Embedding Java

Are current implementations of Java too big, too slow, too unpredictable, and functionally incomplete for use in embedded systems? Ron Workman looks at alternative ways to implement Java virtual machines to address these legitimate concerns.

Sun Microsystems estimates there are more than 400000 developers using Java today. While there are many documented success stories of Java's use for corporate or enterprise applications, there are few public illustrations of Java technology for embedded systems development.

In order to look at the current situation more objectively, I will separate the term 'Java technology' into two aspects – specification and implementation.

As a long time developer and provider of virtual machine technologies, it is my opinion that the current specifications for Java – including Enterprise Java, Personal Java, and Embedded Java – are not at fault, **Fig. 1**. But rather, most current implementations – particularly for Personal Java and Embedded Java – are not viable for many embedded systems.

Sun Microsystems introduced Java in 1995 as a new programming language and runtime environment ideally suited for Internet-related applications. Building on its long-standing corporate themes, 'The network is the computer,' Sun recognised that more powerful microprocessors were beginning to be used in many consumer devices, and increasingly, they have been connected to networks.

Network connectivity began with workstations and personal computers, but was quickly followed by printers, scanners, copiers, and various other types of office equipment.

More recently, newer devices such as personal digital assistants, Web televisions, pagers, smart cell phones and even wristwatches have all been enhanced with microprocessors and connected to networks.

Sun's network vision

Prior to Sun's introduction of Java, the network was viewed primarily as a vast system for storing and serving up relatively static information. Then, the phrase, 'The network is the virtual disk drive' was probably more fitting.

Today, Java's promise is to extend the usefulness of networks by providing an efficient means for storing and distributing dynamic and extensible functionality to networked computing devices.

The Java architecture is comprised of several distinct but inter-related technologies, each of which is defined in specifications from Sun Microsystems. These technologies include the Java programming language, the Java virtual machine, and the Java API, **Fig. 2**.

This article focuses on the Java virtual machine and aspects of its implementation that affect Java's suitability for embedded applications.

As with all virtual machines, the Java virtual machine defines an abstract computer. Its specifications define the functionality that every Java virtual machine must provide, but allow almost unlimited freedom to the designers of each implementation.

For example, each Java virtual machine can use any technique to execute Java 'bytecodes'. In fact, the Java virtual machine can be implemented in software or hardware, or varying degrees both. This flexibility in the specification for the Java virtual machine was intended to allow for implementation on a wide variety of computers and embedded devices.

Classes at the top

At the top level, a Java virtual machine's primary purpose is to load Java class files and execute them, **Fig. 3**. During execution, there are two aspects of any Java virtual machine implementation that have the most impact on the suitability of Java for the specific application. One is the methods used for executing the bytecodes, and the other the management of system resources – primarily memory.

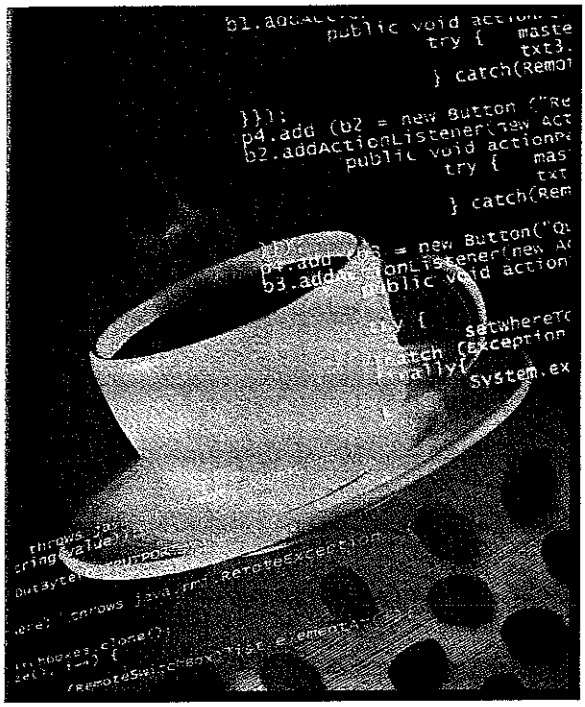
The default method for executing bytecodes by a Java virtual machine is interpreted execution. Whereas C/C++ applications are compiled to a native instruction set, and are stored in a single executable file, Java applications are compiled to Java bytecodes, and are stored in separate class files for loading and execution by the Java virtual machine.

When the Java application is run, the Java virtual machine loads the class files and interprets the bytecodes as a stream of instructions. Typically, an interpreted-only method for execution by a Java virtual machine will result in performance which is 10 to 15 times slower than an equivalent natively compiled C/C++ program.

