TUTORIAL

JAVA FOR EMBEDDED SYSTEMS

DEEPAK MULCHANDANI
Wind River Systems

Java requires infrastructure to run on embedded devices. This requirement must be traded against the powerful support for dynamic behavior that it offers embedded system developers.

Embedded systems have traditionally been differentiated from desktop systems on the basis of functionality: desktop systems provide a wide spectrum of technologies to serve a broad range of application needs, while embedded devices are fitted with just enough software to handle a specific application. The emergence of the Internet, however, is shifting the idea of fixed-function embedded devices toward more open systems offering some form of network connectivity.

Consumers are driving this shift by demanding Internet-style access to data. Additionally, certain Internet-based technologies fill gaps in existing embedded systems technology. For example, embedded Web servers and small Web browsers provide more flexible and less expensive user interfaces than traditional embedded system clients; and component software offers rich functionality when developed to work in conjunction with an embedded real-time operating system.

EMBEDDED JAVA

Embedded system developers have embraced Java over the past few years because the language is abstracted from underlying hardware, making its applications portable. With Java, developers can target a platform-independent API and migrate their applications to different devices without recompiling or reverifying them. Further, the object-oriented nature of Java supports well-structured development and software reuse. Finally, the Java Virtual Machine provides a dynamic platform with a secure execution environment.

Java is not just a programming language; it's a complete dynamic platform that requires extra infrastructure to run on embedded devices. Accordingly, its suitability to an application depends on the device requirements. Developers must consider resource, integration, and real-time performance requirements to determine Java's suitability to an application. For example, the bias of the language and platform toward 32-bit processors would not be appropriate for devices with 4-, 8-, or 16-bit CPUs. Issues related to memory management and the interpreted Java VM would hinder applications that demand strict real-time performance.

1089-7801/98/$10.00 ©1998 IEEE

Another consideration is the size of the complete Java platform. Many people believe that support of a Java VM is all you need to run Java applications. However, to compute the total size of the Java platform correctly, you must add the size of the Java VM, Java API package, Java application, and associated native-code libraries. Current Java API packages tend to be large, so the specific API selected for your device will significantly impact its size. Added components can also affect platform size.

Java nevertheless offers powerful support for embedded devices that must maintain some form of dynamic behavior. This article focuses on information pertinent to the runtime aspects of Java, specifically the Java APIs and VM and how they interact with a real-time operating system (RTOS). It also addresses architecture-specific issues such as memory management, RAM resources, and performance. These issues are critical to successfully adding or developing Java-enabled embedded devices.

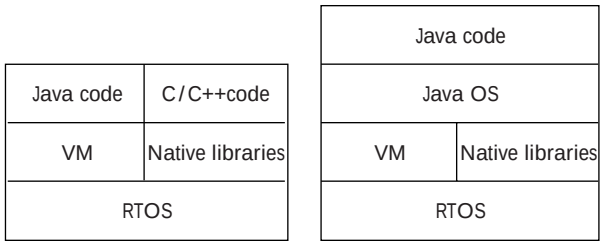

Figure 1. Two Java-enabled devices: on the left, a lightweight configuration; on the right, a solution based on Java OS.

Embedded Java Platform Architecture

Figure 1 illustrates two sample Java-enabled devices. The example on the left is a lightweight configuration, where the Java VM is integrated as part of the software environment. The configuration on the right is more common for devices like a network computer; it includes the JavaOS, which is well suited to environments that base the entire programming and user environments on Java.

