By Jay Kamdar,
Vice president of marketing and COO,
Nazomi.

Embedded Java in Information Appliances

Embedded devices - from smartcards to embedded servers - are connecting to the Internet. For the embedded Internet to succeed, it needs software, and getting a large volume of software requires a standard platform that lets developers build their applications without having to create a special version for each manufacturer's device. Java holds great promise as a platform that can support Internet-enabled information appliances. It offers portability and improved productivity, is compact and is verifiable. However, Java's flexibility and portability come at too high a price - in terms of performance, memory and power consumption - for most embedded applications. This presentation will explore the use of Java in embedded Internet applications and briefly overview the features, advantages and disadvantages of the various approaches to implementing Java. It will go into greater detail on Java coprocessors as a the most practical way of addressing performance issues and will focus on JSTAR as the specific solution.

INTRODUCTION

Information appliances are a new class of specialized devices designed to interact with the Internet or a local network. There is incredible momentum behind such smart devices, with many industry analysts predicting that these appliances will outnumber PCs on the Internet within the next two years. What is equally interesting is that in every application category, manufacturers who are building information appliances are making Java an integral part of their devices. A few examples include mobile phones, screen phones, set-top boxes, smart cards, digital cameras, game machines, digital televisions, automotive (navigation, communication and entertainment), and thin servers. The Java solutions market represents tens of millions of consumer and embedded Internet devices.

A PARADIGM SHIFT

The role Java plays in information appliances signals a dramatic move away from the traditional model of development for those focused on embedded devices. The embedded world has always focused on the practical aspects of the devices they build involving some combination of cost, size/weight and power consumption. Reducing these means finding low performance components that are just good enough to do the job. If a device will be sold in volume, cheap components mean lower costs and higher profits. Software development costs are not nearly so important. Reducing the development cost for a device is easy: just sell more units.

This fact has kept embedded projects from enjoying the kind of productivity gains seen by virtually every other class of software developer. But, with the advent of Internet appliances, all this is beginning to change and Java is at the forefront of that change. These devices demand a more flexible and open development approach which Java makes possible bringing a new level of productivity and flexibility to the embedded world. Here are a few ways that Java is turning embedded development on its head:

- **Dedicated & standalone vs. flexible & networked:** Traditionally, embedded devices were designed for a single purpose. These devices interacted with their user but rarely had a need for complex interaction with other devices, especially devices that had yet to be invented. That will change with the development of smarter products. These products must be able to be used in ways not imagined by their designers and to interact with a variety of other products. Smarter interconnects among devices will be as important to the user as the capabilities of the individual devices.
- **Custom & proprietary vs. universal & standards-based:** Where efficiency of the end product is the ultimate goal, it is better to create a custom solution that does exactly what you want and nothing you don't need. Every implementation, every algorithm can be tuned to the specific requirements of the device under construction. But flexible, networked devices demand a different approach. Standards are a vital part of that approach. Each device can't dictate its communication with other devices; the only answer is to agree on a set of protocols that everyone will use. And if there is benefit in a device's software being customizable, it is essential that it make use of a universal software platform. The more standardized the environment, the easier it is to adapt it to new situations.
- **Monolithic vs. layered:** A device's software was generally designed as a single monolithic structure. Such a design is more efficient for the end product, although it tends to take far longer to develop. Increasing market pressure requires product teams to deliver their products faster. And the best way to write code faster is not to write it at all: reuse instead of recoding. Monolithic applications are giving way to layered approaches that write new code only where it's unavoidable. And programmer productivity becomes much more important when it is reflected in reduced time to market.

Despite its advantages, Java's flexibility and portability often come at too high a price for many of the information appliance applications. The JVM, combined

Copyright 2000 by Dedicated Systems Magazine - 2000 Q4 (http://www.dedicated-systems.com)

with class libraries, can cause Java to require more memory than C. In addition, the JVM is an interpreter and the overhead of byte code interpretation and extra error checking requires a more powerful processor to get to the required level of performance. More memory and faster processors mean higher cost. Worse, for mobile wireless applications they also mean higher power consumption and shorter battery life.

THE ISSUE OF JAVA PERFORMANCE

Today, there are many different approaches to executing Java code. What follows is a survey of the different techniques commonly in use, each of which has benefits and tradeoffs. Keep in mind that these approaches are not mutually exclusive; a single Java solution may benefit from a combination of approaches.

