Java and Real Time Storage Applications

Gary Mueller
195 Garnet St
Broomfield, CO 80020-2203
garymueller@qwest.net
Tel: +1-303-465-4279
Janet Borzuchowski
Storage Technology Corporation
2270 South 88th Street
M. S. 4272
Louisville CO 80028
janetborzuchowsk@qwest.net
Tel: +1-303-673-8297

Abstract

Storage systems have storage devices which run real time embedded software. Most storage devices use C and occasionally C++ to manage and control the storage device. Software for the storage device must meet the time and resource constraints of the storage device. The prevailing wisdom in the embedded world is that objects and in particular Java only work for simple problems and can not handle REAL problems, are too slow and can not handle time critical processing and are too big and can't fit in memory constrained systems.

Even though Java's roots are in the embedded application area, Java is more widely used in the desktop and enterprise environment. Use of Java in embedded real time environments where performance and size constraints rule is much less common.

Java vendors offer a dizzying array of options, products and choices for real time storage applications. Four main themes emerge when using Java in a real time storage application; compiling Java, executing Java with a software Java Virtual Machine (JVM), executing Java with a hardware JVM and replacing a real time operating system (RTOS) with a JVM.

The desktop and enterprise environment traditionally run Java using a software JVM that has been ported to a particular platform. The JVM runs as a task or process hosted by the platform operating system. With the performance and memory available on most workstations and personal computers, running an application on a software JVM is not an issue. However, many desktop and enterprise applications are not faced with the critical time and space constraints of an embedded application. Because of these constraints, running an embedded application on a software JVM incurs the additional overhead of software running software. Although it might be possible to run some embedded applications on a software JVM because of the tremendous speed of some processors, for most embedded applications, this configuration will not met timing or space constraints.

For a real-time storage application, running a JVM in software is typically only used for tasks which are not time critical. Typical tasks include hardware configuration,

maintenance and diagnostics, or upgrading or loading new code. For these tasks, a software JVM can meet the performance and space requirements. The software JVM typically runs as a low priority task. Other time critical tasks are written in C or C++ and do not use the intermediary JVM.

Compiled Java is an acceptable option since the JVM is eliminated and the functionality of the JVM such as garbage collection is wrapped into a set of runtime libraries. Compiling Java gives you the benefit of an object-oriented language without the performance penalty of an interpreted language.

The ultimate in speed and performance is attained when the JVM is cast in silicon. Several hardware vendors are planning or currently offering coprocessors or custom chips that execute Java directly in hardware.

Since the JVM provides the runtime environment for Java, in essence an operating system, one interesting approach is to use the JVM as a replacement for a RTOS.

This paper discusses the advantages and disadvantages of each approach as well as specific experiences of using Java in a commercial tape drive project.

1 Why Java for Real Time Storage Systems?

Java is an object-oriented language which gives you all the advantages of object technology, including faster delivery to market, more maintainable code, and easier adaptation to change. Java enforces the discipline of object design. Using Java in an embedded environment presents several challenges. Embedded applications have both functional and timing requirements and run in resource constrained environments. Java must meet the performance and space requirements of the embedded application. Some questions to answer include:

- Space the final frontier, will the JVM and class libraries fit?
- Performance, can the JVM run fast enough to meet hard real time deadlines?
- Scheduling, is the JVM deterministic and can non-deterministic tasks, such as garbage collection be scheduled?

2 Java Basics

Java is both a language and an environment which supports compilation and execution of the language.

Java, the language, supports single inheritance, polymorphism and other object concepts. Java is *compiled* to an intermediate language, Java byte codes, the assembly language for the JVM. The output of the Java compiler is a class file, which contains the Java bytecodes.

Java, the environment, is a virtual machine that has been ported to many operating systems and processors. The JVM interprets and executes the Java bytecodes and is usually written in C or C++. The JVM loads the Java class with a class loader, links the class files, verifies the bytes in a class file for correctness, prepares the class files for execution, initializes the class, resolves method references and determines when to garbage collect unused classes. A typical Java environment is shown in Figure 1.

