Faster than compiled Java?

Pedro G. Gonnet (pedro@vis.ethz.ch)

April 25, 2001

Abstract

For quite some time now, makers of Java Virtual Machines (JVMs) based on Just-In-Time (JIT) and Ahead-Of-Time (AOT) compilers with on-the-fly optimizations have been flaunting their Products as running Java programs "faster than compiled c++", well knowing that nobody was going to write a complete test-suite for both languages.

Fortunately enough, the Open Source community has been working on a Java front-end to the popular Gnu Compiler Collection (GCC) and it's own version of the Java Application Programming Interface (API) for compiling Java programs to native machine code, using all the optimizations available to gcc.

This report tries to compare four JVM-type packages: Sun Microsystems' JDK 1.3 [?], IBM Research Laboratories JDK 1.3 [?], the Open Source Kaffe Java Runtime Environment (JRE) [?] and gcj, the Gnu Java Compiler [?].

Instead of using commercially available benchmarking suites for which the compilers could have been optimized, a new suite of test programs – almost all with freely available sources – has been comprised, based loosely on the Standard Performance Evaluation Corporation's SPEC JVM98 benchmarking package.

Chapter 1

Introduction

The Java Programming Language first appeared in late 1995, it was not presented so much as a language, but a whole new concept in computing. Programs written in Java could only be compiled to a platform independent byte-code (class-file) and could only be executed on a platform-specific virtual machine.

Since the purpose of the Java Programming Concept (JPC) was platform independence, a whole set of libraries – the Java API – was written and distributed with the development kits and runtime environments.

1.1 Performance Problems and JIT-Compilation

Although platform independence was probably what made the language so immensely popular in its beginnings, it had one major drawback: byte-code interpretation was too slow.

In an effort to counter this problem, a concept shift was needed: the Java byte-codes passed from being the final product of compilation to a mere intermediate representation of the Java program. The virtual machine then passed on to be a back-end compiler, producing native code on the fly from the byte-codes to be executed.

Since compilation is a rather time-consuming exercise, only those parts of a program were compiled that were actually needed, when they were needed, hence the term Just-in-Time compilation.

1.2 On-the-Fly Optimizations

As the engineers behind the compilers started getting bolder, the concept of on-the-fly optimizations was added. If the runtime environment noticed that one segment of the program was being called repeatedly, it would be recompiled, paying more attention and investing more time in optimizations.

Now, unfortunately, the engineers weren't the only ones getting bolder: due to the impressive financial effort behind the use of Java, the marketing departments of the companies involved started to flaunt the advantages of their JIT-compilers and Java in general in increasingly aggressive terms. This was the birth of the "faster than compiled c++" prediction for JIT compiler technology. This slogan was quite easy to pass off uncontestedly, since nobody in their right mind would write an extensive test suite in both languages (c++ and Java).

1.3 Performance Compared to Natively Compiled Programs

Although Sun Microsystems and others have long since backed off from this claim, it is still interesting to know where the JIT-compilation of Java programs stand performance-wise. To this effect we shall refrain from comparing them to native compiled c++. We now have a much more powerful means of comparison: native compiled Java.

Since 1998, an Open Source movement has been active writing a Java front-end for the immensely popular Gnu Compiler Collection (GCC). In an effort to gain complete independence from Sun Microsystems, the Java API was also rewritten. The result is a native Java compiler which can produce binaries for all the platforms supported by GCC.

Although the compiler (gcj) and the associated API (libgcj) are still a few steps away from full maturity, it is possible to compile and run large and complex Java programs, therefore making a performance comparison possible.

Chapter 2

Java Runtime Environments

2.1 Sun Java 2 SDK, Enterprise Edition 1.3 Beta Release

Sun Microsystems, the inventor of Java, is putting considerable effort into the marketing of their brainchild. Therefore it is not surprising that their compiler and runtime environment are amongst the most popular. The compiler and runtime used are the ones distributed with the beta release of the Java 2 Software Development Kit (SDK), Enterprise Edition 1.3.

2.1.1 The `javac` Java Compiler

This is the program that generates .class-files form the Java source-code. Although the compiler performs some sophisticated flow analysis (i.e. for variable initialization and return value checks), no optimizations are performed, resulting in some pretty thick byte-code.

The compiler does have a `-O` option for optimizations, this however does nothing as can be read in the `javac`-documentation [?]. This approach underlines the shift of the byte-codes from an executable to a mere intermediate representation.

2.1.2 The HotSpot VM

According to the Java HotSpot VM whitepaper [?], a program is executed by the following steps:

- **Hot Spot Detection:** The code is loaded and, in a first step, is only interpreted. This allows the VM to detect critical "hot spots" and pass them to the global native-code optimizer. This monitoring is continued throughout the lifespan of the running program.