- **The classic interpreter:** Most Java implementations continue to ship with the kind of interpreter that appeared in the first JVM with an instruction processing loop where efficiency is paramount. In a loop that executes hundreds of thousands of times before the first bit of user code executes, even a tiny inefficiency becomes significant.
- **Just-in-time (JIT) compilers:** JIT compilers replace portable Java byte code with the equivalent native machine instructions the first time a piece of code is run. Doing the translation at run time maintains the benefit of portability, security and validation. JIT compilers provide dramatically better performance than interpreters, often reducing run time by a factor of ten. But they can be a mixed blessing. Using a JIT compiler means adding some translation overhead up front to get better performance later in the execution. And a JIT means big increase in memory footprint. JIT compilers running on Intel x86 processors create native code that averages four times the size of the original byte code, while the code produced for RISC processors like MIPS and Sun's SPARC average eight times the size. In server applications where large memory configurations are the norm a tradeoff of memory for time may well be acceptable. But in an embedded application more memory means higher cost and more drain on limited power.
- **Native code compilers:** A Java native compiler violates some of the rules of Java in exchange for better performance. It treats Java byte code the way other language compilers treat source, analyzing it, optimizing it and then converting it into native code. This object code is linked with Java run time support and other system libraries to create an executable program. Such a program gives up many of the advantages of Java. It isn't portable, can't be validated for safety and may not support run time integration with new classes into dynamic networks of components. Java native applications will have better performance than JIT compiled code, although the difference may not be significant.
- **Faster processors:** The usual solution for slow software is to throw more hardware at it. Processors continue to improve both their performance and their price/performance; eventually they will reach a stage where slow applications run fast enough. The questions are: how long must we wait,and what is the cost? Faster processors demand more power and generate more heat and radio frequency emissions, making them a real challenge for cell phones and other portable wireless devices. When faster processors arrive they won't completely eliminate the need for some kind of JIT solution and its increase in memory that comes with it.
- **Java-specific processors:** Instead of attempting to emulate the JVM on an existing processor, a Java chip implements that conceptual computer in real silicon and uses Java byte code as its machine language. There is no need to translate to a different instruction set or to layer the JVM's stack-based architecture on top of the register architecture used by most microprocessors. Java processors seem the simplest and most natural way to run Java efficiently. The challenge for a Java chip is the same as for any new processor with a new instruction set: software. One can assume that a supplier of a Java chip will provide a real-time operating system (RTOS) and a JVM for their chip, whether as custom software or ported versions of existing packages. Note that any new processor, even a Java processor, needs someone to implement a JVM for it before it can run Java applications. This can be a significant effort: while Java applications are portable, the underlying Java platform software is not. This raises important questions each embedded device designer must answer: Is the software my device will need available on this processor? Is my first choice of RTOS (either because of its feature set or my organization's familiarity with it) available? What about the availability and quality of JVM(s) for that chip? And what about other software: applications, libraries, etc.? In short, how much of the software already runs on the chip and how much time and money will it take to get the missing pieces in place? Designers of embedded devices have large investments in hardware, software and, even more significant, in the experience and expertise of their people. Java processors run counter to that investment, requiring designers to start again with new hardware, new and potentially less stable software and a possibly short but still not insignificant learning curve. Many developers do notconsider the new investment worthwhile.

Clearly each approach to running Java has its strengths and its weaknesses. None emerges as the clear winner for embedded devices, as designers are forced to trade off processor speed against memory requirements against the availability of software.

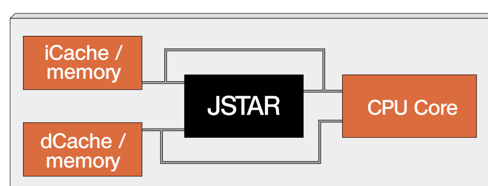

Figure 1. The JSTAR-enabled processor.

Copyright 2000 by Dedicated Systems Magazine - 2000 Q4 (http://www.dedicated-systems.com)

Using a Java Coprocessor

The ideal Java solution would run Java applications at least as well as the alternatives available today, offering performance five to ten times higher than interpreters and would do so without the need for a faster CPU clock or extra memory. It would be compatible with existing processor architectures taking advantage both of existing expertise and a wide variety of available software.

