Java Performance Advancing Rapidly

Just-In-Time Compilers Show Java Can Compete with Compiled Code

by Brian Case

Java got the computer industry's attention because of its superior integration with the Internet and the World-Wide Web. With Java, it is easy to build a portable, GUI-based applet that can be embedded in a Web page and then used on any platform that has a Java-enabled Web browser. The recent release of some good Java development environments makes it even easier to deploy Java applets.

But, as anyone who has done more than a cursory investigation of Java applets on the Web will confirm, Java currently suffers from two deficiencies: a complete lack of substantive applets and dreadfully slow performance. Some Java proponents have proposed building Java microprocessors that directly execute the Java VM *(see* **1005VP.PDF** *)* to achieve compelling Java performance and reduce memory requirements. The early results of benchmarking just-in-time compilers (JITs), however, indicate that Java integer code on a standard microprocessor with a JIT is only about a factor of two slower than optimized C code. As JITs mature, this gap will likely narrow. Thus, Java microprocessors will probably have only a small performance advantage over general-purpose microprocessors unless Java chips can beat optimized C performance, which seems unlikely.

Java Performance Is All Over the Map

Users of Windows 95 are the most affected by poor emulation, as Table 1 shows. Netscape's Java performance on Windows 95 is so bad, in fact, that it almost seems intentional. The current version of Netscape runs one of the programs—the Loop component of CaffeineMark—less than 1/70 the speed of the fastest JIT. Results not included here indicate that Netscape's Java performance has actually declined compared with previous versions of the browser.

There is good news for Java devotees, however. First, the recent release of two commercial JIT compilers dramatically improves Java performance. Second, the generally poor performance of Java on Windows 95 appears to be due to overhead in Windows 95's implementation of the Win32 graphics API. By using Windows NT instead, Java junkies can reclaim some of the lost performance without sacrificing access to the growing portfolio of Win32-based Java tools.

Third, the much-improved performance provided by JITs makes it possible to develop useful Java applets and even standalone applications. The widespread deployment of JITs may take a few months, but it now seems indisputable that Java has the potential to be a force across nearly the entire spectrum of computer software. Netscape has licensed Borland's JIT, and as soon as a JIT-enabled version of Netscape is released, the race will be on to develop commercial applications in Java.

Table 1 shows the results of running three different benchmarks on 13 different combinations of Java VM implementations and operating systems. All benchmarks were run on a fast Pentium machine (see sidebar), but such a machine will be considered midrange by the end of 1996.

As the table shows, the performance of nongraphical Java programs spans a wide range and is heavily dependent on the quality of the VM implementation. The other striking implication of the data shown in Table 1 is that the Win32

Execution Environment		Pentominoes		Linpack		CaffeineMark						Average Ratio
		Seconds	Ratio	MFlops	Ratio	Sieve	Loop	Graphics	Image	Mark	Ratio	
Windows 95	Symantec JIT	64	28.0	4.2	11.0	386	936	101	102	381	9.1	16.0
	Borland JIT	75	24.0	5.5	14.0	338	1307	97	94	459	11.0	16.0
	Symantec VM	610	3.0	0.98	2.5	75	79	94	88	84	2.0	2.5
	Sun JDK 1.0.1	980	1.9	0.69	1.8	54	50	91	81	69	1.6	1.8
	Netscape Atlas	1820	1.0	0.39	1.0	18	28	74	51	42	1.0	1.0
Windows NT 3.51	Symantec JIT	62	29.0	3.5	9.0	1116	1247	331	158	713	17.0	18.0
	Borland JIT	74	25.0	5.4	14.0	809	2138	298	147	848	20.0	20.0
	Symantec VM	600	3.0	0.98	2.5	90	83	288	125	146	3.5	3.0
	Sun JDK 1.0.1	980	1.9	0.65	1.7	60	51	262	116	122	2.9	2.2
	Netscape Atlas	1830	1.0	0.40	1.0	34	29	174	63	75	1.8	1.3
Linux 1.3.81	optimized C	34	53.0	16.0	41.0	not directly portable from Java to C						47.0
	Kaffe JIT 0.3p1	67	27.0	3.9*	10.0	won't run						19.0
	Sun JDK 1.0	950	1.9	0.60	1.5	64	58	200	162	121	2.9	2.1
	Netscape Atlas	1300	1.4	0.52	1.3	47	42	188	122	99	2.4	1.7

Table 1. This table summarizes the results for three Java benchmarks with five Java implementations on three operating systems. For Pentominoes, a lower number of seconds is better; for the other benchmarks, a higher score is better. The Ratio columns show performance relative to Netscape under Windows 95. *The Kaffe Linpack result is suspect because this trial gave abnormal error statistics. (Source: MDR)

System Setup

All benchmarks were run on a 166-MHz Pentium-based PC with 512K pipelined-burst cache and 32M of memory. The system used a relatively pedestrian PCI-based ATI Graphics Xpression video card with 2M of DRAM for the frame buffer; video resolution was set to 1280 × 1024 at 8 bits per pixel.