- **Hot Spot Compilation:** Once a "hot spot" is identified as such, it is compiled by a fully optimizing compiler. This compiler performs all the classic optimizations: i.e. dead code elimination, loop invariant hoisting, common subexpression elimination and constant propagation. It also inlines virtual method calls where it can. For all optimizations, runtime information gathered in the detection phase is used.

Other features touted by the whitepaper are improved memory allocation and garbage collection and better thread synchronization.

2.1.3 The Client and Server VMs

The Java HotSpot VM can be invoked in two flavours, Client and Server. The difference between these is not quite clear – or very well documented [?, ?]. The main difference appears to be the optimization level when compiling "hot spots" to native code.

For this report, both modes of operation were tested, yielding significant performance differences.

2.2 IBM Developer Kit for Linux, Java 2 Technology Edition, Version 1.3

As with the Sun JDK, the `javac` compiler does not try to perform any optimizations - even if the `-O` flag is on. The byte-codes produced are identical to those produced by the Sun `javac`, leading to assume that they are one and the same.

2.2.1 The IBM Java Just-in-Time Compiler

The IBM JIT compiler follows pretty much the same strategy as it's Sun counterpart: the byte-code is interpreted at first and once a method has been called sufficient times (methods with loops also count loop iterations) it is passed to the optimizing compiler.

As in the Sun HotSpot compiler, profile information resulting from code interpretation is used to guide the optimizations when compiling. As an extra bonus, the IBM JIT compiler also tries to apply optimizations best suited for the underlying processor, i.e. preferring memory operand instructions on a PentiumPro-family processor to exploit its out-of-order execution.

Alongside the standard optimizations, the IBM JIT compiler also does the following:

- **Method inlining:** Recursive calls to sparse or empty methods are inlined in the caller. This is often the case for constructors and simple variable access methods.

- **Exception Check Elimination:** Flow analysis is performed to try to avoid array index bound checks and type checks. NullPointerExceptions are handled by the native trap mechanisms.

It should furthermore be noted that the IBM JIT compiler is specifically geared for compilation speed [?], i.e. using other, simpler algorithms than graph-colouring for register allocation. IBM also includes its own version of the Java API, replacing most classes, notably the `java.util` and `java.lang` packages.

2.3 Kaffe OpenVM 1.0.6

Kaffe is probably the most popular open source Java environment. The project was started in early 1996 by Transvirtual (http://www.transvirtual.com) with the goal of producing an extensible, multi-platform Java virtual machine with JIT compilation capabilities.

2.3.1 The kjc Java Compiler

The Java compiler included with Kaffe is `kjc`[?], an extremely pedantic Java 1.1 compiler written in the Java programming language itself. In contrast to the Sun and IBM compilers, kjc actually tries optimize the given Java program if told to do so with the `-O<int>` option.

2.3.2 The JIT3 Kaffe Compiler

The JIT compiler in the Kaffe OpenVM works in the following steps:

- **Load Class file:** The byte-code is loaded and converted to an internal format called `icode`[?].
- **Initialize Trampoline Functions**[?]**:** A method table is set up for every class. The table entries are initialized to the JIT compilation function.
- **Method Invocation:** When a method is invoked, control is transfered to the function in the method table. If this is the JIT compilation function, the method is compiled int native code and the entry in the method table changed to the later.

As can be seen, every method is compiled, even if it is invoked only once. The JIT compiler is not especially geared for speed, but for portability – hence the use of the `icode` intermediate. The platforms supported by the JIT3-compiler include i386, SPARC, ARM, MIPS, m68k, Alpha and ia64.

Kaffe also has its own version of the Java API, including support for the Active Windowing Toolkit (AWT).

2.4 The GNU Compiler for the Java Programming Language

In September 1998 work started to produce a Java front-end to the Gnu Compiler Collection (GCC)[?]. To this date, the compiler – `gcj` – can pretty much chew through any Java program given to it.

The compiler translates the Java source or class-file to a GCC intermediate and passes it on to the back-end. Since the GCC supports virtually every processor that was ever built, this means you can port Java to almost every platform around – provided it has a posix-compatible operating system.

The binaries built by the Gnu Compiler for Java (`gcj`) link to the `libgcj` library – an implementation of the Java API. The API has almost complete support for Java 1.2, a noticeable exception being however Swing and many AWT classes.

The compiled binaries run just as any other program would. You do not need to recompile them every time you want to run.

Chapter 3

Benchmark Programs

The biggest question the astute reader will be asking himself/herself right now is probably “why wasn’t a standard benchmarking suite used in the comparison?” This has a few very good reasons:

- **Availability:** Most commercial benchmarks are rather expensive.
- **Program Size:** Most benchmarks – SPEC JVM98[?] being the notable exception – base their benchmarks on small, highly repetitive programs that do not usually reflect the software we use on a daily basis.
- **Special Tuning:** SPEC JVM98 has been around since 1998. This has given compiler constructors more than enough time to tune their optimizations to the programs used therein.