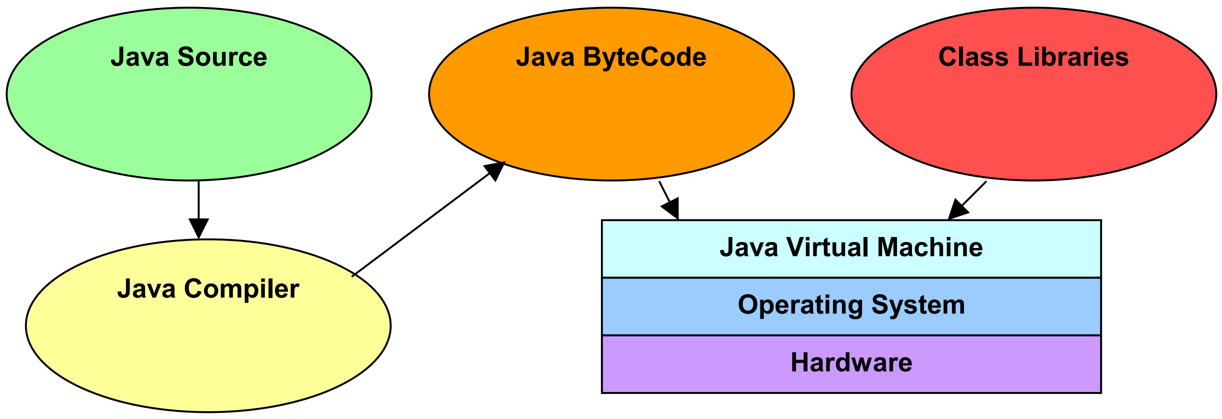

Figure 1 - Java Runtime Environment

3 Flavors or Java for Embedded Systems

There are four flavors of Java for embedded systems:

- Software Java Virtual Machine
- Compiled Java
- Hardware Java Virtual Machine
- Java as a Real Time Operating System

3.1 Software Java Virtual Machine

A software JVM is an application, process or task that typically is hosted by another operating system. Software JVMs are typically used for desktop or enterprise applications. Most desktop applications execute Java using a JVM running as a process or task on the desktop. Browsers execute Java with a JVM in the browser. This is the *classic* use of Java.

Since Java is interpreted by another program, the software JVM, there is a concern about the performance of the application which the JVM is executing. In particular, embedded applications must execute within specific time frames. Executing the embedded application on the JVM which itself is being executed raises the question of how fast the embedded application is executing and whether it can meet its required deadlines. One might speculate that there may exist embedded applications which given enough hardware horsepower will meet their required deadlines with a software JVM.

For those embedded applications which rely on and use a RTOS, a software JVM could be executed as a set of tasks or processes on the RTOS. Assuming the JVM tasks have a sufficient priority, some non real-time or slow real time embedded application tasks could be run using a software JVM such as:

- Hardware configuration

- Maintenance and diagnostics
- Code upgrades and loads

This method of executing Java is typical for desktop and enterprise applications where performance, although a concern, is not a driving factor. An example of this flavor of Java is WindRiver® Personal JWorks™ [1].

3.1.1 WindRiver® Personal JWorks™

As shown in Figure 2, Personal JWorks™ includes the PersonalJava Core Libraries, the JVM, the VxWorks Real Time Operating System (RTOS), the Supporting Native Libraries, a board support package (BSP) and device drivers for the particular processor and RTOS.

The PersonalJava Core Libraries include the applet, awt, beans, io, lang, math, net, rmi, security, sql, text and utl packages. The Personal JWorks™ application environment is based on the Java Development Kit 1.1.8 and adds security as specified in the Java 2 Software Development Kit, version 1.2.

Personal JWorks™ supports and fully implements the Abstract Windowing Toolkit (AWT) and fully supports the Java AWT graphics system. The WindRiver Media Library (WindML) glues the Personal JWorks™ environment to an applications graphics hardware. WindML supports 2D graphics primitives, fonts and provides audio and video support.

