condition codes, the bigger problems occur when emulating an ISA with condition codes on a host platform that does not support condition codes. Hence, it is preferable to avoid condition codes in ISAs, as in current V-ISAs.

5.6.6 Instruction Discovery

With arbitrary ISAs, code discovery is problematic (see **Section 2.6**). The root cause is indirect jumps to potentially arbitrary locations. When combined with variable-length instructions and embedded data, it is hard (or impossible) to discover code that goes beyond an indirect jump. Clearly, the solution is to restrict indirect jumps. In particular, in V-ISAs they are restricted to be explicit procedure (or method) calls and returns where the return addresses are protected from modifications. Then, because all conditional branches are to PC-relative addresses where the offset is a constant, control flow is statically discoverable. V-ISAs also separate data from code, although they do permit variable-length instructions. However, because all control flow is exposed and explicit and there is no embedded data, variable-length instructions by themselves do not pose a problem. The net result is that the virtual ISAs are explicitly designed so that at the time a method is first invoked, all the code contained in the method can be discovered immediately.

5.6.7 Self-Modifying and Self-Referencing Code

For most user applications, self-modifying and self-referencing code are not essential; in fact they are typically discouraged. Consequently, self-modifying and self-referencing code are most easily dealt with by simply making them impossible to do; such operations are not embodied in the commonly used V-ISAs. Interestingly, and somewhat ironically, the emulation engine running on the host platform may rely heavily on self-modifying code in order to give good performance. For example, writing translated

(or JIT-compiled) code into a code cache is a form of self-modifying code.

5.6.8 Operating System Dependences

As discussed in earlier chapters, when performing process emulation, it is guest operating system dependencies that are probably more difficult to deal with than instruction set dependencies. All instruction sets contain some elemental instructions that make them functionally complete—any function can be performed; it's just a matter of how efficiently it will be done. With operating systems, however, this may not be the case. Different systems may perform process management, file I/O, networking, graphics, etc. differently.

Hence, in HLL VMs, the preferred way to deal with OS dependencies is to arrive at some kind of least common denominator set of functions and to implement them with standard libraries. These standard libraries must be ported to all desired host platforms. This approach is more easily said than done, however, and on some platforms the fit may not be perfect. However, the library interface is usually at a higher level than a conventional OS interface, so the library writer has more flexibility in achieving at least an acceptable level of compatibility.

---

1. However, Sun (McGahn and O'Connor 1998) and ARM (**ARM 2002**) have developed processors that support the Java ISA.

2. Binary classes are often referred to as *class files,* but they do not, strictly speaking, have to be stored as files; hence, the more generic term *binary classes.*

3. However, system methods written in Java do not have to be trusted, strictly speaking; they can be verified just as any other method. Doing so would tend to increase system integrity, but it would also slow down the JVM start-up time, so it is not commonly done.

4. In the context of HLL VMs, translation is usually referred to as *compilation.*

[1] In the context of HLL VMs, translation is usually referred to as *compilation*.

Previous chapter
Chapter 4: Dynamic Binary Optimization

Next chapter
Chapter 6: High-Level Language Virtual Machine Implementation

Chapter Six

High-Level Language Virtual Machine Implementation

The collection of binary classes or modules that make up a program specify its operation in terms of the HLL VM *architecture*. It is up to the HLL VM *implementation* to carry out the specified operations. This chapter discusses HLL VM implementations, primarily by describing the Java VM, the best known of the HLL VMs. A CLI implementation is similar in many respects. We describe the major implementation components and then look at ways of improving performance in HLL VM implementations. We complete the chapter with a case study of a specific high-performance JVM, the Jikes research VM.

The organization of a typical JVM implementation (**Venners 1998**) is shown in **Figure 6.1**. The three major components are the class loader subsystem, the memory system, including a garbage-collected heap, and the emulation engine (sometimes referred to as the *execution engine*). These major components are, in turn, implemented with lower-level components. We first describe the major components briefly and then provide greater detail.

Memory and State Registers

The memory consists of an area for program code, a global memory area, and stacks for both the architected Java code and for native library code. The program area is implicitly defined for the most part; common sense tells us there must be some region for holding the program. However, program memory can only be accessed via the program counter; it is never explicitly accessed with load and store

instructions. Furthermore, the program counter contents can never be directly inspected.

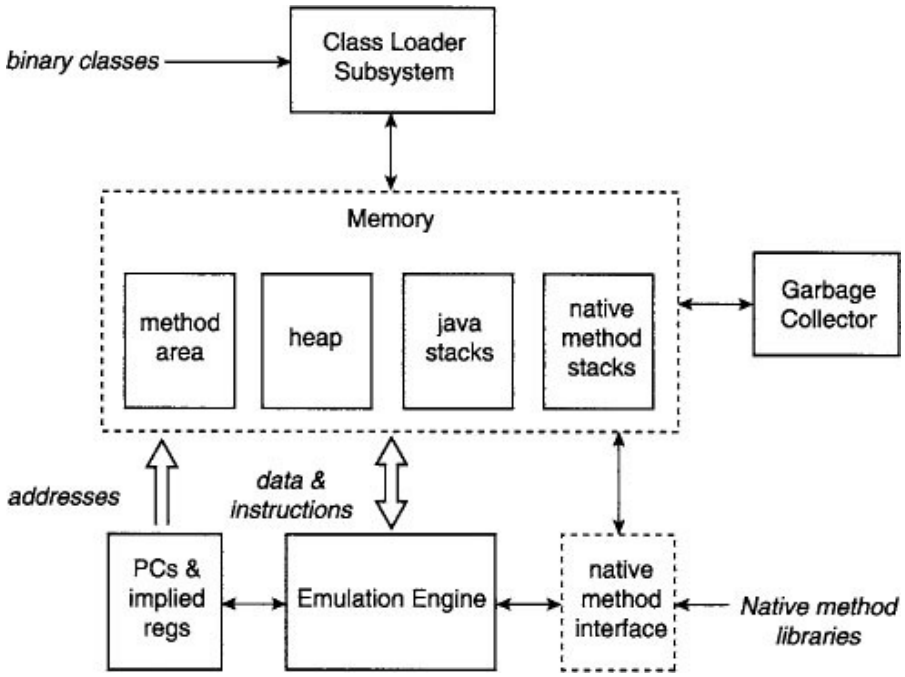


Figure 6.1 A Java Virtual Machine Implementation.

The Java stack area was described in **Section 5.3.2** and needs no further explanation. The native stack and its interaction with the Java stack are described subsequently as part of the native method interface. Associated with the stack is an implied register, the stack pointer. Just as with the program counter, the stack pointer is used by instructions for accessing memory, but it is not explicitly accessible; i.e., there is no way for a program to inspect the contents of the stack pointer. The exact size of the stack is not architected; that is, it is implementation dependent. If its size is ever exceeded by a running program, the StackOverflowError exception is thrown.

The global memory is a dynamically allocated heap. When a new object instance is created, its memory is allocated from the heap. As with the stack, the heap is of unspecified size. If the heap runs out of space, an OutOfMemoryError is thrown. To reduce the likelihood that this will happen, a garbage collector can be used to reclaim heap memory no longer needed by a running program

heap memory no longer needed by a running program.

Garbage Collector

Many Java objects are created, used, and then are no longer needed by the program. This occurs when the last (or only) reference to an object is destroyed or is overwritten. At that point, the object becomes *garbage,* because it is no longer useful to the running program. Because any real JVM implementation will have a limited amount of memory, it is advantageous to reclaim, or collect, the memory space used by the garbage objects so that the physical memory resources can be reused. Consequently, most JVM implementations have a garbage collector, which is responsible for finding those objects no longer needed in order to make room for other, new objects.

Emulation Engine

The emulation engine, supported by the native method interface and implied registers (i.e., the program counter and the stack pointer), is responsible for emulating the Java bytecode instructions. It can be a simple interpreter, or it can perform a full or partial translation (compilation) to native host instructions. Higher-performance implementations may use profiling and staged emulation techniques to first detect hotspots and then translate to native instructions held in a code cache, as discussed in earlier chapters. Even if a simple interpreter is used, some pretranslation of the program file is performed. For example, some of the indirection in the program file (such as the use of the constant pool) would likely be removed, with immediate values being embedded directly in instructions.

Native Method Interface

To access operating system–managed functions, the JVM uses a set of standard libraries. These libraries can perform file I/O and graph-

of standard libraries. These libraries can perform file I/O and graphics operations, for example. Many of the libraries are written in Java, and their associated binary classes are loaded and emulated in essentially the same way as the application-specific methods. However, in practice, at least some of the library code is written in the host's native code in order to bridge the gap between the platform-independent and platform-dependent parts of the overall system. Many calls to the host OS are made through native methods, for example. Providing these native methods (along with other standard library methods) is part of the overall JVM implementation process.

Class Loader Subsystem

The class loader subsystem performs a number of vital functions for supporting a networked computing environment. Of course, it converts the class file containing metadata and instructions into an implementation-dependent memory image. It is also responsible for finding the binary classes in the first place, often dynamically and on demand, either from the system or from other systems in a network. The loader subsystem is responsible for verifying correctness and consistency of binary classes and is an integral part of maintaining overall security within the network environment. We discuss class loading in more detail in the next section.

6.1 Dynamic Class Loading

When a method is called for the first time by a program, the class loader locates the requested binary class; typically it is held in a file. Then it checks the integrity of the binary class and performs any translation of code and metadata in order to make the requested method ready to run.

An important aspect of the network environment is that it must be possible to identify (name) variables, methods, and other data items

in a standard, universal way. All variables and methods must be declared as part of a class (which must have a name). Each compiled class is stored as a separate entity, e.g., in a separate file, and one or more classes can be combined into a logical entity known as a *package.* Thus every method or variable has what is called a *fully qualified name,* which consists of the package name, the class name, and the method or variable name, separated by periods, for example, testpackage.block.mass. For accessing packages across the Internet, there is a naming convention that is based on Internet domain names. For example, a package might be given the following fully qualified name:

edu.wisc.ece.jes.testpackage.shape.area

Here, the Internet domain name **ece.wisc.edu** (in reverse order, by convention) is prefixed to an internal path name, jes.testpackage, to provide a full package name. The access rights to packages, classes, and fields are defined in the following way.

- A package is accessible according to the access rights on the local system, for example, if the read or execute permission allows public access when it is being accessed by an external (nonlocal) user.
- Classes within a package are accessible to all other classes within the same package (this scoping property is one of the reasons for having packages in the first place).
- A class that is declared as public is accessible from other packages; a nonpublic class is only accessible from within its own package.
- All fields of a class are accessible from within its own class. Fields are accessible from a different class in the same package if they are not private.

The class loader system implements dynamic loading and is a critical part of the security system. A Java VM implementation must contain a *primordial class loader,* whose operation is defined as part of the JVM specification. In addition, users may also define class loaders as part of a class loader subsystem. The primordial class loader can be relied upon to do the right thing with respect to security; i.e., it is a trusted JVM component.

