

Implementing Java in Internet Appliances

A core component of many Internet Appliances is the Java™ Virtual Machine (JVM). The desired behavior of a JVM in an embedded device goes beyond the correct execution of Java programs. The JVM must operate robustly and predictably in the face of inevitable resource restrictions while minimizing interference with other native functionality inside the device. It must adapt smoothly to properties of device features, and show optimal performance under conditions pertaining to embedded devices.

The purpose of this tutorial is to help the reader evaluate JVM products by pointing out critical areas in which Java Virtual Machines may differ. We will explore the consequences of three architectural aspects of the JVM. The first aspect regards the division of labor between the Operating System (OS) and the JVM. How independent is the JVM, and what are the virtues of being independent? The second architectural aspect is configurability. To what extent is the JVM implemented as a collection of loosely coupled software components, and why would that be important? The third aspect regards the architecture of Just-In-Time compilation technology. Which technology scales best with available resources?

1. Division of labor between the Operating System and the Java Virtual Machine

Significant parts of the functionality of a JVM can be implemented straightforwardly by expressing them in terms of similar functionality offered by the Operating System under which the JVM runs. However, such implementations may have drawbacks that make it advisable to include the functionality inside the JVM itself. We consider three examples:

Graphics: The shortest way to implement Java's graphics library, the Abstract Windows Toolkit (AWT), is to express it in terms of a general graphics library provided by the OS. Alternatively, the graphics can be handled completely inside the JVM, except for a simple pixel-level function that transforms a JVM screen buffer into the appropriate format required by the platform. If designed well, the latter approach leads to superior graphics performance, as the use of a usually heavy-weight software layer (e.g., X-Windows) is avoided. For a JVM vendor that supports multiple platforms, a built-in graphics library is a virtual necessity, because it all-but eliminates the porting effort related to the AWT. The important advantage for the user is the availability of virtual machines with the same look-and-feel on different platforms.

Multi-threading: The Java language supports and in a sense promotes the use of parallelism to structure programs. Java programs must therefore be expected to create many, often small, threads. Operating systems provide an entity parallel to Java threads, OS tasks. Hence, the simplest way to implement Java threads is to map each thread to an OS task. However, the kernel services used to manage OS tasks are generally optimized to handle a limited number of tasks, and may use priority schemes and communication primitives that may not suit Java. Alternatively, the JVM may be implemented as a fixed number of OS tasks (one, for instance), and manage Java multi-threading internally. In this way, the JVM has complete freedom to tailor multi-threading in Java programs, to the needs of Java. For instance, NSIcom's JSCP™ implements parallel execution through

a platform-independent polling machine and supports dynamic thread stacks. Dynamic thread stacks can grow and shrink as circumstances require, and eliminate the problem of stack overflow. The mechanism obviates the need to permanently allocate a worst-case size stack for every thread, and provides a level of robustness that cannot be attained otherwise. A JVM with complete built-in support of threads becomes independent of the idiosyncrasies of the OS task support. Again, this contributes to the ease of porting the JVM to another platform. In addition, the JVM vendor will be able to guarantee that a Java program will display identical timing behavior on altogether different platforms.

Multi-tasking: Many embedded products will need to run more than one Java application, independently, at the same time. One simple way to do this would be to instruct the OS to start a new instantiation of the JVM for every Java application. The problem with this approach is that every instantiation of the JVM duplicates the JVM core and all Java System classes. The associated resource costs are prohibitive for most embedded systems. By making the JVM reentrant, duplication of the JVM core is avoided, but the reuse of Java System classes remains a problem. The most thorough solution is to provide multi-tasking at the JVM level. This makes the JVM into a multi-processor, which is able to run parallel Java programs that have private resources and private instantiations of program classes but share Java System classes. Devices that run more than one Java application will likely need a program manager task to start, monitor, and manipulate (suspend, resume, change priority, stop) Java programs. Therefore, a multi-processor virtual machine must come with a JVM-monitor module that provides the services required by such a program manager.

In summary, the encapsulation of OS-like functionality inside the JVM is a generally attractive proposition. It simplifies the interface between OS and JVM, and thereby the effort of porting the JVM to another platform. It maximizes the platform-independency of Java, and the likelihood that Java programs developed on a desk-top machine will run in the embedded device in the exact same way. Because the JVM is implemented as a fixed number of tasks with fixed resources, interference between the JVM and the other native functionality can easily be controlled. If no other native functionality exists, it may be possible to strip down the OS to a very small kernel that provides only the simple primitives needed by the largely self-contained JVM.

2. Configurable Virtual Machines: the use of Software Components

The architecture of a JVM should take into account the diversity of embedded systems. That is, the JVM must be designed to be highly configurable to device features. Hence, it is important to assess to what degree a JVM can be fine-tuned to a given platform. Two dimensions of configuration should be considered:

1. The ability to leave out code that relies on features that are not supported by the hardware, and to so minimize the size of the JVM for the given platform. For example, internet access needs only be supported if there is a TCP/IP stack, AWT only if there is a screen, File IO only if there is a file system.
2. The adaptability of the JVM to characteristics of device features (the speed of the internet connection, number of sockets, the size of the monitor, the number of

colors, the speed of File IO, available memory) and user wishes (debug options, look-and-feel of AWT, settings for garbage collection).

