

Now, if instruction *d* happens to trap, the register state can be backed up to instruction *c* (as before), and the operands that produce the condition codes (registers r1 and r2 at the time the translation block is entered) can also be restored for condition code evaluation. Interpretation is then used for materializing the correct register state and condition code state at instruction *d*.

4.5.3 Superblocks Versus Traces

As mentioned earlier, most dynamic translation systems use superblocks rather than the more general traces. To understand why, we need to examine a number of issues. First, we consider instruction cache and branch prediction performance, and then we consider issues that come into play when dynamic superblock/trace formation and optimization are implemented.

An obvious consideration with respect to instruction caching is that a superblock may add replicated code due to tail duplication, which can increase the working set size and decrease instruction cache efficiency. Mitigating this effect is that only heavily used regions of code are converted to superblocks. Also, superblocks lead to more straight-line code fetches, which will enhance the cache-fetch efficiency. If code expansion due to tail duplication becomes excessive (and the runtime can keep track of this as it forms superblocks), then additional constraints can be put on superblock formation, such as restricting their length.

With respect to branch predictors, as with instruction caches, there is a working set size disadvantage; e.g., in **Figure 4.22**b block G is triplicated so the branch at its end will have three different entries in a branch predictor table. On the other hand, superblocks may also be beneficial because, in effect, a superblock encodes some global branch history information. For example, the three branches

corresponding to the end of G belong to three different paths. Hence, superblocks essentially embed path correlation information and perhaps a simpler branch predictor can be used, or a given predictor may work better.

As far as optimizations are concerned, both side exit and join points may inhibit interbasic block optimizations involving code motion because it may not always be possible to add compensation code (see **Table 4.1**). For example, code often cannot be moved below a join point. Getting rid of join points, therefore, removes some code reordering constraints.

Another advantage of removing join points becomes evident by considering the trace/superblock formation and optimization process. It is important to keep in mind that the basic blocks that are observed during trace/superblock formation are dynamic basic blocks, not static basic blocks. Consequently, determining whether a code region join point is already part of an existing trace adds complexity and bookkeeping to the formation algorithm. In particular, because only dynamic basic blocks are observed, it is not known *a priori* that a point in the middle of the block may later be discovered to be a join point. Hence, during the code-discovery process, checking to see if a branch target has already been translated requires not only looking at known branch targets, but also keeping track of, and searching, address ranges within already-translated basic blocks.

A related problem occurs because a preexisting translated trace has presumably been optimized, and this optimization may have been done without knowledge that a join point would later be discovered and added. Consequently, the optimization may no longer be safe after a join is discovered and added, so the optimization may have to be undone or at least revised, adding further complication.

It is perhaps worth noting that trace scheduling was originally implemented in a static compilation environment (**Fisher 1981**) where

plemented in a static compilation environment (Fisher 1981) where the complete control flow graph could be analyzed when forming traces and where there were fewer compatibility constraints that would inhibit optimizations—that is, only compatibility with the high-level language program had to be maintained, not compatibility with an existing binary.

From a different perspective, one thing that makes superblocks simple is that they require compensation code only at exit points, which is relatively straightforward. Furthermore, the only entry point is at the top, where the initial conditions (i.e., register state) are more easily maintained. In effect, the tail duplication required by superblocks is like adding superset compensation code at all join points and skipping all the analysis; i.e., the compensation code is "pre-added." Because it performs exactly the same state updates in the same order, it does not have to be segregated as special compensation code but remains main-line code.

4.6 Code Optimizations

There are a number of optimizations that can be applied within translation blocks to reduce execution time. Even if the original source binary code was optimized when it was produced, additional optimization opportunities are often present in the dynamic environment. For example, superblock formation removes control flow join points, creating a locally different control flow than in the original code. Partial procedure inlining converts what would originally be interprocedural analysis into intrablock analysis.

In general, many code optimizations have been proposed and used in practice (**Aho, Sethi, and Ullman 1986**; **Cooper and Torczon 2003**), and we will focus on ones that are more likely to be present during dynamic optimization. We begin by describing a set of common optimizations through examples. It is often the case that performing an optimization of one type enables additional optimiza

forming an optimization of one type enables additional optimizations. The examples are structured to illustrate this point.

