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.

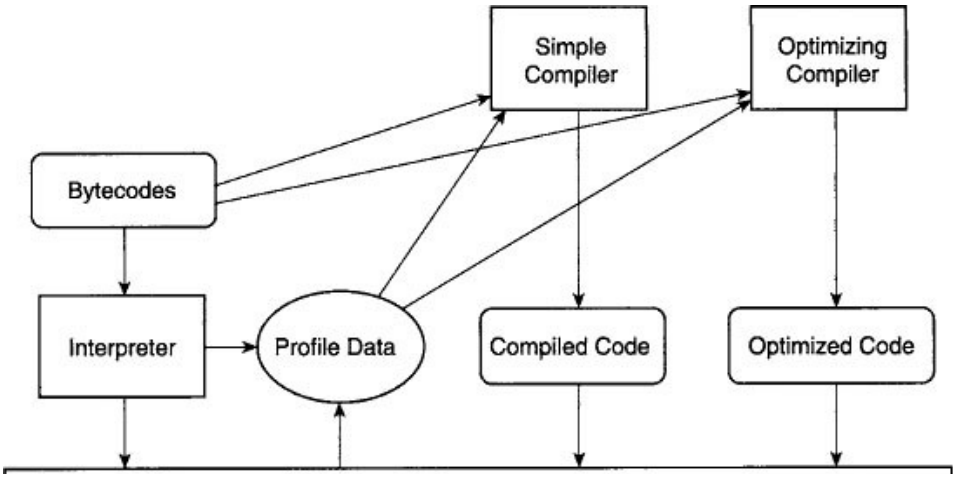




Figure 6.11 A Typical Dynamic Optimization Framework.

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

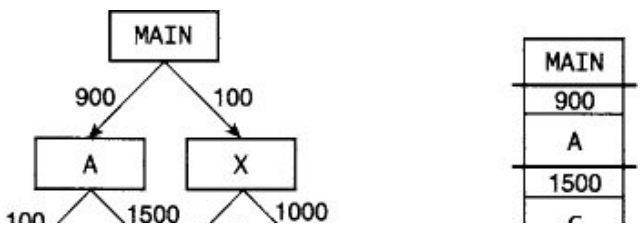
Method inlining is referred to as *procedure inlining* in **Section 4.3.1**. With inlining, a method call is replaced with the actual code contained in the method; i.e., the method code is placed "inline" with the calling code (**Suganuma, Yasue, and Nakatani 2002**). The overheads of passing parameters, 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.



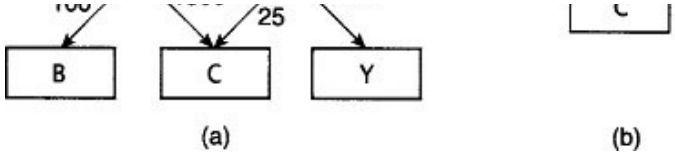


Figure 6.12 Tracking Method Calls. *(a) With a call graph; (b) via stack frames.*

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.

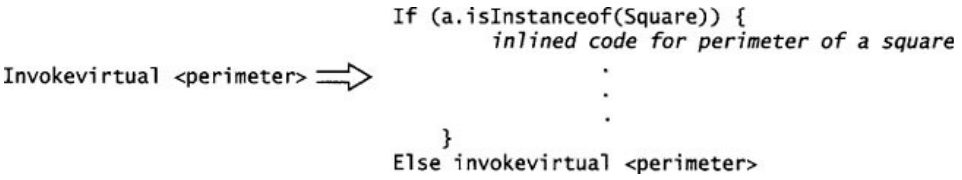


Figure 6.13 An Inlined Method Call Protected by a Guard.

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.

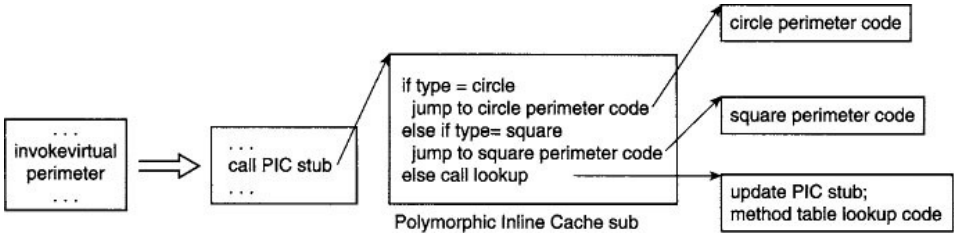


Figure 6.14 Example of Polymorphic Inline Caching.

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.

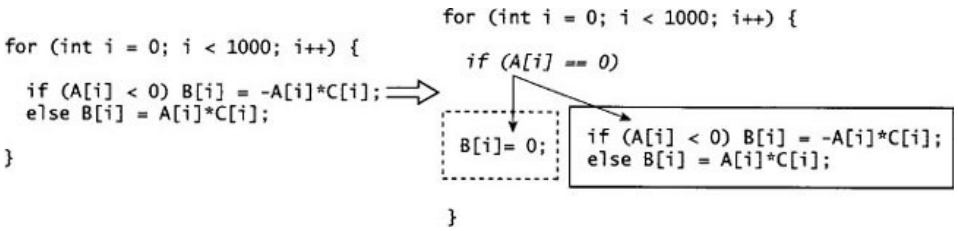


Figure 6.15 Multiversion Code. *For the common case where C[i]=0, there can be a simpler, specialized code version.*

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.

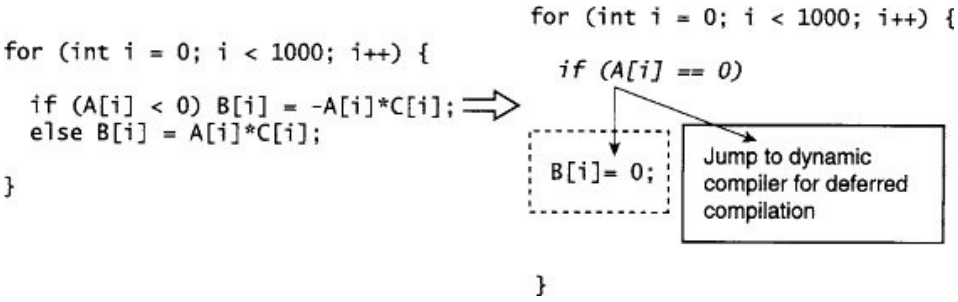


Figure 6.16 Deferred Compilation. *With deferred compilation, code is not generated for the uncommon case until it actually occurs.*

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]

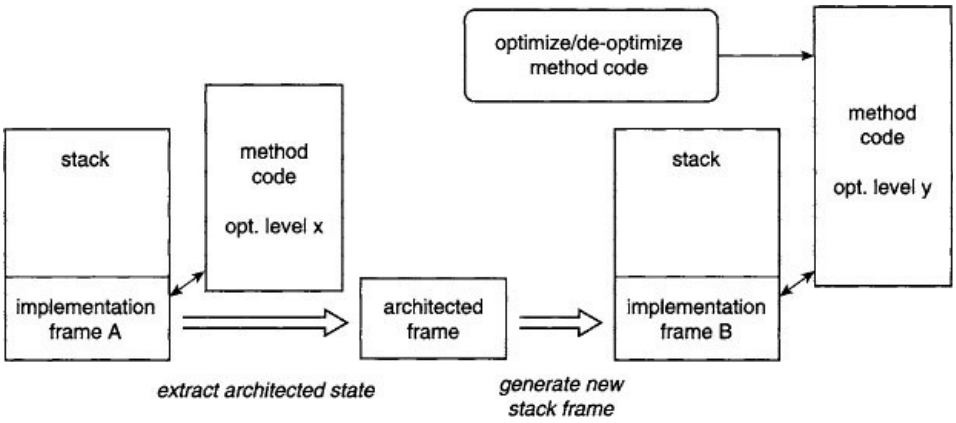


Figure 6.17 On-Stack Replacement. *With on-stack replacement, one version of a stack frame is replaced by another. This typically occurs when the optimization level of the code accessing the stack frame changed dynamically.*

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).

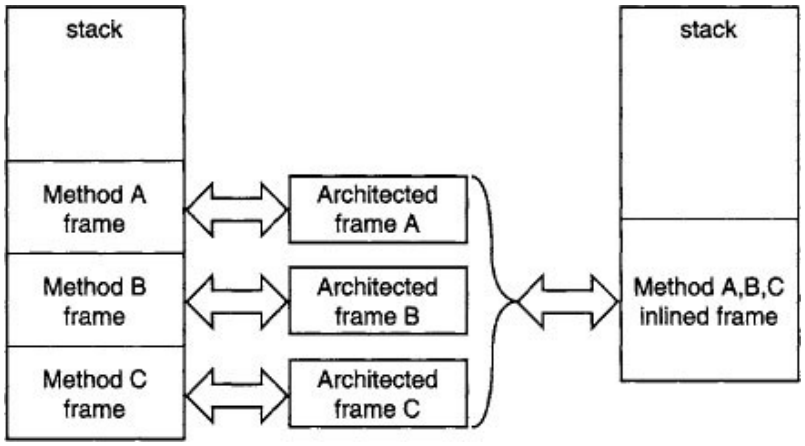


Figure 6.18 On-Stack Replacement. *On-stack replacement can be used to replace multiple stack frames with a single frame (or vice versa).*

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.