In the plus column, an interpreted-only version of a Java virtual machine generally has a relatively small memory footprint – implementations are typically 1/2 to 2Mbyte of ROM.

However, there are other techniques that can improve the speed of bytecode execution. For example, just-in-time, or JIT, compiling can enhance application performance up to ten times over interpreted-only execution.

Rather than only interpreting bytecodes, each time the Java



virtual machine encounters new bytecodes for the first time, it invokes the JIT compiler to compile that code to native instructions. These native instructions are then stored by the Java virtual machine and are reused the next time the code is required by the Java application.

Performance comes at a price though. JIT technology typically requires four times the amount of memory of an interpreted-only version of a Java virtual machine. It compiles all code that it encounters, and often requires a disk and virtual memory for paging the compiled code segments.

Because of this overhead, JIT-based execution is suitable for computing platforms that have powerful processors, substantial system memory – 32Mbyte or more – and generally a fast disk drive to support swapping. It is not suitable, however, for the vast majority of embedded systems which are generally far more resource constrained.

Java for embedding

A bytecode execution method more suitable for embedded systems is 'adaptive dynamic compilation'. Similar to the just-in-time alternative, adaptive dynamic compilation uses on-the-fly compilation technology to improve the performance of bytecode execution.

Unlike a JIT implementation however, adaptive dynamic compilers are generally smaller, more selective of the bytecodes that they compile to native code, do not require virtual memory, and adapt to the available system memory.

The size of an adaptive dynamic compiler is typically measured in terms of tens of kilobytes of memory for the compiler itself. During interpreted execution of the bytecodes, the Java virtual machine monitors the application and determines where the execution bottlenecks reside. The Java virtual machine then invokes the adaptive dynamic compiler – which may be implemented as a thread – to compile the segment of bytecodes that are executing repeatedly.

Depending on the implementation, the adaptive dynamic compiler may compile the entire class, a single method, or only a block within a method. The resulting native instructions are stored in memory only for fast access.

The Java virtual machine may then find and invoke compilation for the next most frequently executed code. This process is repeated until all available code buffers have been used.

Because the dynamic compiler is adaptable, when it encounters code that is executing at greater frequency than some previously stored native instructions, it will compile these new bytecodes. It will then store them in a code buffer containing less frequently used code.

The user can configure the amount of system memory used

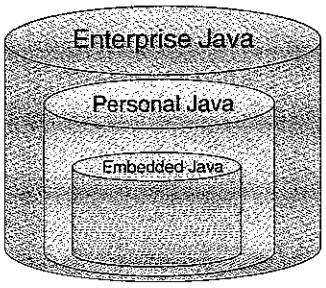


Fig. 1. There's a version of Java to suit all needs.

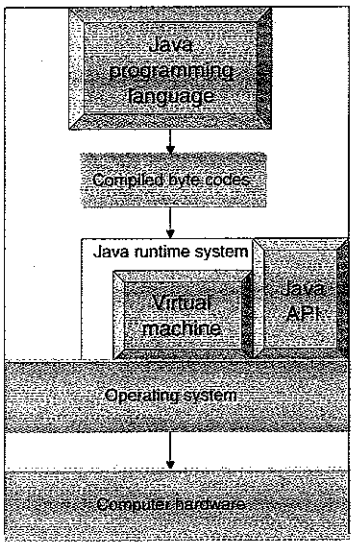


Fig. 2. Java's architecture comprises several distinct but inter-related technologies.

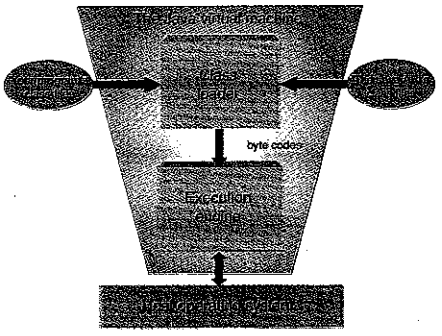


Fig. 3. At the top level, a Java machine's primary purpose is to load Java class files and execute them.