The programs used for this report were chosen to reflect the selection used in the SPEC JVM98 benchmark suite. All programs, with the exception of InstantDB and javac are available as Java source-code. All programs are also freely available.

3.1 The Mauve Project Test Suite

name	The Mauve Project Test Suite
version	CVS Snapshot March 14th
input	none
runs	1
url	http://sources.redhat.com/mauve/

The Mauve Project is a collaborative effort to write a free test suite for the Java Class libraries. This test serves two purposes:

- Check conformance of the JREs and their respective APIs to the Java standard.

- Test the efficiency of the API implementations.

The test consist of 137 test sets that are run once each. This test replaces the *_200_check* in SPEC JVM98, which is not, however, part of the performance metric. For this report it will be evaluated as to compare the implementations of the various APIs.

3.2 BZip2 for Java

name	Aftex Software BZip for Java
version	0.2.2
input	mauve-snapshot-2001-03-18.tar
runs	1
url	http://www.aftexsw.com/aftex/products/java/bzip/

This test program is a Java implementation of the popular `bzip2` compression tool. The original sources were altered slightly to remove the graphical user interface so that GCJ could also compile it without warnings.

The input file used was a snapshot of the Mauve Project Test Suit. Its original size is 1'914'880 bytes, compressed it is only 221'420.

This Test replaces the *_201_compress* benchmark in SPEC JVM98.

3.3 Knights Tour in jProlog

name	jProlog
version	0.1
input	Knight.pl (5x5 knights tour)
runs	1
url	http://www.cs.kuleuven.ac.be/ bmd/PrologInJava/

JProlog is a small prolog interpreter written in Java. The program only accepts prolog queries, facts are converted from prolog to java classes by means of the prolog program `comp`.

The knights tour program is relatively simple:

```
main :- statistics(runtime,_),
        write('started looking for tour on 5x5 board...'), nl,
        knight5(Tour), write(Tour), nl,
        statistics(runtime, [_,Time]),
        write('CPU time = '), write(Time), write(' msec'), nl,
        halt.

knight5(Tour) :-
        Vs = [(1,1), (1,2), (1,3), (1,4), (1,5),
              (2,1), (2,2), (2,3), (2,4), (2,5),
              (3,1), (3,2), (3,3), (3,4), (3,5),
              (4,1), (4,2), (4,3), (4,4), (4,5),
              (5,1), (5,2), (5,3), (5,4), (5,5)],
        tour(Vs, Tour).

tour(Vs0, Tour) :-
        select((1,1), Vs0, Vs),
        tour(1, 1, Vs, Tour).

tour(I, J, [], [(I,J)]).
tour(I, J, Vs0, [(I,J)|Tour]) :-
        next(I, J, I1, J1),
        select((I1,J1), Vs0, Vs),
        tour(I1, J1, Vs, Tour).

next(I, J, I1, J1) :- I1 is I-2, J1 is J-1.
next(I, J, I1, J1) :- I1 is I-2, J1 is J+1.
next(I, J, I1, J1) :- I1 is I-1, J1 is J-2.
next(I, J, I1, J1) :- I1 is I-1, J1 is J+2.
next(I, J, I1, J1) :- I1 is I+1, J1 is J-2.
next(I, J, I1, J1) :- I1 is I+1, J1 is J+2.
next(I, J, I1, J1) :- I1 is I+2, J1 is J-1.
next(I, J, I1, J1) :- I1 is I+2, J1 is J+1.

select(X, [X|Zs], Zs).
select(X, [Y|Ys], [Y|Zs]) :- select(X, Ys, Zs).
```

The program is started by executing the query `main`.

This test replaces the *_202_jess* test in SPEC JVM98. Solving a knight's tour problem doesn't seem to compare to well to an expert system at first sight, but both are actually highly repetitive, recursive programs using brainless heuristics to solve a problem.

3.4 InstantDB version 3.26

name	InstantDB
version	3.26
input	people.data
runs	10
url	http://instantdb.enhydra.org

InstantDB is a relational database management system implemented in Java. It has several tools to parse SQL commands, either in the form of a script (`ScriptTool`) or as commands at a prompt (`commsql`).

The database used was taken from the SPEC JVM98 example and consists of a list of 5000 addresses. The queries used for the test are the following:

- `drop table people`
- `import people from "../people.data" using "../people.schema"`
- `select * from people order by last_name`
- `select * from people where last_name like "%e" order by first_name`
- `select * from people where last_name like "%ee%" order by city`
- `select * from people where city in (select city from people where last_name like "Mc%") order by pcode`

This sequence of commands is run 10 times.

3.5 Javac Java Compiler

name	javac
version	Sun Java 2 SDK
input	mauve sources
runs	1
url	http://java.sun.com

This is probably the largest test program in the series. It takes the class-files of Sun's Java Compiler `javac` (probably the most-used Java program ever) and executes them with the mauve sources as input. The overall speed of the compilation is measured.

