VIEWPOINT

Java Virtual Machine Should Stay Virtual

Hardware Not Needed To Serve Java's Tantalizing Brew

by Brian Case

When I first learned that the Java language is compiled to an abstract virtual machine (VM) to achieve portability *(see* 100402.PDF *)*, the microprocessor and system-software designer in me immediately embraced the idea of building new microprocessors and clever software implementations of the Java VM. This kind of technical challenge is always attractive.

At first glance, the popularity and pervasiveness of Java seem to create an opportunity for investments in high-performance Java VMs. Assuming that performance-hungry Java applications will appear, the decision to build a microprocessor implementation of the VM seems like a "no-brainer." In PCs and other systems that need a legacy microprocessor for compatibility, high-performance software implementations of the VM will be required.

The Java Paradigm Is Compelling

To understand the demand for Java and the derived demand for Java execution environments, we need to understand what makes Java valuable. In my opinion, the advantages of the Java paradigm fall into two categories. First, there is the VM-based execution model. The ability to use Java applets and programs simply by porting the VM execution environment is what got the Java steamroller moving. Currently, a Java applet can be run using the Netscape Internet browser, for example, on platforms from PCs to Macs to workstations from many different manufacturers. Netscape is even available for Linix. As a front-end for a corporate data-base application, a single Java applet can solve many thorny accessibility problems. This ability is powerfully compelling.

The second category of advantages centers around the safety of programming in Java. The Java language is object-oriented with many built-in safety checks. There are no arbitrary pointers (as in C), array bounds are always checked, and memory allocation is automatic. By eliminating several of the most annoying and bug-producing problems of C, Java proponents suggest that Java will displace C in many cases. In addition, a degree of security for transmitted applets is achieved by a verifier at the receiving end that tries to make sure the applet code is not malicious.

I have no argument with the Java paradigm; I agree that it is compelling in many application areas. I do not, however, agree that Java microprocessors enhance in a unique way the delivery of Java's advantages. First, I believe that for those applications where Java is compelling, Java will be chosen independent of performance issues. Second, I believe that a standard microprocessor coupled with an appropriate Java VM implementation—emulator, dynamic compiler, or conventional compiler—is technically as good as a Java microprocessor and superior in other ways.

Execution Method		Relative Speed	Relative Mem. Size
Pure emulation	Pure emulation	1	1
Dynamic compilation	Unaggressive	3	2
	Aggressive	7	5
Static re-compilation	Unaggressive	5	3
	Aggressive	10	5
Conventional compilation	Highly optimizing	15	1
Java microprocessor	Low-end	3	1
	High-end	14	1

Table 1. This table compares estimates of the relative performance and memory footprint of some possible Java implementations. The conventional and Java processors are assumed to have comparable internal designs and manufacturing technology.

Many Ways to Run Java

Table 1 lists the basic categories from the broad spectrum of possible Java implementations. In the category of dynamic compilation, a variety of performance/memory-footprint trade-offs is possible, though probably only a few will satisfy the majority of uses. Sun, and possibly others, are working on Java microprocessors. Another possible technique—enhancing a conventional microprocessor to speed Java—has not been the subject of any press releases to date.

Most of the techniques listed in Table 1 are based on the Java VM, that is, Java programs are first compiled to the VM, then VM instructions are executed in some manner. It is possible and desirable, however, to compile Java programs in the conventional way to the native instruction set of a conventional microprocessor. Excellent performance can be gained with a compiler, but what's lost is portability: the code can't be blindly distributed on the Internet, for example.

In an embedded product not connected to a communication medium, however, portability is not an issue, making conventional compilation reasonable. Sun has mentioned that Java might be used in consumer devices as common and ubiquitous as shavers and cameras; it seems reasonable to assume that these devices will not download applets from Web sites. (I hope I'm not wrong about this!)

The performance and memory-footprint numbers in Table 1 are only educated guesses that vastly oversimplify the issues involved. In fact, each number in the table should be a range, and the categories have significant overlap. Still, they are probably accurate enough for my purposes. One point of these numbers is to show that Java microprocessors appear to combine the best performance with the best memory footprint. Another point, however, is that high performance, small memory footprint, and various compromises between the extremes are possible without Java microprocessors.

