Java's Virtual World

Java Components Include High-Level Language and Virtual Machine

by Mark Lentczner, Glyphic Technology

After several years of research and shifting directions, a small group at Sun Microsystems brings us the Java language and Java virtual machine. While virtual-machine implementations of languages are nothing new, this is the first time one has been at the core of a system receiving so much attention —even before version 1.0 was released.

The Java story gets confusing because there are really several different parts that make up the Java scene, as Figure 1 shows. At the core, there is the Java language, which is in the C family of languages. It has object semantics similar to those of C++ but adds an object memory that is heap allocated and garbage collected. It is most notable for its support of dynamic linking, run-time code loading, and safe code execution.

The Java language could be compiled to any machine architecture. However, the other core piece of the Java universe is the virtual machine (VM). The Java VM implements an abstract processor architecture. This virtual machine can then be implemented in software on a variety of operating systems and hardware. A Java program compiled to the Java VM instruction set can be loaded and run on any conforming implementation of the Java VM. This technique allows compiled Java programs to run on different platforms without recompiling, so long as each platform has a virtual-machine implementation.

On top of this core—the language and virtual machine—are the more famous pieces: HotJava, JavaScript, Java programs (sometimes called applets), and the libraries that support them. The Java class libraries provide Java programs with standard objects and messages for manipulating common data structures, operating-system entities (such as files and network streams), and user-interface components.

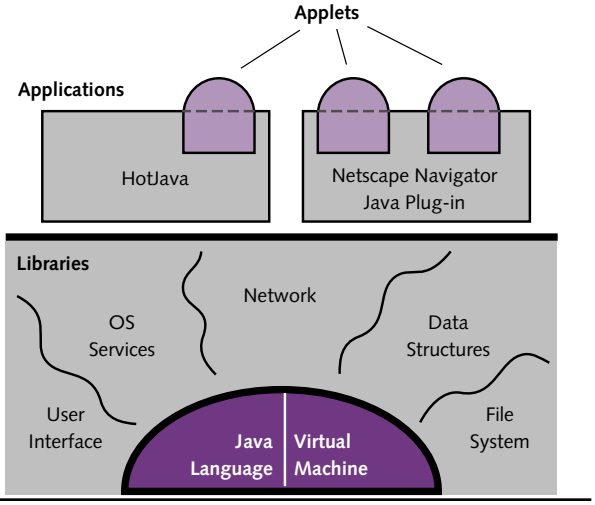

Figure 1. Java applets can be executed using HotJava or other Web browsers. These browsers and other Java software execute using the virtual machine (VM), with some library calls executed using the operating system.

HotJava is a World Wide Web browser written in the Java language, making use of the Java libraries. By relying on the architectural neutrality of both the Java VM and the Java libraries, it can download and execute Java applets graphically embedded in a Web page. The Netscape browser, which is not written in Java but includes a Java VM implementation, will do the same thing.

JavaScript is entirely another beast. A language that Netscape was working on, it was later renamed as part of the Java family. It bears only superficial similarities to Java and is different in many significant ways: object model, libraries, and implementation. In fact, it makes no use of the Java VM or Java language at all!

Java Language Links Modules at Run Time

The Java language is yet another object-oriented extension to C. Java shares most of C's heavy emphasis on scalar data manipulation (integers, floats and characters), structured programming constructs, and type checking. Beyond C and C++, Java requires object-oriented techniques for all programming (see sidebar), has stricter type rules, and does away with many extraneous language features.

The biggest difference is that Java programs go through a different life cycle than programs written in more traditional languages. Individual program units, consisting of one or more classes, are compiled independently to the Java VM instruction set in a format called bytecode in Java jargon. At this stage, these modules (called .class files) can be exchanged around the network.