4.6.1 Basic Optimizations

Figure 4.36 illustrates a combination of common optimizations. The first is *constant propagation.* Here, the constant value 6 is assigned to R1; then R1 is used in the following instruction, so the constant 6 is propagated into the next instruction (which would effectively become R5 ← 6 + 2). Then an optimization known as *constant folding* is performed. The constants 2 and 6 are combined into the constant 8, and the add instruction is replaced with the assignment of the constant 8 to register R5. Following constant propagation and folding, the multiplication in the third instruction is converted via *strength reduction* into a shift by 3.

The foregoing example may at first appear to be far-fetched. What compiler would have generated such nonoptimal code in the first place? Actually, this example illustrates a point made earlier. Consider **Figure 4.37**, which illustrates the original source code. It is the same as in **Figure 4.36**a, except we see that the assignment to R5 was at a control flow join point in the original code. Hence, in the original sequence, the compiler could not perform constant propagation. It is only because superblock formation removed the join point that the constant propagation (followed by other optimizations) was enabled.

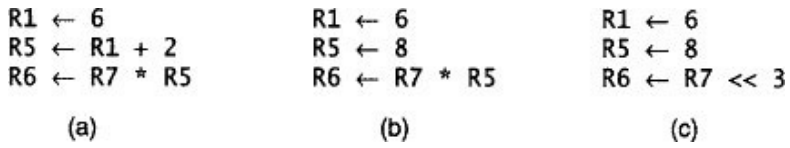


Figure 4.36 Constant Propagation and Folding (a) to (b) Followed by Strength Reduction (b) to (c).

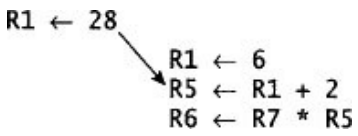


Figure 4.37 A Control Flow Join Point That Inhibits Code Optimization

Figure 4.37 A Control Flow Join Point That Inhibits Code Optimization.

Another example where superblock formation leads to a potential optimization is shown in **Figure 4.38**. Often the assignment of a value may be partially dead. That is, it is dead on one control flow path but not on others. This is illustrated in **Figure 4.38**a. Here, the assignment to R3 above the BNE is dead if the branch is not taken, but it is not dead if the branch (to L1) is taken. This suggests an optimization known as *code sinking*. If the first assignment to R3 is moved below the branch, with a compensating copy being placed at destination L1 (**Figure 4.38**b), then the partially dead assignment leads to a fully dead assignment on the not-taken path. Consequently, the assignment to R3 on the not-taken path can be removed (**Figure 4.38**c). This optimization is most beneficial if the not-taken path is the most common one. In terms of superblock optimization, the sequence of instructions on the not-taken path might be part of the superblock. Then code sinking is simply a reordering, with compensation code being added at a superblock exit.

Figure 4.39 illustrates another pair of common optimizations. In the original code sequence, there is a copy from register R1 into R4. *Copy propagation* causes register R1 to be substituted for R4 in the multiply instruction. This does not reduce the number of instructions, but in a superscalar processor it will increase parallelism because the copy and the multiplication can be performed in parallel after copy propagation. Finally, after copy propagation, the value held in register R4 is dead; i.e., it is not read by any following instructions before it is overwritten. As a second optimization, therefore, *dead-assignment elimination* removes the copy to R4.

```
R1 ← 1
R3 ← R3 + R2
Br L1 if R7!=0
R3 ← R7 + 1        L1: R3 ← R3 + 1
```

(a)

```
R1 ← 1
Br L1 if R7!=0
R3 ← R3 + R2
R3 ← R7 + 1        L1: R3 ← R3 + R
                       R3 ← R3 + 1
```

(b)

```
R1 ← 1
Br L1 if R7!=0
R3 ← R7 + 1     L1: R3 ← R3 + R2
                    R3 ← R3 + 1
```

(c)