The benchmark numbers in Table 1 are the best result produced from at least five trials of each combination of benchmark and environment. Results varied by up to 10% on different runs, so all numbers shown are rounded to two significant digits.

Since Java is still new, few benchmarks are available. Two were found on the Web and one was ported from a C program posted to a Usenet newsgroup. The benchmarks are:

- Pentominoes: an integer-intensive puzzle-solving program (ported from C to Java).
- Linpack: a standard floating-point intensive benchmark (see *www.netlib.org/benchmark/linpackjava/*).
- CaffeineMark: a mix of integer-intensive and graphics-intensive programs (see *www.webfayre.com/pendragon/pscaffeine.html*).

The Java environments consist of three different operating systems: Window 95, Windows NT Workstation 3.51, and Linux (kernel version 1.3.81). Five different Java virtual-machine implementations were tested on the two Windows versions: the Sun Java Development Kit (JDK) version 1.0.1, Netscape's Atlas version pr2, the Symantec Cafe interpreted VM, the Symantec Cafe JIT, and the Borland JIT that ships with the Borland C++ 5.0 Development Suite. To help gauge current state-of-the art performance, C-language versions of the Pentominoes and Linpack benchmarks were compiled using GCC with full optimization enabled and run under Linux.

The Symantec JIT is an add-on to Symantec's Cafe Java development environment for Windows. This JIT can also be plugged into Sun's JDK environment if desired. Borland's JIT, called AppAccelerator, ships only with its full C++ Development Suite product. The Kaffe JIT for Linux is being developed by Tim Wilkinson as a spare-time project and is far from complete. It is available on the Web at *web.soi.city.ac.uk/homes/tim/kaffe/kaffe.html*.

graphics API is pitifully slow on Windows 95: the JITs provide only a meager improvement on the Graphics program in the CaffeineMark suite.

JITs Shine on Integer Code

Not surprisingly, the authors of the current JITs seem to have focused on improving the performance of integer code. As shown by the Pentominoes program, Symantec's JIT delivers the highest Java performance. Compared with Symantec's own interpreted VM implementation, which is already the fastest of its kind, the JIT improves performance by roughly a factor of 10. The Borland JIT is slower but not far behind. Even the experimental Kaffe JIT performs very well.

The native C-language version of Pentominoes delivers roughly twice the performance of the JITs, and while a factor of two is a substantial margin, a couple of considerations make this disparity less disappointing than it first seems. First, the Gnu C compiler is mature and stable, while the Symantec and Borland JITs have been available for only a few weeks and will probably improve over the coming months. Second, the Pentominoes benchmark makes heavy use of array accesses, which are bounds-checked in Java; this situation probably puts the Java code at an unfair advantage. There may be room to improve the JIT's optimization of bounds-checking code.

Floating-Point Performance Acceptable

Linpack has been a standard floating-point benchmark for nearly two decades, so it's not surprising that it is one of the first benchmark to be ported to Java. The Java version does not use loop unrolling (and is thus equivalent to using the C-version's "rolled" option) and uses double-precision floating-point arithmetic (Java does not provide a single-precision FP data type).

Here, Borland's JIT takes the prize, improving performance over the fastest interpreted VM by more than a factor of five. Even compared with the C-language version, the Borland JIT is only a factor of three behind. As the JITs mature, floating-point performance is likely to improve.

Graphics Programs Expose Windows 95 Overhead

CaffeineMark is a suite of four programs; higher scores are better. The overall CaffeineMark is simply the average of the four components. The Graphics program draws rectangles of various sizes and colors in the applet window. The Image program copies a small bit-mapped image (a coffee cup, of course) to random locations in the applet window a large number of times. Both programs run for a few seconds. From its name, Sieve would appear to be a simple CPU-intensive program—it always has been—but the improvement provided by the JITs under Windows 95 does not track the improvement on Pentominoes. On the other hand, the improvement on Loop, which also appears to be a simple CPU benchmark, gains more performance from the JITs than does Pentominoes.

The quality of the graphics API and interfaces is directly reflected in the Graphics and Image results. For the Graphics component of CaffeineMark, simply moving to NT improves performance by a factor of three; NT improves the Image score by about 50%. Furthermore, notice that the score on Graphics produced by Netscape—which uses an interpretive

Java VM—under Linux/X-windows beats all Windows 95 scores, even those for the JITs.

Java VM: ANDF-2000?