It is from this problem definition that Nazomi Communications began. Founded in late 1998, Nazomi, formerly JEDI Technologies, set itself the task of eliminating the technical barriers to widespread use of Java in embedded devices. Nazomi's approach was to develop a Java accelerator that could be added to existing microprocessor designs. Acting like an on-the-fly JIT compiler, this accelerator, called JSTAR, provides similar levels of performance without any need for memory to hold the translated code. And, because JSTAR enhances an existing processor, it gets all the benefits of using that processor, including the catalogue of software that supports it.

Architecturally, JSTAR is a coprocessor that interfaces to the native microprocessor core and its cache or memory subsystem. JSTAR acts as a Java interpreter in silicon, retrieving byte code instructions from memory and executing them in conjunction with the native processor. JSTAR operates directly on Java byte code, eliminating the extra memory JIT compilers need to hold the native code they generate.

THE JSTAR-ENABLED PROCESSOR

Adding JSTAR to a microprocessor requires no modification to the native core; the native instruction set and pipeline architecture of the processor are unchanged. Operating systems and native applications, software components and tools run on a JSTAR-enabled processor just as they do on the original chip.

Even Java native methods compiled for the native processor run without modification. JSTAR was designed to integrate with existing Java Virtual Machine implementations from Sun, HP and others. The JVM is modified to initialize JSTAR and then to give it control of the main fetch/decode/execute instruction processing loop. Making a JVM work with JSTAR is greatly simplified by JSTAR's ability to adapt to the internals of the JVM. In particular, JSTAR does not impose any specific layout for local variables, call stack frames or the Java operand stack.

JSTAR also works with the JVM's implementation of garbage collection, native method invocation and thread switching and synchronization. It is designed to work with the variety of thread schedulers used by JVMs, including both native threads and cooperative thread schemes such as Sun's green threads. It also supports multiple JVMs running concurrently on the same processor. JSTAR's flexibility minimizes the work required to support a new JVM . It also permits JSTAR to benefit from all the work being done to optimize specific Java run-time implementations.

THE JSTAR ADVANTAGE

Building an embedded device involves making choices: technical, economic and practical. Each choice is defined by a series of tradeoffs and implicit decisions regarding other choices. Choose a processor and you either limit yourself to software that already runs on that processor or to software that can be moved to it within the time available and at an acceptable cost. Add an operating system and you restrict your options further. And so it goes with each new layer of software and each new component.

Developers working with embedded Java must balance their need for Java performance against other requirements: cost, power, size, the ability to run native code, the ability to interact with the outside world and so on. Java technology that intrudes on the device's ability to satisfy the non-Java requirements of a device is not a viable solution.

JSTAR represents the low risk, low cost Java solution most developers need. It offers a high performance engine for running Java applications without placing unacceptable demands on scarce resources. And it offers this benefit without giving up the high C/C++ performance of a native processor, its large catalogue of available software or the years of expertise built up by embedded developers. This symbiosis between JSTAR and a native core means the best of both Java and traditional computing. Details on JSTAR can be found at www.nazomi.com ■

Jay Kamdar has 18 years of technical marketing experience in the semiconductor and computer industry. Prior to founding Nazomi Communications, he was a group marketing manager for Java processor products at Sun Microsystems. His responsibilities included licensing of picoJava technology, development of new net-centric markets leveraging the Java paradigm, and development of long-term strategic plans for the information appliances market (e.g., i-TV, set-top boxes, Webphones, smartcards and advanced handhelds).

Kamdar spent 14 years with National Semiconductor in a number of positions, including product marketing director for a division producing logic, mass storage controllers, interface products and very advanced bus controllers, and North American marketing director for National's microcontroller and microprocessor division. In these roles, Kamdar successfully led National into new markets and helped the company achieve number-two positions in the logic and interface markets. He has also worked at Nestar Systems, a local area network start-up, and at IBM.

Kamdar has a B.S. in electrical engineering from S.P. University of India; an M.S. in computer science from Stevens Institute of Technology, Hoboken, New Jersey; and an M.B.A. in international management from Golden Gate University, San Francisco.

Copyright 2000 by Dedicated Systems Magazine - 2000 Q4 (http://www.dedicated-systems.com)