As just noted, there can be multiple user-defined class loaders, but each one defines a separate *namespace.* A loaded class is "tagged" with the name of the loader that loaded it, so if two classes loaded by different loaders happen to have the same name, they are in different namespaces and can be kept separate. Classes loaded into one namespace cannot interact with classes in other namespaces; indeed they cannot even become aware of the presence of the other namespaces. Hence, in effect, there is a barrier between the different namespaces. Different class loaders are typically used to load classes from different origins; hence, the namespace becomes identified with the origin.

Unlike the primordial class loader, which must be designed to be trustworthy, the user-supplied class loaders are only as trusted as the user who supplies them. This issue can be simplified somewhat because a user-supplied loader can be designed so that it relies on the primordial class loader for assistance in loading binary classes, for example, as a means for ensuring security.

After it locates a binary class, the loader parses it and translates it into internal data structures for emulation by the execution engine. This involves converting the metadata contained in the binary class into an implementation-dependent form that is more amenable to emulation. As part of this process the loader performs some consistency checks. It first checks the magic number at the beginning of the binary class to make sure the given data is at least claimed to be a binary class (this check will mostly turn up simple errors where the VM is being invoked for a "wrong" file). More importantly, the loader checks to make sure all the components are of the sizes indicated in the binary class and that proper formats are used in the various metadata structures. It can also check to make sure the numbers and types of arguments match between calling and called methods. Within the binary class, fully qualified references are resolved. This is often done on demand. The fully qualified references

are symbolic. After being resolved, the fully qualified (symbolic) reference is replaced with a direct reference.

Another important function of the loader is to verify the integrity of the bytecode program. That is, it checks to make sure the stack values can be tracked statically, as required of all well-formed programs. Then it performs all the static type checking to make sure there are no program errors and that the protection boundaries will be respected by the program. This process is described in more detail in the next subsection. The loader checks to make sure all references in the constant pool are within the bounds of the programs, and it checks to make sure all branch instructions are to addresses within the procedure where they reside. Finally, after the program is verified as being properly formed, memory is initialized and control passed to the emulation engine.

6.2 Implementing Security

The basic protection sandbox was described in general terms in **Section 5.2.1**. This is the default security model used in Java, although the model has been greatly enhanced in the Java 2 specification. In this section we first elaborate on the basic security model and then describe the Java 2 enhancements.

In the sandbox model, the overall objective is to form a barrier around the Java execution environment so that a user application is confined to operate within the bounds of the sandbox and cannot affect any other system or network resources, whether unintentionally or maliciously. As discussed in **Section 5.2.1**, there are three main parts to the sandbox.

First, remote files must be protected. As pointed out earlier, the protection of remote files is within the province of the remote system. Any files on the remote system are given access privileges by the

owner of the file, and, ultimately, these are maintained by the remote OS, whether Windows, Linux, or any other OS. These are the same privileges as for any type of remote access, whether inside or outside the Java execution framework.

Second, local files must be protected from a running Java program. That is, the owner of the local files, typically the user running the Java program, must control access to the files. This part of the protection sandbox is implemented via the security manager, described in more detail in **Section 6.2.2**.

Third, the JVM code and data must be protected from the running Java program. That is, the Java program should not be allowed to read or write data associated with the Java VM side tables, for example. Furthermore, the program should not be allowed to branch or jump to arbitrary locations in the JVM. This third part of the protection sandbox is implemented via a combination of static and dynamic checking, with static checking performed by the loader playing a key role in this process. The next subsection discusses this intraprocess protection part of the sandbox, and this is followed by the discussion of the security manager.

6.2.1 Intraprocess Protection

One of the more challenging aspects of building a modern network-oriented HLL VM is the ability to take untrusted program code from the network, combine it with trusted VM code to form a single host-supported user process, and then execute the untrusted program efficiently. A basic objective is to make sure the application code only accesses its own memory locations and can interact with the JVM (and the rest of the host platform) only via method calls to libraries containing locally trusted code.

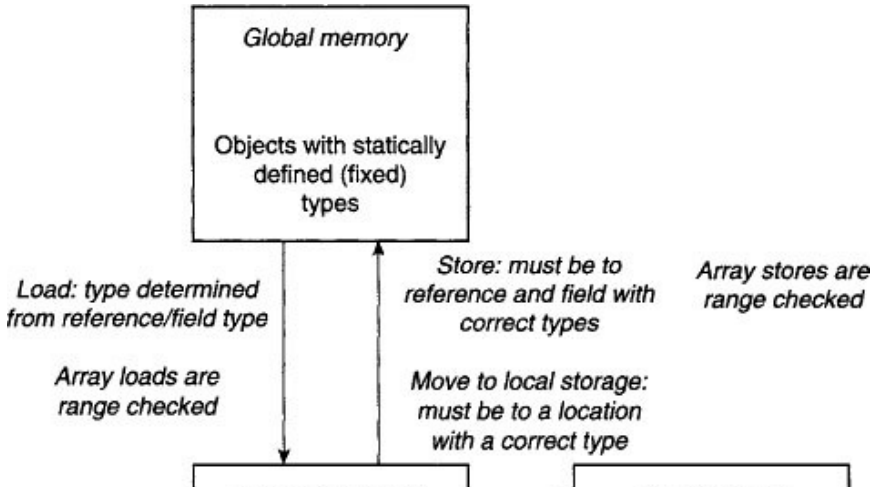
It is the ability to construct a protection sandbox that most clearly separates the modern HLL VMs from the earlier P-code-like VMs. To

separates the modern HLL VMs from the earlier P-code like VMs. To understand this, consider the typical P-code environment. Here, the compiler is a trusted element; i.e., it is assumed by the user to be correctly implemented. The Pascal language is designed so that the compiler can check data-type information at compile time, based on the data declarations made by the programmer. After this checking is complete, the P-code is generated. Then the static data-type information is no longer needed; rather, it is built into the code that accesses the data, and the user can be assured that the P-code will only access data in the correct way. The P-code alone can be passed to other users to run on their computer systems, thereby achieving platform independence. It is up to the individual users to trust the provider of the P-code program.

The earlier scenario would be fine if other users were always willing to trust the source of their programs, but of course this level of trust is often not present in the networked computing environment. Yet a user would like to be able to execute programs that are made widely available for network-based computing. A solution to the problem of untrusted code is to transmit not only the application code but also a checkable description of the data structures on which the code will operate, i.e., the metadata. Then when the program is loaded by the end user, the (trusted) loader will perform static checks of the program with the declared metadata information. If the program code is consistent with the metadata, then the user can be assured that the program will only access its data in a proper (protected) way. Furthermore, the loader can check the control flow information of the program to make sure it branches only to locations within the program itself or performs procedure calls using proper protocols. This is the reason that HLL ISAs stress static type checking and statically traceable control flow information. And, of course, the ability to perform this high degree of static checking derives from the strong typing features incorporated in the high-level language in which the program was written in the first place (e.g., Java or C#).

Now we will illustrate in more detail the features of the Java ISA that allows intraprocess protection. There are two main elements of intraprocess protection. First, it must be guaranteed that the program only accesses its own memory; i.e., it can never make data accesses outside the bounds of its declared memory region. Second, it must be guaranteed that the program will not branch outside its own code region. It can only leave the code region via a library call, and the libraries are trusted code.

We first give an informal inductive argument for the protection of data accesses. Just as data flow is centered on the stack (see **Figure 5.9**), our argument will revolve around the stack (**Figure 6.2**). Before any memory accesses have been performed, and with an empty operand stack, the only way to put a reference onto the operand stack is from the constant pool or from local memory. The constant-pool entries are statically checked by the loader, as noted earlier, so they can be guaranteed to be valid references (or are null). All local memory locations holding reference types are initialized to null references. Furthermore, all movement between the stack and local memory uses absolute addresses for the local memory locations. Hence, the loader can check the local memory types with the data-movement opcodes to make sure that only reference types are placed into local memory reference locations.



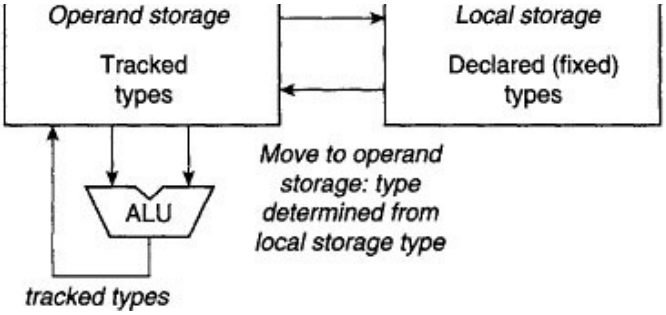


Figure 6.2 Data Flow and Intraprocess Protection Checking.

Once on the stack, the types of all elements can be statically tracked (as described near the end of **Section 5.3.3**). In particular, recall that for any valid program, the number and types of elements on the stack can always be determined from the static code itself. The loader verifies this is the case at the time the program is loaded. Among other things, this means that any data movement of references from the stack to local memory can be checked for the correct types statically.

Pulling all the foregoing together, starting from initial conditions, before any global memory accesses have been made, any reference values on the operand stack either are null or are verified reference values taken from the constant pool (either directly or indirectly through local memory). Furthermore, the types of these references are statically known.

Next, consider accesses to global memory. The first access to global memory is to a location specified by a reference on the operand stack. Because of the ability to track reference types on the stack and in local memory, this reference type is known statically. Hence, if an object is accessed, the field information for the access can also be checked statically (there is an exception for arrays, given in the next paragraph). If the access is a load, the data type being loaded is known statically; if it is a reference type, then the type is also known and can be tracked statically while it is on the stack or in local memory. Similarly, if a value (data or reference) is stored to an object field memory, it can also be checked prior to being stored because of

field memory, it can also be checked prior to being stored because of static tracking in local storage and static field definitions; hence the value stored to memory will be consistent with the static field information. By a similar argument, and to complete the induction step, if all data types can be determined from static information prior to a memory access *i;* then the next memory access, $i + 1$, must also yield statically trackable type information.

An important exception to the argument just given involves array accesses. All loads or stores to arrays must use an array reference; these can also be statically checked for type information. However, arrays differ from other objects in that the exact element being accessed may be computed at run time. This means that the actual index into an array must be checked dynamically to make sure it is within the bounds of the array. Furthermore, all references, whether for an array or for a general object, must be checked for null values, in the event that they have not been assigned to an object when they are first used.

Another important place where dynamic checking is required occurs with dynamic casting or conversion of an object reference to a different type. Allowing arbitrary casting of references can break that part of the protection sandbox enabled by static checking, because static checking depends on knowledge of object types. However, casting can be done in situations where the reference is being cast to a superclass (*upcasting*) or a subclass (*downcasting*). The safety of upcasting can be checked at compile time; however, downcasting requires a run-time check to make sure the reference being cast is to a subclass (or the same class) as the object type to which it currently refers.

To summarize, a program, even if it is from an untrusted source, can be checked to make sure it makes no unallowed memory accesses. This is done by a combination of static (load-time) and dynamic (run-time) checks. In Java, much of the checking is done stati-

cally. By relying on static type checking, dynamic checking is mostly confined to null pointer checking, bounds checking for array accesses, and dynamic casts to make sure they are being properly done within the class hierarchy. In the high-performance Java discussion to be presented later, we will include analysis methods and optimizations that are able to remove some of these dynamic checks.