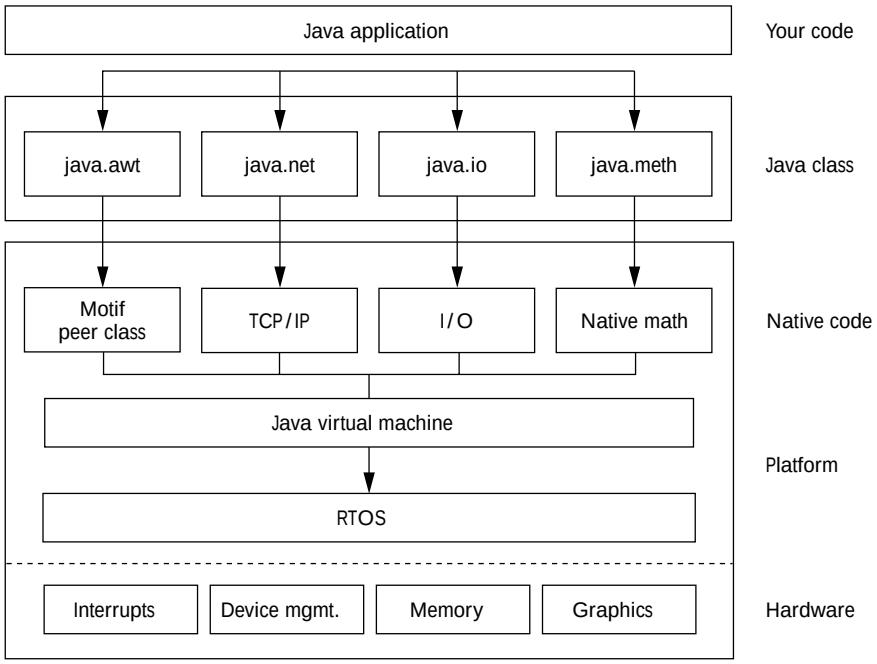

Figure 2. Architecture of the Java platform.

The Java runtime environment can be integrated into almost any embedded device. As Figure 1 shows, an RTOS is often used as a platform for running the Java VM. An RTOS establishes a customized foundation for your device that is highly reliable, scalable, and configurable. The RTOS supports multithreading (scheduling), memory management, networking, and peripheral management for the Java VM.

The Java VM includes interfaces that allow it to be readily integrated with an RTOS and other native libraries. Written in C, the Java VM is compiled with the RTOS and interprets the Java bytecodes as the application executes. There are two audiences for the system interfaces:

- Developers using the Java API layer to write applications; these users do not worry about hardware-specific functionality.
- System or firmware developers, who need to understand how the Java VM and Java packages work on top of their embedded device, specifically with the RTOS, software libraries, CPU, and memory configuration.

Figure 2 provides a more detailed picture of the Java platform architecture. It features five layers:

- Your code: the ZIP or JAR file that contains your application code.
- Java class: the Java API class package you receive with your Java VM. When you write Java appli-

cations you will reference the API calls provided by this class package. A majority of the functionality in this class package is written in Java itself, but some portions rely on native-code libraries. Examples are packages like "java.awt" (for graphics components) and "java.net" (for networking).

- Native code: any native-code libraries referred to by Java code in the class package. This layer co-exists with the Java VM and RTOS.
- Platform layer: the Java VM, which loads and executes Java classes from memory. The Java VM will utilize the facilities of the RTOS to manage the Java application operations.
- Hardware layer: the complete hardware infrastructure, managed by the RTOS. The RTOS resolves all runtime needs of the Java VM and also manages scheduling.

The Java VM depends on an RTOS to provide hardware-specific functionality for running Java applications. This functionality includes threading, memory management, and execution of native code. For example, the Java VM threads must be mapped onto the native RTOS threading model so that users can write Java threads. The RTOS handles all the low-level tasks such as boot sequence and initialization of the device, which isolates the Java VM and the application code from such activities.

The device in Figure 2 supports four Java API packages: java.awt, java.net, java.io, and java.math. The packages are layered directly (and transparently) on top of native-code libraries that are responsible for implementing the required functionality. This technique supports the abstraction of Java applications from platform-specific functionality and allows the Java API package to remain portable across platforms.

The Java APIs

Initially, JDK 1.0.2 was the only Java technology available for application development. Because Java was initially positioned for the Internet market, JDK 1.0.2 was designed to support a broad set of applications. As JDK 1.0.2 evolved to JDK 1.1, the Java API package was enhanced at a penalty of size. As a result, many embedded manufacturers recognized the benefits of Java but hesitated to pay the added resource requirements it places on their devices.

To address this weakness, **JavaSoft** developed four APIs that targeted different market segments: JDK, PersonalJava, EmbeddedJava, and Java Card. The sidebar "Java platform APIs" summarizes each one. These APIs provide a lower "barrier to entry" by eliminating the need to support Java APIs irrelevant to a specific market. They are designed to be upwardly compatible. For example, EmbeddedJava applications can run on a PersonalJava Java VM and JDK Java VM without any rewrite of the software.

Table 1 presents the resource requirements for each API. However, the numbers do not reflect the complete infrastructure required to run Java on embedded devices. For example, they do not include the underlying native infrastructure to run the Java VM and classes, which includes the RTOS, graphics stack (for AWT applications), and networking stack (to name a few). The sizes of these resources must also be factored in to get an accurate estimate of the final requirements for the Java API used for an application.