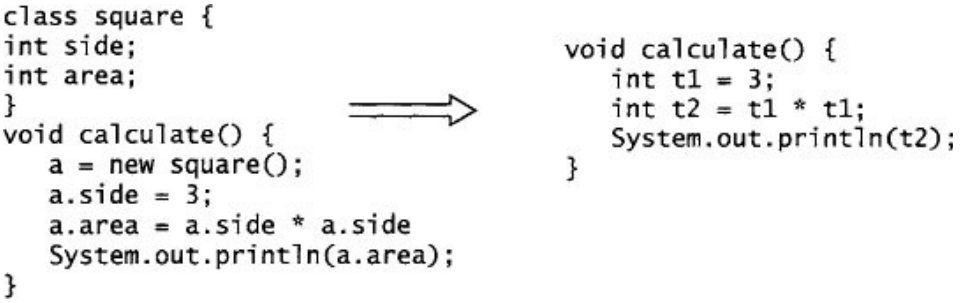


Figure 6.19 Scalar Replacement. *With scalar replacement, an object field is replaced by a scalar variable so that access delays are reduced.*

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).

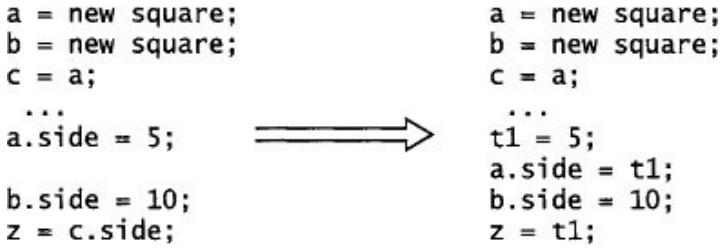


Figure 6.20 Example of Redundant getfield (load) Removal.

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

them as if they are regular instructions and then to perform the analogous optimizations. For example, just as a redundant instruction can be removed or an invariant instruction hoisted out of a loop, so, too, a redundant or invariant check can be removed or hoisted. A simple example is in **Figure 6.21**. Here, the same object reference (held in p and r) is used for a series of getfield and putfield operations. If the first putfield (p.x = ...) passes the null check, then the getfield using the same reference will as well. The null check property can also be propagated to reference r. Thus, only the first null check for p is needed, and the later checks for p and r become redundant. After the redundant null checks are removed (right-hand side of the figure), then the putfield to q. x can be rescheduled after the putfield to r. x.

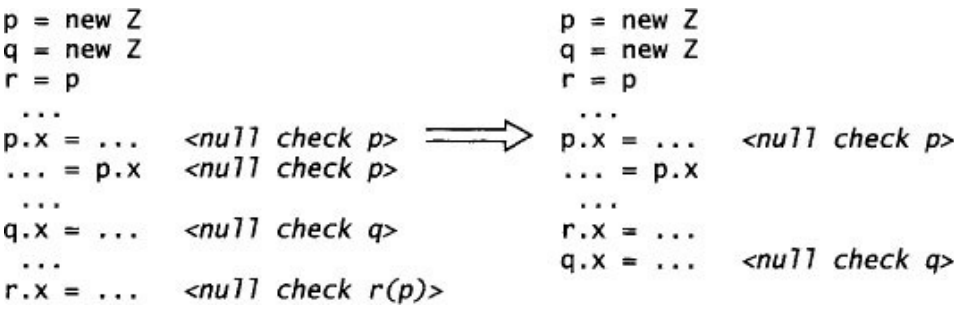


Figure 6.21 Removing Redundant Null Checks. *After redundant null checks are identified and removed, optimizations involving code motion are enabled.*

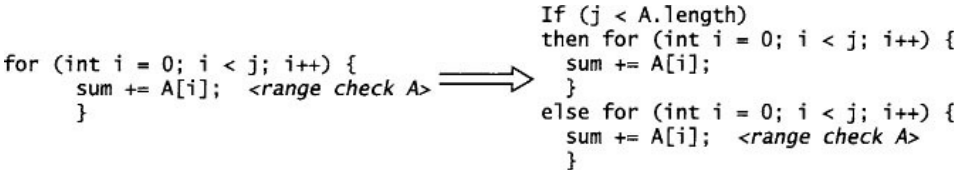


Figure 6.22 Hoisting an Invariant Check. *An array range check is hoisted outside a loop. Then in the common case, no inner loop range check is required.*

An example of hoisting an invariant check is in **Figure 6.22**. In the code on the left, an array range check is performed every loop iteration. However, the check can be hoisted outside the loop (right-hand side of the figure) and only needs to be performed once. After the check has been hoisted outside of the loop, it takes the form of a guard for two versions of the code, one that does not perform the

check and the other that does.

As a final example, a technique known as *loop peeling* can be performed to avoid null pointer problems. In **Figure 6.23** the loop body contains an object reference with a null check. Hence, code motion around the reference is inhibited. However, the first loop iteration can be peeled (right-hand side of the figure); if this first iteration does not cause a null pointer exception, none of the other iterations will. Hence, the null check is not needed for the remaining loop iterations, and other code optimizations can be enabled.

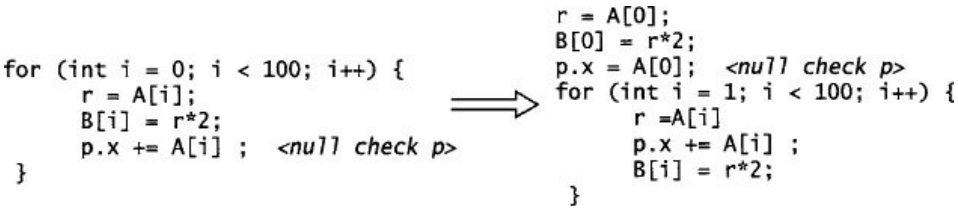


Figure 6.23 Loop Peeling. *By peeling the first loop iteration, the null checks in the remaining loop iterations can be removed.*

Optimizing Garbage Collection

Garbage collection is a key part of a high-performance VM implementation, and there are a number of ways the compiler can provide assistance to the Java runtime to enhance garbage-collection efficiency. First, there may be times when the heap state is temporarily inconsistent, for example, when object references are being modified. At these points, to avoid errors, garbage collection should not be initiated. Consequently, the compiler can provide the garbage collector with “yield point” at regular intervals in the code. At these points a thread can guarantee a consistent heap state so that control can be yielded to the garbage collector. Also, the compiler can help with specific garbage-collection algorithms. For example, if a generational collector is used, then write barriers must be provided by the compiler.

6.7 Case Study: The Jikes Research

Virtual Machine

The Jikes research virtual machine (RVM) was developed at IBM Research and has been made available to the general VM research community (**Arnold et al. 2000**). Jikes is based on an earlier research effort, Jalapeno, and was initially described in research papers using that name. The discussion here is based on descriptions of both the Jalapeno and the Jikes systems.

The overall optimization strategy is to compile only; there is no interpretation step. First, there is a baseline compiler that translates bytecodes directly into native code. This compiler performs no register allocation *per se;* rather, the generated code simply emulates the Java stack. Then for optimization, there is a dynamic compiler that supports three levels of optimization, invoked in stages, depending on an estimate of cost–benefit.

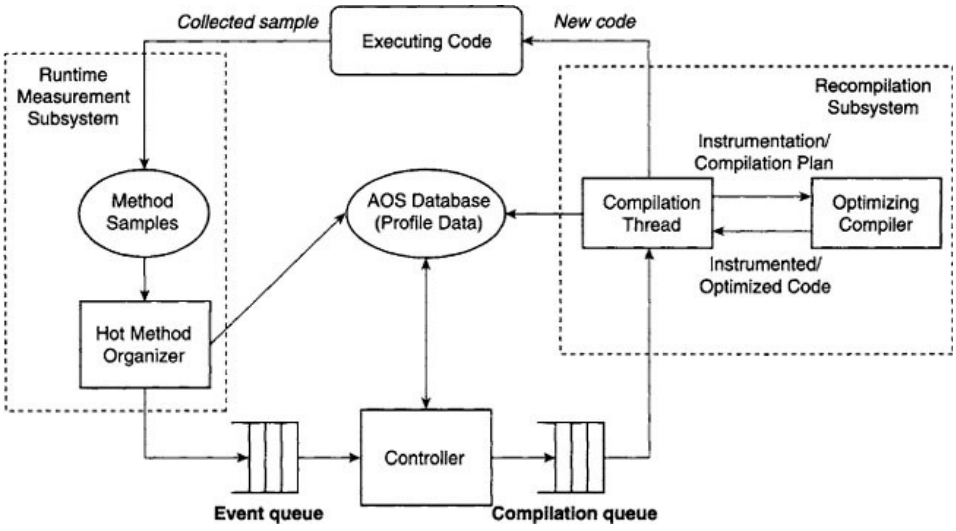


Figure 6.24 The Jikes Adaptive Optimization System (AOS).

Jikes is a multithreaded implementation, with threads responsible for optimization running concurrently with executing program threads. Jikes multiplexes Java application and runtime threads onto threads supported by the underlying host platform, e.g., AIX

pthreads. To allow preemptive thread scheduling, the compiler places yield points, typically at method prologs, epilogs, and loop back edges. At a yield point, the code tests a control bit; if the thread scheduler wants to preempt a thread, it will set the thread's control bit. At the next yield point, the thread will call the thread scheduler.

The overall architecture of the Jikes adaptive optimization system (AOS) is shown in **Figure 6.24**. The major subsystems are the runtime measurement system, the recompilation system, and the controller. These subsystems all interact with the AOS database, which contains profiling data and a history of optimization decisions.