Finally, consider the protection provided for control transfers, i.e., the constraint that there never be a branch or jump that leaves the program's valid code region. Control transfers are only through branches, switch statements, and method calls. All the branches and switch statements are to PC relative constant values. Therefore, all the branches and jumps can be statically checked to make sure they branch to instructions only within the given procedure. The only way to exit a method is through a method call/return (or when the program terminates). Therefore, if all methods are internally checked, the overall program is checked.

6.2.2 Security Enforcement

In a Java implementation, the security manager is a class belonging to the java.lang API that contains a number of methods for checking to make sure potentially unsafe operations cannot be performed. These operations include reading a specified file, writing a specified file, opening a socket connection to a specified host and port number, and creating a new process, among many others. Such operations are made available to untrusted user programs via methods in (trusted) Java libraries, e.g., the java.io API, that interact directly with the host OS. Before performing a requested action involving an OS call, the trusted library methods first check with the security manager, via the appropriate method call, to make sure it should be allowed.

A security manager check method simply checks the requested op-

eration to see if it is permitted and either throws a security exception if it is not permitted or returns if it is. The security manager is attached when a Java application is initiated and cannot be changed, deleted, or replaced with another thereafter. Hence, an individual user can specify the checks made via the security manager, i.e., which files should be accessed and how, which network ports can be used, etc. Then whenever a Java application is started, these checks will be in effect as long as the application runs.

If the user chooses to use no security manager, then the application has unrestricted access to the user's resources. Of course, the Java application cannot exceed the privileges of the user, because the JVM runs as a user process. Any attempt to go outside the user's resources will be caught by the underlying OS on the host platform.

The security manager cannot protect against everything, overallocating memory or spawning too many threads, for example. In situations like this, it is difficult or impossible to discern the difference between an erroneous or malicious program and one that has large resource requirements. For example, how could one determine the difference between runaway recursion and recursion that is simply very deep? In effect, detecting the difference would be equivalent to solving the classic Turing Machine halting problem. Consequently, some denial of-service attacks are possible, despite the presence of a security manager.

The security manager can be written to implement relatively complex policies. The loaders and security manager may be customized by a software developer or user. For example, the security manager can check to see if a request is made from a locally provided method or from one loaded over the network and can allow different resource accesses depending on which is the case.

6.2.3 Enhanced Security Model

The basic security sandbox was extended by adding signing in JDK 1.1 and then enhanced with access control in Java 2 (**McGraw and Felten 1999**). Adding the notion of identity and signing gets away from the all-or-nothing sandbox approach and allows more flexible, finer-grained security policies, where code from different sources may be granted different access privileges to user resources.

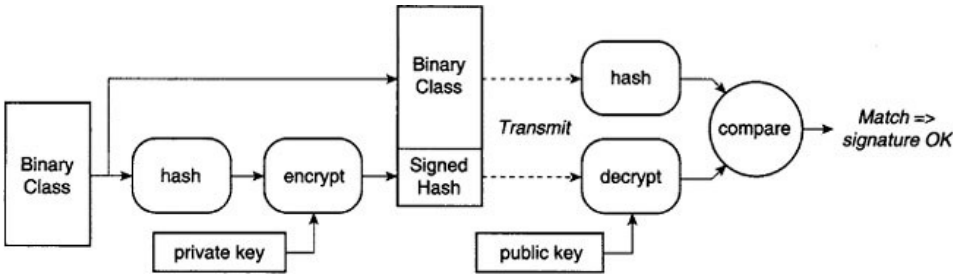


Figure 6.3 Using a Public Key Encryption System to Sign a Binary Class File.

If particular outside program sources can be securely identified, then a wide variety of security policies can be implemented, each one depending on the specific source of a given application program. A technology that supports secure identification of sources is *signing*. The basic idea of signing is simple: if a piece of code brought in from the outside can be signed in a nonforgeable way, then the local security system can inspect the signature and provide access privileges appropriate for the given (signed) piece of code.

Signing is based on public key encryption systems (**Diffie and Hellman 1976**). In a public key encryption system there is public key/private key pair. A message is encrypted with the public key and is decrypted with the private key. The critical feature of the system is that the private key cannot be determined from the public key in any practical way. Consequently, only the owner of the private key can decrypt a message.

The application of public key encryption technology to signing application code (and other types of data) is illustrated in **Figure 6.3**. The source of application code (a binary class) first hashes the code

down to a smaller size (for reasons of overall efficiency) and then encrypts the hashed version with the private key. The hashed, encrypted version is a signature appended to the binary class and is sent across a network to someone who wishes to execute it. At the receiving end, the received binary class is again hashed (with the same hash function as before), and the signature is decrypted. The two versions of the hashed code are then compared. If they are the same, then the user is assured that the owner (sender) of the binary class is as claimed. The security manager can then grant access to local resources in accordance with a policy established by the local user.

Because the identity of a program's source can be securely identified, the application of the all-or-nothing sandbox for all program sources is no longer necessary. Fine-grained, configurable security policies can be implemented. For example, programs from one outside source may be allowed to open network connections, while others cannot. One program may be given access only to file A, while another may be given access only to file B.

This flexible approach does pose some potential problems when binary classes from different sources are allowed to interoperate as part of the same program. For example, what if a method belonging to a binary class from a source not having access privileges to a given file calls a method from a source that does have access privileges to the file and then performs a file access indirectly through the called method? The security manager can solve this problem by inspecting the method call stack to check on access privileges not only of a method making a request but of all earlier methods in the call sequence (**Wallach and Felten 1998**).

Stack inspection is illustrated in **Figure 6.4**. Each stack frame is appended with *principal* information indicating whether the method came from a trusted system source or an untrusted source. Also, there is information regarding the access privileges that reside with

the source of the method. After a sequence of method calls, the resulting stack frames are shown in the figure. Untrusted method 2 had permissions to write file A but is restricted from accessing other files. On the other hand, method 4 does have permission to write to file B. Now, when method 4 attempts a write to file B, it will do so via the I/O API method 5 that first calls a security manager check method. The security manager walks the stack in reverse order, checking access permissions of all the methods that it finds on the stack. When it comes to method 2, it finds that permission to write file B should not be granted and throws an exception.

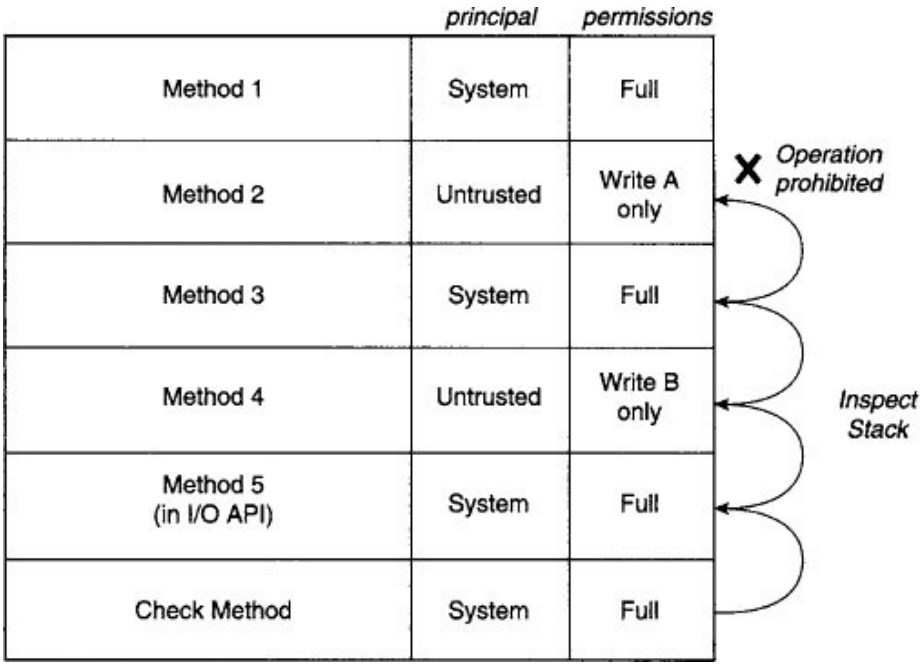


Figure 6.4 Method Call Stack Extended with Resource Access Permissions. *The security manager inspects the stack to enforce protection.*

6.3 Garbage Collection

In an object-oriented programming environment, objects can be freely created, used, and later discarded when they are no longer needed. The programmer is given the illusion of an unbounded memory space and does not have to worry about explicitly allocating and managing a fixed-size memory. Of course, in reality, mem-

ory resources are not unbounded, and eventually a program can run out of memory resources (which would result in an OutOfMemory exception being thrown). To prevent this from happening (or at least forestall it), objects that are no longer accessible (i.e., "garbage") can be collected and then reused for new objects.

In our simple example in **Figure 5.5**, when the reference a is assigned to a new Rectangle object, the prior Rectangle object it pointed to becomes inaccessible. This is the typical way that an object becomes garbage—the last (or only) reference to it is overwritten with a reference to a different object of the same type. In other cases, the last reference may simply be discarded, e.g., by popping it from the stack.

A garbage collector is part of essentially every JVM implementation, although it is not strictly required as part of the JVM specification. When the JVM is running low on memory resources or at periodic intervals, the garbage collector is invoked to find the inaccessible garbage objects and collect their memory resources for later use. A typical situation is illustrated in **Figure 6.5**. Here, a *root set* of references point to objects held in the global memory heap. Some of these objects, in turn, contain references that point to other objects, and so on. Also, there are some objects that cannot be reached through a sequence of references beginning with the root set; these are the garbage objects.

To begin a garbage collection operation, it is first necessary to identify the root pointers. By referring back to **Figure 5.7** (and considering the Java instruction set) we can see that the root set must contain references somewhere on the stack, including both local storage and the operand stack, or in the constant pool. The root set must also include references contained in static objects. Any access to global memory, i.e., through a getfield or putfield instruction, must get its reference from one of these places, and any object that cannot be accessed via a series of references beginning with some

not be accessed via a series of references beginning with some member of the root set simply cannot be reached. Hence, garbage collection finds all the objects that can be accessed, beginning with the root set of references; any objects not found during this process are garbage that can be collected. There are a number of approaches for performing these operations (**Jones 1996**; **Wilson 1992**; **Appel 1991**), and in the following subsections we summarize some of the major classes of garbage-collection techniques.