This test is equivalent to *_213_javac* in SPEC JVM98 and differs only in the version number of the compiler (SPEC JVM98 uses version 1.0.2) and the source-files, which are more diverse.

3.6 JavaZoom MP3-Decoder

name	JavaZoom MP3-Decoder
version	0.0.8
input	input.mp3 (8'578'322 bytes, 44.1 kHz at 256 kbit/s)
runs	1
url	http://www.javazoom.net/javalayer/javalayer.html

The JavaZoom MP3-Decoder takes an ISO MPEG Layer-3 audio file and converts it to the WAV format. This test is almost identical to *_222_mpegaudio* in SPEC JVM98, differing only in the program used for decoding and the input file.

The input file is a live recording of the song ”Idioteque” by the group “Radiohead” on the 24th of September 2000 at Victoria Park, London.

3.7 LRay – a Java Raytracer

name	LRay
version	none
input	none
runs	1
url	http://www.l-shaped.freeserve.co.uk/computing/lray/

LRay is a small ray-tracing applet that renders an anti-aliased 320x240 scene consisting of textured spheres and a mirroring, textured plane. The source was adapted as to be able to run as a stand-alone application that does not display the image but renders it into a buffer.

This test corresponds to *_227_mtrt* in SPEC JVM98.

3.8 the JavaCup Parser Generator

name	CUP Parser Generator for Java
version	0.10j
input	java11.cup
runs	1
url	http://www.cs.princeton.edu/ãppel/modern/java/CUP/

The CUP Parser Generator for Java is a popular yacc-based LALR parser generator written in and for the Java language.

This test corresponds to *_228_jack* in SPEC JVM98. The input used in the test is the grammar for the Java programming language itself.

Chapter 4

Results

All tests ran more or less successfully. They were all timed with the UNIX `time` command. Each test was performed 10 times and the average times (user, system and clock) registered. All input and output was to a local file-system.

The machine used for all tests had the following setup:

- **Processor:** PentiumIII 500MHz
- **Main Memory:** 128MB RAM
- **Disk Subsystem:**
- **Operating System:** GNU Linux 2.2.12-20

4.1 The Mauve Project Test Suite

program	compilation commands
gcj	gcc -O4 -c -g -I../mauve ../mauve/gnu/testlet/.../*.java gcj –main=gnu.testlet.SimpleTestHarness *.o -o mauve
kaffe	javac -O4 -d ../kaffe @files
ibm	javac -d ../ibm @files
sun	javac -d ../sun @files

Since too many errors were encountered during the build (even with mauve's own makefiles) with all compilers, this test was abandoned.

4.2 BZip2 for Java

program	compilation commands
gcj	gcj -O2 -g -c -I../src ../src/com/aftexsw/util/bzip/*.java gcj –main=com.aftexsw.util.bzip.Main *.o -o bzip2
kaffe	javac -d . ../src/com/aftexsw/util/bzip/*.java
ibm	javac -d . ../src/com/aftexsw/util/bzip/*.java
sun	javac -d . ../src/com/aftexsw/util/bzip/*.java

The Kaffe compiler `kjc` was unable to compile the bzip sources due to an internal exception, so Sun's `javac` was used instead.

program	execution command
gcj	time ./bzip2 ../mauve-snapshot-2001-03-20.tar
kaffe	time java com.aftexsw.util.bzip.Main ../mauve-snapshot-2001-03-20.tar
ibm	time java com.aftexsw.util.bzip.Main ../mauve-snapshot-2001-03-20.tar
sun client	time java -client com.aftexsw.util.bzip.Main ../mauve-snapshot-2001-03-20.tar
sun server	time java -server com.aftexsw.util.bzip.Main ../mauve-snapshot-2001-03-20.tar

All programs ran without major problems and produced identical output.

program	user	system	user + system	wall-clock
gcj	10.832	0.152	10.984	10.991
kaffe	24.563	0.140	24.703	24.713
ibm	7.850	0.245	8.095	8.109
sun client	8.130	0.208	8.338	8.767
sun server	8.887	0.231	9.118	13.342

The clear winner in this test is the IBM JIT compiler. Since the Kaffe OpenVM performs almost no optimizations, it was not expected to do to well in this highly repetitive code.

4.3 Knight's Tour in JProlog

program	compilation commands
gcj	gcj -O4 -g -c -I../src ../src/*.java gcj –main=Prolog *.o -o jprolog
kaffe	javac -O4 -d . ../src/*.java
ibm	javac -d . ../src/*.java
sun	javac -d . ../src/*.java

program	execution command
gcj	time ./jprolog ¡ ../src/run.pl
kaffe	time java Prolog ¡ ../src/run.pl
ibm	time java Prolog ¡ ../src/run.pl
sun client	time java -client Prolog ¡ ../src/run.pl
sun server	time java -server Prolog ¡ ../src/run.pl

program	user	system	user + system	wall-clock
gcj				11.554
kaffe	50.355	0.057	50.412	50.522
ibm	13.450	0.151	13.601	13.617
sun client	6.565	0.130	6.695	7.412
sun server	7.344	0.129	7.453	10.819

The user and system times for the gcj-program could not be evaluated due to the use of lightweight threads on Linux.