When users want to run a Java program, they load the module into an implementation of the Java VM (for example, running as part of Netscape Navigator). The Java VM may then load additional .class files as needed (assuming the user has them, or they can be obtained across the Internet) and runs the program. Only at this point are references between different modules resolved, a step performed in other programming environments by a linker long before the user gets the program. Because this link step happens at run time, and the particular versions of modules are not determined until run time, systems like Java's are said to have dynamic linking. In Java, this step is known as resolution.

Virtual Machine Memory Is Object Oriented

To understand how the Java VM operates, one has to understand how it views memory. Because the Java VM specifica-

tion is of an abstract machine, it gives few details on the layouts of memory data structures or the particulars of allocating and using physical memory. These details, and trade-offs of various techniques, are left to the implementer of a Java VM. The specification describes at a high level how instructions operate on three distinct regions of memory: objects, stacks, and class-constant pools.

While little is said about memory for objects in the Java specification, the operational description implies certain implementations. All program-defined and -allocated memory structures are stored in objects. These objects can contain scalar data and references to other objects.

Programs specify what data fields each of their objects must have. The actual memory layout of an object, however, is determined at run time according to the policies of the VM implementation that is running it. The layout of an object must be determined at this time because it can depend on information from other modules that are not loaded until run time. This is a consequence of dynamic loading.

There are several operations that various VM instructions must be able to perform on objects: get and set fields, perform message lookup (finding the particular method for a given message), and test for object class membership. As long as these operations are defined, an implementation of the VM is free to use any technique for storing objects.

While objects are explicitly created by the NEW instructions, there are no explicit freeing operations: object memory is garbage collected, freeing unused objects automatically. Garbage collection trades increased system overhead for ease of programming and the elimination of some programming errors (no dangling pointers).

This trade-off, and techniques for its implementation, are thoroughly discussed in computer-science literature and can be argued in many different ways. But one thing is clear: good garbage collection is a complex task. It will consume significant processing resources in any VM and will require a large coding effort. In hardware versions of the Java VM, garbage collection will probably be implemented in software with some assistance from the silicon.

Stack Contains a Frame for Each Method

The Java VM's stack holds a last-in-first-out (LIFO) sequence of frames. Just as in a conventional processor, each stack frame holds information needed for the execution of a message (or function). The Java VM specification says even less about the stack and frames than about objects; it doesn't even specify what is stored in them. A careful reading of the specification, however, shows what information is needed for each frame.

Stored directly in the frame are:

- Message arguments
- The receiver of the message (the *this* variable, stored as argument 0)
- Local variables
- The value stack

Object-Oriented Programming

In an object-oriented program, each data type and the functions that operate on it are grouped together into units called classes. When the program allocates a section of memory to use for one of these data types, the data is called an object. An object always remembers its class (data type). When the program needs to call one of the functions associated with a class, it does so by sending a message to an object. This message invokes the actual code for the function, known as a method, to execute.

An advantage of programming with objects is that two or more classes can define their own versions of the same function. For example, a binary-tree class and a linked-list class could both define their own insert functions. Then, when the program sends the insert message to an object, the appropriate insert function will be executed by checking the class of the object receiving the message at run time.

The frame also contains pointers to:

- The frame of the caller (with a saved PC)
- The class-constant pool
- Method information: bytecode instructions, exception handling, and debugging information

The layout shown in Figure 2 is but one way to store this information in memory. When a program sends a message to invoke a method, a new frame is created on the top of the stack. The arguments of the message are placed in the frame, and the local variables of this new frame are set to zero. A reference back to the calling frame is stored, so when the called method completes, the calling method can be resumed. Additional information that the virtual machine will need is also stored in the frame, including references to the method's code and its class-constant pool.

The Java VM is a stack-based machine: arguments are popped from the top of the value stack, and results are pushed back onto it. This value stack while specified as a separate stack, can be at the end of the frame stack. Java assumes that all stacks can grow without bound as needed.

The frame stack supports a single thread of execution. A multithreaded implementation could contain multiple stacks. Little is said about threads in the Java VM specification. Threads are mostly supported by the run-time libraries and routines. Support from the virtual machine is limited to instructions for manipulating synchronization monitors that mark uninterruptible portions of code.