Figure 4.38 Example of code sinking. *(a) The assignment to R3 is partially dead; (b) after code sinking, the assignment on the not-taken branch path is fully dead; (c) the dead assignment to R3 can then be removed.*

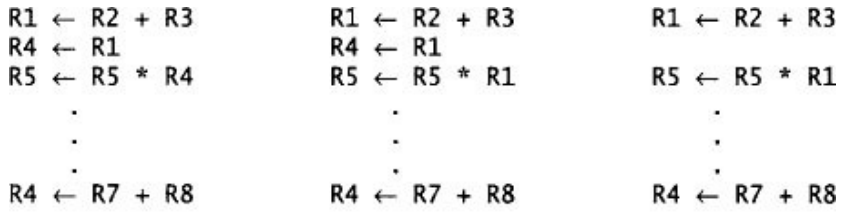


Figure 4.39 Copy Propagation (a) to (b) Followed by Dead-Assignment Elimination (b) to (c).

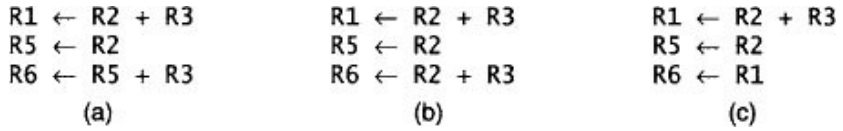


Figure 4.40 Copy Propagation (a) to (b) Followed by Common-Subexpression Elimination (b) to (c).

Another example of an optimization enabled by copy propagation is in **Figure 4.40**. Here, the copy from R2 to R5 is propagated to the second add. Then R2 + R3 becomes a common subexpression that can be eliminated from the second add. At this point, the copy to R5 may become a dead assignment and can be removed (not shown in the figure).

The final example optimization is the *hoisting of an invariant expression out of a loop,* **Figure 4.41**. Here, if neither R2 nor R3 is modified in the loop, then the add to register R1 is invariant throughout the loop's execution. Hence, it needs to be done only once, before the loop is entered.

```
L1:  R1 ← R2 + R3          R1 ← R2 + R3
     mem (R4) ← R1   L1:   mem (R4) ← R1
     R4 ← R4 + 4           R4 ← R4 + 4
          .                     .
          .                     .
          .                     .
     Br L1 if R7!=0        Br L1 if R7!=0
```



Figure 4.41 Hoisting a Loop Invariant Expression Out of a Loop. *(a) The computation of R1 done repeatedly inside the loop is (b) moved outside the loop and done only once.*

This completes our survey of common optimizations: redundant branch removal, constant propagation, copy propagation, constant folding, code sinking, strength reduction, dead-assignment elimination, and hoisting of invariant expressions. Although any of these optimizations, as well as others, can be applied, the different optimizations have different effects on compatibility, and this is often a critical factor in deciding whether they should be applied.

4.6.2 Compatibility Issues

An optimization is *safe* with respect to traps if, after the optimization is performed, every trap in the original source ISA code is detected in the translated target code and either traps to a handler in the runtime or branches directly to the runtime. Furthermore, the precise architected state at that point must be recoverable. There are no hard-and-fast rules for determining exactly which optimizations are safe, but optimizations that do not remove trapping operations tend typically to be safe. For example, copy-propagation, constant-propagation, and constant-folding optimizations are usually safe. There may be some end cases where compatibility becomes an issue, for example, if constant folding happens to result in overflow. This case can be determined at optimization time and can be disabled in those rare cases where it occurs.

On the other hand, a number of optimizations may not always be safe. Dead-assignment elimination, if it removes a potentially trapping instruction, is not safe because it may break trap compatibility —e.g., the original source code might trap if an instruction performing a dead assignment is executed. If the dead-assignment elimination removes a nontrapping instruction then it is likely to be safe. However, if such nontrapping dead-assignment code is removed,

then it may still be necessary to extend the live ranges of its input values, just in case some other instruction traps and the values are needed for precise state construction.

