Java: Too Much for Your System?

by Michael Barr

Sun's introduction of the Java programming language, in 1996, was a welcome development in the world of networked computing. Java is an easy-to-use, object-oriented programming language designed specifically for platform-independence and protection from common programming errors. These advantages and others help software developers produce new applications more quickly than they ever could in C or C++. And the resulting programs are more robust and immediately able to run on multiple computing platforms.

Unfortunately, some of the benefits of the language, particularly platform-independence, are less applicable to embedded software developers. And the costs associated with robustness and programming ease—code bloat and greater run-time overhead, for example—often outweigh the benefits in such resource-constrained environments. In addition, most of the default garbage collection algorithms are incompatible with the (hard and soft) real-time demands placed upon these systems.

Embedded software developers generally lag their more mainstream brethren in programming tools and techniques by 5 to 10 years. We develop software for systems with older processors, less memory, and a non-uniform set of peripherals. The target hardware almost always lacks a keyboard and display and, as a result of this and the reduced computing power, we need a general-purpose computer on which to develop the embedded software and from which to debug it.

Is Java applicable to resource-constrained and/or real-time environments like these? And if so, are the necessary tools available to make embedded software development in Java viable today? In this paper, I will attempt to answer these and many other questions you may have about developing embedded software in Java.

THE BENEFITS OF JAVA

Let's start by answering the most fundamental question: What's so great about Java? This is a bit of a more complicated question than you might at first suspect. Some of the things which are nice about Java are features of the programming language, while others are features of the virtual machine concept. Let's start with the language itself.

Java Programming Language

Although platform independence has often been hailed as Java's greatest strength, it is equally important to note that it is easier to produce bug-free software in Java than in C or C++. Java was designed from the ground up to produce code that is simpler to write and easier to maintain. And, though they based their language on the syntax of C, the creators of Java eliminated many of that language's most troublesome features. These features sometimes make C/C++ programs hard to understand and maintain, and frequently lead to undetected programming errors. Here are just a few of the improvements:

- All of Java's primitive data types have a fixed size. For example, an `int` is always 32-bits in

Java, no matter what processor lies underneath.

- Automatic run-time bounds-checking prevents the program from writing or reading past the end of an array.
- All test conditions must return a Boolean result. Common C/C++ programming mistakes, such as `while (x = 3)`, are thus detected at compile-time, eliminating an entire class of bugs.

In addition, Java is an object-oriented language, which allows software developers to encapsulate new data types and the functions that manipulate them into logical units called classes. Encapsulation, polymorphism, and inheritance (the three pillars of object-oriented programming) are all available and are used extensively in the built-in class libraries. Java simplifies inheritance by eliminating multiple inheritance and replacing it with *interfaces*. It also adds new features that are not available in C++, most notably:

- Automatic garbage collection simplifies dynamic memory management and eliminates memory leaks.
- Built-in language support makes multithreaded applications written in Java more portable than those written in other languages, by providing a consistent thread and synchronization API across all operating systems.

Java Virtual Machine

It is also important to recognize those additional benefits of Java that arise from the virtual machine architecture. But first you must understand what that architecture is. The virtual machine concept is not unique to Java; it arises from the marriage of two simpler ideas. First, that the work of computer programmers everywhere would be much easier if there were just one processor architecture that was used in every system. And, second, that simulation of one processor by another is always possible and, given sufficient computing power, often reasonable.

The creators of Java designed both a new language and a mythical processor on which all of the programs written in that language would be run. Since this processor did not actually exist, they termed it a *Java Virtual Machine*. The written specification of this "processor" (called The Java Virtual Machine Specification and available in book form[1]) describes a full set of machine-language instructions and their behaviors and reads much like the Programmer's Guide for a real processor. The instruction set recognized by the virtual machine is called the Java *bytecodes*.[2]

Since the Java processor did not actually exist in hardware (at the time, anyway) it was necessary to simulate it in software. Toward that end, the first Java Virtual Machine (JVM) was developed by Sun. The first JVM was an *bytecode interpreter*, which translates each Java bytecode into one or more of the opcodes of the underlying physical processor, at runtime. An interpreter like this retranslates a bytecode each time it is fetched from memory. Obviously, this slows down the execution of the Java program, or requires additional processing power. (Consider the impact of reinterpretation on a simple `for()` loop and its inner statements.)