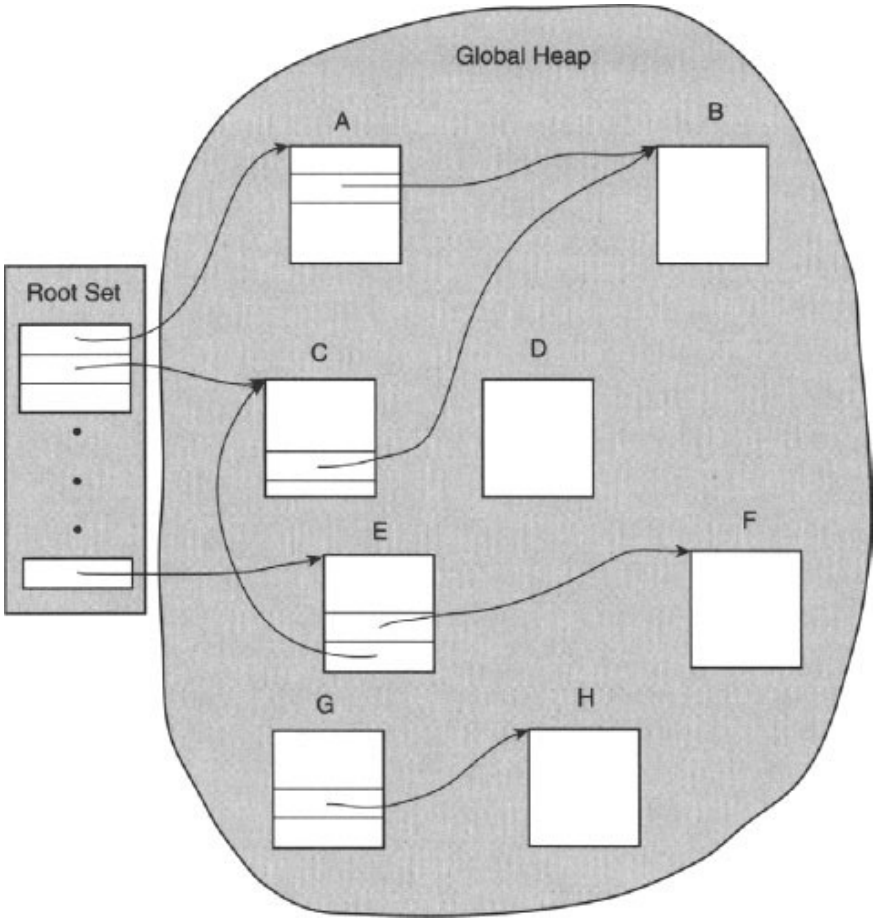


Figure 6.5 A Garbage-Collected Heap. *The objects G, D, and H are garbage.*

During the discussion, a number of tradeoffs are considered. Time spent collecting garbage is time not spent computing, so the overhead of garbage collection is an important overall implementation consideration. Moreover, different methods organize the heap differently, collect the garbage into free space differently, and may implement object references differently. Consequently, the important

tradeoffs to be considered involve garbage-collection time, object-allocation time, object-access time, and the efficiency with which the heap space is used.

Although there have been garbage collectors that keep a count of the number of references to each object on the heap and consider as garbage any object whose reference count goes to zero (**Collins 1960**), such reference counting collectors are relatively uncommon in JVMs. Most JVMs use the general method suggested earlier, that is, to start with a set of root references and then *trace* through chains of references in the heap to find all the objects that are reachable or are "live." Then all the unreachable objects are considered garbage and are recycled. The following discussion of garbage collectors will focus on tracing collectors.

6.3.1 Mark-and-Sweep Collectors

The mark-and-sweep collector is a basic collector that starts with the root references and traces through all the reachable objects, *mark*ing each one as it is reached. Marking may consist of setting a flag bit contained as part of the object implementation or in a separate bitmap, where each entry in the map is associated with an object. If an already-marked object is reached, then tracing down that particular path stops. After all live objects have been found and marked, there is a *sweep* stage, where all the objects are inspected and the unmarked ones are determined to be garbage and can be reused. As garbage objects are found during the sweep stage, they can be combined into a linked list of free objects.

Overall, this is a relatively fast way of identifying and collecting garbage. However, the free objects are of varying size and are scattered around the heap space, interspersed with the live objects. Simply linking together free objects leads to a memory fragmentation problem that in turn leads to a significant inefficiency when a new object is created and a suitably sized free space must be found

new object is created and a suitably sized free space must be found for it. Specifically, the allocation algorithm must search the linked list to find an appropriately sized block of contiguous free memory). Eventually fragmentation becomes so great that compaction is required.

Inefficiencies can be reduced, however, by using segregated free lists. That is, by dividing the heap into a set of fixed-size chunks having a range of sizes, e.g., from 16 bytes to 2KB, and maintaining the free space as multiple linked lists, one for each chunk size (**Comfort 1964**). Then the object allocator can simply go to a free list and get a chunk of the appropriate size (i.e., the smallest one that is large enough to hold the object being allocated). Provisions must be made for occasionally rebalancing the space in each size category, but overall allocation efficiency is significantly improved. In general, dynamic memory allocation is very closely associated with garbage collection. An excellent survey of memory allocation techniques, including segregated free lists, is **Wilson et al. (1995)**.

A conceptually straightforward approach to improving allocation efficiency is to consolidate all the garbage space into a large contiguous region from which new objects can be created. There are two ways of consolidating the garbage space: by compacting and by copying. These are described in the next two subsections.

6.3.2 Compacting Collectors

A compacting collector, as its name suggests, essentially “slides” the live objects to the bottom (or top) of the heap memory area so that all live objects are adjacent. What is left is a contiguous region of free space. For example, **Figure 6.6** shows the contents of memory with live objects (shaded in dark gray) and garbage objects (shaded in light gray). During compaction, the live objects are moved to a contiguous region of the heap (at the bottom in this example), and the unused space also becomes a contiguous region, from which

new objects can be allocated.

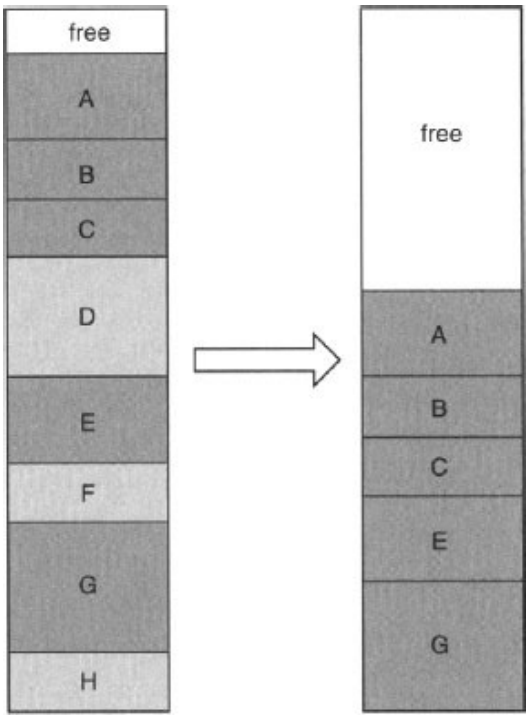


Figure 6.6 Example of Garbage Compaction.

Although conceptually simple, a compacting collector is relatively slow, in that it makes multiple passes through the heap. One pass does the marking, and then subsequent passes compute the new locations for the live objects, move the objects, and update all references to point to the new locations. Other methods, to be summarized shortly, improve efficiency by reducing the number of passes and/or analyzing only a subset of the heap during each collection step.

Compacting collection also highlights an issue that occurs with any of the schemes that move objects in memory as a part of garbage collection. That is, the references to the objects must be changed when objects are moved, and this complicates (and slows) the overall process. However, to reduce the number of reference updates, some systems have used a *handle pool* for consolidating pointers to each of the individual objects. Then a reference to an object points to the associated pointer in the handle pool. This is illustrated in

Figure 6.7. With the handle pool, it is not necessary to find and update all the references to an object when the object is moved; rather, only the pointer in the handle pool has to be changed, and all references are then automatically modified in the process. The big disadvantage of this approach is that every object access includes an additional level of indirection.

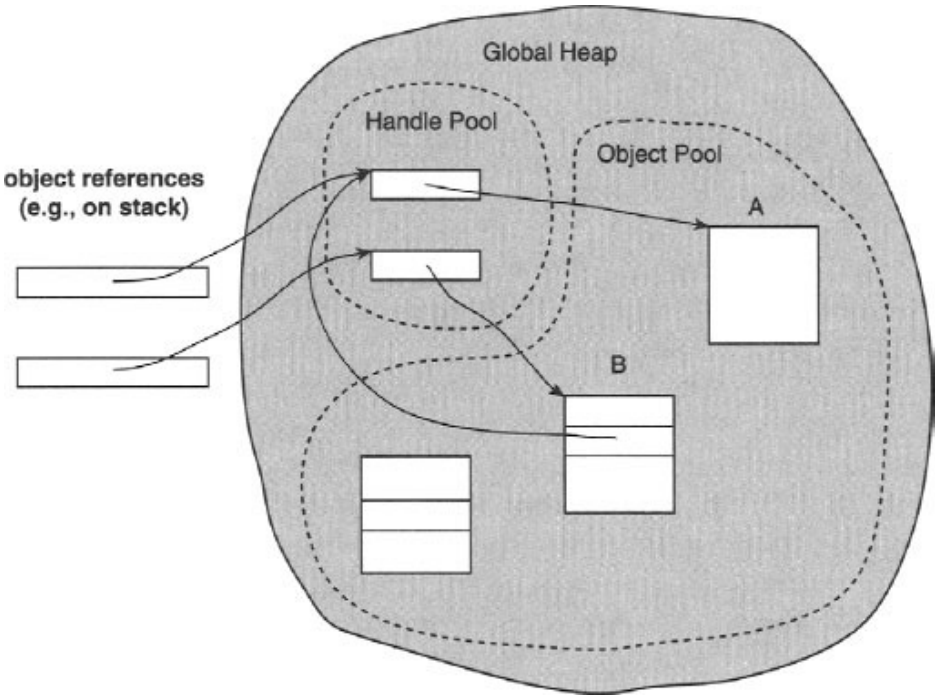


Figure 6.7 Handle Pool. *Using a level of indirection (through a handle) can simplify pointer updates when an object is moved during garbage collection.*

6.3.3 Copying Collectors

To reduce the number of passes through the heap during collection, a copying collector trades memory space for collection time. It does this by dividing the heap into two halves. At any point in time, one half is unused while the other contains the active heap. When the active half fills up, the collector makes a pass through the heap just as it does during a mark phase; however, it combines the sweep with the mark phase. When it finds a live object, it immediately moves it into the unused half of the heap and continues the pass through the heap. When the pass through the heap is complete, the

live objects are in contiguous locations in what was formerly the unused half of the heap, the remainder of the second half is free, and the first half of the heap (which was formerly active) becomes unused. A copying collector is illustrated in **Figure 6.8**. A copying collector is faster than a compacting collector, but it also has higher memory requirements because half of the heap space, by definition, is unused at any given time.