Personal JWorks™ uses a software JVM that runs as a set of tasks on VxWorks®. Using the Java Native Interface (JNI), JVM services such as thread and memory management (garbage collection), synchronization mechanisms, networking and graphics are mapped to VxWorks tasks through the Supporting Native Libraries. As a result, the VxWorks scheduler is able to prioritize and preempt the Java threads in the

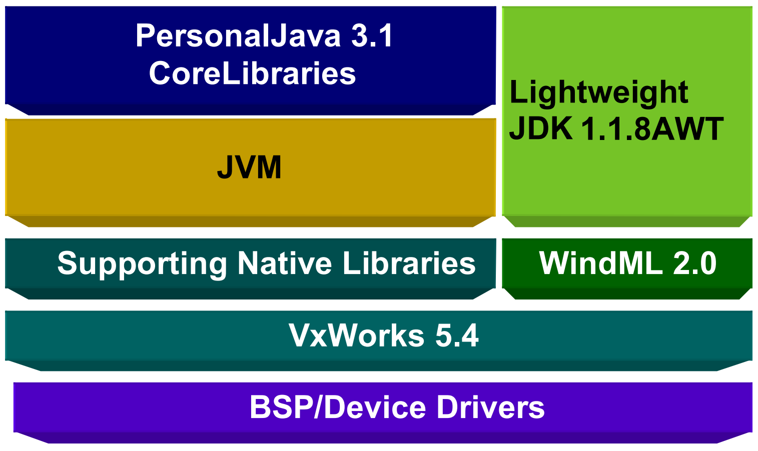

Figure 2- Personal JWorks™ Architecture

same way as it does VxWorks tasks. Although Personal JWorks™ does not provide real-time response, any VxWorks native task placed at a higher priority than a Java thread will execute without impact. Personal JWorks™ thus retains the determinism of VxWorks®. Using the JNI, Personal JWorks™ applications can access any C/C++ function in the VxWorks operating system including system calls.

3.2 Compiled Java

Compiled Java removes the environment portion of Java and treats Java as a language. Java is simply compiled to either native code or to an intermediate language such as C or C++. Compiled Java provides the benefit of an object-oriented language without the performance penalty of an interpreted language. Garbage collection and other JVM services are implemented through runtime libraries. Two examples of compiled Java are the Gnu Compiler for Java™ and WindRiver® Diab™ FastJ®.

3.2.1 Gnu Compiler for Java™(*gcj*)

Java applications are compiled and linked with the *gcj* runtime library, *libgcj*. The *libgcj* supplies the core classes, the garbage collector and the bytecode interpreter. The *libgcj* must be ported to the processor in your environment. The *gcj* allows three types of compiling:

- Java source code to native machine code
- Java source code to Java bytecode
- Java bytecode to native machine code

3.2.2 WindRiver® Diab™ FastJ®

FastJ® compiles C, C++ and Java source code to native machine code. As shown in Figure 3, the FastJ® compiler compiles, optimizes and generates assembly code for the desired target CPU and runtime environment using the Global Optimizer, Code Selector and Code Generator. External assembly source code and external libraries may be assembled and linked with the C, C++ and Java code. To reduce code size only needed core libraries may be configured. The Assembler together with the Linker produce an ELF format executable image for the desired processor.

FastJ® supports three memory management options:

- Explicit memory management, similar to C/C++, eliminates garbage collection.
- Standard, non-incremental garbage collection, runs when memory is low or explicitly called.
- Preemptive, incremental garbage collection, runs as a preemptable, low priority background task.