Similarly, hoisting instructions out of loops may not be safe because trap compatibility may be violated. For example, in **Figure 4.41** a potentially trapping instruction is hoisted. In a static sense the instruction has not been removed, but in a dynamic sense instructions have been removed—in the original code, the hoisted add was executed every time through the loop.

Strength reduction may be safe in some cases but not in others. For example, the reduction of a multiplication to a shift in **Figure 4.36** is likely to be unsafe, because an overflow during the multiplication will not occur when the substituted shift is performed. On the other hand, reducing a multiplication by an addition (e.g., replacing R1*2 with R1 + R1) would likely be safe—an overflow will be detected.

4.6.3 Inter-superblock Optimizations

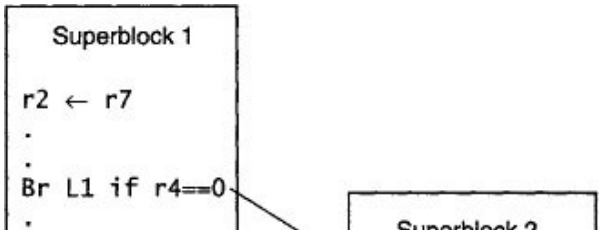
All the optimizations discussed thus far have a scope that is restricted to a single superblock. During the optimization process only a single superblock is buffered and analyzed. The register state at the time the superblock is entered and exited is precisely consistent with the register state at the same points in the source code. Even though superblocks are relatively large and provide good optimization opportunities, there may be room for additional optimizations that span superblock boundaries. One solution is to use tree groups, as described in **Section 4.3.5**. Another solution is to optimize across superblocks incrementally.

At the time two superblocks are linked together, they can both be reexamined and reoptimized, based on the new knowledge of control flow and a more complete context. In theory, this could be a complete reoptimization and reformulation of both superblocks that are

being linked. However, this would lead to additional optimization overhead and could affect the recovery of precise state (i.e., superblock exits may no longer be precise exception points with a consistent register mapping). Furthermore, the original superblocks are optimized to take into account the most common control flow path, and modifying some of these optimizations to account for early exit paths may be counterproductive. Consequently, it is probably a better approach to stick with the original superblock optimizations and optimize across superblocks only at the "seams."

An example is given in **Figure 4.42**. Here, register r2 is dead within the first superblock but may be live on exit path L1. If, however, the destination superblock of the early exit branch is examined when the superblocks are linked, it is apparent that the assignment to r2 is in fact dead within the original superblock and may be removed (subject to safeness requirements discussed earlier).

This optimization can be implemented via two side tables (**Bala, Duesterwald, and Banerjia 1999**). An *epilog* side table keeps a mask indicating the registers that are potentially dead at the time a superblock is exited, along with a pointer to the last instruction to assign the register a value (this is the potentially dead register). A *prolog* side table keeps a mask indicating the registers that are written before being read when a superblock is entered. **Figure 4.43** illustrates the tables for the example given earlier in **Figure 4.42**. When two superblocks are to be linked, the bit masks of the exited and entered superblocks are ANDed. Any bits that remain set indicate dead register values. The pointer in the epilog can then be used for finding them.



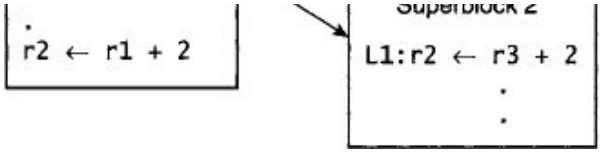


Figure 4.42 Two Superblocks to Be Linked. *After linking, the first assignment to r2 can be determined to be dead.*

Another possible optimization is to remove some of the register copies at exit points that are intended to maintain a consistent register state mapping. In the final code for the extended example given in **Section 4.5.2**, the copies of r5 to r3 (both at the bottom of the superblock and at the early exit) are present only to provide a consistent mapping of register ecx to r3. Here, if the mapping could be changed (so that ecx is mapped to r5) when the subsequent superblock is entered, then the copy instructions are no longer needed. To implement this optimization, there should be a side table for each superblock entry point, indicating the register mapping when it is entered (or indicating those mappings that differ from the standard consistent mapping). This table is also be consulted when there is a trap or interrupt to ensure that the correct precise state is restored.