Class-Constant Pool Contains Code, Static Data

The class-constant pool is the equivalent of the code segment in other architectures. This area stores all information about a compiled class needed by the virtual machine: bytecodes, constant data, class descriptions, method descriptions, and

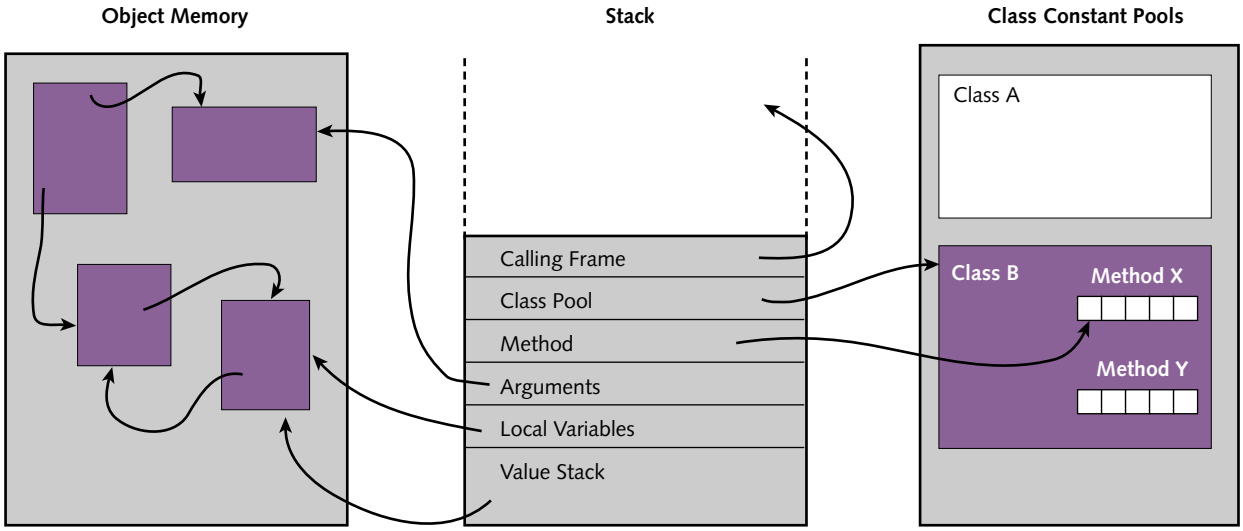

Figure 2. The Java VM uses three areas of memory. The stack holds a set of frames, which contain arguments and pointers. Program variables are stored in object memory, while constants and other static information are in the class-constant pool.

static-value descriptions. The specification defines the precise layout and byte ordering of this information for file storage. This definition ensures that compiled Java code can be moved from one system to another without recompilation. Once loaded into memory, however, this information can be stored in any format needed by a particular VM.

The class-constant pool is organized as a densely packed array of variable-length items. Because many instructions reference this array, however, any efficient implementation of the VM will need to use another format when storing it in memory. This need for an efficient memory format increases the memory requirements of a Java program, and the format translation adds to the load time.

Whether stored in the stack or in objects, Java data is strongly typed. Data is either one of the scalar types (integer, float or character) or it is a reference to an object. Although an object reference is like a pointer, it is not the same. Object references can be implemented in a number of ways—direct pointers, indirect table indices, etc.—and they do not allow arithmetic, unlike pointers in C.

Many virtual machines use tags to type data, but the Java VM does not, instead relying on the compiler to ensure type consistency. If the compiler emits an instruction to push an integer onto the operand stack, the compiler must not generate any instruction treating that stack location as a reference to an object. Given the design of the Java language, this restriction is easy to enforce.

