

WHITE PAPER

ACCELERATING TO MEET THE CHALLENGE OF EMBEDDED JAVA™

Steve Steele, Java Program Manager, ARM Limited

Cambridge, UK

Abstract :

Why does Java™ present such a challenge in the embedded space? This paper explores the need to accelerate the Java virtual machine, examining the various available techniques and design constraints.

The ARM® Jazelle™ solution for Java acceleration is studied in detail, including an overview of Jazelle technology and the reasoning behind design decisions.

The issues surrounding integration of Jazelle technology into real products are considered. Finally, we look at the roadmap for Java in embedded devices.

WHY DOES JAVA PRESENT A CHALLENGE IN THE EMBEDDED SPACE ?

Java was developed in the mid-1990s by Sun Microsystems to support the thin client computing model. As PCs grew powerful enough to support their own applications, thin client computing within the office environment declined. Programmers, however, discovered Java's write-once-run-anywhere (WORA) model of application development and portability made it the ideal mechanism for delivering mobile applications. But portability comes with a price: Java defines a portable, byte code instruction set architecture, which must be either implemented physically (at the cost of extra area and power) or emulated by the target device in software (which is inherently inefficient and slow). As a result, Java's initial integration into portable devices was difficult, inefficient and expensive.

The solution to these problems is to build a Java virtual machine into the mobile device, which will allow the latest applications to be portably downloaded and then used offline, providing

cheaper and more reliable access as well as excellent user interaction.

Integrating and accelerating Java to execute sophisticated applications capable of supporting intensive visual content presents significant technical hurdles for developers faced with cost, power, memory and space constraints. Integrating Java into devices designed for world-wide markets requires low-cost platform processors capable of running Java applications at a high level of performance alongside existing OSs, middleware and application code. The designer's ultimate challenge, therefore, is integrating and accelerating Java for optimal system performance while minimising space, costs and power consumption.

A REVIEW OF DESIGN CONSTRAINTS FOR ACCELERATION TECHNIQUES

Traditional Java acceleration systems consist of a variety of software solutions such as optimized Java virtual machines and just-in-time (JIT) compilers, as well as hardware solutions such as dedicated Java processors and Java co-processors. Depending upon the system, a level of acceleration ranging from 2-10 times faster can be achieved using these methods. Delivering this enhanced performance, however, has typically compromised power, memory, space or cost efficiencies. New hybrid solutions, called architectural extensions, can now execute Java byte code directly in the processor core, offering optimal performance along with OS and application compatibility, without requiring additional hardware or memory.

Software Acceleration

Software solutions for accelerating Java execution are widely available, but generally regarded as unsuitable for portable or low-cost systems.

- Optimized Java virtual machines typically offer sufficient memory efficiency, however, they are incapable of providing adequate performance for high-end applications unless a high-performance

processor is used. This is generally incompatible with the cost and power constraints of mobile devices.

- JIT compilers bypass the Java virtual machine for much of the byte code interpretation, translating such code directly into the existing core's instruction set. Although effective for desktop computers, JIT compilers are unsuitable acceleration mechanisms for mobile applications because they require extra resources. Typical compilers are more than 100 Kbytes in size, and compiled code typically expands by a factor of six or eight – requiring a large RAM cache. Although native code can run faster than Java code - a JIT compiler is typically slow to initiate, resulting in pauses and user input disruptions. JIT compilers also make heavy demands on the CPU during the compilation phase, which means greater memory requirements, more processing power and ultimately, more expense.

Hardware Acceleration

Hardware solutions for accelerating Java execution typically require additional space and power. And because they require external memory, they do not maximise speed.

- Dedicated Java processors directly execute the Java byte code within the processor. Although they appear to offer acceptable performance, dedicated Java processors represent a significant overhead and additional integration and development complexity. Because they do not support existing applications or established operating systems, they must always operate alongside another processor.
- Java co-processors translate Java byte code into the existing core's instructions. This acceleration process often requires a significant hardware and software integration effort and is difficult to incorporate into the existing OS. Co-processors also require extra space for the gates, extra power to operate, and are expensive to manufacture. In addition, they tend to run relatively slowly because they are loosely coupled with the core processor.

Architectural Extension

An architectural extension is a combination hardware/software solution for accelerating Java byte code execution directly in the processor core. Extensions offer the possibility of achieving significant performance within the cost and power envelope of a typical mobile device.