You must also analyze the sizing estimates provided by JavaSoft very carefully. According to Table 1, the requirements for the JDK runtime are 4 Mbytes of RAM and 4–8 Mbytes of ROM. However, the actual Java class package shipped with the JDK tools is approximately 9.5 Mbytes (in a Zip

JAVA PLATFORM APIs

JavaSoft has defined four JDK-based APIs that target different markets.

- **Java Development Kit.** Aims for use on desktop systems such as Solaris and Microsoft Windows. The JDK API is relevant to embedded systems that must provide a complete Java user environment. This would include applications such as Network Computers and set-top boxes. In general, only a subset of the JDK API is relevant to embedded systems. For more information, see the JavaSoft JDK page at http://java.sun.com/products/jdk/1.1/.
- **PersonalJava** (or Java Applet Environment). Targets the connected device market that needs a GUI and the capability to execute applets. Examples of such devices are smart phones and handheld PDAs. The first version of this platform, PersonalJava1.0, was released in December 1997. JavaSoft has recently announced a specification for PersonalJava1.1 as well. For more information, see the JavaSoft PersonalJava page at http://java.sun.com/products/personaljava/.
- **EmbeddedJava.** Targets low-end Java devices, specifically those lacking a general-purpose display (specifically java.awt). Developers can customize the Java platform to suit their needs. For more information, see the JavaSoft Embedded Software page at http://java.sun.com/products/embeddedjava/.
- **Java Card.** Targets the highly embedded/reduced-feature market such as smartcards. These applications often do not require the support of anything but the java.lang packages and ISO support packages for encryption. For more information, see the JavaSoft JDK page at http://java.sun.com/products/javacard/index.html.

Table 1. Operating parameters for Java environments defined by JavaSoft.

Operating Parameters	Java JDK	PersonalJava	EmbeddedJava	Java Card
RAM size	> 4 Mbytes	1 Mbyte	< 512 Kbytes	16 Kbytes
ROM size	4–8 Mbytes	< 2 Mbytes	< 512 Kbytes	512 bytes
Packages	All	Removed java.security Removed java.rmi Removed java.sql Removed java.text Modified java.awt Modified java.net Modified java.zip Optional java.math	The class package image will depend on the classes used by the application. Java.applet not allowed.	Java card classes Java ISO Classes
CPU type	100 MHz +	50 MHz +	25 MHz +	300 KIP

Note: Numbers do not reflect the complete infrastructure needed to run Java on embedded devices. For example, they do not include the size of the underlying native infrastructure required to run the Java VM and classes.

file format). In general, this poses a problem for devices supporting the complete JDK 1.1 implementation since class files can potentially take up a majority of the available storage on the embedded target (based on JavaSoft suggested parameters).

Ultimately, very few embedded devices benefit from availability of the complete JDK API set; most require a smaller, customized version of it. There are several ways to achieve this. First, developers should make sure they choose the proper API from the four offered. PersonalJava or EmbeddedJava were, in fact, defined solely to serve embedded system needs.

Table 2 (next page) provides a breakdown of the JDK 1.1.4 class package, along with a comparison of the requirements for the PersonalJava and EmbeddedJava APIs. The "Helper package" label for most of the Sun and Sunw packages means that they are mainly support packages for the Java packages. The "Depends on application" label for Java packages in EmbeddedJava means that developers can include the specific Java class subsets required on their embedded device. JavaSoft will provide development tools with the EmbeddedJava platform to help generate a customized Java class package. Unfortunately, developers writing applications for the JDK or PersonalJava platform cannot officially customize the Java class package to suit their device requirements.

As Table 2 shows, the Sun packages (especially Sun.io) contribute most of the size of the overall Java class package. Sun.io contributes more than half the size itself! The second step to saving resources is to scale out unnecessary Java packages. However, this depends on the needs of your Java application as well as the characteristics of your device.

There are two types of analysis for scaling Java class packages. The *static method* analyzes the application code and compiles a list of the referenced Java API classes and packages. Using the compiled list, you can go through the Java class package and remove the unnecessary packages and classes. The static method is quick, but prone to some error in missing classes that are not directly referenced through the source code. Examples are calls to Class.forName(), which loads classes using a classloader.