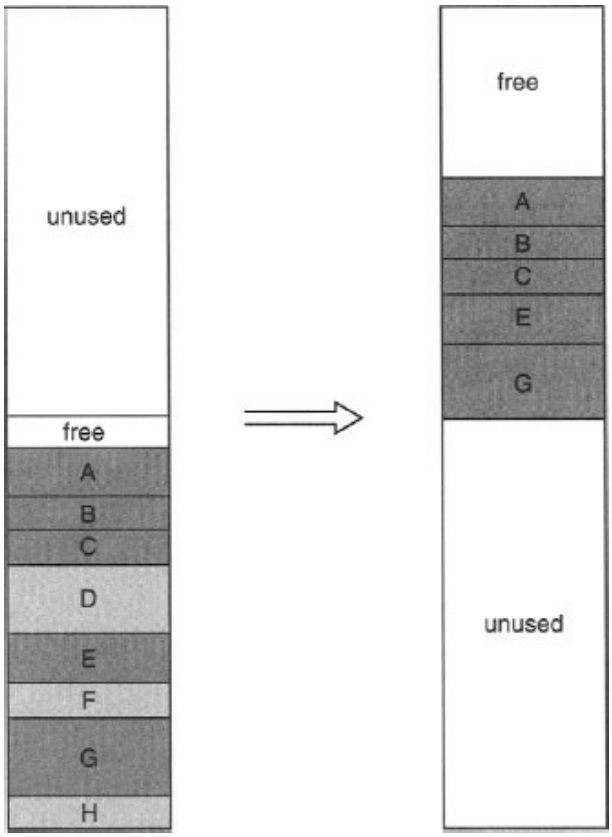


Figure 6.8 Example of Garbage Collection via Copying.

6.3.4 Generational Collectors

Both compacting and copying collectors move a very large fraction of dynamic objects every time collection is performed. A long-lived object may be moved many times during its lifetime; this rather wasteful movement can be avoided by observing that object lifetimes have a bimodal distribution. First, many objects have very short lifetimes; this is often a by-product of good object-oriented

programming practice. Second, objects that do not have short lifetimes tend to have very long lifetimes. Consequently, generational garbage collectors attempt to group objects according to their age in order to avoid repeated copying of the long-lived objects.

In a basic generational collector, the heap is divided into subheaps. For simplicity, we describe the implementation with two subheaps, but generalization to more than two subheaps is straightforward. Of the two subheaps, one is intended to hold the older, or *tenured,* objects. Meanwhile, the other subheap serves as a *nursery* for newly created objects. The nursery is garbage-collected much more frequently than the tenured subheap. If an object survives a certain (usually small) number of collections in the nursery, it is moved to the tenured subheap. Consequently, the longer-lived objects eventually are placed in the tenured section of the heap, where the garbage collections are infrequent, thus avoiding unnecessary object movement.

Not only is the total overhead for collecting reduced, but a smaller portion of the heap is collected each time the collector is invoked. This means that if the running process is stopped while collection takes place, the “pause” time is much reduced. With a large heap and a conventional compacting or copying collector, the time that a program is stopped for collection can be noticeable to the user.

The two subheaps of a generational collector do not necessarily have to be managed the same way. In fact, they can be managed differently to take advantage of performance tradeoffs. A specific hybrid algorithm studied as part of the Jikes JVM project (**Attanasio et al. 2001**) uses a copying collector for the nursery, which yields fast allocation of new objects, and a mark-and-sweep collector for the tenured objects, which reduces collection time by eliminating pointer updates.

6.3.5 Incremental and Concurrent Collectors

6.3.3 Incremental and Concurrent Collectors

All of the basic collectors described earlier stop program execution while they perform collection and then return control to the program. Collection can be time consuming (even if generational collection is used), so program execution may pause for a significant time period while the collector is working. Collection time may be spread out if collection is done incrementally as the program runs rather than all at once. Furthermore, in real-time applications, the garbage-collection time may be limited to provide adequate response latencies. If multiple processors are available, it may be advantageous to collect garbage concurrently with one thread while normal program execution proceeds using other threads. In both cases, a partially collected heap may be in a state of flux while the program is running. This implies there must be some synchronization between the running program and the garbage collector, so the program does not attempt to reference an object at the time it is being moved; i.e., at a time when pointers may be temporarily inconsistent. Or the program may be changing references at the time the collector is tracing a reference path, leading to similar synchronization problems.

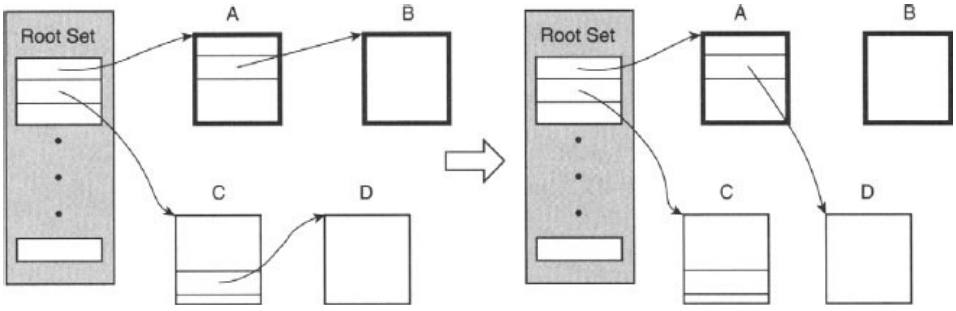


Figure 6.9 A Problem with Concurrent Garbage Collection. *A reference in an object may be modified after the object has been marked.*

Many of the conventional stop-and-collect methods can be converted to incremental versions. A basic problem with concurrent or incremental collections is that at any given time, an object may have been scanned and marked. Then while collection is still in progress, a reference in the already-scanned object may be changed to point

a reference in the already-scanned object may be changed to point to an object that has not yet been marked. An example is in **Figure 6.9**. In this example, the objects A and B have been marked (as indicated by the bold boxes). The other objects have not yet been marked. Then before they can be marked, the pointer to object B is replaced with a pointer to object D. If this is the only pointer to D, then D may never be marked and will be incorrectly discarded as garbage. The object B may also be incorrectly retained, but this is not a problem; it will be collected on the next garbage-collection pass.

There are a number of solutions to this problem; they all provide some form of synchronization between the running application (which may change the contents of the heap) and the collector. One of the common solutions is to provide *write barriers* for references to objects that have already been marked. These write barriers basically check for the case where a pointer in an already-marked object is overwritten. When this occurs, the object pointed to is marked (and put on the queue of objects whose pointers should be followed as marking proceeds).

6.3.6 Discovering the Root Set

As stated at the beginning of this section, before beginning garbage collection it is necessary to identify the root pointers. The root set for Java programs (and similarly for CLI) consists of references held in the stack, the constant pool, and contained in static objects. However, this is what one might call the “architected” root set. In a particular VM implementation, the actual pointers may reside in registers and various implementation-specific runtime memory locations. For example, the architected stack elements may be assigned to registers by the dynamic compiler, and some of these could be spilled to memory.

Therefore, at the time garbage collection is invoked, it is necessary

to construct the architected root set from the implementation storage locations that may potentially hold root set pointers. The basic solution for finding the root set is for the compilation/optimization system to keep side tables, or maps, that indicate where the architected root set elements may be found, if needed for garbage collection. This solution requires that the VM runtime and compilers keep track of the architected root values as dynamic compilation and optimization are performed (**Stichnoth, Lueh, and Cuerniak 1999**).

The foregoing approach is referred to as *type accurate* or *exact* because it uses the exact root set as a starting point. A less conservative solution is to take all the implementation registers and memory locations that can *possibly* hold root pointers and assume they *do* hold root pointers (**Boehm and Weiser 1988**). These will form a superset of the true root set. Some of these may be eliminated from consideration (e.g., if they hold small integer values that are clearly not memory addresses). Then the garbage collector begins with this root superset. This approach frees the compilation/optimization system from keeping track of root set references and saves runtime side table space for keeping track of the mappings at all potential garbage-collection points. A problem with this superset approach, however, is that a garbage collector that moves objects, e.g., a compacting collector, cannot be used, because a region of memory may be identified as an object of a given type when it is not really an object. Then if such an apparent "object" is moved, it will damage any real objects that contain the apparent object as part of their memory region.

6.3.7 Garbage-Collection Summary

No one collector works best across all programs because programs vary in working set size, object sizes, heap sizes, and the rate at which objects are created (and freed). Hence, we cannot arrive at hard-and-fast conclusions regarding the "best" collector (as the hundreds of papers on the topic attest). We can summarize the impor-

tant performance tradeoffs in general terms, however. **Table 6.1** qualitatively compares the major collector categories in terms of (1) collection time, (2) object allocation time, (3) object access time, and (4) memory efficiency (i.e., the relative total heap space that can maintain a working set of a given size). In addition to these criteria, all the "moving" collectors, i.e., the compacting and copying collectors, require an exact root set, as noted earlier.

The use of a generational collector is a somewhat orthogonal consideration because a generational collector can be used with any of the basic approaches (or as a hybrid). The advantage of a generational collector is that it reduces collection time by focusing only on those objects where garbage is most likely to be found (the nursery). Today, generational collection is usually considered to be a winning strategy. A study of garbage collection in the Jikes JVM (**Attanasio et al. 2001**) describes the performance tradeoffs with respect to specific benchmark programs.

TABLE 6.1 Comparison of Basic Garbage Collectors

Collector	Collection Time	Object Allocation Time	Access Time	Memory Efficiency	Comments
Mark and sweep	Good	Poor	Good	Medium	Allocation requires search for proper-size free memory block
Mark and sweep (multisizes)	Good	Medium	Good	Medium	Multisizes help allocation time more than it benefits memory efficiency
Compacting	Poor	Good	Good	Good	Collection requires multiple passes
Copying (with handles)	Medium	Good	Poor	Poor	Memory efficiency poor due to unused space
Copying (without handles)	Medium to poor	Good	Good	Poor	Pointers must be found and updated for all moved objects

6.4 Java Native Interface

The Java native interface (JNI), as described in Section 5.3.6, is a way of allowing Java programs and native compiled programs to interoperate. An important consideration when implementing the JNI is

the accessing of Java data structures in a safe way, i.e., in a way that maintains the integrity of Java objects and the stack.

From the perspective of a Java program, a call to a native method is in some respects similar to executing a single (very complex) instruction. There are arguments and a return value, but the Java stack is not used by the called method; rather, the native methods use their own platform-dependent stack. The management of this native stack must be done by trusted JVM code. The JVM implementation supports both a Java stack and a native stack, with one native stack per Java thread (see **Figure 6.10**). When there is a call to a native method, it is intercepted by the JVM, which then sets up the native stack frame, transfers arguments, and then transfers control to the called method. Similarly, on a return, it places a result onto the Java stack frame and returns control to emulation. In the JNI, a native method may "call back" a Java method; when this happens, a Java stack frame is placed on the stack frame of the calling native method.