- An architectural extension is an enhanced single processor solution that directly executes Java byte

code alongside existing OSs, middleware and application code. By placing an additional instruction set inside the processor, an architectural extension reuses all existing processor resources without the need to re-engineer existing architecture or add cost, power or memory resources. An extended core can efficiently run both Java and native code, giving developers the ability to leverage the existing base of applications and operating system expertise while achieving a successful balance of Java portability and native performance for their application.

JAZELLE - THE REASONS BEHIND THE DESIGN DECISIONS

Initially, ARM looked for generic solutions for executing Java in hardware but couldn't find one that fitted all the requirements. Those considered either would not work on cached processors; they were simply too large to fit the power and die size requirements; or they led to unacceptable increases in memory consumption.

Increased customer pressure

As pressure from ARM's customers for improved Java performance increased, it became obvious that ARM would have to provide its own solution for executing Java in hardware. It was decided that the only way to go was to integrate Java execution into our ARM core. Thus, Jazelle was born. The objectives were clear:

- In order to reduce die size and improve performance, Jazelle would be implemented in the ARM pipeline as an FSM (Finite State Machine) rather than a traditional microcoded engine. It would be implemented inside the cache, which has important benefits both in terms of power consumption and performance.
- Jazelle would be implemented in a way that allowed all Java instructions to be restartable. That is, an interrupt can be taken in the middle of an executed Java instruction in such a way that interrupt latency is not affected.
- Typically, a Jazelle implementation would dynamically re-map Java stack locations to ARM registers, thus avoiding the need for a translation stage. Similarly, such an implementation of Jazelle would perform 32-bit fetches in order to fetch up to four Java bytecodes at once.

Jazelle - ARM Architecture Extensions for Java
ARM processors support two instruction sets: the ARM instruction set, in which all instructions are

32-bits long, and the Thumb instruction set, which compresses the most-commonly used instructions into a 16-bit format. The Thumb instruction set typically offers 35-40% code compression compared to ARM code, which reduces performance slightly. The instruction set supports procedure calls between ARM and Thumb code, so application programmers typically choose at compile time whether parts of the application should be compiled for performance or code density.

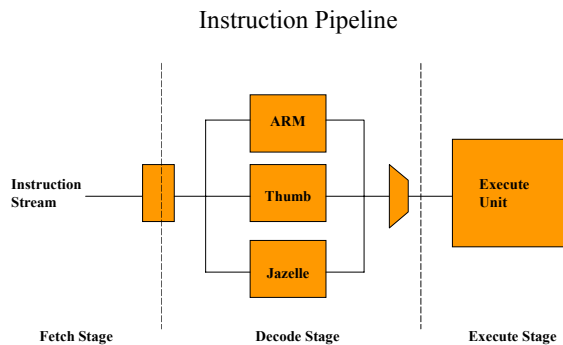


Figure 1: Jazelle Technology

To create Jazelle, a third instruction set - Java bytecode – was added to the processor, with instruction set support for entering and exiting Java applications. This instruction set creates a new state in which the processor behaves like a Java machine - it fetches and decodes Java bytecodes and maintains the Java operand stack. That is, once in Java State, the processor is in every way a Java processor, but it can switch easily between Java State and ARM/Thumb State. In essence, we have made a two-in-one processor - one is an ARM/Thumb processor and the other is a Java processor - but with all the performance, memory use, battery life, and space and cost advantages of a single processor.

Entering and exiting Java applications is simple, and can easily be put under the control of any operating system. Interrupts are handled normally, and cause an immediate return from Java State to ARM State to run the interrupt handler. At the end of the interrupt routine, the normal return mechanism will return the processor to Java State. This ensures real-time interrupt performance.

A BRIEF OVERVIEW OF JAZELLE TECHNOLOGY

There is a single new ARM instruction: ‘Branch-to-Java’ for entering Java State. This instruction first performs a test on one of the condition codes. If the condition is met, it puts the processor into Java State, branches to specified target address and begins executing Java bytecodes.

Once in Java State, the ARM PC is extended to 32-bits to address Java bytecode. Bytecodes are fetched and decoded in two stages (compared to a single decode stage when in ARM/Thumb State). A new Current Processor Status Register (CPSR) bit records the Processor State. This is an important feature, as the CPSR is automatically saved and restored when handling interrupts & exceptions, so Jazelle is compatible with the existing ARM interrupt/exception model used by operating systems.