A few years ago, there was an OSF initiative called ANDF (architecture-neutral distribution format, see MPR 10/2/91, p. 17) with the goal of producing a virtual machine that could be used to distribute application programs in a completely portable format. While ANDF was technically compelling, it had the drawback of inserting one more layer of software that could contain bugs. Thus, from the application developer's point of view, ANDF offered a solution to the relatively unimportant portability problem but exacerbated an already difficult one of ensuring software quality.

The Java .class-file format (used for Java executable files) is not a true ANDF because it lacks some key capabilities, such as good support for a variety of languages. Nonetheless, there are efforts under way to compile other languages to the Java VM, and some compilers are already available.

Java seems to be force-feeding the essential concepts of ANDF down the throats of developers and users. Whether developers will choke and reject Java because of support issues—e.g., is a user's problem caused by the program, the operating system, or the Java VM?—remains to be seen.

It seems likely, however, that at least some off-the-shelf, shrink-wrapped applications will become available in the portable Java binary format. In the extreme case, it is possible that businesses and consumers will prefer applications that are distributed as Java .class files and that developers will be forced to develop Java versions of their popular applications to remain competitive. Indeed, perhaps Java will finally give PC developers easy access to the workstation market, assuming they still want it.

Given the ubiquity of x86-based PCs, the most probable scenario is that the Java .class file format will be preferred for applications that must be cross-platform, while the native x86-binary format will be sufficient for single-user PC applications. Java proponents and Internet visionaries would say that all applications will need to be Internet-enabled and cross platform, which may be true, but access to Java applets will soon be available to every application regardless of its heritage. Most OS vendors have committed to embedding the Java VM into the operating-system structure, which means even applications not written in Java can be Java-enabled (i.e., able to run Java applets).

How all this plays out will be interesting to see. Carried to its logical conclusion, this trend would result in an evolution toward the irrelevance of native instruction-set architecture, just as the ANDF dreamers (this author included) once envisioned. Such a situation would presumably allow the "best" microprocessor to be chosen for a product without regard to its native instruction set. This extreme case seems unlikely, at least in the short term, but the current pace of Java activity makes predicting the outcome difficult.

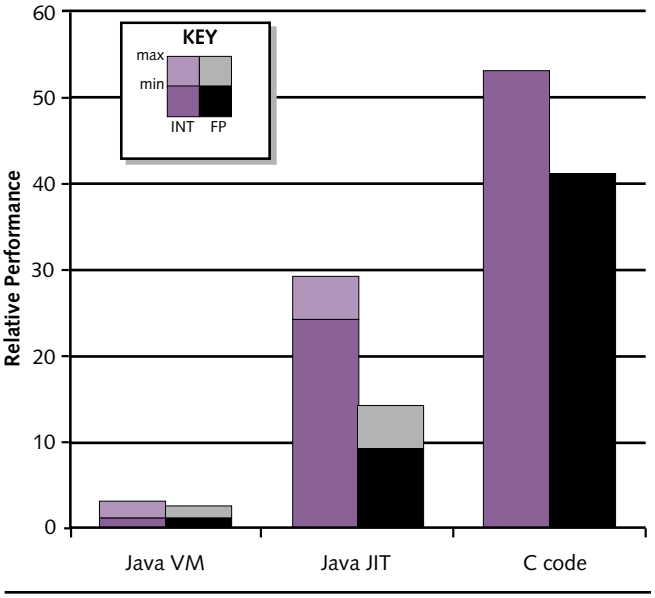

Figure 1. While Java JITs are an order of magnitude faster than virtual machines (VMs), they trail optimized C code by a factor of two. See Table 1 for details.

Dynamic Compilation Finally Gets Attention

The viability of many potential Java applications depends on high-performance VM implementations. Java JITs seem able to provide the needed performance. JITs translate Java VM code as it is run, every time it is run. Static recompilers, if they existed, would translate Java applications once and save the compiled code on disk, offering the potential of even higher performance without any delay associated with on-the-fly dynamic compilation.

Dynamic compilation has been waiting in the wings for its chance to take center stage. The first hints that this technology was ready for widespread commercial deployment came in the form of dynamic compilers for SmallTalk implementations and Connectix's Speed Doubler upgrade to Apple's PowerPC-based 68000 emulator. With Java, dynamic compilation can at last move into the mainstream.

The benchmark results summarized in Figure 1 demonstrate that one of the impediments to widespread Java deployment—poor performance—can be addressed by JITs. They also indicate that the potential performance advantage of a Java microprocessor may be limited to a factor of two or less. There is still much work to be done on both software and hardware, but because of JITs and possible static compilers, Java chips will be competing head-on with standard microprocessors.

Now it is up to OS vendors to provide the promised transparent support for Java applications, and up to application developers to test the market to see if Java applications appeal to businesses and consumers. Within the next year, the market may decide if Java will be "just" an Internet language or a force across the entire computing spectrum. M