Virtual Machine Provides Portability, Safety

There is nothing new to the virtual-machine, or interpreted, approach to language implementation. Among the more well known systems that use a virtual machine are the UCSD Pascal P-Code system, the Smalltalk-80 system, various Scheme implementations, PostScript, and Microsoft's P-Code engine for C and C++. With Java, Sun has followed in these footsteps for a number of reasons:

Dynamic Linking—This feature enables different parts of a program to be combined at run time without an explicit link step. Dynamic linking offers the portability needed for compiled Java code to be downloaded from the World Wide Web and executed by a Web browser running on a variety of platforms.

The size of the entities that can be dynamically linked determines the flexibility of the system: larger link units mean less flexibility. Java's unit of linking is the .class file, which contains one class and its methods. This choice puts Java at the flexible end of the scale. Other software systems are often too restrictive, requiring larger units for dynamic linking. (Most Unix systems, for example, require whole programs to be linked with certain libraries at run time.) Although an even more flexible design would have been better for interactive software development, Java's design works well for end users running applets.

Portability—The Java literature touts portability. To get Java applications to run on a new platform, says Sun, only the Java VM must be ported. Porting the VM, using Sun's reference implementation, involves writing only about 5,000 lines of code. One must also port the native portions of the standard Java libraries to a given platform, however, and that adds another 15,000 lines of code. Still, once this porting effort is done, any compiled Java program or method can be loaded and run.

This porting effort pales in comparison to the task of individually recompiling each program and library, as required by traditional programming languages. Other virtual machines that have portability as a goal, however, require a fourth or less porting effort than Java, based on the amount of newly written code required.

Another aspect of portability is the ease with which programs written in Java can run on different machines. This is an area where Java shines: due to dynamic linking, the well-defined VM specification, and the consistent libraries, programs written in Java can be executed on other platforms without even so much as a recompile.

Safe Execution—To ensure the safe execution of programs, the Java VM, like many virtual machines, does extensive error checking. For example, it is impossible to access the fifth element of a four-element array—if this is attempted, rather than fetch bad data, the virtual machine will detect the error and force an exception to system software. In an environment where bits of code are being downloaded from the Internet for execution, programs must be kept from crashing as much as possible.

The Java VM design has many built-in safeguards. For example, integers cannot be treated as pointers into memory; 16-bit values cannot be stored accidentally in the middle of 64-bit floating point variables, and functions cannot jump to random memory locations and execute data as code. Many of these safeguards reduce run-time performance. To reduce the run-time burden, the Java VM relies heavily on the Java compiler to generate code guaranteeing that many of these constraints are followed.

Realizing that Java bytecode could be generated in other ways, Sun added a module called the verifier. It checks the instructions in .class files when they are loaded into the Java VM to ensure they don't violate any constraints. This task is not simple: it is akin to proving a program correct without running it. The Java literature indicates that the verifier is very picky and errs in favor of code sequences it knows are correct. Although this method will work if the verifier knows all the tricks of the current compiler, how it will handle future compilers is unclear.

Another aspect of safe execution comes from the inherent safety of a system without pointer operations and with garbage collection. These two features eliminate many bugs before they even happen. It is impossible to reference an area of memory that isn't a valid, allocated object. It is equally impossible to reference objects that were prematurely freed from memory.

Code Size—Some virtual machines reduce code size, an important metric for some applications. Indeed, the primary purpose of Microsoft's P-Code system is to reduce program code size by an average of 40%.

In contrast, Java .class files can be up to 25% larger than the equivalent C++ code compiled for the x86 architecture. The figure gets worse when you realize that, due to the highly compact nature of the .class file, any fast implementation of the Java VM must use more main memory to represent its contents (perhaps as much as 50% more), while the x86 compiled code will only get smaller after linking.

In general, most of the extra size can be accounted for by the structures and information needed for dynamic linking (although other software systems achieve this feature with a somewhat lower memory cost). Many users would consider this trade-off reasonable. Sun representatives, however, have claimed a Java processor would be good for small embedded devices because of the greater density of Java code. If one were to look at just the instructions, this claim could be true, but adding the class-constant pool overhead (which is required) yields memory footprint figures that don't support the claim.