The *dynamic method* requires you to run the application code and generate a log of Java classes your application references. To perform this analysis, you can run your Java application on a Java VM with the -verbose flag enabled. The -verbose flag causes the Java VM to print out the list of classes it reads and verifies as it executes Java code. This technique will generate a fairly complete list of all the Java classes referenced in an application. The dynamic method is a lot more accurate than static analysis since it compensates for classes loaded from other sources such as the network. Using a combination of static and dynamic analysis, you can put together a concise list of requirements of your application.

From this analysis, it is possible to break up the JDK 1.1.4 classes.zip file into two separate archives: rt.jar (1.3 Mbytes) and i18n.jar (3.3 Mbytes). The rt.jar archive contains the core classes that a majority of Java applications need, and the i18n.jar file contains a majority of the Sun.io package. The Sun.io package contains a lot of classes specifically for internationalization, so breaking it out in a separate archive allows developers to include such sup-

port on an as-needed basis. The impact of a 1.3-Mbtye class package is a lot easier to absorb on an embedded device. Note that the 1.3 Mbyte figure is for the JDK class package. The PersonalJava and EmbeddedJava class package will be much smaller.

BASICS ON NATIVE METHODS

So far we have focused on the Java platform and how it integrates into embedded devices. The architectures of the Java VM and APIs make them easy to add to suitable embedded devices. However, once the integration of the Java VM and other components is complete, specific technical issues must be answered regarding how Java affects the operation of an existing device and what constraints it places on real-time performance.

As pointed out earlier, the Java class package depends on native software libraries for specific functionality. The mapping of the Java class package to native software libraries is achieved using a well-defined protocol, the Java Native Interface. JNI provides a framework by which the Java VM can reference and execute native methods.

Briefly, the first step is to write the Java class, defining the native methods with the Java "native" keyword. For example

```
public class NativeTest
{
native int foo();
native int bar();
}
```

Once you have compiled this Java class, you need to run the javah tool on the resulting class file. The javah tool is shipped with the JavaSoft JDK; it will generate a C source file that contains stub definitions for calling native code segments. For the two methods shown above, the stub functions will be

```
Java_NativeTest_foo_stub()
Java_NativeTest_bar_stub()
```

You will have to implement the functions, NativeTest_foo() and NativeTest_bar(), so that execution of the Java application will route any

Table 2. Breakdown of JDK 1.1.4 classes with requirements for PersonalJava and EmbeddedJava.

Java Package Name	JDK 1.1.4 Size (in bytes)	PersonalJava 1.0	EmbeddedJava 1.0
Java.awt	330,849	Required	Depends on application
Java.io	111,875	Required	Depends on application
Java.lang	171,494	Required	Depends on application
Java.text	606,272	Not required	Depends on application
Java.util	101,943	Required	Depends on application
Java.applet	3,931	Required	Not allowed
Java.beans	54,140	Required	Depends on application
Java.net	60,372	Required	Depends on application
Java.math	31,731	Optional	Depends on application
Java.rmi	46,595	Unsupported	Depends on application
Java.security	40,960	Partly optional	Depends on application
Java.sql	36,624	Unsupported	Depends on application
Sun.applet	106,283 (helper)	Helper package	Helper package
Sun.awt	418,759 (helper)	Helper package	Helper package
Sun.io	5,541,077 (helper)	Helper package	Helper package
Sun.audio	18,830 (helper)	Helper package	Helper package
Sun.misc	53,898 (helper)	Helper package	Helper package
Sun.beans	19,029 (helper)	Helper package	Helper package
Sun.net	150,150 (helper)	Helper package	Helper package
Sun.rmi	221,062 (helper)	Unsupported	Helper package
Sun.security	178,743 (helper)	Helper package	Helper package
Sun.jdbc	152,157 (helper)	Unsupported	Helper package
Sunw.io	231 (helper)	Helper package	Helper package
Sunw.util	537 (helper)	Helper package	Helper package

method foo() calls to Java_NativeTest_foo_stub(), which in turn will call NativeTest_foo().

The stub architecture has been in place since JDK 1.0.2 and was enhanced in JDK 1.1 with a more powerful implementation. The JavaSoft Web site gives more detailed information about using Java NI for implementing native methods.

Using the description of the Java NI, the java.awt package works on an embedded device. It routes calls to methods in the java.awt package to the underlying X Windows/Motif libraries via the Motif Peer Class. This class is a set of Java interfaces that allow AWT components to be mapped to underlying graphics libraries present on the device. As a result, when a developer uses the following statement in a Java application

```
Frame f=new Frame();
```

the call is translated into a sequence of X Window/Motif calls that are responsible for actually creating and rendering the frame on the screen. In this scheme, the Java VM is responsible for executing only the Java code; when the Java VM encounters a native call, it hands the call off to the native software library for execution. Another example of a similar Java package mapping is the java.net package, which is layered on top of a platform-specific TCP/IP implementation.