[1] Lindholm, Tim and Frank Yellin. The Java Virtual Machine Specification. Addison-Wesley.

[2] The term "bytecode" derives from the fact that each instruction is just one byte wide.

The overall development and execution infrastructure for a Java program is shown in Figure 1. A program written in the Java programming language is compiled into a set of bytecodes. Those bytecodes are then loaded and executed by a Java Virtual Machine. If the program makes calls to other Java classes, the bytecodes for those classes will likewise be loaded and executed. Some of these libraries (`java.lang`, `java.math`, `java.io`, etc.) are intended to be a built-in part of any standard Java execution environment, much as the C standard library is considered a part of the ANSI C language.

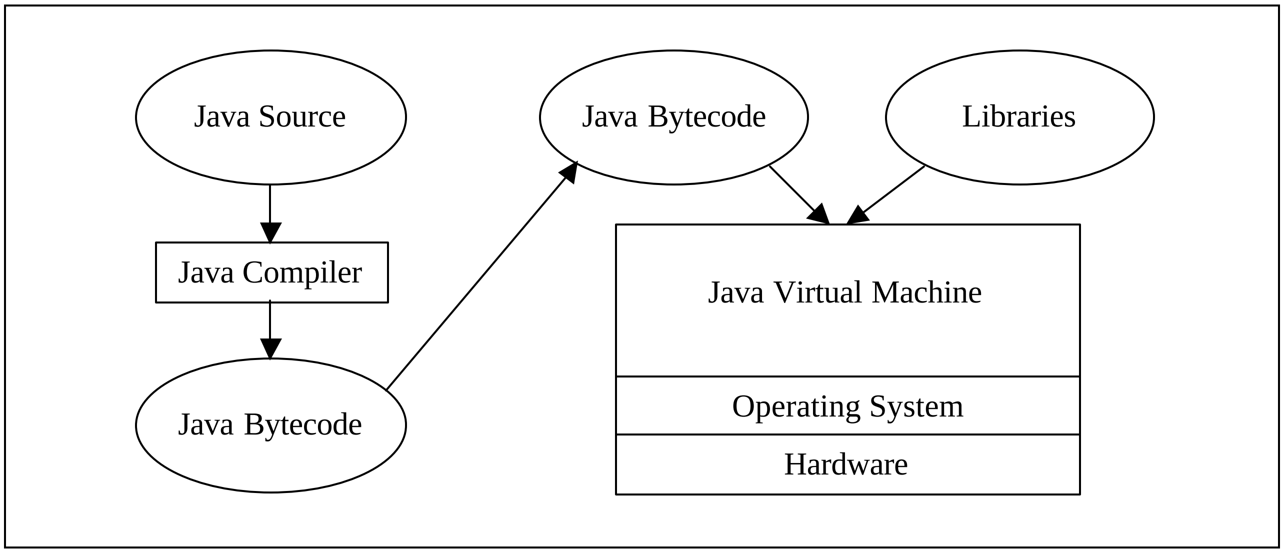

Figure 1. The Java Development and Execution Infrastructure

The principal advantage of the virtual machine concept is portability. Programs written in Java can be executed on any processor for which a simulator (JVM) exists. If the behavior of the virtual machine is defined well enough to ensure consistency across all platforms, applications will produce the same result in each such computing environment.