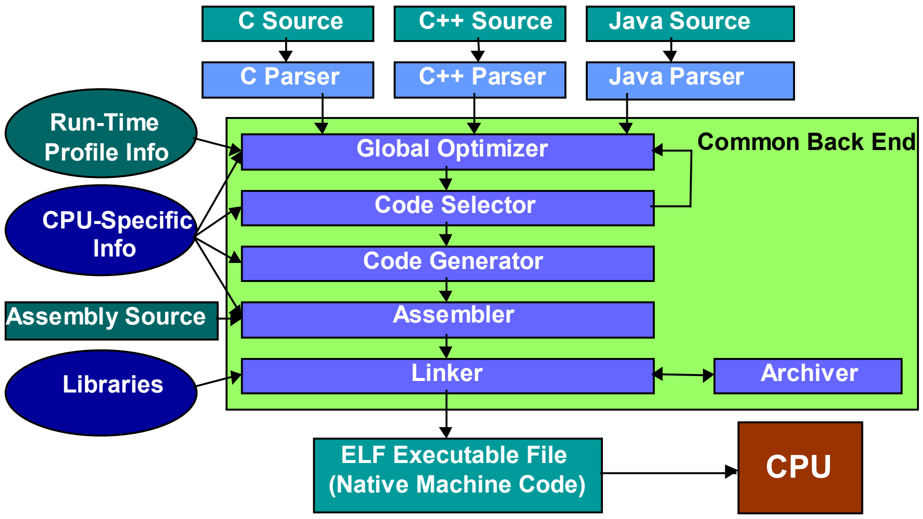

Figure 3 - FastJ® Compiler Architecture

3.3 Hardware JVM

The ultimate in performance is achieved by executing or running the JVM in hardware. The JVM is implemented in silicon as either a co-processor or separate processor on a custom chip. Specially designed or custom hardware is required which directly executes the Java bytecodes. This is similar to assembly code being executed on a particular processor. Several chip vendors including ARM from England, Ajile from the United States, Vulcan Machines LTD from England and NTT Docomo from Japan offer hardware JVMs. [2]

Several variations of the hardware theme are currently available. Some hardware implementations use a co-processor to execute Java bytecodes. Other implementations use specialized hardware, which is called when Java bytecodes are detected. An example of a hardware JVM is the ARM® Jazelle™[3].

3.3.1 ARM® Jazelle™

Jazelle™ is a product from ARM®, which includes a hardware JVM for the ARM® family of processors and a runtime environment to support Java applications. The Jazelle™ runtime architecture, as shown in Figure 4, allows Java applications to access the Java Class libraries available in the particular Java development kit, either the Java 2 Enterprise, Standard or Micro Edition. Each edition of Java has a virtual machine which executes the Java bytecodes. Jazelle™ currently supports the pJava, KVM and CVM virtual machines. Jazelle™ provides a Java Technology Enabling Kit for porting other VM's.

The Jazelle™ Supporting Code replaces the Java virtual machine interpreter loop and enables execution of the Java bytecodes directly in hardware. A condition bit in a new ARM® instruction puts the processor in the Java state. The processor then executes the Java byte code directly in hardware. Jazelle™ supports execution of both Java bytecodes and ARM® machine codes. This allows existing application written in C and C++ to continue to execute alongside the Java applications. The main difference between a software JVM such as Personal JWorks™ and a hardware JVM such as Jazelle™ is how the Java bytecodes are executed. In Personal JWorks™, the bytecodes are translated to native machine code and then executed. With Jazelle™, the bytecodes are executed directly in hardware.

Since the JVM must be supported by the underlying RTOS, Jazelle™ also supports WindowsCE, SymbianOS, PalmOS, Linux and many real time and proprietary operating systems.

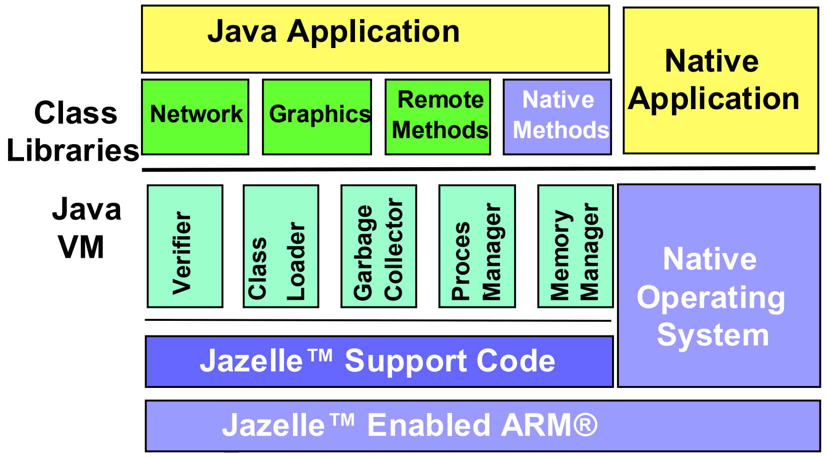

Figure 4 - Jazelle™ Run-Time Architecture

3.4 Java as a Real Time Operating System

An interesting variation is viewing the JVM as an operating system. The JVM is the RTOS. Since the JVM is essentially a machine, simulated or executed on another machine, it makes sense to eliminate the other machine and execute the JVM directly on hardware. An example of this is Jbed™ from Esmertec [4].