ROMIZERS FOR COMPRESSION

The Java VM places two types of runtime requirements on RAM resources in the embedded device: storage of Java objects (Java Heap) and execution of Java bytecodes (Java Memory Pool). When Java applications execute, the Java VM copies the bytecodes from ROM (or other storage) into RAM. This is required since the Java VM has to verify each Java class file prior to execution. When the Java VM verifies a section of code, it converts all the bytecodes into an alternate "quick" format. This technique ensures that the Java VM does not execute malicious code as it loads applications not only from ROM but also from sources such as the network. The Java VM keeps the verified code in RAM, which can quickly drain embedded device resources. Additionally, if the device needs to support multiple applications in memory simultaneously, performance may degrade.

This dependency on RAM poses a problem for embedded devices, which often rely extensively on ROM for cost reasons. To solve the problem and reduce the overhead incurred, new embedded-friendly tools called ROMizers are being developed.

ROMizers are embedded-friendly tools that let Java VMs execute precompiled classes from ROM.

ROMizers "precompile" Java classes and allow Java VMs to execute these classes directly from ROM. This reduces the copy-verify-execute cycle. It also reduces the interpretation overhead.

The ROMizer technology mimics the operation of the Java VM on a host computer and precompiles Java classes into an alternative native representation (usually a C data file). The ROMizer loads, verifies, and links entire sets of Java class files (similar to the Java VM) and generates output that can be linked in with an RTOS image. ROMizers can be used to convert selected pieces of the Java platform into a format more suitable for embedded devices.

Depending on the features provided, a ROMizer can perform the following tasks:

- Merge the "constant pool" data section of each class file. This removes redundant information and points all references to the constant pool so they point to the same constant pool section.
- Convert all the bytecodes to their corresponding "quick" bytecode format. This removes the need for the Java VM to verify the bytecodes when it is executing ROMized Java code. By performing the verification ahead of time, the Java VM does not have to copy the bytecodes into RAM. Rather, it can execute them out of ROM.
- Create runtime images of the class structures for the Java VM. This provides a runtime enhancement as the Java VM does not have to create these structures upon startup.
- Resolve the class hierarchy on the target. This allows you to place only the required subset of Java classes on your embedded device.

Figure 3 compares running a small Java application that creates a single Frame on the screen without and with ROMized classes. (The Java VM used is running on **Wind River System's VxWorks RTOS**.) The ROMized Java VM needs to load and verify only two classes before starting execution of the Java application. By comparison, without ROMized classes, the Java VM must load and verify 44 classes before it actually loads the first class for the appli-

cation. This difference will be reproduced throughout the life cycle of the running application. Without ROMized classes, every reference to classes from the java.lang, java.io, java.util, java.awt, or sun.awt will have to be loaded into memory and executed. Calls to ROMized classes, however, do not need to be loaded in the ROMized Java VM. This technique definitely speeds application performance.

Note that ROMizing is just one technique that developers can use to improve the Java runtime performance. Other techniques such as Java native compilation and JIT compilers are quite popular as well. But there is a unique difference between the techniques. ROMizers are useful for developers who wish to place more of the Java infrastructure in ROM for cost reasons. The important distinction is that ROMizers do not perform any optimizations on the Java classes while building the ROM image. Therefore, ROMized classes should run and behave exactly the same as interpreted classes. On the other hand, JIT compilers and Java native compilers usually take Java bytecodes and convert them (with optimizations) to native code. The native code generated is not guaranteed to run or behave the same as the original Java classes.

ROMizer can also help by reducing the startup time of the Java VM. By design, the virtual machine loads and verifies classes from the java.lang, java.io, and java.util packages when it first starts up. If you were to ROMize these specific packages, the Java VM would locate these precompiled packages on startup and immediately begin executing the Java application. The output from the ROMizer is a C source file, which basically contains data structures and character arrays that resemble the format of the Java classes in RAM. In Figure 3, the compiled ROMized class package image size is approximately 1.3 Mbytes, created from compiling the generated ROMized C file.

