

Virtual Machine Technology: Managing Complexity and Providing Portability for Embedded Systems

IBM's VisualAge for Embedded Systems Virtual Machine provides an independent Virtual Machine that is both designed to be compatible with the Java Virtual Machine standard and developed from the ground up for embedded system applications. Using a Virtual Machine, such as VisualAge, provides embedded developers with the tools needed to design and build devices with communications facilities, to download and run applications and new platform components, while maintaining compatibility with the Java platform. Designing to a Virtual Machine reduces the complexity of porting to other processors and hardware devices, and can improve time to market now and for future releases. This article gives you the details.

The key to integrating Java(TM) successfully into embedded systems is the virtual machine technology. A Java virtual machine is software that enables programs written in the Java programming language to execute unmodified on a wide range of operating systems and hardware platforms, by acting as a software "computer" that presents features of a single hardware and operating system to the Java program.

A Java virtual machine is a layer of software that runs on top of a host operating system and provides a single, consistent API to Java programs. It loads the compiled bytecode files (called class files), dynamically loading and linking additional class files as necessary. The program executes entirely within the environment provided by a Java virtual machine, obtaining all of its services from that Java virtual machine. This makes Java programs highly portable between different computing architectures, and different types of embedded devices. The Java API which the program uses to obtain services is historically defined by Sun Microsystems, and consists of a set of classes, organized in groups called packages.

Sun provides its own version of a Java virtual machine, as a part of its Java platform implementation, that usually represents the standard for VMs. But there are no secrets in the technology behind virtual machines (VMs), and it's possible to develop a VM independent of Sun's version of a Java virtual machine. Sun's Java virtual machine is a generic platform for a wide range of uses, and not optimized for any specific hardware or application.

For embedded systems use, a VM should be optimized for small and resource-constrained devices. Using a generic platform is often not feasible when the needed computing resources aren't available. Having a stable yet customizable VM is especially important in embedded systems, where the myriad of different processor, real-time operating system (RTOS), and packaging choices can make design decisions both complex and time-consuming. The VM insulates com-

munications routines and Java applications from any changes that are implemented to the underlying hardware platform.

A VM optimized for embedded systems provides for application portability among devices, while maintaining a compact and highly flexible package that can be adapted to new and innovative designs. And using a VM as an application platform has significant advantages in time to market, reliability, and life cycle costs. Engineers can be assured a stable and common platform across processors and RTOSs, so applications developed for one system may not have to be rewritten for new generations or derivatives of the device.

SIZE AND CUSTOMIZABILITY

A complete Java virtual machine workstation platform is big - up to 10 megabytes for the full Java package. Often, desktop workstations require significant processing power and memory in order to run the full Java virtual machine. This fact may give some engineers the mistaken belief that any Java platform implementation must be too big for any embedded system.

But that is misleading for two reasons. First, much of a Java virtual machine is simply not needed for many embedded systems. Much of the graphics and windowing toolkit, for example, probably wouldn't be required for a small handheld device with a limited user interface, or not needed at all for a device like a network router. For small or limited-purpose applications, a Java virtual machine can be substantially reduced in size. The minimum size is highly dependent on the application, but it need not be considered out of the reach of memory and storage capabilities of today's embedded devices.

Second, a big Java virtual machine obscures the fact that Java programs in general are far smaller than natively compiled programs. If the VM, its supporting native code components, and the class libraries are thought of as Java "entry fee," then the application code can be viewed as "payload." The Java language

makes a small payload go a long way. For example, a simple "Hello, world" executable image using C++ with Motif is about half a megabyte. The same program written in the Java language is about 12 kilobytes.

Of course, the VM for Java programs adds much more to the memory overhead than is required by the C++ image. For just a single Java program, the overhead may well outweigh the small size of the Java application in many cases. But in many designs this may be the exception rather than the rule. The balance between the overhead of the VM and the capability of the applications involves the following factors:

- The number of applications to be run on the device
- The complexity of those applications
- Whether the applications are loaded from a high-speed device like a disk drive or flash memory, or a low-speed device like a modem

If the purpose of the device is to run several different applications, and at least one of these applications is of moderate complexity, or if applications are routinely downloaded from servers, the overhead of a Java VM is tolerable. In these circumstances, total memory and storage requirements can actually be less than that of a more traditional language.

One VM that address the execution needs of the Java language on embedded systems is IBM's VisualAge[®] for Embedded Systems virtual machine. Developed specifically for embedded devices, the VM supports running Java programs on a highly customizable virtual machine platform, with the ability to choose only those class libraries and support files needed by the target device. This platform has the infrastructure to dynamically download components from a server if they are needed to execute a given program. New programs and updates to the VM can also be downloaded on demand. This makes the VisualAge for Embedded Systems VM an effective alternative to standard embedded systems designs.

DETERMINISM AND REAL TIME

Possibly the biggest limitations to the use of the standard Java language in embedded devices is its historical lack of determinism and the inability to guarantee a real time response. Much of the reason for these limitations is due to the automatic allocation and deallocation of memory by the VM. Rather than requiring the program to make calls to functions like `malloc()` and `free()` for managing memory, a Java VM automatically allocates memory when a variable is instantiated, and periodically calls a routine to collect unused memory.

Garbage collection is the automated process of searching through memory for unreferenced memory, and reclaiming it for the free memory list. A Java virtual machine uses a software algorithm for garbage collection to return memory to the system when it's no longer in use. This saves the programmer from manually having to return memory to the system, reducing the likelihood of memory errors in the program.