Here, the client version of the Sun HotSpot VM is the fastest. This is probably due, given the highly recursive nature of the test program, to a better inlining strategy and method call optimizations.

4.4 InstantDB version 3.26

program	compilation commands
gcj	gcj -O4 -g -c -I../Classes ../Classes/org/enhydra/instantdb/db/*.class gcj -g -c -I../Classes ../Classes/org/enhydra/instantdb/db/Search.class gcj -g -c -I../Classes ../Classes/org/enhydra/instantdb/db/expression.class gcj -O4 -g -c -I../Classes ../Classes/org/enhydra/instantdb/jdbc/*.class gcj -O4 -g -c -I../Classes ../Classes/javax/transaction/xa/*.class gcj -O4 -g -c -I../Classes ../Classes/org/enhydra/instantdb/ScriptTool.class gcj -O4 -g -c -I../Classes ../Classes/org/enhydra/instantdb/SampleThread.class gcj -O4 -g -c -I../Classes ../Classes/org/enhydra/instantdb/TestObject.class gcj –main=org.enhydra.instantdb.ScriptTool *.o -o ScriptTool
kaffe	nothing
ibm	nothing
sun	nothing

Since only class-files were available, these were given to gcj for compilation. The other programs didn't have to compile anything. Due to a compiler error, the files `Search.class` and `expression.class` could only be compiled without optimizations.

program	execution command
gcj	time ./ScriptTool test.sql > test.out
kaffe	time java org.enhydra.instantdb.ScriptTool test.sql > test.out
ibm	time java org.enhydra.instantdb.ScriptTool test.sql > test.out
sun client	time java -client org.enhydra.instantdb.ScriptTool test.sql > test.out
sun server	time java -server org.enhydra.instantdb.ScriptTool test.sql > test.out

All programs ran without major problems, except for the Kaffe OpenVM which left the database in an inconsistent state every other run. 20 runs were made and only the "clean" ones were counted.

program	user	system	user + system	wall-clock
gcj	93.365	2.077	95.442	99.535
kaffe	100.741	2.204	102.945	107.059
ibm				37.171
sun client	24.262	2.174	26.436	39.525
sun server	27.008	2.245	29.253	55.298

Once again, the user and system times did not make much sense, since the program is multi-threaded. What is noticeable here is that gcj is clearly beaten by the Sun and IBM VMs. This is probably due to the two crucial class-files that could not be compiled with optimizations in gcj.

4.5 Javac Java Compiler

program	compilation commands
gcj	gcj -O4 -c -I../src ../src/sun/tools/java/*.class gcj -O4 -c -I../src ../src/sun/tools/tree/*.class gcj -O4 -c -I../src ../src/sun/tools/asm/*.class gcj -O4 -c -I../src ../src/sun/tools/util/*.class gcj -O4 -c -I../src ../src/sun/tools/javac/*.class gcj -O4 -c -I../src ../src/sun/io/MalformedInputException.class gcj -c -I../src ../src/sun/tools/java/Parser.class gcj -c -I../src ../src/sun/tools/java/Scanner.class gcj -c -I../src ../src/sun/tools/asm/Instruction.class gcj –main=sun.tools.javac.Main -o javac *.o
kaffe	nothing
ibm	nothing
sun	nothing

As with the InstantDB-test, gcj could not compile the class-files `Parser.class`, `Scanner.class` and `Instruction.class` with optimizations, the first two being rather crucial in a compiler.

program	execution command
gcj	time ../gcj/javac -d ../gcj -classpath $CLASSPATH:../gcj @files > javac.out
kaffe	time java -classpath $CLASSPATH:../src sun.tools. javac.Main -d ../kaffe -classpath Klasses.jar:. @files > javac.out
ibm	time java -classpath $CLASSPATH:../src sun.tools. javac.Main -d ../ibm -classpath Klasses.jar:. @files > javac.out
sun client	time java -client -classpath $CLASSPATH:../src sun.tools. javac.Main -d ../sun -classpath Klasses.jar:. @files > javac.out
sun server	time java -server -classpath $CLASSPATH:../src sun.tools. javac.Main -d ../sun -classpath Klasses.jar:. @files > javac.out

program	user	system	user + system	wall-clock
gcj				16.485
kaffe	27.348	0.174	27.522	27.545
ibm	18.266	0.321	18.587	18.618
sun client	6.690	0.223	6.913	11.320
sun server	15.738	0.238	15.976	33.818

In this test, the Sun HotSpot VM is both first and last, lagging, in server mode, even behind the unoptimized Kaffe OpenVM. As with the InstantDB test, it should be noted that the gcj version was using an unoptimized parser and scanner, which very probably had a strong influence against it.