The runtime measurement subsystem gathers raw performance data by sampling at yield points. That is, when a thread switches, a back-edge yield point increments an activity count for the method containing the edge. If it is at a method prolog, the activity count for the calling method is incremented; if it is at a method epilog, the activity count for the called method is incremented.[2] The system can also support other types of compiler-inserted instrumentation to perform edge profiles, basic block profiles, or value profiles, depending on the optimizations that are under consideration. Performance data is initially kept in raw form. Periodically, the raw method samples are analyzed by a separate thread, the hot method organizer, which summarizes and puts the profile data in a form more easily used by the rest of the optimization system.

The controller is responsible for coordinating the activities of the runtime measurement and the recompilation subsystems. It instructs the measurement subsystem to initiate, continue, or change profiling activities. When a method is sampled, that is, found to be active during a thread switch, the hot method organizer places it on the event queue for the controller to consider for recompilation. The controller then makes decisions regarding the recompilation, e.g., whether recompilation of the method should be performed and, if so, at what optimization level.

An important part of the controller's function is to determine whether a method should be recompiled at a higher optimization level. To do this, Jikes uses an analytical cost-benefit model. Say that a given method *m* is currently compiled at optimization level $i < N$, where level *N* is the highest optimization level. Then the controller estimates (1) *Ti,* the expected time the program will spend executing method *m* if it is not recompiled; (2) *Cj,* the cost (time required) of recompiling method *m* at level $j \geq i$; (3) *Tj,* the expected execution time the program will spend in *m* after recompilation. Note that a recompilation at the same level, *i,* is considered because there may be more profiling data available than what was available the previous time *m* was compiled at level *i;* this additional profiling data may change the particular optimizations performed at level *i.* Then the controller determines the optimal level *j,* where $Cj + Tj$ is minimized. Recompilation is performed at level *j,* unless, of course, $Cj + Tj \geq Ti$ and no recompilation is done.

Although this cost–benefit analysis seems very straightforward, the real difficulty comes in estimating the *C* and *T* values. Consequently Jikes uses a combination of heuristics and experimentally derived parameters. To arrive at an estimate of the *T* values, the controller assumes that at its current optimization level, the method will consume as much execution time in the future as it already consumed; i.e., that $Ti = Tcurrent$. The *Tj* values are determined by first estimating the speedups of the different optimization levels using off-line benchmarks. The speedup of optimization level *k* versus level 0 is *Sk*. Then $Tj = Ti * Si/Sj$. Finally, the recompilation cost *Cj* is estimated to be a linear function of the size of the method to be recompiled. The multiplying constant is again determined by off-line benchmarking. If the controller decides that recompilation should be done, it places the pertinent information into the recompilation queue, to be serviced by one of the compilation threads.

The recompilation subsystem consists of a number of threads that

The recompilation subsystem consists of a number of threads that remove compilation plans from the compilation queue. Because they run as separate threads, they can execute concurrently with the Java application threads. A compilation plan consists of the optimizations that should be performed, profile data that will be used by the optimizer, and an instrumentation plan, i.e., the profiling instrumentation the compiler should put in the code it generates (for potential future optimizations).

The AOS database contains a history of profile data and the compilation plans, status, and history of recompiled methods. The controller and recompilation subsystem can query this database to guide further recompilation decisions and optimizations.

The Jikes compiler has three levels of optimization. A brief summary of the optimizations performed at each level is given next.

Level 0: Includes many of the conventional compiler optimizations, such as copy and constant propagation, common subexpression elimination, dead-code elimination, branch optimizations, and several others. So-called *trivial* methods are inlined—these are methods where the code in the method is smaller than the method's calling sequence. In addition, some simple code relayout is performed, and register allocation follows a simple linear scan algorithm.

Level 1: Because many of the classical optimizations are done in level 0, the additional optimizations deal with higher-level code restructuring—there is more aggressive inlining, based on profile information, and there is more aggressive code relayout.

Level 2: Uses a static single assignment (SSA) intermediate form, where each register variable is assigned a value only once (**Cytron et al. 1991**; **Cooper and Torczon 2003**). Using this form allows a number of global optimizations (or more effective versions of conventional optimizations, such as common-subexpression elimination). It also includes some optimizations that can lead to significant

duty). It also includes some optimizations that can lead to significant code expansion unless judiciously applied (with the benefit of profile data). For example, loop unrolling replicates the body of a loop, thereby increasing opportunities for other optimizations and eliminating a number of loop-closing branches.

The developers of Jikes have published a number of benchmark results. We give some of the results for the SPECjvm98 benchmark suite. There are seven programs with class files ranging from 10KB to nearly 1.4MB. Performance results are given to measure "start-up" performance and "steady-state" performance. There are two differences between start-up and steady state. First, the steady-state data sets are approximately an order of magnitude larger than the start-up data sets. Second, each steady-state benchmark is run for five iterations by the same JVM instance and the best iteration performance is reported. The start-up behavior will be influenced much more heavily by initialization of data structures, class file loading, and compilation overheads. The steady-state performance is closer to what one might expect in a more realistic production environment, where the start-up overheads have more time to be amortized.

For comparison, a number of compilation scenarios are benchmarked and compared with the simple baseline compilation. In one set of scenarios, the compiler with each of the three optimization levels is run as a JIT compiler. That is, the first time a method is invoked, it is immediately compiled at the given optimization level and remains there throughout the program's execution. The AOS implementation uses all optimization levels in a staged manner, and optimization is performed on a per-method basis. With AOS + FDO (Feedback Directed Optimization), more complete profiling information is maintained, so better inlining decisions can be made and optimizations can be better targeted within methods.

Performance is evaluated as speedup over the simple nonoptimizing

baseline compiler. Figure 6.25 shows start-up results for each of the benchmarks and the harmonic mean speedup for all seven. For the JIT compilers, the lower the optimization level, the better the performance (compress is the only notable counterexample where JIT 1 outperforms JIT 0). For the start-up benchmarks, the extra time required for optimizations cannot be amortized, due to the relatively short run times (with the exception of compress, where code quality is more important than in the other benchmarks). All the JIT compilers are below the baseline, because they provide at least level-0 optimization for all methods, even those executed only once. The AOS and AOS + FDO dynamic compilation methods perform best, giving a speedup of about 1.75, because they only take the time to perform compilation beyond baseline, possibly followed by additional optimizations, where there is good likelihood of performance benefit.

Figure 6.26 shows results for the steady-state benchmarks. Here, the results are quite different than for the start-up case. Now, the higher the optimization level for the JIT compilers, the better the performance. Using level 0 as a JIT gives a speedup of 1.9 over the baseline, and going up to level 2 yields a speedup of about 2.25. Interestingly, using the AOS staged optimization typically falls slightly short of the level-2 JIT. Because the same code is executed during each iteration, the JIT performance runs can complete compilation during the first iteration and then execute four iterations with no profiling or compilation overhead. On the other hand, the AOS performance runs profile and recompile through all five iterations. For two benchmarks, AOS without FDO actually beats JIT2, however, because AOS sometimes recompiles a method at a later time (in program execution) than JIT2 does. This later compilation allows more class loading to take place, eliminating some unresolved references, and thus the code generated by AOS is more highly optimized. Finally, because the AOS + FDO is able to perform additional profile-guided optimizations, for some benchmarks it provides significant speedup over the method-granularity optimizations used by the AOS implementation.

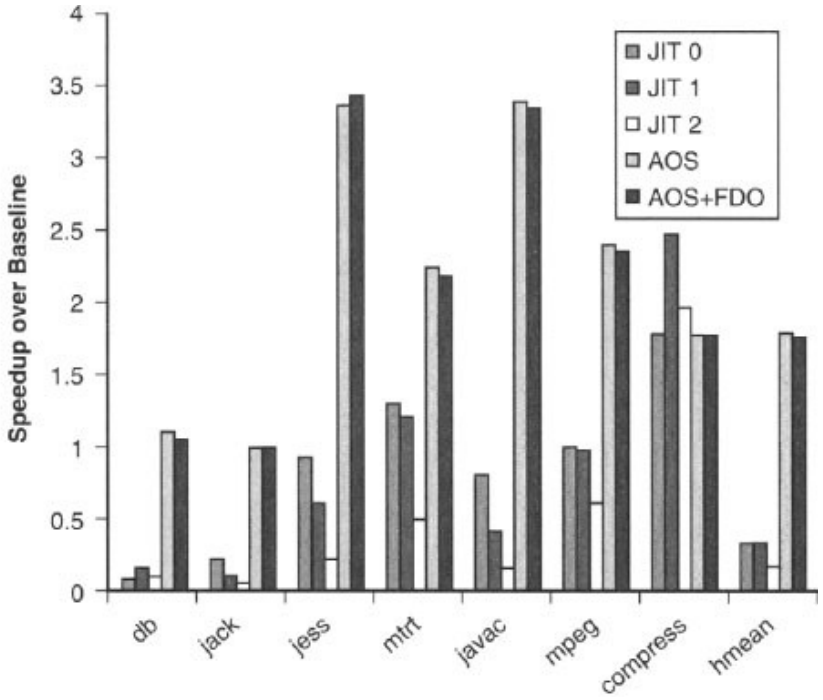


Figure 6.25 Jikes Start-up Performance on Seven SPECjvm98 Benchmarks.