3.4.1 Esmertec ™

Jbed™ combines the JVM and a real time operating system into a single entity. Jbed™ has a four layer architecture. The Java applications have access to lang, io, util as well as the connection framework in the javax.microedition package and is PersonalJava 3.0 (JDK 1.1) compliant. As shown in Figure 5, Jbed™ supports many of the popular

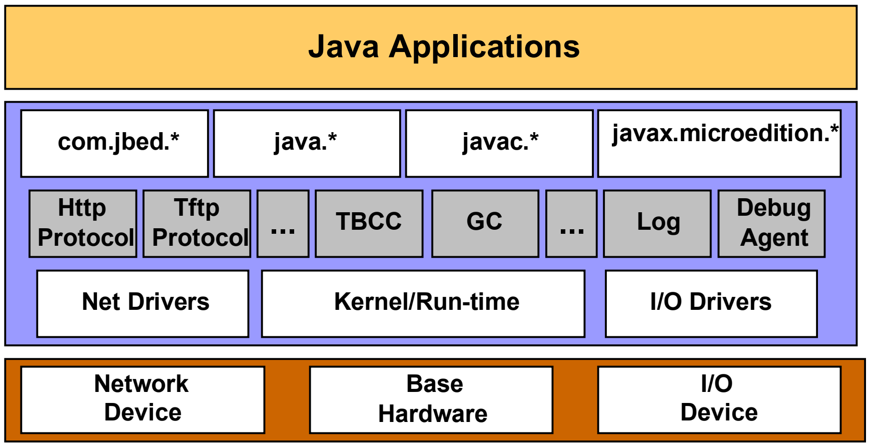

Figure 5 - Jbed™ Run-Time Architecture

Internet protocols such as HTTP, TFTP, TCP/IP, PPP and UDP. JVM services such as garbage collection (GC) are supported without the intermediary JVM. Jbed™ does not execute or interpret Java bytecode. Instead, bytecode is translated into fast machine code prior to downloading or upon class loading with the Way Ahead of Time compiler and the Target Bytecode Compiler (TBCC). This avoids the speed and size penalty of a JVM, yet stills provides advanced Java features such as dynamic code loading and automatic garbage collection. Jbed™ extends the Java thread package to provide priority based scheduling using the earliest deadline first algorithm. A device driver support package supports driver development in Java. Thus, the entire application including device drivers can be written in Java.

4 On the Road to Java

The 9840 and 9940 family of StorageTek tape drives use an ARM7® 32 bit processor, with 2-4MB of RAM for loading the code image. A 32MB - 64MB data buffer is used for data transfer and the drives support the SCSI, ESCON, and Fibre Channel interfaces. Specialized Application Specific Integrated Circuits (ASICs) are used to control the tape drive. All of the code is written in C with Vertex serving as the RTOS.

C++ and object design have been introduced into the time critical tape microcode. Initially, the classes have been written in C++ and are mirrored in Java for unit testing. The Java classes form the basis for a hardware simulator.

Since FastJ™ is similar to current development environment, FastJ™ will be the first step to introducing Java in our real time system. It is the least disruptive and does not require hardware changes. FastJ will be used to compile the Java classes used in the hardware simulator and tape microcode. Since the current RTOS is old, the next step will be to investigate Jbed™ which is a Java RTOS, a combination of hardware/software. Finally, since Jazelle™ requires hardware changes, the last step will be Jazelle™.

5 Summary

Recently, there has been a resurgence in the use of Java for embedded systems. Options ranging from software Java Virtual Machines offered by real time operating system vendors to chip vendors developing Java chips are available to the embedded storage developer. Java will be used in the next generation Personal Digital Assistants (PDA), such as the Palm Pilot, and in the next generation of mobile phones.

We believe that Java has now become a viable option for building real-time storage applications. Issues involving the space, performance and scheduling problems of Java for embedded systems are being solved. Almost daily, a new vendor or company announces its plan for Java in the embedded environment. With the many options available, at least one flavor of embedded Java will work for your application.

6 References

[1] WindRiver web site - http://www.windriver.com

[2] EE Times, January 29,2001, “Java Vendors set to skirmish over cellular”, page 1

[3] EE Times, October 16, 2001, “ARM tweaks CPU schemes to run Java”, page 20

[4] JavaPro, February, 2002, “A Comfortable Jbed”, page 72