ROMizers provide an elegant, low-risk mechanism to reduce runtime resource requirements of Java applications. They are a host-based tool rather than a target-resident component, so they give the developer a lot more control in making appropriate trade-offs than other performance enhancement tools do. The suitability of Java native compilers and JIT technology for embedded systems will be discussed in further sections.

MEMORY MANAGEMENT AND GARBAGE COLLECTION

Java applications do not have any explicit mechanisms for memory management. The allocation and deletion of objects is performed with cooperation between the Java VM and the garbage collector. On

Application Execution Without ROMized Classes

```
All Java packages were run interpreted.
Target CPU: Intel x86
ROMized class package image size: 0 bytes
Size of JDK Class package used:1,351,207 bytes
Overhead incurred: 0 bytes
Java("-verbose Frame1")
value=0=0x0
[Loaded java/lang/Thread.class from /java/lib/rt.jar]
        //*37 additional [Loaded java . . .] instructions.*//
[Loaded sun/io/CharToByteConverter.class from
/java/lib/rt.jar]
[Loaded sun/io/CharacterEncoding.class from /java/lib/rt.jar]
[Loaded java/util/Locale.class from /java/lib/rt.jar]
[Loaded sun/io/CharToByte8859_1.class from /java/lib/rt.jar]
[Loaded java/io/BufferedWriter.class from /java/lib/rt.jar]
[Loaded java/lang/Compiler.class from /java/lib/rt.jar]
[Loaded Frame1.class from /java/lib/frame.jar]
<output deleted>
```

Application Execution with ROMized Classes

```
The following packages were ROMized: java.lang,
java.io, java.util, java.awt and sun.awt. The rest were run
interpreted.
Target CPU: Intel x86
ROMized class package image size: 1,302,823 bytes
Size of JDK class package used: 823,287 bytes
Overhead incurred due to ROMizing: 774,903 bytes
Java("-verbose Frame1")
value=0=0x0
[Loaded java/lang/NoClassDefFoundError.class from
/java/lib/rt.jar]
[Loaded sun/io/CharToByte8859_1.class from
/java/lib/rt.jar]
 [Loaded Frame1.class from /java/lib/frame.jar]
<output deleted>
```

Figure 3. Example output from a Java application that creates a single frame on screen: (left) without ROMized classes and (right) with ROMized classes. Note that the code on the left has been abbreviated.

the one hand, this isolates the developer from the process of keeping track of memory use in an application. On the other hand, the garbage collection process is nondeterministic and cannot be scheduled. As a result, you cannot be sure exactly when the Java VM will clean up the object heap, since it runs the garbage collector as a low-priority thread and does not include mechanisms to directly control the behavior of the garbage collector. The Java API package does provide a method, System.gc(), to call the garbage collector, but this method does not guarantee anything; it merely sends the Java VM a suggestion to clean up the object heap.

The garbage collector provided by JavaSoft in JDK 1.1 is based on the "mark and sweep" algorithm. When it is time to clean up dead objects, the garbage collector locks out all other Java threads and then starts marking up the object heap. Once all the objects eligible for deletion have been marked, the garbage collector sweeps away the dead objects. The Java VM internally maintains state information on the object heap.

The information is based on ratios and divided into three categories: green, yellow, or red state. When there is no shortage of memory, the Java VM operates in the green state. When memory drops below a specified threshold, the Java VM enters the yellow state, which means it should schedule a garbage collection soon. If the application keeps executing, the Java VM enters the red state, which means the object heap is almost saturated and garbage collection needs to be done immediately. If the Java VM runs out of memory, then most likely a Page Fault will occur on the embedded device.

Figure 4 presents an example to help understand the architecture of the Java VM threading model. The example is based on running the HotJava 1.1 browser on Wind River System's JDK 1.1.4 VxWorks-based Java VM. The VxWorks Java VM supports the "native threads" package, which means that each Java thread is mapped directly to underlying VxWorks tasks. This implementation allows Java threads to freely interoperate with other VxWorks-specific system tasks. Also, the RTOS can manage resources for the entire device rather than handling only the real-time aspects of the device.