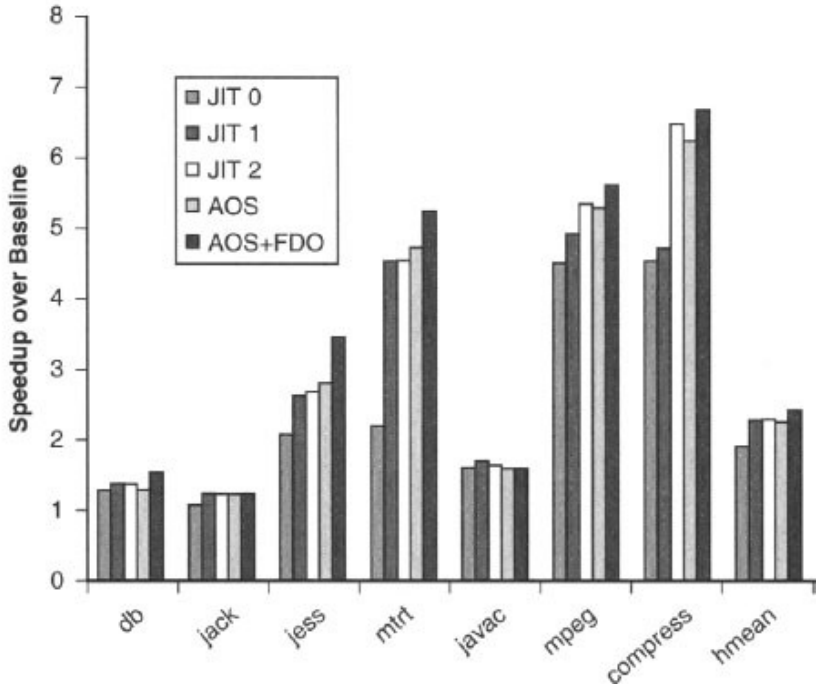


Figure 6.26 Jikes Steady-State Performance on Seven SPECjvm98 Benchmarks.

It is also interesting to see where an advanced JVM spends its time. An execution profile for Jikes is given in **Figure 6.27**. This particular

profile is for the entire SPECjvm98 benchmark suite, with each being executed for one iteration. Across different applications, the execution profile may vary significantly from this average performance, especially in time spent in garbage collection. Over the entire benchmark suite, performing optimized recompilation consumes only 7% of the total time, and all the other tasks related to adaptive optimization, i.e., the various control and bookkeeping functions, consume negligible time. We see that 86% of the time is spent in execution application threads and 6% is spent in the garbage collector. This tends to highlight the importance of garbage collection in a JVM; about as much time is spent in garbage collection as in performing optimizations on the code.

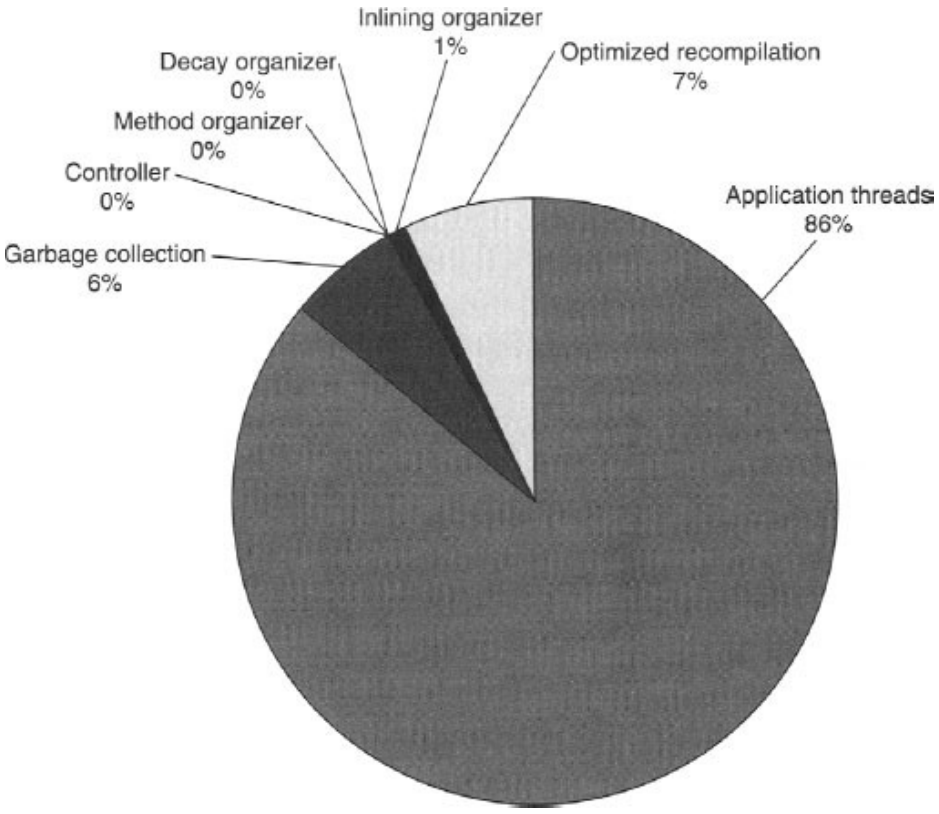


Figure 6.27 Where the Jikes System Spends Its Time.

6.8 Summary

The conventional belief is that programs running on an HLL VM are

relatively slow. This is a misleading notion. Some of the early Java VMs *were* slow. A heavy reliance on interpretation is naturally slow during steady-state emulation, and dynamic optimizing methods that are not selective, i.e., simple JIT compilation, will have high start-up times. However, modern HLL VM implementations rely on compiled code and are much more selective regarding when to apply optimizations and where. This leads not only to less overhead but to better optimizations in the steady state. Furthermore, slowdown due to the dynamic VM paradigm should not be confused with slowdown due to support for object-oriented programming. Object-oriented programming leads to more robust programs and a high level of code reusability (among other advantages), but sometimes there is a performance cost for these important software engineering advantages. Inherent in object-oriented code are additional levels of indirection that may not be found in conventional C-type code, for example. But this is usually caused by the object-oriented nature of the software, not the fact that it is running on a VM.

With the development of the Java platform and the more recent Microsoft .NET framework, HLL VMs already have achieved a "critical mass." At some point in the (not-too-distant) future, virtually all application programs are likely to be developed for a platform-independent HLL VM. This will change common programming practice (programming in C will be viewed in much the same way as programming in assembly language is viewed today), and it will likely change the way that hardware platforms are implemented. If all software is developed for a platform-independent HLL VM, the level of standardization is raised and the underlying hardware ISA becomes much less relevant.

---

1. It is not clear, however, that this type of behavior happens often enough in real programs to make such a technique worthwhile. This type of behavior is sometimes observed in kernel benchmarks (also known as *microbenchmarks*), and this may in fact be the prime motivation for considering such an optimization method in the

fact be the prime motivation for considering such an optimization method in the competitive benchmark-oriented environment in which high-performance VMs are often developed.

2. These yield points and profiling heuristics reflect the current Jikes implementation, not the one in the paper describing Jikes (Jalapeno) (**Arnold et al. 2000**).

Previous chapter
Chapter 5: High-Level Language Virtual Machine Architecture

Next chapter
Chapter 7: Codesigned Virtual Machines

Chapter Seven

Codesigned Virtual Machines

Both hardware and software have undergone radical changes since the first commercially available computers appeared in the early 1950s. In contrast, the basic character of the interface between hardware and software has been virtually unchanged during that time. Consequently, the instruction set architectures (ISAs) currently in use reflect a perspective and a division of labor between hardware and software that is decades old. At a time when hardware resources were very expensive and hardware was relatively simple, it made sense for the ISA to be a direct reflection of the specific hardware implementation. If the hardware had an accumulator, then the ISA had an accumulator. Over the years, however, hardware resources have become plentiful and inexpensive. Today, hundreds of millions of transistors are available on a single chip. As a consequence, underlying processor hardware has grown to be quite different from the image presented by the commonly used ISAs. For example, consider the Intel IA-32, one of the older microprocessor ISAs in use today. The IA-32 uses a CISC ISA, originally conceived for sequential in-order execution with relatively few general-purpose registers and stack-based floating instructions. A modern IA-32 implementation, however, converts the CISC ISA into what is essentially a RISC ISA running on a dynamic superscalar microarchitecture having lots of registers and register-based floating-point instructions (**Hinton et al. 2001**; **Keltcher et al. 2003**). This CISC-to-RISC conversion is done entirely in hardware (see **Figure 7.1**).

The reason behind hardware-intensive dynamic translation of legacy ISAs is the need to maintain their role as the interface between hardware and software. Today's ISAs are valuable interfaces

tween hardware and software. Today's ISAs are valuable interfaces, ones in which huge software and infrastructure investments have been made. Compatibility has also become the largest obstacle to implementing new ISAs that are better suited to today's technology. Fortunately, virtual machine technologies permit new ISAs by enabling a different approach to general-purpose processor design. Virtual machine technologies can turn processor development into a *codesign* effort, where the host architecture (the target ISA in particular) is designed concurrently with the VM software that runs on it. Because the host hardware is developed as an integral part of the VM, it is not necessary to use an existing host platform or target ISA. Taken together, however, the codesigned hardware and software support a conventional source ISA (and all the software developed for it).

