Java vs C++ "Shootout" Revisited

June 15, 2004

Summary
"I was sick of hearing people say Java was slow," says Keith Lea, "so I took the benchmark code for C++ and Java from the now outdated Great Computer Language Shootout (Fall 2001) and ran the tests myself." Lea's results three years on? Java, he finds, is significantly faster than optimized C++ in many cases.

By Jeremy Geelan

Keith Lea writes of the benchmark, on his results page, "I was sick of hearing people say Java was slow, when I know it's pretty fast, so I took the benchmark code for C++ and Java from the now outdated Great Computer Language Shootout and ran the tests myself."

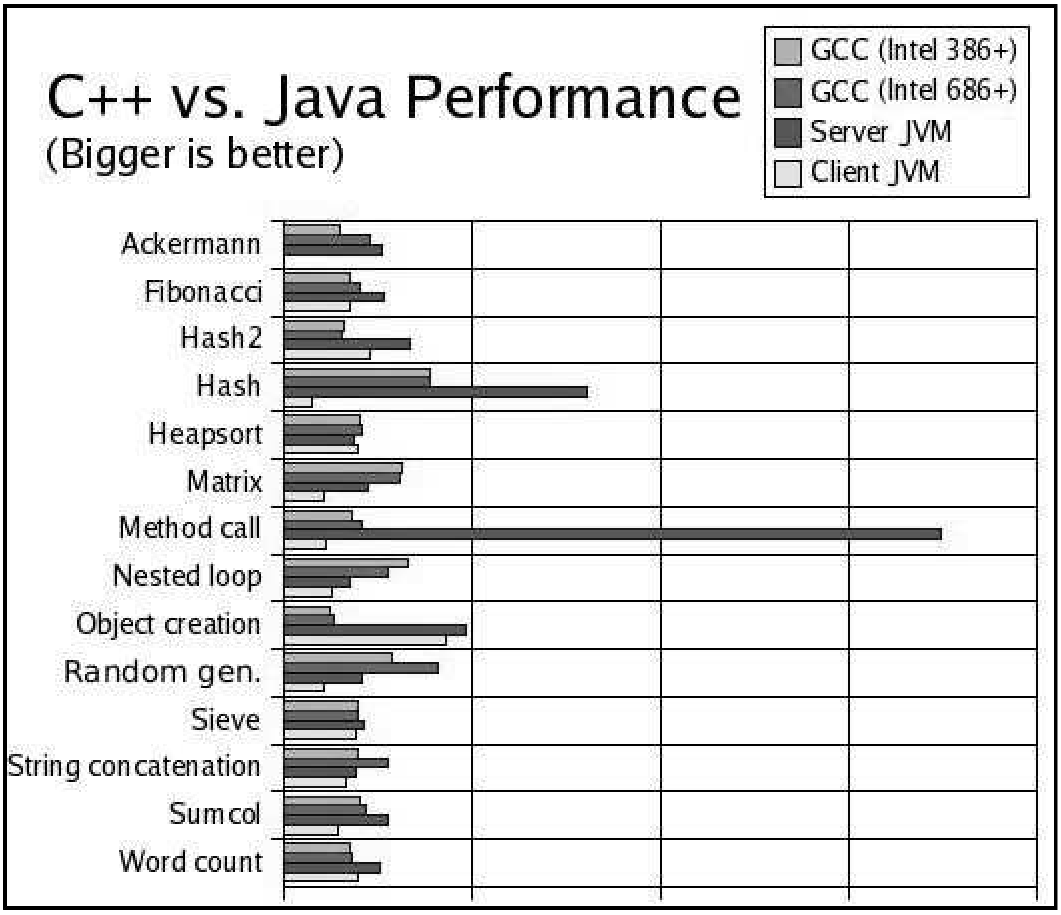
Lea used *G++ (GCC) 3.3.1 20030930* (with *glibc 2.3.2-98*) for the C++, with the `-O2` flag (for both `i386` and `i686`). He compiled the Java code normally with the Sun *Java 1.4.2_01* compiler, and ran it with the Sun 1.4.2_01 JVM. He ran the tests on *Red Hat Linux 9 / Fedora Test1* with the *2.4.20-20.9* kernel on a T30 laptop. The laptop "has a Pentium 4 mobile chip, 512MB of memory, a sort of slow disk," he notes.

The results he got were that Java is significantly faster than optimized C++ in many cases.

"They also show that no one should ever run the client JVM when given the choice," Lea adds. ("Everyone has the choice," he says. To run the server VM, see instructions in the **Using the Server JVM** section below.)

JDJ has agreed to post online anyone else's results as long as they use Java 1.4.2 or higher and any version of GCC that produces faster or equivalent code than the 3.3.1 I used. We encourage you to download the source and/or the binaries and perform the tests yourself, with your favorite compiler and on your favorite platform.