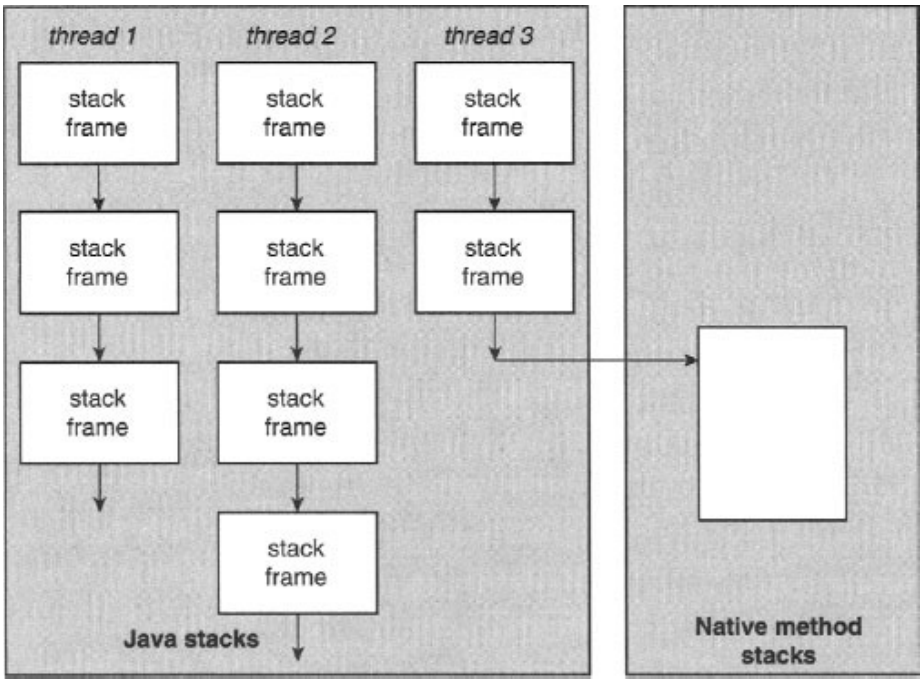


Figure 6.10 Java and Native Stacks. *In a VM implementation, the native library stack is managed separately from the architected VM stack.*

A native method can be given access to objects and arrays via the JNI. However, once the native method has an array or object reference, the array or object must be protected from garbage collection. That is, the native method assumes the object to be at a particular memory location, and an object-moving garbage collector would violate this assumption. Even with a nonmoving collector, if the native method should happen to have the only reference at some point, then the garbage collector may attempt to collect it (because the reference will not be in the garbage collector's normal root set). To prevent this from happening, the JNI method that passes the native method a reference also "pins" the referenced object so that it will not be moved or collected. With a non-moving collector, the JVM could also add the reference to the root set.

6.5 Basic Emulation

The emulation engine in a JVM can be implemented in a number of ways, with different complexities and performance levels. The simplest method uses straightforward interpretation of bytecode instructions. In general, this is no different than interpretation of a conventional ISA as described in detail in **Chapter 2**. A more advanced, and commonly used, emulation method performs just-in-time (JIT) compilation (**Tabatabai et al. 1998**; **Aycock 2003**). With a JIT compiler, methods are compiled at the time they are first invoked, i.e., "just in time" for execution. This compilation essentially performs a translation from the bytecode instructions to native host instructions. Method-at-a-time JIT compilation is enabled because, in contrast to a conventional ISA, the Java ISA is designed so that all the instructions in a method can easily be discovered at the time the method is first entered.

A JIT compiler bears a very close relationship to the binary translators discussed earlier in this book and could just as easily be called a

JIT “translator.” In fact, it seems that the reason one is called a “compiler” and the other is a “translator” is that two different groups of people came up with the names. A JIT compiler differs from a conventional compiler in that it doesn’t have a frontend that parses a high-level language program and performs syntax checking before converting it to an intermediate form. The bytecode program, having passed through the loader, is assumed to be syntactically correct. The bytecode instructions are essentially an intermediate representation themselves, although most JIT compilers will transform the bytecode instructions into a different intermediate form before performing optimizations.

A dynamic, run-time compiler can perform most (if not all) of the optimizations performed by a classical static compiler as well as others that are more specific to Java programs. Many optimizations are time consuming, however, and this adds to the run-time overhead of executing a Java program. Consequently, a JIT compiler may include multiple optimization levels, with more sophisticated optimizations being applied to methods that are more frequently executed. This leads to a form of staged optimization applied at the method level. An even more efficient strategy is to apply optimizations selectively only to code regions that are heavily used rather than to the entire method that may contain such a region.

A typical high-performance emulation engine begins with interpretation, supplemented with profiling to locate frequently used methods. Then when a usage threshold is reached for a given method, the method is compiled with minimal optimization. Later, depending on the level of use, selected code sections within hot methods may be further optimized. The Sun HotSpot (**Meloan 1999**; **Paleczny, Vick, and Click 2001**) and the IBM DK (**Suganuma et al. 2000**) follow this overall strategy. Some systems skip the initial interpretation step in favor of a simple compilation; this approach is taken in the Jikes RVM (to be described in more detail later), for example. A more complete discussion of dynamic optimization in HLL VMs is given in

the next section.

6.6 High-Performance Emulation

As with most other virtual machine applications, performance is an important consideration in HLL VMs. There are two challenges when dealing with HLL VMs. The first is the same as in other dynamic optimizing VMs: to offset the run-time optimization overhead with the program execution-time improvement. The second challenge is to make an object-oriented program go fast. Object-oriented programs typically include frequent use of addressing indirection for both data and code, as well as frequent use of small methods (which suffer the relatively high overhead of method invocation). In this section we will discuss optimization techniques for HLL VMs. These will be discussed in the context of Java, using Java examples, but similar techniques are equally applicable to the CLR, running C# programs, for example.

We begin with a brief discussion of the overall framework of a high-performance HLL emulation engine. Then we describe optimizations that can be performed by a run-time compiler and the virtual machine run-time system. A comprehensive survey of adaptive optimization, including an extensive bibliography, can be found in **Arnold et al. (2005)**.

6.6.1 Optimization Framework

The general flow of a dynamic optimization framework is given in **Figure 6.11**. This flow should look familiar—it is similar to the one used for dynamic binary optimization in process virtual machines discussed in **Chapter 4**. Many of the same principles hold, so we will not discuss them in detail here. The framework supports a progression from simple interpretation to successively higher levels of compilation, depending on the frequency at which sections of code (i.e.,

methods) are executed. In some schemes, the interpretation stage is skipped, and the number of optimization levels also varies depending on the particular implementation.

Profiling is an important part of the overall optimization strategy. Profile data can be collected by the interpreter, or it can be provided by compiled code as it executes. A spectrum of instrumentation and sampling techniques can be used, as described in **Section 4.3**. Often, profiling is done at the method level rather than the basic block level as in **Chapter 4**. One can construct a call graph, similar to a control flow graph, which has methods as nodes and arcs connecting caller and callee nodes. An example is given later as part of **Figure 6.12**.

At a minimum, the profile data should track method usage, e.g., node counts in the call graph. When a method usage threshold is reached (or some more complex cost-benefit analysis indicates that it is called for), the next level of optimization is applied to the "hot" method. Other profile information can track edge counts in the call graph to guide method inlining, or conventional edge profiling can be used for code layout optimizations. Finally, dynamic data and pointer information may be profiled in order to generate specially optimized code that takes advantage of the values and/or data types that occur very frequently within a code region.

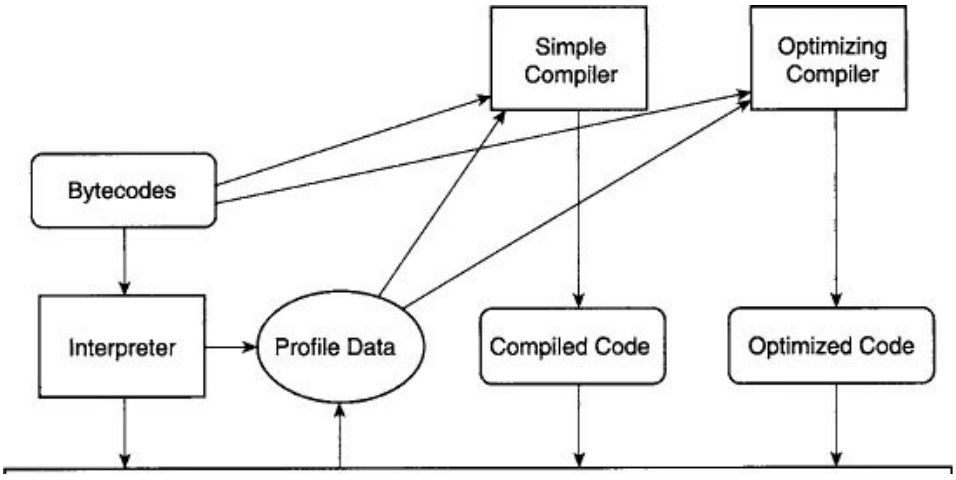




Figure 6.11 A Typical Dynamic Optimization Framework.

The compilers can provide a range of optimizations, some of which are applied at a relatively low level, e.g., involving redundant-subexpression elimination or strength reduction. Other optimizations restructure and specialize code. Some optimizations are rather specific to the object-oriented paradigm, and others are more general. Finally, some optimizations are performed directly via the compiler acting on the bytecode program as input. Other optimizations are performed dynamically by the runtime system, apart from the compiler. These latter optimizations may support garbage collection or enhance data locality by reorganizing heap objects, for example. In the next section we discuss a number of the more important optimization techniques, including both compiler-based optimizations and runtime-based optimizations.

6.6.2 Optimizations

Code Relayout

Just as with dynamic binary optimization discussed in **Chapter 4**, code relayout is a simple and very effective optimization when applied in the context of HLL VMs. **Figure 4.19** illustrates code relayout. Most code relayout algorithms “straighten” the code so that basic blocks along the most commonly followed control flow paths are in contiguous locations in memory (**Pettis and Hansen 1990**). The benefits are more efficient instruction fetching, due to both improved temporal and spatial locality, and improved conditional branch predictability. Code relayout often provides one of the larger performance benefits among all the optimizations.

Method Inlining

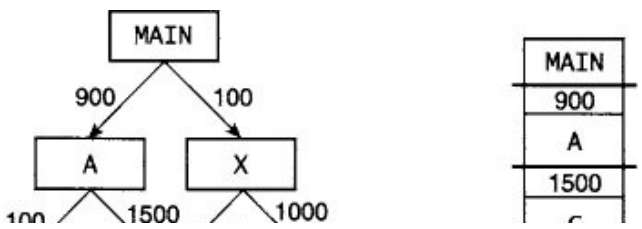
Method inlining is referred to as *procedure inlining* in **Section 4.3.1**. With inlining, a method call is replaced with the actual code contained in the method; i.e., the method code is placed "inline" with the calling code (**Suganuma, Yasue, and Nakatani 2002**). The overheads of passing parameters, managing a stack frame, and the actual control transfers (e.g., a jump and return) are saved, at the possible expense of a larger binary program image. Object-oriented programming tends to encourage many small methods, so performance can often be improved significantly by avoiding all the overhead code, i.e., the *calling sequence* that is associated with a method call. Another significant benefit of inlining is that it increases the scope over which later code analysis and optimizations can take place; e.g., analysis can make less conservative assumptions regarding potential data accesses outside the method being optimized. Inlining very small methods, where the code size of the method is less than the code size of the method's calling sequence, is almost always a win. When inlined, such a small method will not only execute faster, but it will also consume less instruction space than the original (noninlined) code, resulting in improved instruction cache behavior (or at least no degradation).

