Software Issues

Aurora VLSI, Inc.
December 8, 1999

This white paper discusses major software issues:

- Bilingual Capability and Implications
- Support Software

Bilingual Capability and Implications

This processor is a bilingual processor that executes both Java bytecodes and popular legacy binaries based on the RISC instruction set that originated at Stanford*. It is targeted at applications that need better Java performance than software Java implementations provide, given the hardware constraints of various platforms that run Java. For example, inexpensive consumer devices are not powered by high performance and consequently, expensive processors, but by lower performance, low cost embedded processors that can not deliver adequate Java performance for all Java applications. This bilingual processor provides a low cost solution to this Java performance issue.

By executing legacy code in addition to Java, the large code base of existing software can be used on this bilingual processor. Code written in a high level language such as C, is compiled to legacy binaries and run in legacy mode. This avoids a huge software effort to write a C to Java bytecode compiler and/or rewrite existing code in Java. C code will be around for a long time, and this ability to mix Java and C execution is critical to developing a widespread acceptance of Java. If good Java performance were to preclude good C performance and rapid C application deployment (as is the case with pure Java processors), Java would not spread beyond niche applications. And conversely, If Java performance can not match C performance in a C environment, Java's growth will also be slowed. With this bilingual capability, high performance Java and C coexist with minimal software efforts, and therefore, applications can be rapidly deployed regardless of the language that the applications are written in.

Basic Operation

The programming model for this bilingual processor is that only one instruction set is active at any given time. A software programmable mode bit determines the active instruction set. In Java mode, this mode bit is visible in the Java mode Status Register, and in legacy mode, it is part of the legacy mode Status Register.

To change modes, software writes the mode bit and jumps to the instruction fetch entry point of the mode being activated. This jump must be the instruction directly following the mode bit write. Care must be taken so that this jump does not trap. The jump is taken in the new mode's instruction set.

*Aurora VLSI has no affiliation, endorsement, or sponsorship by MIPS Technologies, Inc. None of Aurora VLSI's goods or services originate in any way from MIPS Technologies, Inc.

When switching modes, all state from the previous time that the new mode was active, is retained. Retained state does not need to be saved and restored across mode switches. This facilitates very fast mode switches so that the optimal mode can be used for any code block.

At reset, this bilingual processor will reset into legacy mode.

Operating System

The operating system is run in legacy mode. This leverages existing operating system software. Typically, much of the operating system is written in a high level language such as C. Using an existing, available, industry standard legacy C compiler, the operating system is compiled to a legacy binary that executes in legacy mode. As discussed above, this avoids the necessity of a C to Java bytecode compiler and/or an operating system rewrite in Java, and the associated large development and debug efforts.

Interrupts

Interrupts are fielded in legacy mode. As described above, the operating system is run in legacy mode. An interrupt in Java mode will go to a small extended bytecode interrupt handler stub that will switch the CPU to legacy mode, and allow the operating system to field the interrupt. This leverages existing interrupt handlers and avoids the porting of handlers to Java.

Device Drivers

Similar to interrupts, device drivers run in legacy mode. Device drivers are considered to be part of the operating system that runs in legacy mode. When a device driver is called from Java mode, a small bytecode stub of the device driver will switch the CPU to legacy mode at the device driver entry point. The device driver work is then done in legacy mode. This leverages existing device drivers and avoids porting the drivers to Java.

Native Code

A large portion of the JVM will be native code written in a non-bytecode language such as C. This includes any native functions that call the underlying operating system facilities, the garbage collector, the class file loader, the thread scheduler, etc. In addition to the JVM, one or more Java packages is part of a Java software environment. A Java package is simply a collection of Java class files that provide functionality beyond the bare bones JVM- ie print facilities, APIs, etc. Native functions implemented in native code, are used throughout Java packages also.

Native functions are typically implemented in C or C++. A commercially available industry standard C or C++ legacy compiler will be used to compile these native functions into legacy binaries. When called, these native functions run in legacy mode. Native functions that are part of Java class files, are linked in and called (by the INVOKE* Java bytecodes) as native methods.