The implementations of Java that exist today in Internet browsers, for example, are pure software emulators, but I expect this situation to change soon. Rumor has it that the established suppliers of software development tools have built very high-performance dynamic compilers, and conventional optimizing compilers will likely eventually be included in their Java development products.

I expect forthcoming development tools will provide a variety of Java execution techniques including pure emulation, dynamic compilation, static (just-in-time, or JIT) recompilation, and conventional compilation. At least one of these execution techniques will appropriate for deploying Java programs in most standalone and embedded products.

Products Need Java, Not Java Chips

Embedded products whose designers want to use Java fall into two categories: those products that will run applets and those that will not. For those that don't run applets, a conventional Java compiler for a conventional, high-volume microprocessor offers all the meaningful technical advantages of Java plus the advantage of choosing any standard microprocessor that meets the price, performance, and other specific requirements of the design.

An embedded product that does need to run Java applets would seem an obvious application for a Java chip, but notice that the device's applets are highly likely to be specialized; it probably doesn't make sense to expect the same device to run a tic-tac-toe and database front-end applet from a Web site. Similarly, I doubt it makes sense to run an applet for my shaver, camera, or laser-printer with my Web browser. So, in this case, the advantage of distributing the applet in the form of Java VM instructions is not compelling. In fact, distributing the applet as a conventional microprocessor binary—yes, even with the bounds-checking and other Java-overhead included—will likely result in a smaller file to transmit and store. Despite what Java proponents say, some measurements indicate that Java VM code is not particularly small. A Java microprocessor is not needed here.

In general, each embedded application has a minimum performance requirement, and any extra performance is irrelevant. Most applications that require Java will be satisfied by a conventional microprocessor plus a software VM (or compiler); the others, which require high performance, won't be satisfied by a Java microprocessor. Many applications, such as PostScript emulation and multimedia algorithms, benefit from pointer arithmetic, which is missing from the Java VM.

I have heard Java proponents claim that the existing huge installed base of Java applets on the Web (28,000 by one estimate) is an indicator of the viability of Java chips. First, these applets are irrelevant to embedded Java applications. Second, these applets, by their existence today, don't need Java chips. Third, a Java applet that does need heavyweight computing performance is probably not a generally useful applet, just as an applet for a camera or shaver is not generally useful.

An applet requiring heavyweight performance can probably suffer the overhead of a static recompiler or an aggressive dynamic compiler. These techniques provide relative performance less than a factor of two below the best Java chip, and given the investment required for high-end microprocessor development today and tomorrow, I'm willing to bet that the best commodity microprocessor at a comparable price will have net performance better than the best Java chip. Dynamic compilation has a much larger memory footprint, but in high-end devices, such as PCs, this is probably not a problem.

Are there any areas where Java microprocessors make sense? One possibility is in memory-starved applications that must accept performance-critical Java applets; the applications won't be able to stand the extra memory needed for a dynamic compiler. Another possibility is for designers who want to exploit the dynamic linking and garbage collection of the Java VM but insist that a software solution isn't acceptable. Sun believes that a Java chip can be competitive in the embedded market because designers won't need to repeat the work of implementing dynamic linking and garbage collection, and software overhead won't be required.

We Might All Be Right

Recently, I have come to the realization that my biggest disagreement is with product designers who, according to Sun, are creating the demand for Java microprocessors. I believe the engineers and managers at these companies should be able to see that what's adding value to their products is Java, not Java chips; they should be able to see that there are many other, better ways to tap Java's unique value. But if customers really are ringing Sun's phones off the wall wanting to buy Java microprocessors, I guess Sun has a capitalistic duty to satisfy the demand.

Sun claims it will produce Java microprocessors that are competitive with any standard processor on the market on a price and feature set basis, and that these Java chips will deliver as good or better performance on Java code than even a comparable standard processor compiling Java directly to its native instruction set. If Sun can deliver on these promises, Java microprocessors may have a bright future, but I just can't see how it can happen.

Ultimately, it will be Sun's customers who decide the fate of Java microprocessors. If they design Java chips into high-volume cell phones, cameras, or shavers, those products alone could consume millions of chips and make Java microprocessors successful. Even if Java chips achieve such volumes, however, it still doesn't mean they make sense; too many other successful things don't make sense, either. M