With larger methods, the benefits of inlining are reduced because the calling sequence takes a smaller percentage of the overall execution time and because there is an increase in total code size (and instruction cache requirements). If not applied selectively, the inlining of larger methods, especially those called from a number of different sites, can lead to *code explosion,* resulting in poor cache behavior and performance losses. Therefore, to apply inlining on medium-to-large methods, some kind of cost-benefit analysis is required, and the cost-benefit relationship can be rather complex. The cost in terms of code expansion can be estimated relatively easily, but how this translates to performance cost is more complex and would typically be a programmed-in function derived from off-line experimentation. The benefit is primarily a function of the size of the method and the frequency with which a method is called; that is, the most

frequently called methods will lead to the greatest benefit and are the primary candidates for inlining.

To determine the frequency of method calls, a common technique is to instrument the method call sites with counters to collect profiles of caller-callee pairs. At certain intervals (perhaps fixed time intervals or when some profile counter exceeds a threshold), the optimizing system can construct a *call graph*, with edges annotated with call (method invocation) frequencies. An example call graph is given in **Figure 6.12**a. In the figure, the main method calls methods A and X; method A calls methods B and C, and method X calls methods C and Y. The edges of the call graph are annotated with the number of calls during the sampling interval. For example, in **Figure 6.12**, method C has been called 1500 times by method A and 25 times by method X. After constructing and maintaining such an annotated graph, the dynamic optimization system can analyze the graph to determine which methods should be inlined.

Then, for example, one might define a "start" threshold for triggering inlining analysis as 1500; that is, when an edge in the call graph is traversed for the 1500th time, the dynamic optimization system is invoked. During the analysis, other methods besides the one that triggers the analysis may be inlined, depending on the cost-benefit analysis; e.g., a small method is more likely to be inlined than a larger one, even if they both have the same invocation counts in the call graph. In this example, it may be decided that method C should be inlined in A and that A should then be inlined in main, so main, A, and C are all merged together. Also, it may be decided that method Y should be inlined in X.



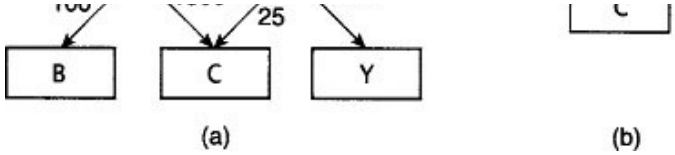


Figure 6.12 Tracking Method Calls. *(a) With a call graph; (b) via stack frames.*

This overall approach to inlining is a good one, but it requires construction of the call graph and typically involves some kind of overall analysis of the call graph. A simpler technique that avoids constructing the call graph and the overall analysis is to wait until a profiling counter reaches the start threshold and then to "walk" backwards through the stack to view what is effectively the currently active portion of the call graph (**Hölzle and Ungar 1996**). In our example, at the time the start threshold of 1500 is reached, the stack is as shown in **Figure 6.12**b. The profile counts are included in their associated stack frames, although they can also be held in a table off to the side. In any event, a walk up the stack may decide to inline B and A, as before. Although this technique uses a very narrow view of the call graph, it is nevertheless effective. In the example, method Y is not inlined as before, but if the X-to-Y call count later reaches the start threshold, it may be inlined at that time.

Optimizing Virtual Method Calls

In general, inlining can easily be applied to static methods and methods declared by the programmer to be final methods. When one of these methods is invoked from a given call site (e.g., via the invokestatic instruction), the method code called never changes. Hence, once inlining is performed on one of these methods, the inlined code will always be the correct code.

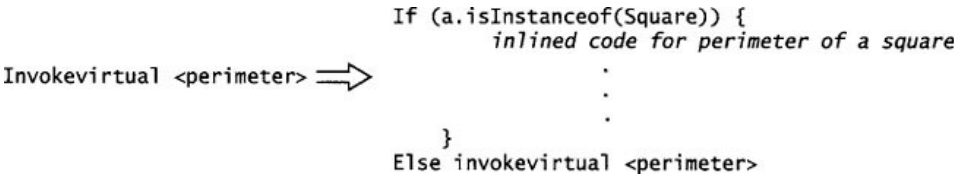


Figure 6.13 An Inlined Method Call Protected by a Guard.

In an object-oriented language, however, many methods are not static or final; i.e., the methods are associated with dynamic classes. Because of class hierarchies and resulting polymorphism, the actual code to be executed by a virtual method invocation can change, depending on the particular subclass of the object being referenced. Returning to the example given in the previous chapter (**Figure 5.5**), if the perimeter method is applied to a Rectangle object, the Rectangle version of the perimeter method code is invoked; in the case of a Square, a different perimeter method is invoked. The determination of which code to use is determined at run time via a dynamic method table lookup. Because a virtual method's code can change dynamically, depending on the type of object it is given, method inlining would appear to be inhibited for any method called via the invokevirtual instruction. However, in many cases a virtual method is invoked with the same type of object the vast majority of the time (or all of the time). For example, say the earlier example has a much larger loop (reading input from a file instead of a command line), the Square shape is much more common than the Rectangle shape; or, in the extreme case, the Square may be the only shape that actually occurs for a given execution of the program. Such a situation can be determined by profiling the types of references for which a given virtual method is invoked.

If one particular method is invoked the vast majority of the time, then the method code for the most common subclass can be inlined, with *guard* instructions placed above the inlined code. The guard simply tests the type of reference on which the following method is to be invoked. If it is of the expected (common) type, then the guard will let control pass to the inlined version of the method. On the other hand, in the rare case when the reference is for a subclass different from the expected one, the guard can branch to a noninlined invokevirtual instruction. For our earlier example, such a guarded inlined method is given in **Figure 6.13**. The If statement checks the type of the object reference on which the perimeter method is being

invoked (via the isInstanceof method) and then performs the inlined code version if it is a Square.

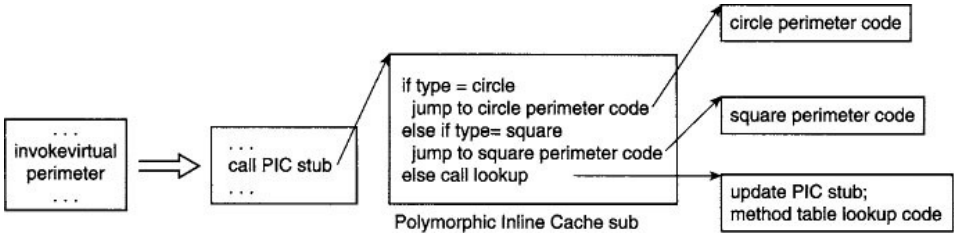


Figure 6.14 Example of Polymorphic Inline Caching.

If the method call is truly polymorphic so that full inlining is not useful, then at least the overhead of a dynamic method table lookup can be avoided using a technique similar to that for software jump prediction during binary translation (**Section 2.7**). This technique is referred to as *polymorphic inline caching* (PIC) (**Hölzle, Chambers, and Ungar 1991**) and is illustrated in **Figure 6.14**. In this example, it is assumed there is a much wider variety of shapes than in our rectangle/square example. With a PIC, the runtime system is responsible for maintaining the stub code, placing the most frequently used jumps at the top of the sequence.

Multiversioning and Specialization

The foregoing approach to inlining of virtual method calls is essentially a form of *multiversioning* (**Artigas et al. 2000**; **Gupta, Choi, and Hind 2000**). With multiversioning there are two (or more) versions of code, and one version is selected, depending on run-time information, for example, data values or type information. With method inlining one version is inlined method code, the other version is an invokevirtual instruction, and the guard selects one of the versions. This same general approach can be applied to other types of code, not just virtual method calls.

For example, in **Figure 6.15**, profiling data values may determine that the elements of the array A are almost always zero. A guard

(shown in italics in the figure) checks to see if A[i] is zero; if so, it uses a version of code that skips the rest of the instructions and simply sets B [i] to zero.

An important aspect of multiversioning is *specialization* (**Grant et al. 1999**; **Suganuma et al. 2001**). If some variables or references are always assigned data values or types known to be constant (or from a limited range), then simplified, specialized code can sometimes be used in place of more complex, general code. This is the case in **Figure 6.15**, where the specialized case occurs when A[i] is zero. Specialization can be used in conjunction with multiversioning, as shown in **Figure 6.15**, or it can be enabled via code analysis that indicates that only a single, specialized version is ever needed.

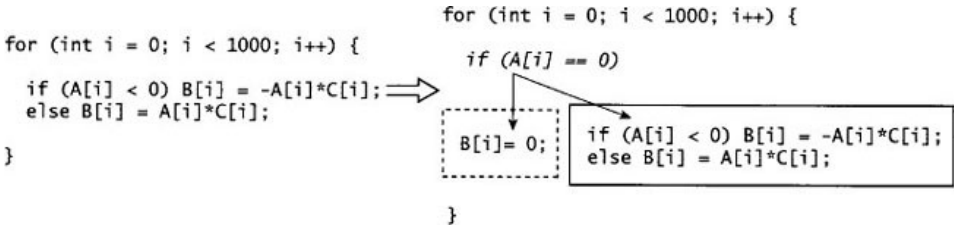


Figure 6.15 Multiversion Code. *For the common case where C[i]=0, there can be a simpler, specialized code version.*

An alternative to constructing multiple versions is to compile only a single code version (or a small number of versions) and to *defer* compilation of the general, more complex case (**Chambers and Ungar 1991**; **Whaley 2001**). For example, profiling may find that only one value has occurred up to some point in a program's execution, so an optimizing compiler may skip the general case code. In effect, it is speculating that only the special case ever occurs. However, there should be a guard to check for the general case; if it occurs, the general code can be compiled at that time. Deferred compilation is illustrated in **Figure 6.16**. This is the same code as in **Figure 6.15**, except that if A[i] is nonzero, the code jumps to the dynamic compilation system so that it can produce the required code to allow execution to proceed.

On-Stack Replacement

The stack is at the center of instruction execution in most HLL VMs, including both Java and the Microsoft CLI. Consequently, the stack is also an important consideration when optimizations are performed. To understand the relationship between the stack and program optimizations, a distinction should be drawn between the *architected* stack, i.e., the stack as specified by a Java or MSIL program, and the *implementation* stack, which is actually used during program execution (after compilation and/or optimization). For example, after method inlining, the actual implementation stack will not contain frames for inlined methods; the frame for the calling and the inlined method are merged. Furthermore, the implementation stack contents may differ from the architected stack contents after some types of optimizations, such as specialization. That the implemented stack differs from the architected stack makes no difference as long as the results are correct. The architected stack is a means for specifying the function to be performed by a program; it does not necessarily reflect the exact manner in which the function is implemented.