Running native functions in legacy mode has significant advantages:

- The C/C++ compiler already exists. A C/C++ to Java bytecode compiler and its development effort, is not needed.
- Porting native functions is very much minimized, thus avoiding another large engineering effort.

Since parts of the JVM and Java packages are written in native functions, there will be a native function interface that is used in all the JVM implementations to minimize the huge porting effort that accompanies each Java package and each release of each Java package.

Support Software

In addition to marketing the processor, supporting software will be available. This includes:

- JVM
  - Small embedded JVM for wireless and handheld devices
  - Larger JVM for television systems and home servers
- Class files
- Software developers' environment
  - C and C++ compiler (to legacy binaries)
  - Java compiler
  - Java assembler, loader, linker
  - Debugger
- Application software- typically provided by third parties, or as an Aurora VLSI extra

JVM

The JVM is the layer between the operating system and the Java bytecode application. It relies on underlying operating system facilities such as file system services, networking protocol support such as TCP/IP, memory management and allocation, signal and interrupt dispatch and handling mechanisms, and processes and threads. One way of looking at the JVM layer is that it masks operating system dependencies from the Java application. Java applications are portable because all the non-portable aspects are addressed in this JVM layer.

There is a wide range of options when it comes to JVM implementations. The garbage collector can be stop-and-go or incremental, copying or non-copying, precise or conservative. This matters greatly for some applications. In fact, one widely supported industry real time Java proposal that is aimed at super low memory configurations such as smart cards, has garbage collection completely disabled! Another area of variation among JVMs is threading support. This can be pre-emptive or not pre-emptive, uniprocessor or multiprocessor, and use native kernel threads or more commonly, use the Green threads package from Sun that emulates threads at the user level with timers and signals. Lastly, JVMs differ in the sets of class files that they include. More class files require more memory. The reality is that there is no "one size fits all" in JVM implementations.

The full JDK2 JVM implementation is sufficiently robust for server applications. It has a large number of class files and consequently its large memory footprint restricts it to hardware systems with large amounts of memory. A trimmed down JVM- the J2ME JVM, is intended for client applications such as TVs and cars but still works best when there is at least 512Kbytes of memory. The KVM JVM is a small memory footprint JVM intended for wireless and handheld devices that have 128Kbytes to 512Kbytes of memory.

Wireless and handheld devices are sufficiently different in their hardware resources, characteristics, and software requirements, when compared to home servers and television systems, that different JVMs are needed to be competitive. Therefore, two JVMs will be implemented. A small 40Kbyte KVM-like JVM for embedded applications is important for the wireless and handheld market. A second larger JVM with wide ranging functionality somewhere between the J2ME JVM and JDK2 JVM will be implemented for set top boxes, personal TVs, DTV, home servers, etc.

Each JVM will be ported to each operating system that is supported by that JVM. Currently, the plan (subject to change based on demand) for JVMs and their operating systems is:

Operating System	JVM
EPOC	small
PalmOS	small
WindowsCE	larger
Linux	larger

Each port will use its target operating system's facilities for I/O, networking protocol support such as TCP/IP, file system services, memory allocations, etc. After the JVM is ported, it is optimized to use Java mode of the processor for bytecode execution, as opposed to the original slow software bytecode interpreter.

JVM implementations have all been in C or C++, not Java (except for one non-commercially viable research project). The Aurora VLSI JVMs will also be in C or C++. Using available industry standard legacy C or C++ compilers, the JVMs will be compiled into legacy binaries. Thus, the JVMs will run in legacy mode, switching to Java mode for bytecode execution. The table below shows the major JVM components and the mode that they run in.

JVM Component	Mode
Class file loader and unloader	legacy
Verifier (linking, verification, initialization)	legacy
Byte code execution	Java
Garbage collector	legacy
Scheduling, thread control	legacy
Memory allocator	legacy
JVM startup and exit	legacy

Native methods for things such as I/O, will run in legacy mode also. An extremely fast mode switch that avoids copying, saves, and restores, is provided so that mode switches have no performance or memory penalties.