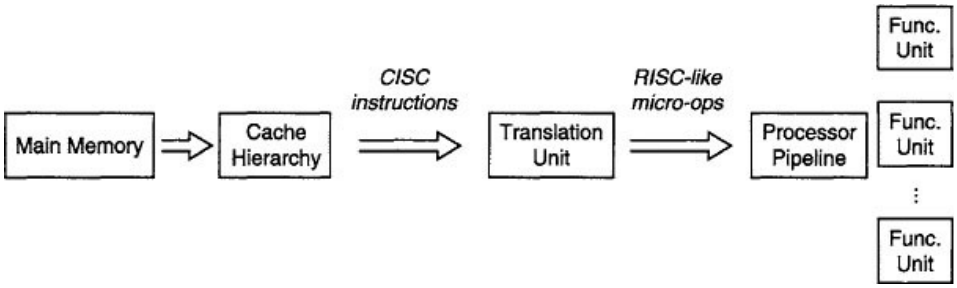


Figure 7.1 CISC-to-RISC Conversion. *Conventional high-performance microarchitectures for a CISC instruction set use hardware to translate CISC instructions into RISC-like micro-ops.*

These *codesigned virtual machines* open new avenues for architectural innovation. Because software becomes part of the "hardware" platform, the interface between hardware and conventional software is shifted upward, and there are new opportunities for dividing the implementation between hardware and software in an optimal way. Because codesigned VMs support an entire system, OS plus applications, they are a form of system virtual machine. However, unlike most other system virtual machines, codesigned VMs are not intended to virtualize hardware resources other than the processor, nor are they intended to support multiple VM environments. Rather, the goals include performance, power efficiency, and design

simplicity.

Perhaps the first codesigned VM was the IBM System/38 (**Bertsis 1980**), later known as the AS/400 (**Soltis 1996**) and currently the iSeries. The System/38 was designed to support a guest ISA with higher-level semantics than could be directly supported by hardware. This reduces the "semantic gap" between a conventional ISA and higher-level software (**Myers 1982**). A second objective was to separate the source ISA from the target ISA so that successive hardware platforms could be redesigned while maintaining software compatibility. The success of this approach was demonstrated as the AS/400 family evolved, changing its host platform ISA from the original proprietary CISC ISA to an extended PowerPC ISA. This migration took place in a completely transparent way from the users' perspective.

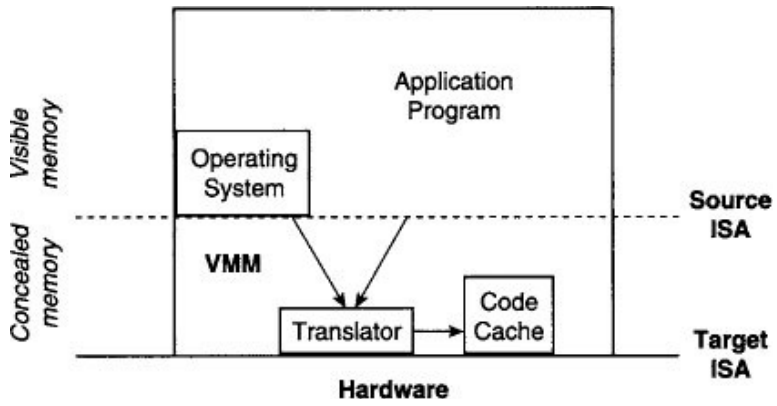


Figure 7.2 Virtual Machine Monitor. *A codesigned virtual machine architecture supports a conventional source ISA via a combination of hardware and software running in concealed memory.*

Another commercially available codesigned VM, developed by Transmeta (**Halfhill 2000**; **Klaiber 2000**) implements a target instruction set where a number of independent instructions placed in a very long instruction word (VLIW). The codesigned software is responsible for finding independent instructions in a conventional source ISA (the IA-32) and packing them into the target VLIW ISA. Because they are independent, the instructions in a VLIW can be issued for execution in parallel without complicated hardware sup-

port. Therefore, the advantage of this codesigned VM implementation is that it does not require the complexity (and power consumption) of an out-of-order issue unit as used in most superscalar processors.

An overview of a codesigned VM is shown in **Figure 7.2**. It is a system VM, as noted earlier, and all the VM software resides in a region of memory that is completely concealed from all conventional software. In keeping with the naming convention used for system VMs, we refer to the virtual machine software as a virtual machine monitor (VMM).

The primary function of the VMM is to emulate the source ISA; in this respect there are many similarities to process VMs. For example, emulation is often done in stages, with the focus on dynamic translation into a code cache. There are differences with respect to process VMs, however, the two most significant being the following.

1. There must be intrinsic compatibility (**Section 3.2**) at the ISA level rather than the ABI level. Because the interface at which virtualization takes place is the ISA, not only must the user-level instruction set be emulated, but the entire system-level ISA must be emulated. This includes the memory architecture, for example, so page fault compatibility must be maintained. It also means that I/O can be dealt with at the hardware level, not the abstracted OS call level as in process VMs.
2. The reason for using a virtual machine implementation is improved performance, power efficiency, design simplicity, or some combination of these. Compatibility is a requirement but is not the motivation for constructing a codesigned VM.

Codesigned VMs also bear some similarity to conventional superscalar processor designs (**Figure 7.1**): Both perform translation from a source ISA to a target ISA implemented in hardware. The difference is that one does the translation in software while the other does it in hardware. We should note, however, there have been a number of proposals for innovative processor implementations that are similar to codesigned VMs, typically relying on microcode or special-purpose coprocessors, rather than VM software to perform

translation or to assist with interpretation (**Debaere and Van Campenhout 1990**; **Nair and Hopkins 1997**; **Chou and Shen 2000**; Patel and Lumetta 2001). Nevertheless, in a conventional superscalar design of a CISC ISA, hardware (in one form or another) must carry the burden of performing the complex decomposition from CISC instructions to the RISC micro-ops that are directly executed. This is no doubt a contributor to the high cost of hardware design verification—a design cast into silicon is hard to debug and expensive to change.

Relying on hardware-based translation also limits the extent of the ISA translation that can be done. For example, inter-instruction optimization is difficult, and in a conventional superscalar implementation there is none. This is illustrated in **Figure 7.3**. With conventional hardware translation, **Figure 7.3**a, each source instruction is translated to micro-ops independent of the other source instructions, i.e., in a context-free manner. On the other hand, with software translation and optimization, as in **Figure 7.3**b, blocks of source instructions can be translated and optimized as a group, leading to more optimization opportunities (**May 1987**). Moreover, with the conventional hardware method, power consumption tends to be higher, due to the large amount of hardware for transforming source instructions into an executable form and scheduling them for execution.

Codesigned VMs are not as widely used as other types of VMs; there are few examples that are commercially available, and they remain the subject of research. They are of interest primarily because of their potential for supporting significant innovations in processor design. Furthermore, many of the hardware-based techniques developed for codesigned VMs can potentially be incorporated as extensions to existing ISAs so that other types of VMs, e.g., process VMs and HLL VMs, can be implemented more efficiently on standard platforms.

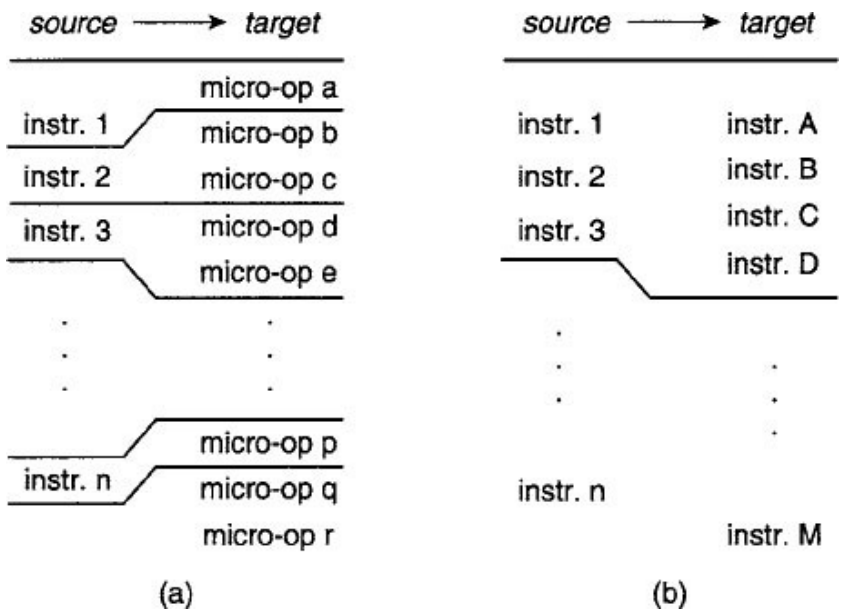


Figure 7.3 Code Translation Methods. *(a) Context-free translation, where each individual source instruction is translated to target instruction(s) (micro-ops); (b) context-sensitive translation, where blocks of source instructions are translated to blocks of source instructions.*

In the remainder of this chapter, we describe the implementation of codesigned VMs. This includes features that make the emulation process in codesigned VMs more efficient as well as specific performance and/or efficiency features that are enabled by the codesigned VM paradigm. The bulk of the chapter focuses on codesigned VMs intended to implement conventional ISAs; e.g., the Transmeta Crusoe implements the IA-32 ISA. It is this type of codesigned VM that has received the most attention in recent years, especially from researchers. Consequently, many of the techniques described in this chapter are centered on fast emulation of conventional ISAs. The IBM AS/400, in contrast, uses a source ISA that is at a higher level than most conventional ISAs and therefore relies more on innovative software than innovative hardware techniques to achieve its goals. The techniques used in the AS/400 are discussed primarily as part of the case study in **Section 7.8**.

7.1 Memory and Register State Mapping

State mapping is easier with a codesigned VM than with many conventional VMs. Because the target ISA is designed specifically for the source ISA, the host register file(s) can be made large enough to accommodate the guest's requirements, with extra scratch registers left over to enhance performance and/or simplify the translation process. An example of PowerPC state mapping for integer registers, as used in the IBM Daisy processor (**Ebcioglu et al. 2001**), is illustrated in **Figure 7.4**. Here, the PowerPC integer registers r0–r31 are mapped directly to host registers R0–R31. Then the branch unit counter and link register are mapped to registers R32 and R33. The PowerPC MQ register was originally used for multiplication and division but is now obsolete; in the original Daisy system it was mapped to host register R34. The constant value of 0 is placed in R35 to allow fast emulation of those cases where r0 is defined to be 0 in the PowerPC ISA. Finally, host registers R36–R63 are scratch registers to be used by the VMM emulator for such things as holding speculative values produced during code optimization, for constants, and for pointers to VMM tables.