Another advantage of this architecture is the ability to distribute and incorporate software libraries. Useful classes can be compiled into bytecode and incorporated into a larger project without the need for source code! Combined with an object-oriented programming methodology, this separation between the executable object and the intellectual property paves the way for a huge potential market for third-party software modules. At long last, programmers have an excellent framework in place for software packaging and reuse.

JAVA WEAKNESSES

Well, you must be wondering, if Java is so great, why should we use any other language ever again? Unfortunately, it is not possible to do everything in Java that embedded programmers may be accustomed to doing in C/C++. And there are also some valid concerns with respect to code size, efficiency, and indeterminism. We'll take a look at these now.

No Access to Registers and Memory

One of the most enduringly useful (and incredibly dangerous) features of the C language is the ability to manipulate the contents of hardware registers and physical memory locations directly. By simply creating a pointer to some fixed memory address, a programmer can control the behavior of complex peripheral devices without the need for assembly language. This feature is an absolute necessity for the developers of device drivers and embedded software—or anyone who works closely with hardware.

But Java does not have pointers. Instead it has *references*, which are merely identifiers for individual software objects. This is like having the name of a friend, rather than his address or phone number. How will you get in touch with him? You'll have to rely on the phone book. Unfortunately, though, the Java "phone book" only has listings for software-created objects and not for hardware registers or physical memory locations. So there is no way to read or write a specific location directly from your Java programs.

Java's saving grace in this regard is that it understands how to call functions written in other programming languages, which are termed *native methods*. So you can still write your device drivers in C/C++ or assembly and call into them from Java. Or you could create a general-purpose peek (poke) function that takes a physical address and reads (writes) a certain number of bytes from (to) that location. Most providers of Java tools for embedded systems provide a capability similar to C's `inport()` and `outport()` for accessing I/O-mapped registers on an 80x86 processor.

Code Size

The central issue with respect to code size is the need for a large Java Virtual Machine and class libraries to execute even the simplest Java program. Together, a poorly-constructed JVM and a full set of class libraries can require more than 1-Mbyte of ROM and several hundred kilobytes of RAM! This is typically not that big of a deal on a general-purpose computer where there is plenty of memory available, but it is the exceptional embedded system that has a significant amount of spare memory. In fact, the vast majority of 8 and 16-bit designs can't even address a full 1-Mbyte of ROM and RAM combined.

The size issue has been addressed by a variety of commercial JVM and tool vendors. Solutions range from smaller (more specialized) JVM implementations to tools that prevent unused class libraries from being loaded into ROM to compilers that turn bytecode into native opcode long before the program is actually executed. The latter tool, called an *Ahead-of-Time Compiler* (AOT), is not much different than the C/C++ cross compiler you may be using today. The only difference is that the input language is Java (or bytecode, since most AOT compilers support both).

Efficiency

As I've already stated, Java is an object-oriented programming language. This and other characteristics require behind-the-scenes work to support the language at run-time. And with Java, there is a lot more such work than for traditional high-level languages like C and C++. Some of the features that require significant runtime support are automatic garbage collection, dynamic linking, and exception handling. There are also run-time checks to be performed, like the array bounds checking mentioned earlier.

By far, though, the biggest slowdown is the result of runtime bytecode interpretation. A variety of tests have shown interpreted Java programs to be at least 10 times slower than equivalent programs written in C/C++! But if you can eliminate the need for interpretation, as you would by using an AOT compiler, you'll find that the run-time checks and other overhead are not nearly so significant.

Depending on the behavior of the application, a precompiled Java program will usually run just a little slower (10-50%) than the C++ equivalent. (In fact, the actual instructions generated by the compilers are equivalent. Most of the extra overhead results from garbage collection and those extra run-time checks.) Such a small performance impact seems like a reasonable price to pay for the increased productivity and decreased bugs offered by Java. But it is clear that interpreted Java programs won't suffice in the majority of embedded systems. For that reason, I think the future success of Java for embedded software development lies in the hands of a small number of AOT compiler vendors.