The tasks highlighted in bold are the specific tasks created by the Java VM for garbage collection and memory management. Specifically, the Async Garbage thread and Idle thread combination allows the RTOS to schedule a garbage collection only when the system becomes idle. The Red Alloc thread allows the Java VM to manage the current

```
-> l
NAME          ENTRY      TID      PRI   STATUS   PC       SP       ERRNO    DELAY
tJreaper      0x1636c0   203e5e4  116   PEND     2536c4   203e590  3d0001       0
Red Alloc     S0x14bfc8  1eaf620  117   PEND     2536c4   1eaf02c  16           0
AWT-Finaliz   0x14bfc8   1bec7a0  118   PEND     2536c4   1bec1ac  16           0
Alloc State   0x14bfc8   1ea5470  118   PEND     2536c4   1ea4e7c  16           0
Request Pro   0x14bfc8   1ebe2e0  120   PEND     2536c4   1ebdcec  3d0002       0
tJmain        0x154208   26a9ec0  122   PEND     2536c4   26a9e50  0            0
AWT-EventQu   0x14bfc8   1f93b40  122   PEND+T   2536c4   1f9354c  d          241
Lite-AWT-In   0x14bfc8   1f89990  122   PEND     2536c4   1f8930c  3d0002       0
Lite-AWT-Ev   0x14bfc8   1f7f7e0  122   PEND     2536c4   1f7f0ec  3d0002       0
Thread-3      0x14bfc8   1bd4400  122   PEND+T   2536c4   1bd3df8  16         292
Screen Upda   0x14bfc8   1d84b90  123   PEND+T   2536c4   1d8459c  3d0004    1846
HotJava App   0x14bfc8   1ce07c0  123   PEND     2536c4   1ce01cc  3d0002       0
Thread appl   0x14bfc8   1c08dc8  124   PEND     2536c4   1c087d4  3d0002       0
Thread-0      0x12dc40   2022678  126   PEND     2536c4   20225b4  3d0002       0
Async Garba   0x1544d4   201a450  126   PEND     2536c4   201a3d0  3d0002       0
Idle thread   0x1542b4   2012228  127   DELAY    252bbe   20121c4  3d0002       6
value=0=0x0
```

Figure 4. Example list of all application threads running the HotJava 1.1 browser.

Table 3. Average execution-time allocation for Java apps.

Application Function	Execution Time
Allocation and garbage collection	20%
Thread synchronization	19%
Running native methods	1%
Bytecode interpretation	60%

(Source: "HotSpot: A New Breed of VM," *JavaWorld*, Mar. 1998)

state of memory in the device. The tJreaper task cleans up VxWorks stack memory for each Java thread after it has terminated.

The garbage collector poses a problem in that it tends to lock down all the Java threads while it cleans out unused objects from the system. The lockdown ensures that the state of the heap remains stable, but it adversely affects real-time performance of Java applications. A simple workaround to this problem is to continue to allow the RTOS to manage operations of the device—interrupts, device management, and task scheduling—with those tasks running at a higher priority. A Java application should not be written to rely on any RTOS or system-specific functionality, nor should it make any assumptions about scheduling or timing. Any specific portions of the Java application that are performance-critical can be implemented in native code (using Java native interface techniques or other Java development tools).

The JDK 1.3 (HotSpot) release will include a new-generation garbage collector, called the "train" algorithm. It allows interruptions as the garbage collector cleans up dead objects. This will improve programmer control in the overall Java VM platform.

At this point a solution for the PersonalJava and EmbeddedJava platforms is not known.

JAVA PERFORMANCE

Java provides its cross-platform promise using a combination of the Java class file format and the Java VM. The Java class file results from the translation of Java source code to bytecodes on the host computer. Once the application is in bytecode format, it can be transmitted to any Java-enabled target for execution.

On the target device, each Java bytecode is executed using the interpreter loop embedded in the Java VM. The Java VM interpreter loop is responsible for translating each Java bytecode into equivalent native-code functions while the application executes. Java bytecode applications tend to run slower than native applications since optimizations cannot be performed at runtime and the translation is a fairly serial process. Based on early benchmarks, Java applications tend to run 20 to 30 percent slower than applications written in C. This is not a significant problem for small applications, but it can be a serious issue when it comes to running any sizable applications.

Table 3 provides average figures from a sample of typical applications performed at Sun Microsystems. As shown, Java applications spend almost 40 percent of their execution time in functions that cannot be directly optimized by performance enhancement techniques: garbage collection and thread synchronization. The table shows the time spent in native code execution at 1 percent of overall execution time. The percentage may be much higher for average embedded Java applications, which must maintain close interaction with the underlying hardware.

A specific optimization technology to improve Java application performance can only be applied to speed the interpretation of bytecodes. The techniques to choose from include the ROMizing technology presented earlier. Other solutions focus on three main areas: Java native compilers, JITs, and Java Chips. ROMizers can also be considered a performance enhancement tool but by design they are mostly used to compress the RAM requirements of the Java VM.