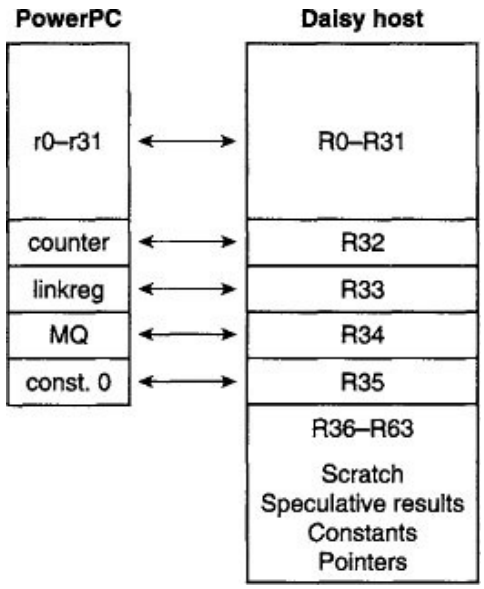


Figure 7.4 Mapping of Guest PowerPC Integer Registers to the Host Platform Register Set, as Done in the IBM Daisy System.

For memory state mapping, the key element of a codesigned system VM is the *concealed memory*. Concealed memory is a reserved region of the memory space where the VMM, code cache, and other emulation software and tables reside. There can be both concealed logical (virtual) memory and concealed real memory. The concealed memory is never made visible to any of the conventional guest software.

Because the VMM takes control immediately after system reset, it essentially has control of the system from the beginning, including the boot process, so it can make sure the conventional software never sees the concealed memory. When the guest OS boots (under VMM control) and checks to see how much real memory is available, e.g., by reading a control register, the VMM intercepts the read and does not inform it of the existence of the concealed real memory. And any attempt to access concealed memory by conventional software results in exactly the same behavior as if the memory were not present (typically a trap).

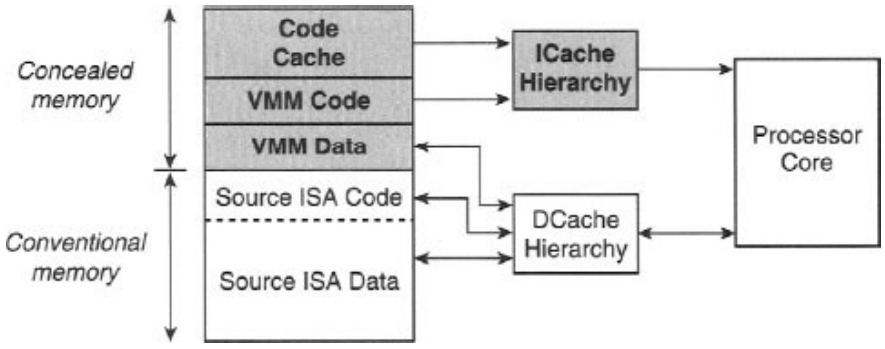


Figure 7.5 Memory System in a Codesigned Virtual Machine. *The shaded areas are those where host instructions and VMM data reside.*

Figure 7.5 illustrates an implementation of real memory in a codesigned VM. The shaded area of main memory is concealed, accessible only via the VMM. The shaded blocks hold target ISA instructions and VMM tables. The concealed memory region is of fixed size and does not change after system initialization. Instructions either are always translated and fetched from the code cache or are VMM

code; consequently the instruction cache hierarchy holds only target ISA instructions, and their presence in the instruction cache is concealed from the guest software.

With respect to logical addressing of memory, it is simplest to allow the guest OS to manage the conventional (nonconcealed) part of real memory as it normally would and to map guest virtual addresses using the guest ISA's address translation architecture. For mapping and addressing of concealed memory, there are a number of options, some of which are illustrated in **Figure 7.6**.

In the first option, concealed logical memory shares an address space with the guest (**Figure 7.6**a). The conventional memory addressing and address translation are maintained as in the guest system, so the host address space must be enlarged, with the guest space fitting inside. Because the guest space is likely to be consistent with one of the "standard" sizes (e.g., 32 or 64 bits), the host space is either the next-larger "standard" size (e.g., 64 or 128 bits) or a nonstandard size (e.g., 33 or 65 bits). This approach is straightforward, but the expansion of the logical address space size may make the host ISA awkward or expensive to implement (which defeats the purpose of a codesigned VM).

A second option is to have two separate logical address spaces, one for concealed memory and one for conventional memory (**Figure 7.6**b). All the VMM software, code cache, associated tables, etc. are placed in the concealed address space, which maps to concealed real memory. This approach implies that load and store instructions must select one of the two mapping tables (e.g., by using different opcodes to select the mapping table or by using a mode bit). Note that all instruction fetches are to concealed memory addresses. Because the VMM controls all instructions that are executed (either through interpretation or because VMM-translated code is executing), the VMM can always ensure that the proper load and store instructions are used.

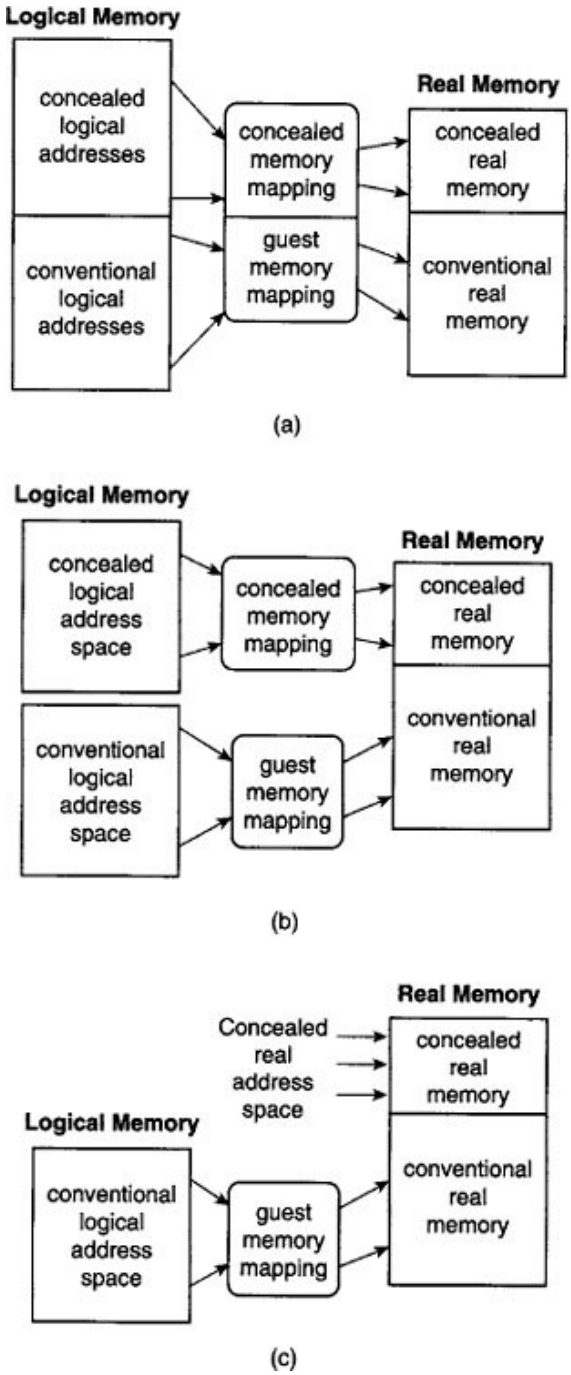


Figure 7.6 Methods for Mapping Concealed Memory. *(a) Extend the host address space; (b) provide a separate address map; (c) use real addresses.*

A third option is to use real addressing for concealed memory, i.e., to bypass address translation when accessing concealed memory (**Figure 7.6**c). In a sense, this is a special-case form of option 2. One way of implementing this is to provide separate sets of load/store in-

structions for accessing memory, one with real addresses and the other via the guest mapping table(s). Also note that the VMM has the option of bypassing address translation even when accessing conventional real memory. In effect, the VMM always operates within the real address space and uses logical addressing only when emulating the guest software. With this approach, a mode bit can then be used for selecting the type of memory addressing to be used, rather than using opcode-based selection.

A key issue regarding concealed memory is how far into the memory hierarchy the concealed memory reaches. It is easiest to keep the concealed portion of memory in RAM and not allow it to extend to secondary storage (disks). If the concealed memory extends to the disk, then the disk (and any bus addresses it uses) must also be concealed from the guest OS. Furthermore, the VMM becomes responsible for managing the concealed secondary memory, including any paging that it might support. Because VMM and the code cache can be kept relatively small (on the order of a few tens of megabytes) and because it simplifies the system design, most of the proposed and implemented codesigned VMs use the diskless option, with VMM software being stored in ROM. The extension of concealed memory to secondary storage is discussed in **Section 7.5**.

7.2 Self-Modifying and Self-Referencing Code

In a codesigned VM, self-modifying and self-referencing code are generally handled using the techniques described in **Section 3.4.2**. As pointed out in the previous section, it is easiest to keep the original guest OS's virtual-to-real page mapping intact. Then, because the source code is held in the guest's memory in its original form, any load or store accesses to an instruction page will naturally proceed correctly.

