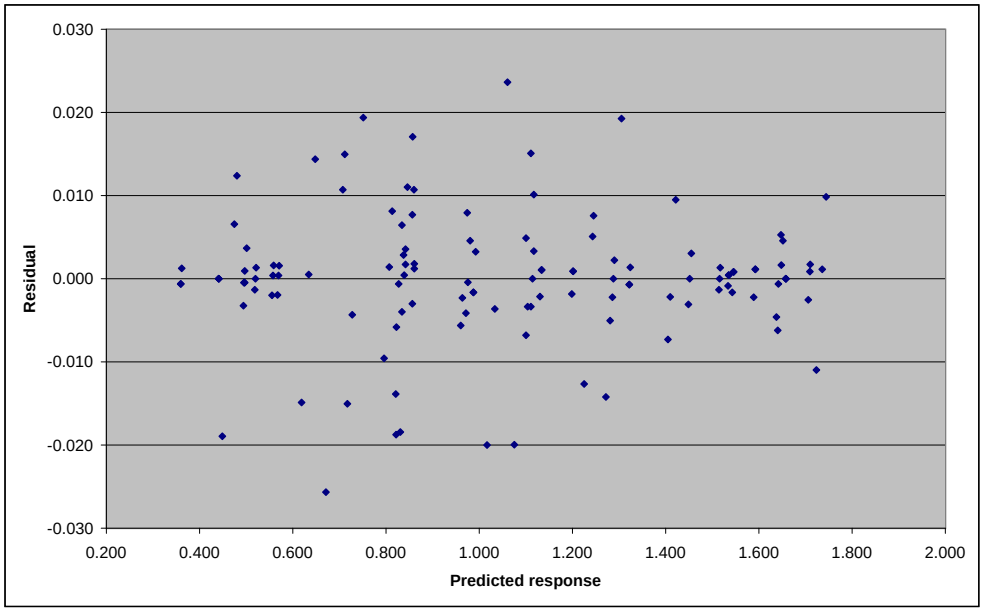

FIGURE 5: Plot of the residuals versus predicted response

machine. It may be *IBM* found some special way of optimization under *Linux*, which causes the parser generator benchmark `_228_jack` to execute much more efficiently...

Looking at the table 33 and at the figure 2, the JVM by *IBM* seems to excel the other JVMs under *Linux* higher than it excels those under *Windows*—although the *IBM* JVM is actually slower under *Linux* than under *Windows*, because of the big difference between *Linux* and *Windows*.

Finally, looking at the figure 3, we see that the foreseen problem that the existence of a *JIT* compiler could be a significant factor has really arisen: the percentage of variation caused by other factors than the selected ones is as high as 24 percent! (It may also be that there are even more factors that have been ignored; maybe the fact that the same byte code is used under all JVMs has also had influence on the results.)

Contents

List of Figures

List of Tables