Java Compilers

Using Java Compilers allows developers to write interoperable Java, C, and C++ code. This approach is similar to standard compilation techniques that you may use to build your application today. The benefits are the increased performance by having your system rely more on the mature native compilers rather than bytecode-based Java compilers. By converting the Java into native code, you can reduce the interpretation overhead of the Java VM. This approach does not eliminate the need for a Java runtime component such as a Java VM. Java applications rely on garbage collection and threading so minimal runtime support will still have to be included.

On the downside, you lose the benefits of the dynamic Java platform as soon as portions are converted to native code. Also, for device updates you will have to rely on native code patches rather than using the Java VM to selectively update certain aspects of your application.

Just-In-Time Compilers

Just-in-Time Compilers are a popular approach to improving Java performance. Unlike traditional compilers, this runtime component replaces the

interpreter loop at the heart of the Java VM. JIT compilers convert Java bytecodes to native code while an application is running. Once a specific bytecode sequence has been converted, the remaining executions will run almost as fast as native code.

JITs capitalize on the locality-of-reference principle. If the Java VM can reuse translated code, then it will not be constantly compiling the Java bytecodes into native code. There are some situations, however, where the JIT approach does not work as well:

- If the application performs operations such as graphics or floating-point operations, then JIT compilers do not give a large performance boost.
- If the application constantly creates objects, the garbage collector would interfere with the operation of the JIT.
- If performance is mostly a function of optimizations, JIT compilers are limited by their status as a runtime component.

In these situations JITs may not perform much better than the default Java VM interpreter loop.

JIT compilers can also take up large amounts of RAM, since they translate bytecodes into native code at runtime. One last consideration: the JIT needs to be reliable as it will be generating code while the application executes.

Java Chips

Java chips have been offered as a hardware option to improve Java performance. Some reference chips currently available are **Sun Microsystems picoJava** and **microJava** processors and **Patriot Scientific PSC1000**. These processors are designed to replace the Java VM interpreter loop and provide other features to boost performance. You still need a Java VM (without interpreter loop) and RTOS to put together a complete Java platform, so the Java chips are primarily performance enhancers.

Java chips are being offered as an alternative to general-purpose CPUs such as x86, ARM, or PowerPC. However, since Java chips focus exclusively on improving raw performance of Java bytecodes, the native code portions of your application may incur a penalty. If your application contains any legacy code (written in C or C++), it will have to be translated into Java bytecodes.

At this time, many details regarding Java chips remain unknown. It is hard to say how such chips will handle the processing requirements of Java applications and native code written in C and C++. One proposed model is to use Java chips as bytecode accelerators for general-purpose CPUs. This provides the benefits of native code execution performance as well as dramatically improved Java application performance.

CONCLUSIONS

Developers can combine different approaches to improve the performance of Java applications. Trade-offs have to be designated based on where you would prefer to optimize the Java software: host or target device. Java compilers and ROMizers are host-based tools that can boost performance by precompiling Java code. JITs and Java chips are target-based optimization tools, which can impose extra memory requirements on your target device, but still provide a dynamic platform.

The integration of Java into existing embedded designs is finally allowing development of embedded devices that offer Web connectivity, multimedia content, and dynamic extensibility. The RTOS from which the Java VM derives some of its functionality allows the Java VM to abstract itself from device-specific architectures. In exchange, the Java VM provides a secure wrapper around the RTOS, keeping malicious content from interfering with the operation of the device.

International Data Corp. estimates that there will be 22 million non-PC Internet access devices by the year 2000. It's a good guess that not many of these devices will look or operate the same way. However, it *is* a good guess that a large part of the differentiation will be based on user-specific requirements: cost, look and feel, multilanguage support, battery life. In the same way that an RTOS operates as the brains behind today's embedded devices, a Java VM tightly integrated with an RTOS will potentially become the standard software platform of tomorrow. ■

Deepak Mulchandani is the engineering manager for embedded Internet products at Wind River Systems. He holds patents related to debugging technology. Readers may contact him at dmm@wrs.com.

URLs for this tutorial

JavaSoft • java.sun.com
Patriot Scientific shBoom • www.ptsc.com/news/
Sun Microsystems picoJava and microJava • www.sun.com/microelectronics/
Wind River System's Embedded Internet Page • www.wrs.com/embedweb/index