In Java State, the processor assigns several ARM registers to functions specific to the Java machine (e.g., R6= stack pointer, R0-R3= top elements of stack, R4=local variable 0). This hardware reuse contributes to the small size of the additional logic (12k gates) required to implement the Java machine, and keeps all of the state required by the Jazelle extension in ARM registers. In addition, it ensures compatibility with existing operating systems, interrupt handlers and exception code.

Keeping the top four elements of the stack in ARM registers is an important contributor to the performance of the processor when executing Java. Application profiling has shown that the working stack depth for most applications is very small, so this technique reduces memory accesses to a minimum. Stack spill and underflow is handled automatically by the hardware.

Hardware / Software Balance for Optimum Cost and Performance

The Jazelle extension divides Java Bytecodes into three classes: directly executed, emulated and undefined. The majority of the Java bytecodes (134 on the ARM926EJ) are executed directly in hardware; the remainder is emulated by short sequences of highly optimized ARM instructions. Jazelle technology removes the interpreter loop from the virtual machine and replaces it with ARM’s proprietary support code called VMZ, which is no larger than the code taken out.

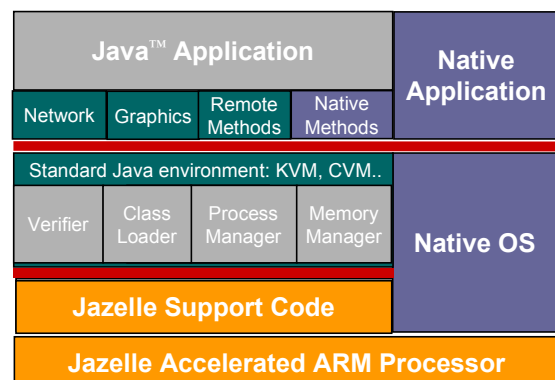


Figure 2: Jazelle hardware and software

This reduces the complexity (cost and power consumption) of the additional logic required to implement the extensions, but without significantly compromising performance. Application profiling has shown that the emulated bytecodes are encountered less than 5 percent of the time in typical application code. With the floating-point option, the number of directly executed bytecodes increases from 134 to 149.

Undefined bytecodes are distinct from emulated bytecodes. Encountering any undefined Java bytecode will cause the processor to leave Java State and return to an exception handler written in ARM code, which is normally part of the VMZ. This also provides a mechanism for supporting future extensions of the Java bytecode set; a software patch can implement a new bytecode function.

Interrupt Behaviour and Real-time performance

In the target applications, it is important for the processor to have good real-time performance so that it can also run the network protocol software and other middleware. In designing the architecture, careful consideration was given to maintaining good interrupt response and compatibility with existing applications. All of the Java bytecodes are re-startable - which means that no special provision has to be made for handling interrupts. Any interrupt returns the processor to ARM State for execution of the interrupt handler.

All of the processor state relating to Java execution is held in the normal ARM register set ('J' bit in the CPSR, PC in R14, etc) so any interrupt routine which saves machine state on entry and restores it on exit, is automatically compatible with the Jazelle technology. When the PC and CPSR are restored on return from interrupt, the processor will automatically re-enter the Java application at the correct point.

Approximately 12k gates are required to implement the ARM Java extensions - much smaller than most dedicated processors or co-processors, which are typically between 60-100k gates. From a technical point of view, integration is very easy - there is only one program counter and all of the Java State is held in ARM registers, so the extensions are consistent and compatible with existing interrupt and exception models.

In the target applications, there are several established platform operating systems - WindowsCE, Symbian OS, PalmOS, Linux, plus many real-time and proprietary OSes. In developing the architecture extensions, ARM has worked with

many of the operating system vendors to ensure that support for the extensions will be available.

The Java-enabled ARM processor makes design simpler and easier, and allows a single processor to run all of the operating system, middleware, and protocol software.

INTEGRATING JAZELLE ACCELERATION INTO REAL PRODUCTS

The market demand for new and exciting products means that development timescales are under constant pressure. If a new technology, such as Jazelle, is to be successful – then it must be accessible to developers and be simple to integrate into their product roadmap.

Ideally, the Java acceleration technology will cause the minimum disruption to the existing product platform. It will be compatible with existing OSes, virtual machine, middleware and applications.