For self-modifying code, any attempt to write into the guest code region must be caught, and this is most easily done by write-protecting the source code region, so that any attempt to write it will cause a trap that can be directed to the VMM. The VMM can then flush the code cache of translations derived from the modified page and allow the page to be written.

In a typical process VM (as in **Chapter 3**), the VMM can write-protect a page via a system call to the host OS in order to change the page table protections. In a codesigned VM, however, the solution is a little more complicated because it is the guest OS that manages its own page tables, not the VMM. This includes the pages that contain the page tables themselves, so changing the page protections in a guest page table would violate the principle of keeping all of the guest memory state intact. The most straightforward solution is to use the TLB to enforce write-protection of guest code pages. If the TLB is managed by the VMM, as it normally would be, then the TLB entries are enhanced with a special "write-protect" bit, for use only by the VMM for detecting self-modifying code. Whenever an entry for a code page is loaded into the TLB, the VMM sets the write-protect bit. In order to keep track of the source code pages, the VMM maintains a table of all the guest virtual pages from which it has translated (or interpreted) code. This is done at the time code is translated or interpreted.

To reduce performance losses due to self-modifying code (both real and pseudo-self-modifying), the software-based methods described in **Section 3.4.2** can be applied. In addition, because of the codesigned paradigm, special hardware support can be provided to further reduce performance losses caused by pseudo-self-modifying code. In the Transmeta Crusoe, a special hardware structure is added to speedup fine-grained write-protection checking (**Dehnert et al. 2003**; **Banning et al. 2002**). The special hardware structure is managed by the VMM and is similar to a small software-managed

TLB. This *write-protect table* can hold the fine-grained write-protect masks (**Section 3.4.2**) for a few source code pages, along with hardware for comparing a potentially faulting store address with the bits in the bit mask (see **Figure 7.7**). After an address translation is performed with the normal TLB, the real address is presented to the write-protect table. If the normal TLB detects a write-protect fault, the smaller write-protect table can automatically filter out those cases where the write is not to a translated code region. As noted earlier, the TLB write-protect bit is a special bit used solely for protecting code pages. The VMM loads the write-protect table with the bit masks for a set of source code pages that are actively being written; this is usually a small number, if there are any at all.

A final issue involves I/O writes to guest code memory. If an I/O device writes to a guest code page, then these writes must also be caught. Again, the VMM helps enforce this by keeping track of all the real guest pages for which active translations exist in the code cache. To do this, the VMM has to maintain a hardware table for I/O writes, most easily held in the memory controller. The VMM makes entries in this table for all the real pages that hold guest code pages. A store to any of these pages results in an interrupt to the VMM, which can then flush the translations that are derived from the guest code page.

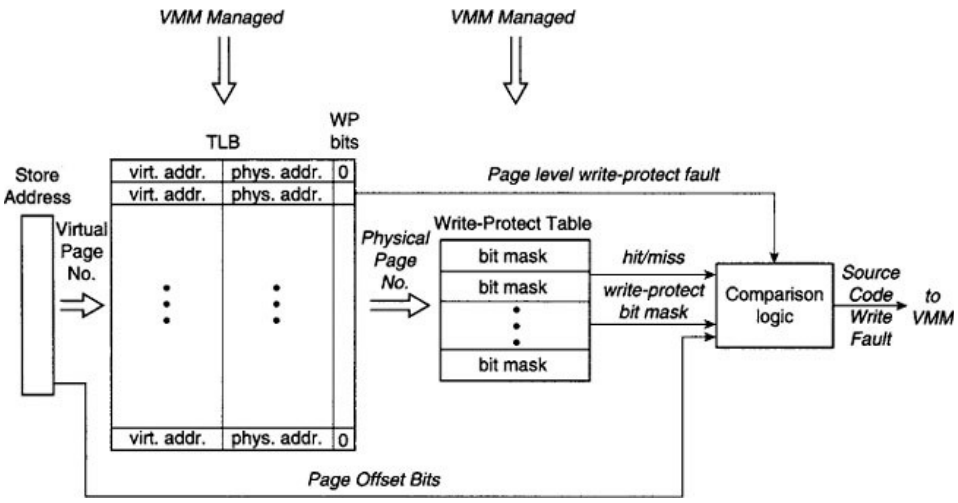


Figure 7.7 Fine-Grained Write-Protection Method for Source Code Regions. *A fault is*

Figure 7.7 Fine-Grained Write-Protection Method for Source Code Regions. *A fault is triggered to the VMM if the TLB indicates a write-protect fault and there is a miss in the write-protect table or the comparison logic indicates the write address is to a write-protected fine-grained code region.*

7.3 Support for Code Caching

A central element of a codesigned VM implementation, as with other emulating VMs, is the code cache. Because performance and efficiency are key factors in a codesigned VM, code cache performance is of utmost importance. It is useful to consider the issues that may lead to performance losses when using a code cache. Because the blocks of translated code held in a code cache are of unequal size, accessing the code cache becomes more complex than accessing a conventional cache, where blocks are of equal size. A code cache access begins by hashing the source PC (SPC) value to an entry in a map table, reading a corresponding SPC value (or tag) from the map table, and performing a comparison to determine if there is a hit (**Figure 2.28**). If so, the target PC (TPC) at the map table entry can be used for accessing the code cache. If the first probe of the map table mismatches, then additional probes of the table may be required in order to handle potential hash collisions.

The access delay for the code cache can be quite long, due to the multiple memory accesses and indirect jumps required. A single dispatch table lookup can take on the order of 10–15 instructions, leading to significant performance losses. In a simple code cache implementation, all control-transfer instructions (branches and jumps) perform a map table lookup. However, to improve performance for direct jumps and branches, superblock chaining (**Figure 2.29**) is an effective solution. With this method, the map table lookup for direct jumps and branches is eliminated entirely. For indirect control transfers the problem is more difficult, however, because register-indirect jumps have their destination addresses stored in a register, and destination addresses can change during the program's execution. Furthermore, the register address is the SPC value, not the TPC

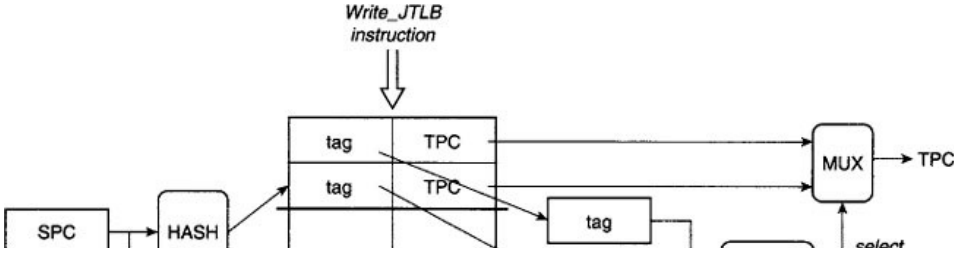
value. This means that the source jump address held in a register must be translated from SPC to TPC every time the translated indirect-jump instruction is executed. A straightforward approach is to consult the map table for every indirect jump; but, as noted earlier, this is slow and hence indirect jumps will add a significant performance cost.

To save table lookup overhead for each and every indirect jump, many dynamic optimizers/translators implement a form of software-based jump target prediction (see **Section 2.7.2**, illustrated again in **Figure 7.8**). Here, a sequence of instructions compares the indirect SPC address held in a register against an embedded translation-time SPC address. A match indicates a correct "prediction," and the inlined direct branch instruction to a TPC is executed; if there is no match, then the code jumps to the slow map table code.

The software prediction method is of somewhat limited value, however. First, if the software prediction is incorrect, then time is wasted by first testing the possibilities when the dispatch table lookup has to be performed anyway. Second, there are a number of indirect jumps that are very difficult to predict using this method, for example, returns for procedures that have a number of call sites and therefore a number of constantly changing destination addresses. The indirect-jump problem is probably the greatest source of performance loss in a software-only code cache system.

```
if ((Rx) == #addr_1) goto #target_1
else if ((Rx) == #addr_2) goto #target_2
else map_lookup (Rx)        ; do it the slow way
```

Figure 7.8 Example of Software Indirect Jump Prediction.



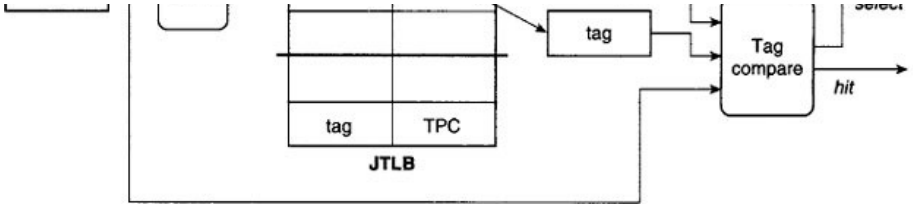


Figure 7.9 Jump Translation Lookaside Buffer (JTLB). *This example uses a two-way set associative table.*

7.3.1 Jump Translation Lookaside Buffers

In a codesigned VM, the high-overhead software map table can be replaced with a specially designed hardware cache of map table entries (**Gschwind 1998a**; **Kim and Smith 2003**). We refer to such a table as a *jump translation lookaside buffer* (JTLB), because it is similar in concept to the software-managed address TLBs used in virtual memory systems. A JTLB is illustrated in **Figure 7.9**. Each entry in the JTLB contains a tag and a TPC value. The JTLB can be fully associative, set associative, or direct mapped. This example is a two-way associative table. Entries are written into the JTLB by the VMM and always contain correct translations (this can be ensured, for example, by requiring that if a translated block is removed from the code cache, then the VMM must remove any associated entries from the JTLB). The JTLB is accessed by hashing the SPC value (this can consist of taking upper address bits as in a conventional TLB or cache, or it can be a more complex hash function). After the hash, the tag value is compared with the full or partial SPC, depending on the hash function used. If there is a JTLB hit, then the TPC value stored in the entry that hits is the desired jump address in the code cache.