Indeterminism

In the embedded systems community, much has been made of Java's indeterminism. And this is indeed an important issue for us to discuss. To be clear, let's first agree on what is meant by that term. A computer system is said to be *deterministic* if the maximum length of time it takes to do something can be determined in advance. The specific length of time is not important to this definition, only that some maximum time can be calculated. However, it should be clear that particular tasks within a program will have very specific deadlines.

What puts Java's determinism at risk is garbage collection. In particular, the problem lies with the tool vendor's selection of a garbage collection algorithm. There are various methods for reclaiming memory that has been previously allocated but is no longer being used by a program. The simplest such algorithm (and a common default for Java) doesn't get started until a memory allocation request is unable to be filled. At that point, the application program is halted, so the garbage collector can safely "walk" through the entire heap, marking each object that is still in use. Once this marking process is complete, unused memory can be reclaimed, the failed allocation request fulfilled, and the application program restarted. The problem with this algorithm (called *mark and sweep*) and some others is that it may preempt the program at any allocation request—even in the middle of executing a high-priority task with a pending deadline. If the maximum length of time that the garbage collection process may take is longer than even one of your deadlines, such algorithms are unacceptable.

However, there are garbage collection algorithms that are better suited to soft, and even hard, real-time systems. For example, an incremental garbage collector is one that runs as a separate task. This task typically has a priority below that of all the real-time tasks in the system. The garbage collection task reclaims unused memory incrementally, while the rest of the application is still running. Though this does not guarantee that every future allocation request will be fulfilled, it can significantly decrease the likelihood and frequency of missed deadlines. And simply by adjusting the priority of the garbage collection task relative to the application tasks, it is possible to optimize the rate of reclamation such that it is, on average, equal to the rate of allocation.

The indeterminism of some garbage collection algorithms should be considered when using Java in any real-time system. However, because there are other algorithms available, this is not in itself a reason for avoiding the Java language altogether. Some proponents of the use of Java in real-time systems have pointed out that the simple addition of a 'delete' keyword to the language would allow elimination of the garbage collector from those systems. However, to date, Sun has remained reluctant to permit such a change.

JAVA USAGE MODELS

It is currently unrealistic to consider implementing an entire embedded software project in Java. For one thing, Java does not include a mechanism for directly accessing I/O ports, memory, or hardware registers. So there continues to be a need for device drivers and other pieces of supporting software written in C/C++ or assembly. This other software might either be a set of native methods that are called from Java or a separate set of C/C++ threads running in parallel with Java threads.

Before designing your system around Java, it is important to think about how the Java programs you write will fit into your overall system architecture. Many Java usage models have been proposed for embedded systems, but it seems that each of them falls into one of four categories: No Java, Embedded Web Server Java, Embedded Applet Java, or Application Java. These four usage models are distinguished by two binary variables: (1) the location of the stored Java bytecodes and (2) the processor on which they are executed. Each of these variables can take one of two values: *target* (the embedded system) or *host* (a general-purpose computer attached to the embedded system). For example, the category No Java includes all scenarios in which the bytecodes are stored on and executed by a host computer; although Java is used in this scenario, it is never actually on the embedded system. All four usage models are illustrated in Figure 2.