4.6 JavaZoom MP3-Decoder

program	compilation commands
gcj	gcj -O4 -c -I../src ../src/javazoom/jl/decoder/*.java gcj -O4 -c -I../src ../src/javazoom/jl/converter/*.java gcj –main=javazoom.jl.converter.jlc -o jlc *.o
kaffe	javac -O4 -d . ../JavaLayer0.0.8/src/javazoom/jl/decoder/*.java javac -O4 -d . ../JavaLayer0.0.8/src/javazoom/jl/converter/*.java
ibm	javac -d . ../JavaLayer0.0.8/src/javazoom/jl/decoder/*.java javac -d . ../JavaLayer0.0.8/src/javazoom/jl/converter/*.java
sun	javac -d . ../JavaLayer0.0.8/src/javazoom/jl/decoder/*.java javac -d . ../JavaLayer0.0.8/src/javazoom/jl/converter/*.java

program	execution command
gcj	time ./jlc ../input.mp3 -p ../output.wav > jlc.out
kaffe	time java javazoom.jl.converter.jlc ../input.mp3 -p ../output.wav > jlc.out
ibm	time java javazoom.jl.converter.jlc ../input.mp3 -p ../output.wav > jlc.out
sun client	time java -client javazoom.jl.converter.jlc ../input.mp3 -p ../output.wav > jlc.out
sun server	time java -server javazoom.jl.converter.jlc ../input.mp3 -p ../output.wav > jlc.out

program	user	system	user + system	wall-clock
gcj				58.928
kaffe	126.927	0.694	127.621	127.661
ibm	55.599	0.965	56.564	56.765
sun client	68.942	0.809	69.751	70.709
sun server	69.396	0.815	70.221	87.022

This is the second-longest-running test in this report. Astonishingly enough, only the IBM JVM was able to beat gcj and the Sun HotSpot JVM in client-mode still out-did its server counterpart.

Since the compilation on Sun's HotSpot JVM occurs in a separate thread and is not registered as user/system time, we can compare the client and server modes via the user + system time. Here we see that the server mode does not outperform the client mode on this test.

4.7 LRay – a Java Raytracer

program	compilation commands
gcj	gcj -O4 -g -c -I.. ../*.java gcj –main=Main *.o -o lray
kaffe	javac -O4 -d . ../*.java
ibm	javac -d . ../*.java
sun	javac -d . ../*.java

The LRay sources had to be adjusted from an applet to a stand-alone, command-line application. The 320x240 scene is rendered into an `int` buffer.

program	execution command
gcj	time ./lray 1
kaffe	time java Main 1
ibm	time java Main 1
sun client	time java -client Main 1
sun server	time java -server Main 1

program	user	system	user + system	wall-clock
gcj	66.569	0.170	66.739	66.786
kaffe	565.934	0.074	566.008	566.041
ibm	109.515	0.374	109.889	109.914
sun client	86.830	0.733	87.563	87.838
sun server	98.277	0.752	99.029	100.246

In this numerically intensive test, gcj clearly beats all of the VMs. This is probably due to the fact that the code consists of only a few classes with very few library calls that could profit from inlining. The extremely bad results for the Kaffe OpenVM are a good indication that it makes little or no effort to optimize floating-point operations.