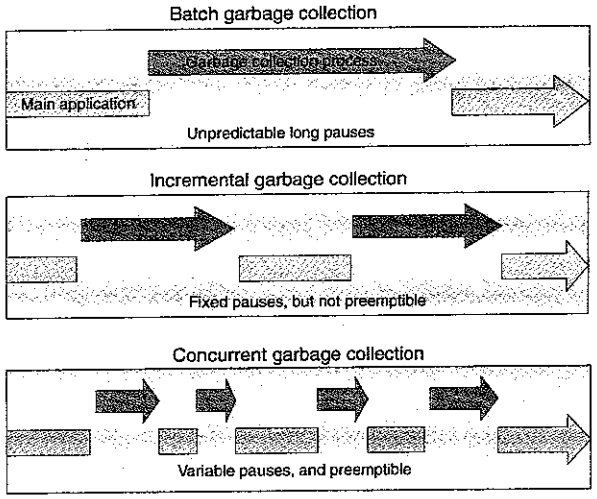


Fig. 4. In Java, garbage collectors aid the task of memory management. There are different ways of implementing garbage colectors, and not all are suitable for embedded applications.

for compilation, and the size and number of code buffers used for the compiled native instructions. Applications may also be given explicit control over the adaptive dynamic compiler's thread priority during execution for greater predictability, providing more suitable behaviour for embedded systems.

Reliability issues

The efficient management of memory is particularly critical for embedded systems, where predictable behaviour is required. The Java virtual machine plays the central role in managing memory. In fact, the application itself only makes requests for memory allocation for new objects and does not explicitly release unnecessary memory.

The release of memory is managed automatically by the Java virtual machine. Each Java virtual machine implements a 'garbage collector' to provide the memory management facilities.

There are several different implementations of garbage collectors, and some are more suitable for embedded systems than others, **Fig. 4**. One of the primary concerns of embedded systems developers is that if the garbage collector runs a complete batch cycle, and cannot be preempted, this will render Java unsuitable for the embedded application. In other words, it will make it unpredictable.

Technical support

Insignia Solutions is a leading provider of virtual machine technology that dynamically optimises the use of available system resources. *JENE*, Insignia's implementation of Java specifically designed for embedded systems, allows developers to create reliable, efficient and predictable embedded devices that are enabled by the company's 'embedded virtual machine.' The publicly held company's US headquarters are in Fremont, California, and its main research and development facilities are in High Wycombe, England. Sales and marketing departments are located in Fremont and High Wycombe.

For additional information on Insignia and its products, call 800/848-7677 in the United States and +44 1628 539 500 in Europe, or visit the company's web site at http://www.insignia.com.

Some implementations of garbage collectors are referred to as 'incremental.' This suggests that they can recycle memory in a stepwise fashion rather than garbage-collecting all memory in a single pass. Although this garbage collector may not be preemptible, this type of implementation should lead to more predictable behaviour than a batch garbage collector.

Since the incremental garbage collector may still block the application, however, the risk remains that the pauses will impact the application. If the garbage collector is defined as 'concurrent,' this suggests that it performs garbage collection incrementally, that it is fully preemptible, and that it should provide the most predictable behaviour of all.

In addition to its mode of execution – i.e. batch, incremental, or concurrent – garbage collectors may perform their duties with varying degrees of efficiency and effectiveness.

There are two basic approaches to separating live objects from garbage: 'reference counting' and 'tracing.' Reference counting is an older garbage collection technique. It involves maintaining a reference count for each object on the heap. Essentially, the reference count is incremented for each new reference to a given object and decremented when the reference to an object goes out of scope.

All objects with a reference count of zero can be garbage collected. However, among other disadvantages, reference counting suffers from the overhead of incrementing and decrementing the reference count each time the object is referenced.

Tracing garbage

A more suitable method for Java virtual machines is the tracing garbage collector technique.

Tracing garbage collectors trace the object reference tree starting with root nodes. Objects that are encountered during the trace are marked. After the trace is complete, unmarked objects can be collected as garbage.

During the tracing process, garbage collectors may either use either a 'precise' or a 'conservative' approach to identifying references to objects. The conservative garbage collector may not distinguish between genuine object references and look-alikes – 32-bit integers for example.

Although this approach may be slightly faster, it may also lead to memory leaks. A precise garbage collector, on the other hand, understands the differences between true object references and look-alikes, and frees all unreferenced objects appropriately.

One final important aspect of garbage collection to consider is whether the garbage collector has a strategy to combat heap fragmentation. Given the limited amount of memory available for most embedded systems, a concurrent, precise, compacting garbage collection strategy should provide for the most predictable system behaviour.

In summary

The specifications for the various Java platforms have generally been well conceived. If not perfect at first, they are certainly evolving in the appropriate direction to address the requirements of the identified classes of computing devices.

Now that the specifications are stabilising, and the vendors are getting down to the business of delivering suitable implementations of Java, I expect to see an increase in the use of Java for embedded systems.

With the implementation of the appropriate technologies described in this article, many embedded developers may soon discover that the benefits of Java can be delivered in a package that is small enough, fast enough, and predictable enough for their next embedded development project. ■