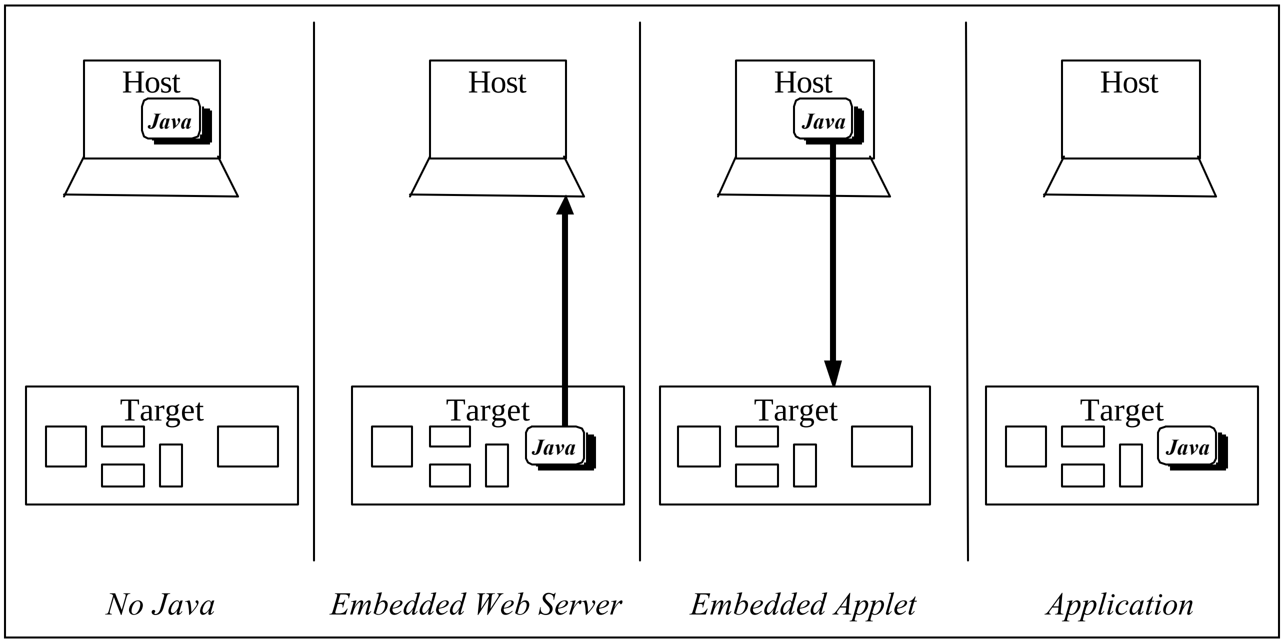

Figure 2. Four Java Usage Models for Embedded Systems

In the Embedded Web Server usage model, the Java bytecodes are stored on the target system (usually in Flash memory or ROM), but executed by the host processor. This model describes networked embedded systems that require a graphical interface. A Java-enabled Web browser—running on the host workstation—executes a set of Java bytecodes that it uploads from the embedded system. In addition to the Java bytecodes, the embedded system in this scenario must store at least one HTML file and include a piece of software called an embedded Web server. However, since Java is not actually executed on the embedded system, no Java runtime environment is required there.

The third and fourth usage models are the most interesting from the viewpoint of this discussion. These are the ones in which Java bytecodes are actually executed on the target processor, and for which an embedded Java runtime environment is, therefore, required. In the Embedded Applet scenario, the Java bytecodes are stored on the host workstation and downloaded to the embedded system via a network or other connection. The embedded system executes the bytecodes and sends the results back to the host. Embedded applets could be used to implement network management functionality (as a replacement for SNMP, for example) or to off-load computations from one processor to another.

In the Application model, Java comprises some or all of the actual embedded software. The Java bytecodes are stored in a nonvolatile memory device and executed by the Java runtime environment in much the same way that native machine code is fetched and executed by the processor itself. This use of Java is most similar to the way C and C++ are used in embedded systems today—to implement large pieces of the overall software. However, because Java lacks the ability to directly access hardware, it may still be necessary to rely on native methods written in C or C++. This is not unlike the way C/C++ programmers use assembly language to perform processor-specific tasks—like context switching—today.

Never forget that languages are just tools. When you hire a contractor to build a house, he will probably bring an electric nail driver and use it often. But you can bet that he'll still have a hammer in his belt at all times. There is a time and a place to use Java and benefit from it. But there will continue to be uses for C/C++ and assembly.