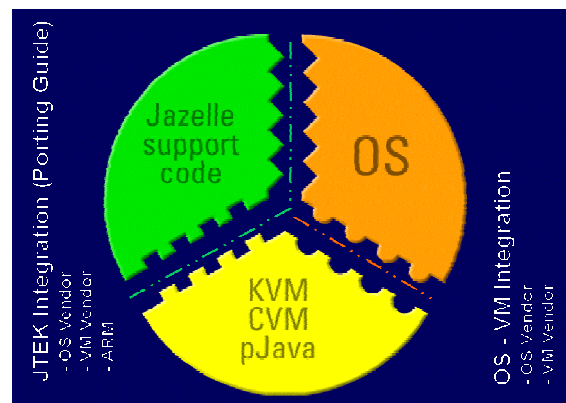


Figure 3: Integration with OS and virtual machine

It is also most important that the technology is supported by a comprehensive set of tools – which are appropriate for the special demands of real-time, embedded development. In short, an integrated acceleration solution with virtual machine and OS compatibility is essential to achieving silicon space, power consumption and memory efficiencies required for portable applications.

Virtual Machine Selection

We stated earlier that designers rely on Java’s WORA model of application development when designing mobile and Internet appliance devices; WORA ensures once the Java code is written, it will run anywhere it is imported. This streamlined design process requires that all devices use a virtual machine certified as Java compliant. Beyond Java compliance, virtual machine selection is driven by the characteristics and functionality intended for the device. Jazelle supports many Java-compliant

virtual machine systems offered by, and licensed from, Sun Microsystems. These include the pJava, KVM and CVM virtual machines.

- pJava (also called Personal Java), is based on the original Java virtual machine from Sun.
- KVM is a scaled-down virtual machine optimized primarily for smaller mobile platforms with minimal user interface requirements such as those found in cellular phones.
- CVM, Sun's most recent Java-compliant virtual machine, is based on the latest Java technology and provides functionality necessary for high-end mobile computing applications such as PDAs.

Once a designer has selected the appropriate virtual machine, the selection of system profiles (WIDP versus MIDP), and device configurations (CDC versus CLCD) can then follow guidelines set by the diagram below depicting the Java 2 and the Micro Edition (J2ME).

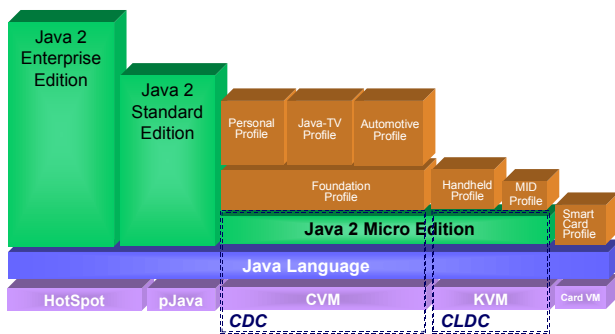


Figure 4: Virtual machine selection

Operating System Compatibility

When incorporating a Java acceleration system (hardware, software, or an architectural extension), designers must confirm it supports their chosen OS. Designers must be confident that complete support and compatibility exists when the processor meets the OS at implementation time. For example, the software component of Jazelle, called the Jazelle Technology Enabling Kit (JTEK), is licensed for integration into a wide range of operating systems.

Development Tools

The market for Java compilers and debuggers is well developed with many excellent solutions available to developers. However, what is less well developed are the tools required for integrating Java virtual machine technology into embedded platforms – with all the usual, complex real-time issues.

ARM is focusing its development effort to produce tools for embedded platform developers; there are no plans to develop a Java compiler or debugger. The resulting ARM Java Debug Solution (AJDS)

provides the facility to debug both Java and native code over a single Multi-ICE interface using the ARM debugger and any Java debugger that conforms to the standard JDWP protocol. The development of AJDS has required ARM to overcome many new and challenging real-time Java debug issues.

WHERE TO NEXT? ENABLING THE EMBEDDED JAVA ROADMAP

It is clear that the Java environment has many advantages for the application developer. Java source is quick to write and the resulting bytecode is easily ported to new platforms. Applications thus developed are typically robust, secure and can be distributed to run on a wide variety of devices.

These benefits make it highly likely that Java will continue to widen its appeal within the embedded developer community. System designers will use Java to attempt ever more ambitious goals and test the capabilities of the Java embedded environment.

The ARM Architecture is constantly being improved and developed through the addition of new features. ARM is committed to continue this process and will include new technologies, such as Jazelle and its successors, in future generations of the ARM Architecture.

Many experts believe that Java is the chief technology for delivering interactive services promised by the 3G revolution. Effective Java implementation and execution, such as offered by the ARM Jazelle technology, holds the key to these services.