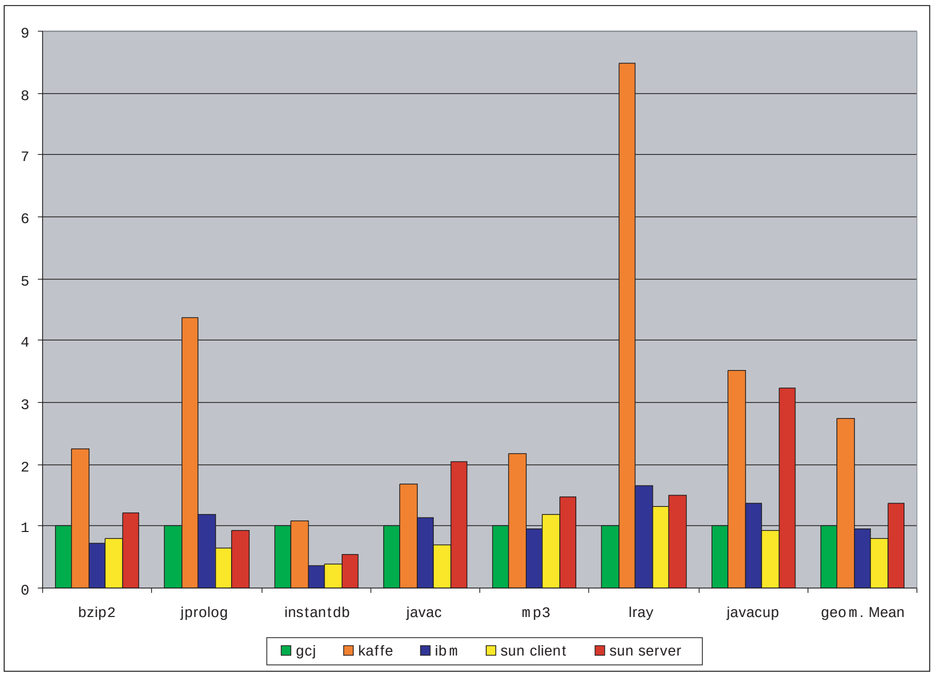
4.8 The JavaCup Parser Generator

program	compilation commands
gcj	gcj -O4 -g -c -I../src ../src/java_cup/*.java gcj -O4 -g -c -I../src ../src/java_cup/runtime/*.java gcj -g -c -I../src ../src/java_cup/emit.java gcj –main=java_cup.Main *.o -o java_cup
kaffe	javac -O4 -d . ../src/java_cup/runtime/*.java javac -O4 -d . ../src/java_cup/*.java
ibm	javac -d . ../src/java_cup/runtime/*.java javac -d . ../src/java_cup/*.java
sun	javac -d . ../src/java_cup/runtime/*.java javac -d . ../src/java_cup/*.java

program	execution command
gcj	time ./java_cup < ../JavaGrammar/Parse/java11.cup
kaffe	time java java_cup.Main < ../JavaGrammar/Parse/java11.cup
ibm	time java java_cup.Main < ../JavaGrammar/Parse/java11.cup
sun client	time java -client java_cup.Main < ../JavaGrammar/Parse/java11.cup
sun server	time java -server java_cup.Main < ../JavaGrammar/Parse/java11.cup

program	user	system	user + system	wall-clock
gcj	4.006	0.085	4.091	4.106
kaffe	14.240	0.121	14.361	14.437
ibm	5.434	0.179	5.613	5.618
sun client	3.142	0.165	3.307	3.846
sun server	6.993	0.168	7.161	13.223

4.9 Accumulated Results



The results for each test were normalized by the times for gcj. The geometric mean was calculated and also set relative to gcj's results.

The variance within the runs for each program in each test was rather insignificant (largest C.O.V.: sun server on javac test, 0.041), giving good confidence in the results.

As shown in the results, only the Sun HotSpot JVM (client) and the IBM JIT Compiler are actualy faster than gcj-compiled Java. If, however, the two test where gcj was not able to compile with optimizations are left out, only the Sun HotSpot JVM wins with a ratio of 0.945 (IBM JIT compiler: 1.135). The failure to compile key classfiles with optimizations is, however, a serious flaw, therefore making gcj's third place well deserved.

Astonishingly enough, Sun's HotSpot JVM in server mode was a complete disapointment, doing noticeably worse than gcj, IBM's JIT and it's own client-mode brother in the longer-running tests LRay and mp3.

The results are summarized in the table below.

Test	gcj	kaffe	ibm	sun client	sun server
BZip2	1	2.248	0.737	0.797	1.213
jProlog	1	4.372	1.178	0.641	0.936
InstantDB	1	1.075	0.373	0.397	0.555
javac	1	1.671	1.129	0.687	2.051
mp3	1	2.166	0.963	1.200	1.477
LRay	1	8.475	1.646	1.315	1.501
JavaCup	1	3.516	1.368	0.937	3.220
geom. mean	1	2.733	0.967	0.798	1.209

Chapter 5

Conclusions

The following is an attempt to interpret the results of this series of tests and elucidate some of the problems and/or features of the different runtime environments.

5.1 Timing Multithreaded Programs on Linux

As the attentive reader may have noted, the test results are divided into user, system and wall-clock times. The purpose was to avoid taking time spent on the disk-subsystem and pagefaults into the results.

Unfortunately, as seen in many of the results, this is not possible on Linux with programs that use posix-threads (`pthreads`) – i.e. all the runtime environments in this test – without major changes to the kerenel or sourcecode. Since however, all tests were preformed on the same system with the localy stored files, the effect is uniform over all tested runtime environments. It does however add a dampening effect on disk-intensive tests, such as javac and InstantDB by adding a constant bias to all the times, therefore reducing the relative distances between the results.

Even for the test that did not explicitly use threads, the user and system times are not reliable, since for the Sun HotSpot JVM they do not include JIT compilation time as with the IBM JIT and Kaffe OpenVM.

5.2 The Gnu Compiler for Java

5.2.1 Some Comments on Pre-Release Open-Source Software

The version of the Gnu Compiler for Java used in these tests was taken directly from the development branch off gcc 3.0. Trying to build a development snapshot of a software project of the magnitude of gcc is an adventure in itself and not for the weak of heart.