JAVA RUNTIME ENVIRONMENTS

If you make the decision to execute Java bytecodes in your embedded system, you will need to create the appropriate software environment. And whether you choose to purchase the parts from a third-party or develop them yourself, you'll need a complete Java runtime environment that consists of the following components:

- A Java Virtual Machine to translate Java's platform-independent bytecodes into the native machine code of the target processor and to perform dynamic class loading. This can take the form of an interpreter, a *Just-in-Time Compiler* (JIT), or something in between. The only noteworthy effects of the JVM internals are the speed with which bytecodes are executed (a JIT is faster because it avoids reinterpreting previously executed methods) and the memory required (a JIT needs more memory, because it must keep copies of the native opcodes for later use). Because of this tradeoff, there are also hybrid JVM implementations that focus their speedup efforts on only those methods that will produce the maximum gain. As a result, they have a much smaller impact on memory.

- A standard set of Java class libraries, in bytecode form. If your application doesn't use one or more of these classes, they are not strictly required. However, some platforms may need to conform to one of the standard APIs, like those described in the next section, which dictate a set of classes that must be available to any Java application.

- Native methods required by the class libraries or virtual machine. These are functions that are written in some other language, precompiled, and linked with the JVM. They are primarily required to perform functions which are either platform-specific or unable to be implemented directly in Java. For example, the `java.awt` graphics library makes calls to a set of graphics primitives to interact directly with the display.
- A multitasking operating system to provide the underlying implementation of Java's thread and synchronization mechanisms.
- A garbage collection thread. The garbage collector runs periodically—or whenever the available pool of dynamic memory is unable to satisfy an allocation request—to reclaim memory that has been allocated but is no longer being used by the application.

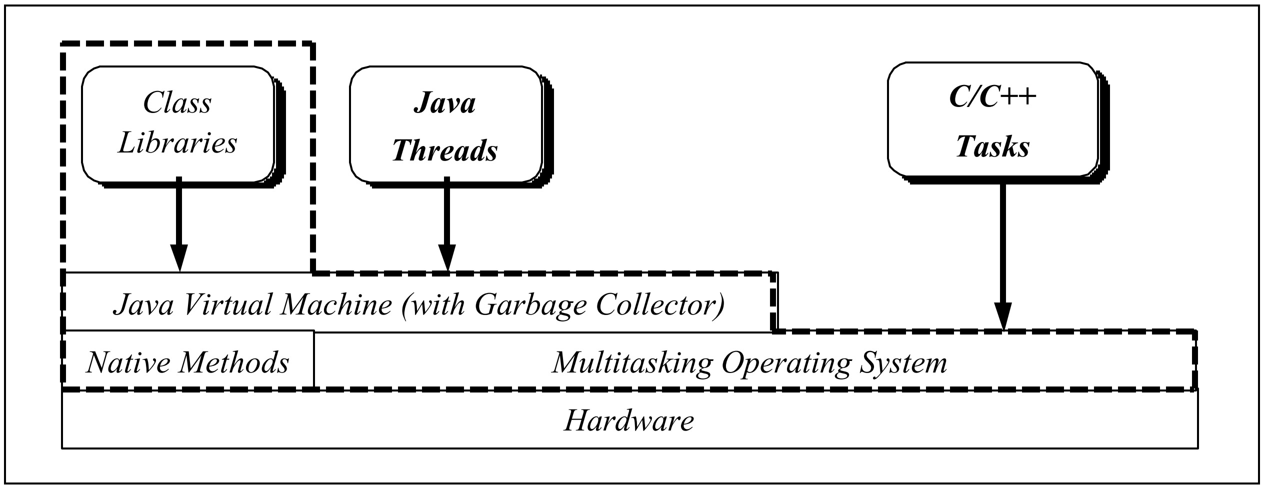
Class Libraries
Java Threads
C/C++ Tasks
Java Virtual Machine (with Garbage Collector)
Native Methods
Multitasking Operating System
Hardware

Figure 3. The Components of a Java Runtime Environment

The relationship of these components to the hardware and other software that makes up a typical embedded system is illustrated in Figure 3. A dotted line surrounds the components of the Java runtime environment.