The standard technique used for Java garbage collec-

tion is termed "conservative." This technique examines memory to determine whether or not it contains a valid address. This works much of the time, and is simple to implement and doesn't have a lot of overhead. But it doesn't always work; if a variable contains an integer that just happens to be in the range of valid addresses, it will be thought of as active, and not reclaimed.

On a desktop system, missing a few bytes of memory until the computer is shut down is no big deal. On a resource-constrained embedded device, this type of memory leak will seriously affect reliability. Rather than the standard conservative garbage collection algorithm used in a normal Java VM, a garbage collector for an embedded VM has to be aggressive. An aggressive VM has the ability to determine if object are no longer referenced and hence no longer required. There are several techniques for doing this. One is tagging memory and using reference counts. Memory will be reclaimed when the reference count reaches zero. Another technique scans memory to find unreferenced objects. While these techniques requires slightly more processing overhead than a conservative algorithm, they prevent memory leaks that a resource-constrained device can't afford.

A garbage collection algorithm can also affect real time performance because another Java task can't run during the garbage collection process in that VM. Depending on the processor, garbage collection might take 100 milliseconds or more. If the application steadily allocates objects, and does not manually force garbage collection often enough, a Java virtual machine will garbage collect when it thinks it has an opportunity, or when it runs out of memory.

Sun's garbage collection algorithm cannot normally be interrupted, making it difficult to use in some real time scenarios. Sun's version of a Java virtual machine has a background garbage collection thread that can start the collection process if it sees that the system is idle. When the garbage collector is invoked from this thread, it can be preempted by other threads, but it could need a long time to complete preemption. Last, if Sun's Java virtual machine does not find enough memory to satisfy a memory allocation request, it will execute non-preemptable garbage collection.

An embedded garbage collection algorithm must be able to be preempted by a higher-priority real time process, so that the device can offer a real time response when required. It must also be able to work incrementally; that is, it must be able to remember where it left off and continue from that point. Otherwise, the system will spend an inordinate amount of time in garbage collection.

Ideally, a system designer should be able to choose the garbage collection algorithm that best meets the requirements of the system. For higher processing requirements where real time response may be less critical, a more conservative algorithm can be included in the VM. Conversely, if real time response is important, an aggressive algorithm using incremental garbage collection can be substituted. This type of flexibility is critical in the world of embedded systems, where radically different designs are targeted toward

very different purposes.

COMBINING PERFORMANCE AND REAL TIME RESPONSE

Until recently, embedded developers have had to wait for Sun to develop requirements for a real time version of the Java language and implement them in one of the Java platform implementations. This presented a time-to-market issue, because Sun took over two years to define the specifications for personal and embedded Java implementations.

But choice with compatibility hasn't really been an alternative, especially in the embedded realm. IBM's VisualAge for Embedded Systems VM provides an independent VM that is both designed to be compatible with the Java VM standard and developed from the ground up for embedded system applications. The VisualAge for Embedded Systems supported runtime environment includes the QNX/Neutrino real-time operating system (RTOS) running on Intel x86, PowerPC, or MIPS microprocessors. The ARM 7100 processor is also supported. Additional RTOS and microprocessor targets are planned to be supported in the future.

The VisualAge for Embedded Systems product provides a high-performance, adaptable VM for the executing Java applications on embedded devices, such as pagers, cellular telephones, screenphones, digital set-top boxes, and car navigation systems.

Thanks to a VM that includes advanced garbage collection and dynamic component downloading, the VisualAge for Embedded Systems product provides the ability to meet both the memory constraints and the real time requirements of modern embedded systems. It also includes the VisualAge for Embedded Systems development environment, which enables developers to both build the application and customize the VM on a development workstation, and then download the desired components to the embedded target.

Using a VM such as the VisualAge one provides embedded developers with the tools needed to design and build devices with communications facilities to download and run applications and new platform components, while maintaining compatibility with the Java platform. Designing to a VM reduces the complexity of porting to other processors and hard-

ware devices, and can improve time to market now and for future releases ■

Mr. Clohessy has over 20 years of embedded system engineering experience in several industries, including telecommunications; air traffic control; air, sea, and land military systems; and more recently, next-generation commercial systems.

As a senior executive managing the engineering and technology departments of a systems engineering company focusing on embedded real-time applications and products, Mr. Clohessy was involved in virtually all aspects of the projects, including the identification of opportunities, the sales cycle, systems design, detailed design, manufacturing, and acceptance.

Mr. Clohessy is one of the founders of DY 4 Systems, a Canadian VMEbus computer firm concentrating on high-reliability embedded systems for demanding government applications. Through extensive standards involvement, Mr. Clohessy was one of the pioneers that brought commercial-off-the-shelf strategies to high-end military and government projects around the world. In 1992, Mr. Clohessy moved to Object Technology Incorporated, to lead the Embedded Systems Group. OTI is a recognized leader in object-oriented programming and embedded systems, with a history in Smalltalk development tools and embedded Smalltalk going back to the mid-1980s.

OTI was purchased by IBM in 1996. OTI provides the base technology runtimes and tools for VisualAge Smalltalk, VisualAge Java, and VisualAge for Embedded Systems. Based on OTI's long history of building virtual machines and tools for embedded systems, OTI is a key element of IBM's Pervasive Computing Strategy.

IBM, PowerPC and VisualAge are trademarks of IBM Corporation in the U.S. and/or other countries. Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the U.S. and/or other countries. Other company, product, and service names may be trademarks or service marks of others.



We welcome news from your company through press-releases that can be submitted on our website at <http://www.dedicated-systems.com/VPR>

For more information contact Nico Van Wijmeersch.

Phone. 32-2-520.55.77 or Fax. 32-2-520.83.09 or Email. info@dedicated-systems.com