Many bugs were discovered in libgcj, the API used by gcj, and duly reported or worked-around in the source-code. This was very time and nerve consuming and not recommended for the weak of heart.

On a positive note, however, I would like to commend the people behind gcj for their good support and lightning-speed response to bug-reports. Anybody willing to look into the ChangeLogs for gcj and libgcj will appreciate the amount of work going into making both products work.

5.2.2 Portability

Although portability of a Java program (in the form of class-files) is lost once it is compiled to native code, gcj provides a different form of cross-platform availability: the incredibly wide range of platforms supported by the gcc back-end. This means that systems otherwise not supported by the big names in the Java marketplace will now also have access to a wide range of software written in Java.

5.2.3 Method Inlining

One of the optimizations that is obviously missing in gcj is function inlining across class foundries. This is due to the fact that the classes are compiled one by one in a stand-alone way. Inlining could then only be performed upon linking, which is however quite a bit more difficult and currently not implemented.

The advantage of this optimization technique is apparent when we consider that the Sun and IBM JVMs had the most trouble keeping up with gcj in code with very few classes and large functions where such optimizations are insignificant or useless.

5.3 The Kaffe OpenVM

The Kaffe OpenVM was by far the worst performer. It should be noted, however, that speed was never the main concern of the developers. The main concerns seems to be portability and size (minamal core JVM with JIT: 124kB, libraries: 743kB), with an obvious interest in embedded platform applications[**?**].

The features that make Kaffe interesting are, first and foremost, the fact that it is an Open Source project and that it can be used to experiment different VM techniques. Other features are a relatively small memory footprint and spartan resource usage.

According to transvirtual, the Kaffe OpenVM (and therefore also the compiler) runs on the x86, StrongARM, MIPS, m68k, Sparc, Alpha, PowerPC and Pa-RISC processor families and supports a number of operating systems.

5.4 The Role of the API

Unfortunately, the test geared at testing the efficiency of the API delivered with each runtime environment could not be run, leaving only room for speculation on the matter.

It should however be noted that in the authors opinion, the Sun Java API is probably the largest freely available collection of design flaws, hacks and downright lousy programming. Since many programs rely heavily on the use of library functions, i.e. the `java.util` package, this would make it an excellent starting point for source-code based optimizations – i.e. better programming.

A great deal of effort has gone into programming and debugging the libraries used in Kaffe and gcj and the author hopes to be able to benchmark these in the near future.

5.5 JIT Compilation Outlook

Although the Sun and IBM JIT compilers perform remarkably well against natively compiled code – even more when taking into account that they compile on-the-fly – there is still much to be done.

The promise of dynamic re-compilation of already compiled code to gain even better performance has yet to be implemented. After the code is compiled, all profiling and analysis stops in the assumption that nothing more can be done. This, as shown in the tests with short, highly repetitive code (lray and mp3), is not necessarily true.

Bibliography

[1] Sun Microsystems. *Java 2 SDK, Enterprise Edition 1.3 Beta Release.* http://developer.java.sun.com/developer/earlyAccess/j2ee/

[2] Sun Microsystems. *The Java HotSpot Performance Engine Architecture* http://java.sun.com/products/hotspot/whitepaper.html

[3] Sun Microsystems. *Frequently asked Questions about the Java HotSpot Virtual Machine* http://java.sun.com/docs/hotspot/PerformanceFAQ.html

[4] Sun Microsystems. *The Java HotSpot Client and Server Virtual Machines* http://java.sun.com/j2se/1.3/docs/guide/performance/hotspot.html

[5] Sun Microsystems. *javac - Java programming language compiler* http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/javac.html

[6] IBM developerWorks. *IBM Developer Kit for Linux, Java 2 Technology Edition, Version 1.3.* http://www-106.ibm.com/developerworks/java/jdk/linux130/

[7] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu and T. Kakatani. *Overview of the IBM Java Just-in-Time Compiler.*

[8] *Kaffe OpenVM 1.0.6.* http://www.kaffe.org

[9] The Kopi Project. *Kopi Java Compiler* http://www.dms.at/kopi/kjc.html

[10] Samuel K. Sanseri. *The Kaffe JIT icode Instruction Set.* http://www.cs.pdx.edu/ sanseri/kaffe/k1.html

[11] Samuel K. Sanseri. *Trampolines in the Kaffe JIT Compiler* http://www.cs.pdx.edu/ sanseri/kaffe/k2.html

[12] Transvirtual Inc. *Kaffe Datasheet.* http://www.transvirtual.com/datasheet.pdf

[13] *The GNU Compiler for the Java Programming Language.*
http://gcc.gnu.org/java

[14] *The GNU Compiler Collection.*
http://gcc.gnu.org

[15] Standard Performance Evaluation Corporation. *SPEC JVM98 Benchmarks.*
http://www.spec.org/osg/jvm98/