JAVA APPLICATION ENVIRONMENTS

As stated earlier, a Java runtime environment includes a set of standard class libraries. But these Java classes are not strictly required unless your application actually uses them. In that sense, they are very similar to the standard C libraries. For example, if you've ever used `strcmp()` or `strlen()` in an embedded program, you were relying on the standard C library to be linked with your application. Similarly, if you want to manipulate strings in Java, you will need a class library called `java.lang` in your runtime environment.

In order to promote and encourage the "Write Once, Run Anywhere" nature of Java, Sun has defined several standard runtime platforms (groups of class libraries). Sun refers to these standard platforms as *Java Application Environments.* The following APIs are relevant to embedded software development:

- *Standard Java* - the full set of class libraries included in Sun's JDK. These classes are appropriate for desktop workstations and servers and may require significant hardware and operating system resources.

- *PersonalJava* - a (not-quite proper) subset of the Standard Java API that is appropriate for set-top boxes, PDAs, network computers, and other networked embedded systems with a fairly large amount of processing power and memory.
- *EmbeddedJava* - a "configurable subset" of the PersonalJava API that is better suited to the resource-constrained environments typically found in non-networked and relatively inexpensive embedded devices. Basically, if you aren't using a particular class, it may be excluded. The expectation is that systems built around EmbeddedJava do not usually allow new applications to be downloaded in the field; "Write Once, Run Anywhere" behavior is not important in these systems.

The intention of these standard API's is to allow application developers to easily specify the type of platform on which their Java program will run. For example, a program written for use in a PersonalJava-compatible set-top box could also be run on a PersonalJava-compatible network computer or PDA.

ALTERNATIVE TECHNOLOGIES

The Java programming language is wonderful. But because there are problems with the Java runtime environment—it's big, it's slow, and it may be indeterministic—a variety of alternative techniques have been proposed for using the language without the runtime environment. The most noteworthy of these are AOT compilers and Java processors.

Ahead-of-Time Compilers

Generally speaking, an AOT compiler is a cross compiler that runs on the host computer, accepts Java source code (or bytecode) as input, and produces an executable program for the target processor. This eliminates the need for the Java Virtual Machine and speeds up the program execution to a maximum. Of course, you'll still need a garbage collector and some other run-time checking, but the performance will be pretty comparable to an equivalent program written in (object-oriented) C++ and compiled with an optimizing compiler.[3]

Java Processors

Another alternative, the Java processor, was recognized almost immediately after Java was announced. There's no reason that the mythical processor architecture dreamed up by Java's creators could not be realized in silicon. Sun and several other IC manufacturers soon set about doing just that. And the chips and synthesizable cores they've since produced are microprocessors that execute Java bytecodes directly. The speed of execution is comparable to that of any processor executing code that has been compiled for it. And, since there is no JVM, there is none of the associated memory overhead.[4]

[3] Just about every statement in this paragraph is contradicted by one of the actual AOT implementations, but you get the general idea. I'll give more specific details in part 2 of the paper.

[4] Of course, the bytecodes of the class libraries will still take up space in ROM.

CONCLUSION

At this point, the word “Java” is so overloaded with meaning that it can be hard to separate the hope from the hype. Java is a programming language and a set of bytecodes, a virtual machine and a processor, a set of class libraries and a set of application environments, and a whole lot more. I hope that this paper has helped you to recognize each of these different aspects of the technology and to convince you that Java should not be dismissed out of hand by embedded developers. I hope it has also helped you to understand the structure of the Java runtime environment and to see how Java can best be fit into a larger project.

There are certainly many aspects of Java that are undesirable in embedded (and especially real-time) systems. However, the Java programming language is the most significant mainstream language to emerge since C. I think we’re going to see Java used more and more in all kinds of systems, including those that are real-time and embedded.