Not a General-Purpose Machine

Most of the design decisions in the Java VM are clearly motivated by the Java language design. This is common to many virtual machines, which gain power by being operationally similar to the languages they are implementing.

A notable feature of the Java VM is its hybrid approach to data: it has instructions for scalar operations and instructions for object-oriented programming. Most other virtual machines are either all scalar or all object oriented. Java's design leads to relatively good math performance at the expense of leaving math out of the object model: although many object-oriented languages can define new functions for integers, Java cannot. This dualism in the Java VM is clearly reflected in the Java language, and this is one instance where implementation considerations have restricted the language design.

This close coupling with the Java language prevents the Java VM from being a general-purpose machine. There are many constructs of other languages that cannot be implemented on the Java VM, such as C's function pointers, Smalltalk's blocks, Scheme's closures, Pascal's local functions, and PostScript's variable arguments. Other languages have different kinds of classes and objects and also have different variable-scoping rules. Thus, it would be difficult to support other languages on the Java VM. Implementing the Java VM in silicon would have a better chance of market success if the VM were more general purpose.

Execution Performance Is Sacrificed

The other big influence on the design of the Java VM is achieving acceptable performance. The standard VM is a very straightforward implementation of the virtual machine. The interpreter sits in a loop, fetching instructions, decoding them, and executing them. The Java instruction set is large *(see* **100403.PDF***)* and does not rely on tagged data to make a software implementation more efficient, requiring performance-sapping extract and compare operations on a standard microprocessor.

As a result, Java programs executing on the standard VM achieve only a small fraction of native performance, typically 3–10% of the speed of optimized C code compiled for the target processor. Well-known techniques, such as in-line caches and dynamic compilation, have been demonstrated to improve performance to 10–20% of optimized C code for languages with more features than Java offers. Sun is developing a "just in time" (JIT) compiler that recompiles

Java bytecode to the target processor at run time, significantly improving performance but adding memory overhead. Other companies are working on similar techniques, but like Sun's JIT compiler, none has yet been demonstrated.

Newer techniques, such as adaptive optimization and recompilation, have been developed by one of Sun's own research groups. This team has implemented the object-oriented language Self using a virtual machine with only seven instructions. This implementation runs programs at 60% of the speed of optimized C code, yet the Self language is fully object oriented and highly dynamic, and it contains many advanced features. Although these techniques might eventually help improve Java VM implementations, they imply that Java may not have needed to give up language features for speed, and that the large instruction set of the Java VM may be a premature optimization that makes it hard for a smarter implementer to achieve strong performance.

Java Great for Web Applications

The Java language and VM are an excellent application of dynamic, object-oriented ideas and techniques to a C-based language. The Java design sacrifices execution performance, code size, some high-level language features, and support for multiple languages. In return, it gains portability, safe execution, and dynamic linking. These trade-offs are appropriate for environments, such as the Web, that are highly dynamic. In these environments, code fragments must be combined at run time, and most execution is involved with the user interface, which can afford the drop in performance on today's fast desktop processors. Applets that run in Web pages downloaded from the network are a perfect application for Java.

Sun, however, is attempting to position Java to be all things to all people. Unfortunately, Java has missed being really good at some of its own goals: Java code isn't particularly small, and even when implemented in hardware, the Java VM will not be very useful for tasks outside the realm of the Java language. One would be hard pressed to create a feasible implementation of a cellular phone in Java, for example. Given all the uses now claimed for Java and the marketing power behind it, we wish Sun had included support for richer languages and environments and more powerful implementations from the start.

Virtual machines have been used by language designers and implementers for years with relatively little fanfare. Although the Java VM doesn't offer any new techniques, the sudden emergence of the World Wide Web gives Java a growing target market for which it is well suited. Sun must improve its Java technology before it is ready for other markets. M

Mark Lentczner is president of Glyphic Technology, a software consulting firm specializing in programming language design and implementation.