Lea's Data and Results



- Data table and graph (also see above)
- Console run log
- Source:
  - C++ code
  - Java code
- Binaries
  - C++ binaries (Linux)
    - Intel 386 (Basic Pentium or lower)
    - Intel 686 (Optimized for P6-class - Pentium Pro/II, Celeron 266-533MHz, original Athlon, higher)
  - Java class files
- Download all source code and binaries: .bz2, (69KB) .zip (189KB)

JVM startup time *was* included in these results. "That means even with JVM startup time, Java is still faster than C++ in many of these tests," says Lea.

Some of the C++ tests would not compile. "I've never been very good at decoding GCC's error messages," he admits, "so if I couldn't fix a test with a trivial modification, I didn't include it in my benchmarks."

Lea also modified one of the tests, the string concatenation test for Java.

"The test was creating a new `StringBuffer` in each iteration of the loop, which was just silly," he explains. "I updated the code to use a single StringBuffer and appending to it inside the loop."

(The updated tests at the original shootout use this new method.)

"Java lost this benchmark even with the modifications," Lea declares. "So if anyone wants to accuse me of biasing the results, they're going to have to try harder."

Several versions of some of the C++ tests (like `matrix`) were present in the original shootout source, he continues.

"I used the versions without numbers in them, like `matrix.g++` instead of `matrix.g++2`. I don't know which of these were used in the original benchmarks, but from my quick experimenting, the numberless ones generally ran faster than their numbered counterparts."

"Looking at them again," Lea says, "`matrix.g++3` runs faster than the `matrix.g++` that I use. However, it still runs slower than the Java version, so I don't plan to modify the graph/data unless someone asks me to, since getting that graph in the first place was sort of a pain.)"

He continues: "I've been told that the C++ code for the Method Call benchmark returns by value while the Java code returns by reference, and that modifying the C++ code to pass a pointer makes that benchmark faster. However, even with the modification, the C++ version still runs slower than the Java version."

Lea ran the tests many times before running the "official" recorded set of tests, so there was plenty of time for both Java and the C++ tests to "warm up" (both the Java and C++ tests got faster after he ran them a few times).

"I've been told that these tests are invalid because they were run with GCC," he concedes, adding: "I have seen both benchmarks that show GCC producing faster code than Visual Studio's VC++ compiler, and benchmarks showing the opposite. If I update the benchmarks with another compiler added, it will be the Intel C++ Compiler, which I'm pretty sure produces faster code than VC++."

Lea says he's been accused of biasing the results by using the `-O2` option for GCC, "supposedly because -O2 optimizes for space, thus slowing down the benchmark," he explains.

But this is not what -O2 does, he points out, referring to the GCC -O documentation:

> -O2: Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. The compiler does not perform loop unrolling or function inlining when you specify -O2. As compared to -O, this option increases both compilation time and the performance of the generated code.

"On the other hand, -O3 performs space-speed tradeoffs, and -O performs fewer optimizations. Thus, for these tests, I think O2 was the best choice," Lea concludes.

"I don't have an automated means of building and benchmarking these things (and the scripts that came with the original shootout didn't run for me)," he continues. "I really do want people to test it on their own machines, but it's going to take some work, I guess."

Lea compiled the C++ code with:

```
g++ [test].cpp -O2 -march=i386 -o [test]-386

g++ [test].cpp -O2 -march=i686 -o [test]-686
```

and the Java code with:

```
javac [test].java
```

To see how he ran the binaries, see the run log. You can download the source code he used in either .bz2 or .zip format.

Using the Server JVM

Every form of Sun's Java runtime comes with both the "client VM" and the "server VM."

"Unfortunately, Java applications and applets run by default in the client VM," Lea observes. "The Server VM is much faster than the Client VM, but it has the downside of taking around 10% longer to start up, and it uses more memory."

Lea explains the two ways to run Java applications with the server VM as follows

1. When launching a Java application from the command line, use `java -server [arguments...]`

instead of `java [arguments...]`. For example, use `java -server -jar beanshell.jar`.

2. Modify the `jvm.cfg` file in your Java installation. (It's a text file, so you can use Notepad or Emacs to edit it.) This is located in *C:\Program Files\Java\j2reXXX\lib\i386\* on Windows, */usr/java/j2reXXX/lib/i386/* on Linux. You will see two lines:

   ```
   -client KNOWN
   -server KNOWN
   ```

   You should change them to:

   ```
   -server KNOWN
   -client KNOWN
   ```

   This change will cause the server VM to be run for all applications, unless they are run with the `-client` argument.

He can be contacted at keith@kano.net.

Links

- Java vs. C benchmark - the benchmark is in Microsoft Word format
- FreeTTS performance case study - at Sun Research
- Linux Number Crunching - used 1.4.1 VM, got very different results than Lea did, maybe due to floating point optimizations added in 1.4.2
- SNAP matrix benchmark - Java vs. C++ for matrix multiplication
- Nine-Language Performance Round-Up: math and file I/O performance among nine languages
- Performance comparison C++, C# and Java - similar results as Lea's, mostly
- Java vs. C - for bubble sort, IBM's 1.3.0 client JVM comes out on top over GCC with full optimizations

About the author

Jeremy Geelan is i-technology group publisher of SYS-CON Media and the publisher of LinuxWorld Magazine (www.linuxworld.com). (*more*)