Class Files

Class files contain class definitions that provide functionality beyond the bare bones JVM- ie I/O facilities, APIs, etc. Any one application only needs a subset of all possible class files. Some class files are essential to all applications. Others are highly application specific.

Work done at Sun Labs has shown that embedded JVMs (the Spotless JVM that was productized into the KVM) can run in as few as 40Kbytes if one leaves out many of the standard Java classes. It is very difficult to compete in the wireless and handheld embedded space with a full set of Java classes that can take up several megabytes of memory for features such as internalization and graphics, when a smaller, targeted set of classes can run in a much smaller space, and is sufficient. Currently, 40 to 50 classes are planned (subject to change based on demand) for the small KVM-like JVM. These include the classes in three packages- java.lang, java.util, and com.sun.kjava.

The complete set of JDK2 class libraries has hundreds of classes. Several, but not all, of these will be included with the larger JVM. As a JVM for television systems, the Java TV API will be supported. It is expected that the larger JVM will have 100 to 300 classes.

Software Development Environment

Software developers will have a complete software development environment for their work. Software tools are:

- C and C++ compiler (to legacy binaries)
- Java compiler
- Assembler, Linker, Loader
- Debugger

C and C++ compiler

C and C++ code will be compiled into legacy binaries and run in legacy mode. There are several commercially/publicly available legacy C and C++ compilers. Any one of these can be used. Aurora VLSI will not write yet another legacy C or C++ compiler, as there is no reason to do so.

C and C++ code will not be compiled into Java bytecodes, as there is no advantage to this. Therefore, a C or C++ to Java bytecode compiler is not necessary and will not be developed.

Java Compiler

A Java compiler is needed to translate high level Java into Java class files that include the bytecodes. Java compilers are available commercially. Therefore, Aurora VLSI will not write a Java compiler.

Java Assembler, Linker, Loader

There will be Java assembly code that is written. For example, the Java mode interrupt handler stub and device driver stubs that call legacy mode for the main part of the handlers and drivers, will be written in Java assembly code because they need to use the extended bytecode instructions. A Java assembler will be developed and provided by Aurora VLSI. The linker and loader are embedded in the JVM, so there is no need for a separate linker and loader.

Debugger

There is a well defined Java Virtual Machine Debugger Interface (JVMDI) that is defined at the Java Native Interface (JNI) C level. This allows the user to debug at the Java source code level and to step through Java bytecodes. When a native method is encountered, the debugger steps past it in one debugger step.

For out of process debuggers, there is a Java Debug Wire Protocol (JDWP) and an associated com.sun.jdi set of classes defined to access the JVM remotely. By supporting these standard interfaces, the developer can use a hosted development environment and debug Java code running on another board. The Java code can be debugged at the source code level and the bytecodes can be stepped through. Similar to the JVMDI debugger, when a native method is encountered, the debugger steps past it in one debugger step.

Applications

Combining a Java processor with legacy code execution greatly simplifies and speeds up the deployment of applications. Any existing applications can be used as they are. For example, if they exist in C, they will simply be compiled into legacy binaries and run in legacy mode. No rewrite to Java is necessary. If an application exists in Java, it is left in Java and run directly by the hardware in Java mode, leading to the fastest possible performance for it.

New code can be written in Java or C/C++. This offers the ultimate flexibility and performance. Java code can call legacy code as a native method, and then return to Java code by exiting the native method. Thus, a new application can be written partially in Java and partially in C, if desired, for the optimal solution. This bilingual capability also allows for new Java applications that want to include existing C/C++ routines, again speeding deployment of new applications.

Application Example- Bluetooth

With a legacy processor, much of the 12cap layer in Bluetooth can be implemented on the CPU in C, and compiled into legacy binaries to run. There is no reason to rewrite this layer in Java. It is too much work and the software already exists for the protocols, in C. This is a good example of the advantages of legacy code execution by the bilingual processor. It would defeat the purpose of the bilingual capability if the Bluetooth protocols were rewritten in Java.

Once the Bluetooth connection is established, Java mode provides superior performance when running downloaded Java applets. This is something that the user runs on top of Bluetooth, but not as part of Bluetooth itself. It's what one can do with Bluetooth once the wireless connection is established.