A JTLB can be integrated into a codesigned ISA in a couple of ways. The first is a JTLB_Lookup instruction that accesses the JTLB with an SPC address held in a register and then reads out a TPC and places it in a second register. A third register (or condition code) indicates a hit or miss in the JTLB. An example is in **Figure 7.10**. Here, the JTLB_Lookup instruction takes an SPC value in Rk, puts the associated TPC value in Ri (if there is a hit), and places the hit/miss out-

come in Rj. This instruction is followed by a conditional jump on the value in Rj; if there is a miss, control transfers to the map_lookup routine. This routine will place the correct code cache mapping in the JTLB (if the code has been translated); otherwise, the VMM will begin interpretation or will translate the accessed code. A second, simpler way to integrate the JTLB into the ISA is to combine the lookup with the conditional jump, i.e., a Lookup_Jump Rk instruction that performs the jump to the TPC if there is a hit, otherwise it falls through.

```
JTLB_Lookup Ri, Rj, Rk      ; TPC to Ri, hit/miss to Rj
Jump Ri, Rj==0              ; conditional indirect jump
Jump map_lookup             ; do it the slow way
```

Figure 7.10 An Instruction Sequence That Accesses the JTLB.

This method still requires that when a jump is encountered, instruction fetching must stall until the JTLB is accessed and the jump is executed. An additional enhancement is to predict the TPC value immediately after a Lookup_Jump instruction is fetched, using a more or less conventional branch target buffer (BTB) (see **Figure 7.11**). Then the predicted instruction stream is accessed from the code cache immediately. Meanwhile, the Lookup_Jump proceeds up the pipeline (along with the predicted TPC). When the Lookup_Jump instruction issues and accesses the JTLB, the prediction is checked. A misprediction causes a pipeline flush, with fetching being redirected to the correct TPC. A miss in the JTLB causes a flush and a fall-through in the original code sequence (which would then jump to the map_lookup routine).

7.3.2 Dual-Address Return Address Stack

Either the software prediction approach or the JTLB approach, when supplemented with a BTB prediction, can be very effective, as long as the BTB prediction is correct a high percentage of the time. For many indirect jumps this will be the case, but for procedure return jumps it is often not the case. The problem is that a procedure

may be called from a number of places, so a return jump can have a number of different targets that frequently change.

To deal with this problem in a conventional microarchitecture, most modern processors employ a hardware return address stack (RAS) mechanism that can predict a return instruction's target address very accurately (**Kaeli and Emma 1991**). It basically mimics the software procedure stack by pushing the fall-through PC onto a hardware prediction stack whenever there is a procedure-call (jump) instruction. However, in a codesigned VM, this approach cannot be used in its conventional form because the saved return destination address would be an SPC, whereas the corresponding TPC is actually needed for a return destination address prediction. Furthermore, if the procedure jump is at the end of a translated superblock, as it often would be, then the address of the instruction following the jump is not the correct return address anyway (the correct address is at the beginning of a different superblock).

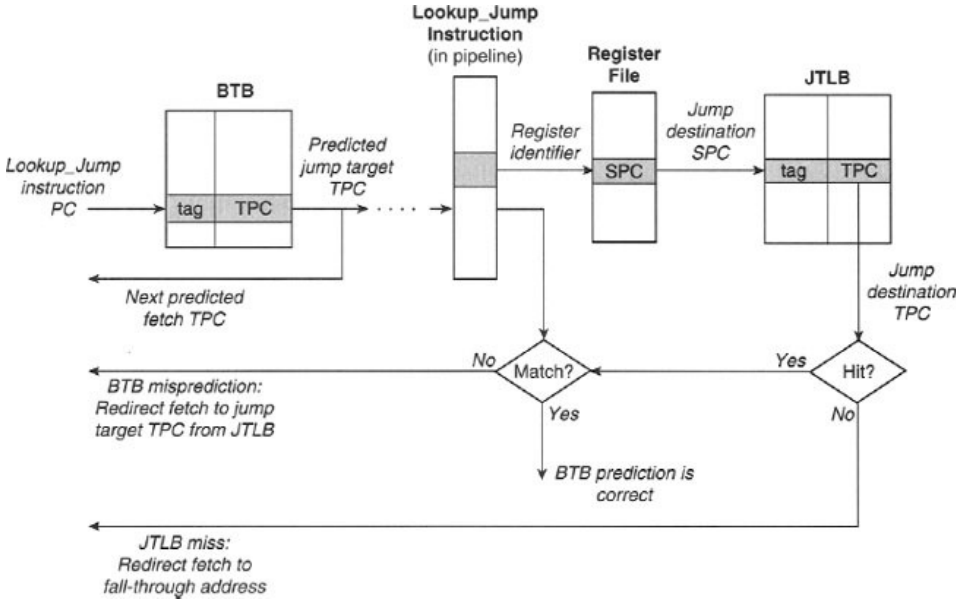


Figure 7.11 An Enhanced Lookup_Jump Instruction That Uses a BTB to Predict the TPC.

In a codesigned VM, a specialized dual-address RAS mechanism can be used for return address prediction (**Gschwind 1998b**; **Kim and Smith 2003**). Each entry in the dual-address RAS (DRAS) contains an

Smith 2003). Each entry in the dual-address RAS (DRAS) contains an address pair, consisting of a return address SPC and its corresponding TPC, as shown in Figure 7.12. The DRAS can be thought of as a hardware implementation of the shadow stack mechanism used in the FX!32 (see Section 2.7.3).

In order to push an address pair onto the DRAS, a special push-DRAS instruction pushes both return addresses. Finding the return destination SPC at superblock construction time is straightforward. Finding the TPC value is similar to setting up superblock linking. If the corresponding TPC is not known at superblock construction time, an invalid address is written in the TPC field of the push-DRAS instruction. Later when the return target superblock is constructed, the invalid address is replaced with a valid TPC. Alternatively, if a JTLB is present, it is accessed at the time of a procedure call, and the TPC value is pushed onto the prediction RAS. In either case, the prediction RAS should be pushed/popped for all procedure calls and returns—sometimes even with an invalid TPC to maintain correct stack ordering.

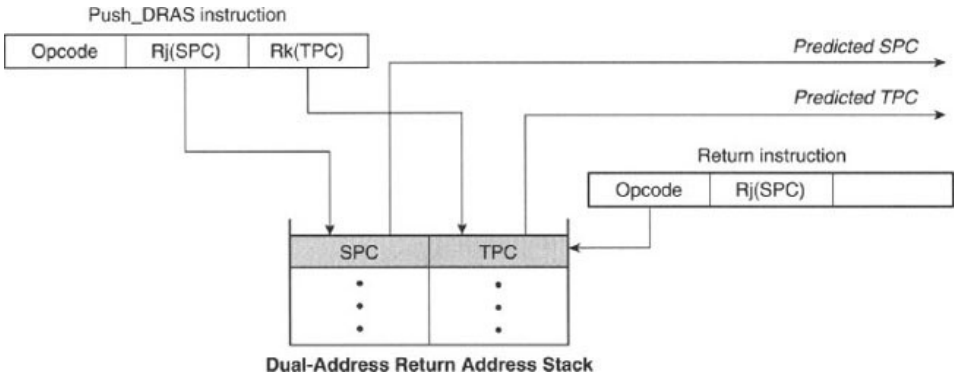


Figure 7.12 Dual-Address Return Address Stack. *The Push_DRAS instruction pushes both SPC and TPC. The return instruction uses TPC as a prediction.*

When a return instruction is fetched, the next fetch address is predicted with the popped TPC. The SPC part of the pair flows down the pipeline with the return instruction and is compared to the register value when the return instruction is issued. If the two values do not match, a RAS misprediction is detected, and fetch needs to be redi-

rected. Fetch redirection is accomplished by a jump to the dispatch table code. This is done by having the return jump instruction "fall-through" to the next instruction if there is a misprediction. The next instruction is a jump to the dispatch table code. Note that if a JTLB is also used, a simplification is to replace the *dual* RAS with a *single* RAS, where only the TPC is pushed. The JTLB can be relied upon to provide the correct TPC.

7.4 Implementing Precise Traps

Just as in other VMs, implementing precise traps is an important problem in codesigned VMs. In general, one can use techniques similar to those described in **Chapters 3** and **4**. That is, during the translation process, the optimizing software maintains software checkpoints. If code motion is involved, register live ranges are extended to ensure that checkpoint values can be restored when there is a trap. Then interpretation beginning at a checkpoint prior to a trapping instruction establishes the correct, precise state. In codesigned VMs this approach is facilitated because the host ISA can be designed to have enough registers so that live ranges can be extended without excessive register pressure. Nevertheless, this software-based approach does limit the types of code motion that can be performed (see **Section 4.5.1**). For example, many instructions cannot be moved below store instructions. By adding hardware support for checkpoints, this restriction on code motion can be removed, thereby relaxing the requirements for register live range extension.

7.4.1 Hardware Support for Checkpoints

In codesigned VMs, the designer has the option of providing hardware support for precise traps in translated code. The basis for the approach is to use hardware to set a checkpoint at the time each translation block is entered (**Figure 7.13**a). Then if there is a trap anywhere in the translation block, the state at the beginning of the