To be able to effectively strip down a JVM it must be constructed in terms of loosely coupled software components that call each other's services. For example, NSIcom's JSCP virtual machine is composed of loosely-coupled software components. Its main components are displayed in Figure 1.

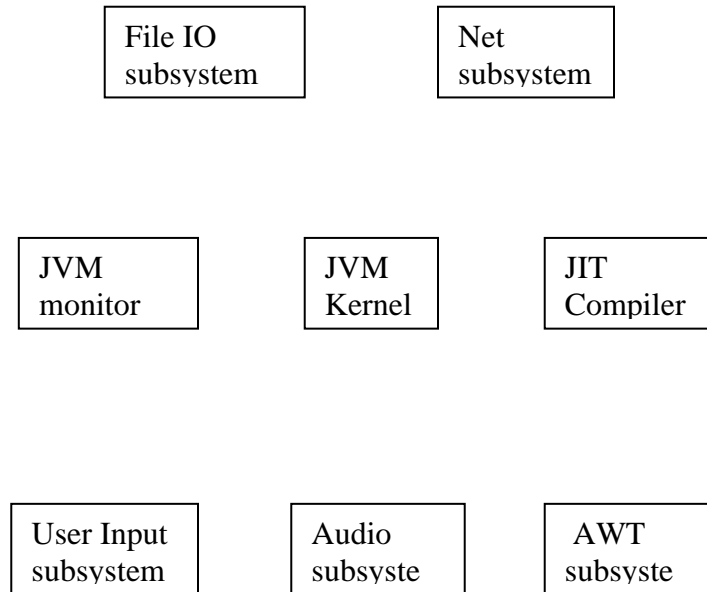


Figure 1: Main components of JSCP Java Virtual Machine. See section 1 for the JVM monitor, the JIT compiler is discussed in section 3.

One may think of loosely-coupled components as Active-X components or Java Beans, though there are many ways to implement them. The essential thing is that components can be left out in a simple way, without any changes to other components. Notice that a JVM may have software components not directly related to device features, but rather to application requirements, or user wishes. Examples are the Just-In-Time (JIT) compiler in Figure 1, or components for Remote Method Invocation and Dynamic Library support.

Configurability is a great asset for a JVM vendor that supports many platforms. At the same time, it enables the end-user to fine-tune the JVM. A JVM that is composed of software components can be distributed as a collection of dynamic libraries to enable the addition or removal of functionality after installation. However, compile-time configuration is generally more practical and gives better results.

3. Precise Just-in-Time Compilation

A major stumbling block for the acceptance of Java technology is the performance of Java software that is executed through a byte-code interpreter. To enhance efficiency, Just In Time (JIT) compilers are used, which, during execution of a Java program, translate some or all of its byte codes into native code for the processor of the platform. State-of-the-art compilers, developed for desk-top machines, take Java methods as units of compilation. That is, they compile Java methods into native functions. The biggest problem of JIT techniques is the increase of memory usage. Most embedded systems simply do not have enough memory resources for the conventional forms of JIT compilation. Truly scalable JIT technology, which achieves maximal efficiency improvement for all memory sizes, may well be the most decisive factor for the acceptance of Java in the consumer market.

The most resource-aware method-based JIT techniques keep in compiled form only those methods that consume most execution time, through a dynamic process that compiles and un-compiles methods as execution characteristics change. This works quite well as long as programs behave according to the 90/10 rule: 90% of execution time is spent in 10% of the code. The implication is that a significant efficiency improvement can be obtained by compiling only the most active methods of a Java application. However, this mechanism does not suffice to optimize speed-up per unit of memory occupied by JIT-compiled code. The reason is that the 90/10 rule not only holds across methods, but within methods as well. That is, methods generally contain a lot of overhead in the form of byte codes that are rarely executed. Put differently, the average branching instruction has quite unequal branching probabilities. An extreme example is the code needed for exception handling in Java methods. Exception-handling code is almost never executed, but nevertheless compiled by method-based JIT compilers.

NSIcom developed a precise form of JIT technology, which is able to limit compilation to chunks of active code within methods, such as most-taken-paths through often-invoked methods, and the bodies of often-executed loops. Rarely executed code, such as code for exception handling, is always left to the interpreter. Compiled code that ceases to be active is un-compiled. The mathematics of precise JIT compilation is complex, but the following computation, based on idealized assumptions, gives an intuitive understanding. Assume that the 90/10 rule holds literally across methods as well as within methods, and assume that JIT-compiled byte codes run 10 times faster than interpreted byte codes. Then, complete compilation of 10% of its methods may make a Java program about 5 times as fast (what takes 10 seconds before compilation, takes $1+9/10=1.9$ seconds after compilation). If a method is compiled through precise compilation, it is similarly sped up by a factor 5 instead of 10. As a consequence, through precise compilation of 10% of its methods, a Java program can be sped up by a factor 3.5 (from 10 seconds to $1+9/5=2.8$ seconds), if there is enough memory to store only 1% of the program's byte codes in compiled form (10% of the byte codes of 10% of the methods).