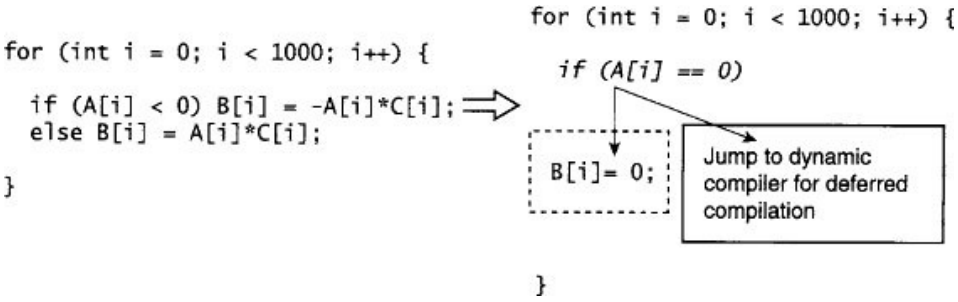


Figure 6.16 Deferred Compilation. *With deferred compilation, code is not generated for the uncommon case until it actually occurs.*

An important corollary to the foregoing is that at a given point in a program's execution, the implementation stack contents (including both the number of frames and the number of elements in each frame) may depend on the optimizations that have been performed on the program. Furthermore, there are situations where dynamic

optimization may require that the contents of the implementation stack be modified on the fly. This process of modifying the stack in response to a dynamically changing optimization level is referred to as *on-stack replacement,* or OSR (**Hölzle, Chambers, and Ungar 1992**; **Fink and Qian 2003**). Some of the situations where OSR may be used to support optimization include the following.

1. When inlining (or any other optimization) is performed, the cleanest way to insert the newly optimized code into the program flow is to wait until the method containing the optimized code is called the *next* time. However, this approach obviously does not provide benefits until the method is next called, and it may be a long time before a long-running method is called again. The extreme case occurs when a program is dominated by a single loop containing the optimized code and the loop is executed many millions (or billions) of times. In this case, the optimized method may never be called again, and therefore the optimized code must be inserted earlier if any benefits are to accrue.[1]

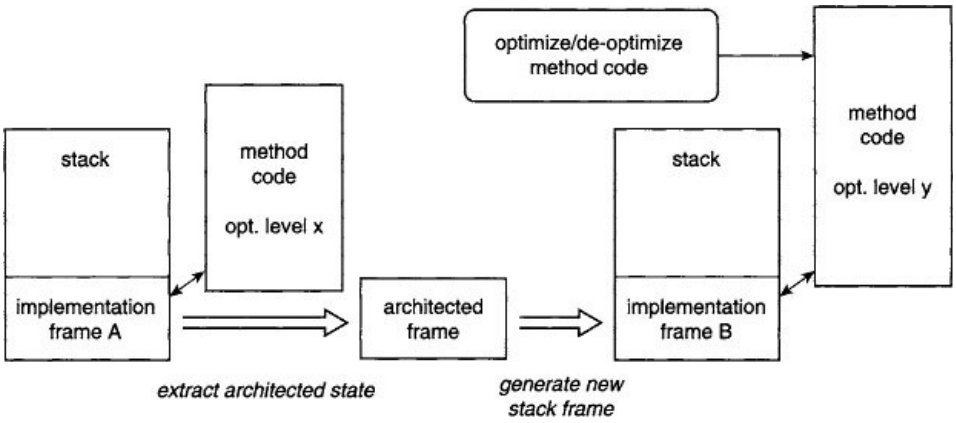


Figure 6.17 On-Stack Replacement. *With on-stack replacement, one version of a stack frame is replaced by another. This typically occurs when the optimization level of the code accessing the stack frame changed dynamically.*

2. When deferred compilation is implemented, new code may be generated while a method is in midexecution. That is, if the guard indicates that an uncommon path is to be followed, new code has to be generated, and the current implementation stack frame may have to modified to account for the newly compiled (and possibly much larger) code region.
3. When a debugger is implemented as part of an overall VM framework, the user would expect to observe the architected instruction sequence and stack contents rather than the results of optimized code and the implementation stack contents. In this situation, it may be necessary to deoptimize a method. This is the opposite

of the two earlier situations; here, OSR may be called upon to modify the implementation stack into a form that is less optimized and more nearly approximates the architected version.

On-stack replacement is illustrated in **Figure 6.17**. Here, the optimization level of code accessing the stack is being modified on the fly, so the structure and/or contents of the implementation stack frame change. The basic steps of OSR are: (1) Extract the architected frame state from the current implementation frame (and any other implementation data that may be maintained for this purpose); (2) generate a new implementation frame that is consistent with the new code version; (3) replace the current implementation stack frame with the new one.

An important application for OSR is illustrated in **Figure 6.18**. Here, multiple stack frames are being replaced by a single frame (or vice versa). This may happen if method inlining is being performed (or is being removed for debugging).

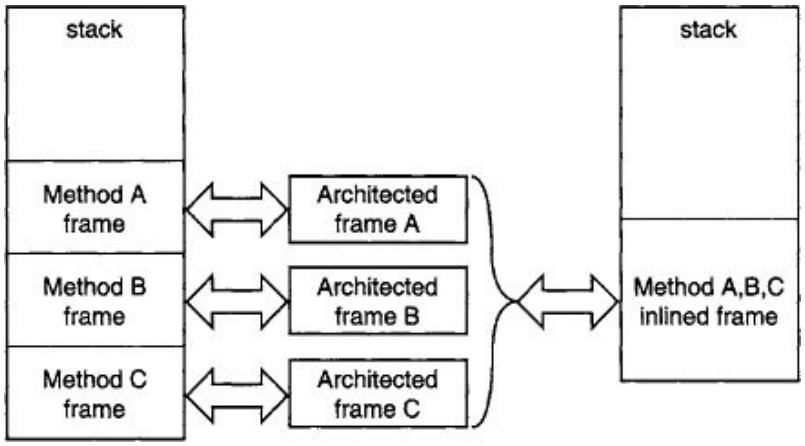


Figure 6.18 On-Stack Replacement. *On-stack replacement can be used to replace multiple stack frames with a single frame (or vice versa).*

In general, OSR is a relatively complex operation, both in constructing the architected stack state and in building a new implementation stack frame. If the initial stack frame is being maintained by an interpreter or a nonoptimizing compiler, then extracting the architected stack state is straightforward. If the stack frame is for optimized code, however, there must often be some special provision

for OSR. For example, the compiler may define a set of program points where OSR can potentially occur and then ensure that the architected values are live at that point in the execution. This is reminiscent of checkpoints and live range extension that are used by some binary optimizers when rescheduling code (see **Section 4.5.2**).

In many cases, the steady-state performance benefits enabled by OSR are relatively small. However, OSR allows the implementation of debuggers that work in the context of optimizing compilation systems. In addition, when used with deferred compilation, start-up times can be reduced by avoiding compilation/optimization of unused code sections, and the instruction "footprint" is smaller, which can improve cache performance.

Optimization of Heap-Allocated Objects

By its very nature, good object-oriented programming freely creates a large number of heap-allocated objects. However, there are a number of overheads associated with objects. Creating objects and garbage collection have relatively high costs. Furthermore, because accessing fields held in an object often involves levels of address indirection, individual object field accesses suffer small overheads that add up. To deal with creation overhead, if profiling indicates a particular type of object is frequently allocated, then the code for the heap allocation and object initialization can be inlined.

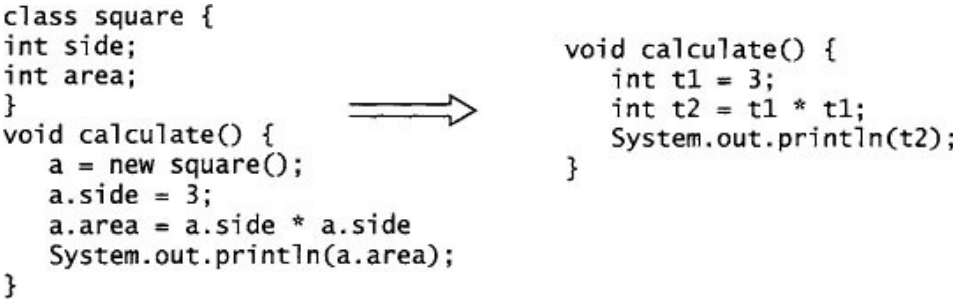


Figure 6.19 Scalar Replacement. *With scalar replacement, an object field is replaced by a scalar variable so that access delays are reduced.*

Scalar replacement (**Carr and Kennedy 1994**) is an optimization that can be very effective in some situations for reducing object access delays. This optimization replaces an object field with a scalar value. An example is shown in **Figure 6.19**. Here, on the left, a small object a is created, its two integer fields are assigned values, and one of the fields is printed. If it can be determined that object a is only used by this piece of code before being discarded, the object creation (and later garbage collection), along with the field references, can be replaced by simple scalar references, as shown on the right in **Figure 6.19**. Scalar replacement when applied to objects requires reference *escape analysis* (**Choi et al. 1999**), that is, an analysis to make sure all references to the object are within the region of code containing the optimization. In this example, no references to a escape from the optimized code region.

The physical placement of data fields within an object is an implementation-dependent feature. Consequently, improved data cache performance can be achieved by ordering object fields according to usage patterns (**Chilimbi, Davidson, and Larus 1999**; **Kistler and Franz 2000**). Furthermore, some field references can be removed entirely by using conventional compiler optimizations. For example, redundant getfield and putfield operations can be found and removed. This is illustrated in **Figure 6.20**. Here, the same field is accessed twice (as a. side and as c. side). Based on dataflow analysis, the redundant access can be found, and the corresponding field value is held in (and copied from) a temporary location, t1 (possibly a register), avoiding the second getfield (to c. side).

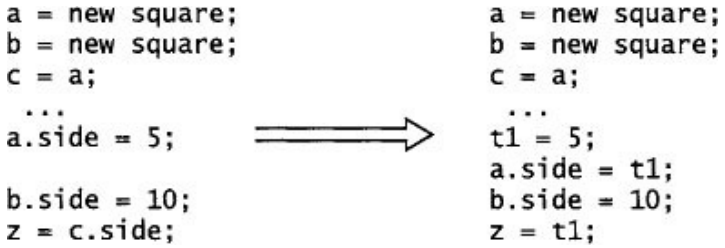


Figure 6.20 Example of Redundant getfield (load) Removal.

Low-Level Optimizations

Besides the optimizations given earlier, which tend to be especially effective when applied to object-oriented programs, many conventional optimizations, discussed earlier, in **Chapter 4**, can also be applied. These include dead-code removal, branch optimization, copy and constant propagation, strength reduction, and code rescheduling, among others. Because they have already been covered we will not go into further detail here. However, we do cover some extensions and other considerations that are more specific to the object-oriented context.

One potentially significant overhead found in object-oriented HLL VMs is the need to perform array range and null reference checking. In theory, every time an array is accessed or a reference is used, these checks must be performed; perhaps just as important, an exception may be thrown. This means there are two potential causes of performance loss. One is the need to perform range/null check itself, and the other is the inhibition of other optimizations because of a potential thrown exception if the check should fail. In the latter case, the architected process state must be precise at the time of an exception, and, just as with binary optimizers, some optimizations may be inhibited if a precise state must potentially be materialized.

The overhead of null pointer checking can be largely eliminated by using an “illegal” out-of-range address to represent the null pointer value, i.e., a memory address for which the Java process does not have read or write privileges. Then an attempt to use a null pointer will result in a trap, which, via an OS-supported signal mechanism, can be reported back to the JVM runtime system. However, the *potential* for an exception (and the need to restore a precise state) remains, so some code optimizations, for example, involving code motion, may still be inhibited.

A common way to deal with range/null check operations is to treat