4.6.4 Instruction-Set-Specific Optimizations

Each instruction set has its own features and quirks that could lead to special optimizations that are instruction-set specific. We now show two examples that illustrate the principles involved.

The first example is unaligned load optimization. Some ISAs, primarily RISC ISAs, are designed so that load and store instructions provide better performance if their addresses are aligned to “natural” boundaries; e.g., a word address has its two low-order bits as 00 (see **Section 2.8.5**). If a load or store instruction should access data that is not naturally aligned, i.e., is “unaligned,” then it will trap. The trap handler can then perform the unaligned access using multiple instructions. Because it relies on a trap, this method is extremely

slow, so an alternative is to use the normal aligned load or store for those accesses that have very high probability of being aligned and using an inlined multi-instruction sequence for those cases that have some significant probability of being unaligned. Of course, the static compiler for such an instruction set strives to align data as much as possible. The compiler also knows when data may be unaligned and inserts the multi-instruction sequences. However, when a source ISA that features unaligned accesses is emulated by a target ISA that does not, handling unaligned accesses leads to a dynamic optimization opportunity.

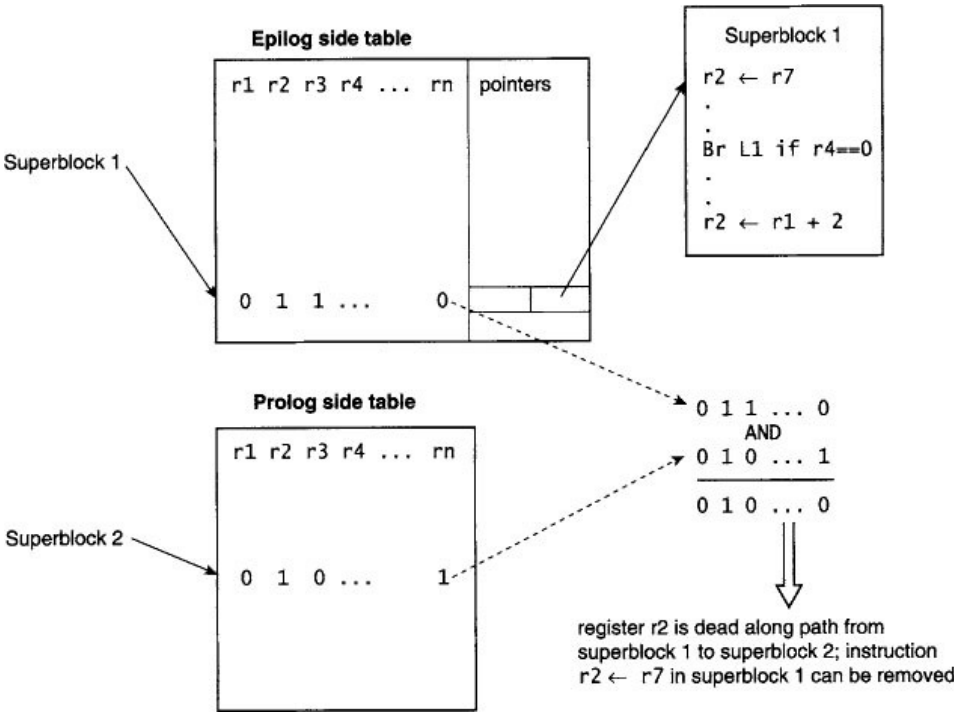


Figure 4.43 Epilog and Prolog Side Tables. *Epilog and Prolog side tables detect dead registers across superblock boundaries.*

If the source ISA allows unaligned accesses, then as part of dynamic profiling, the load and store instructions with unaligned addresses can be identified. That is, at the early emulation stage when profiling is enabled, all loads and stores with unaligned addresses are recorded in a table. Then the binary translator consults this table of profile data. If a load or store has never had an unaligned address, then the translator assumes this will always be the case, and the

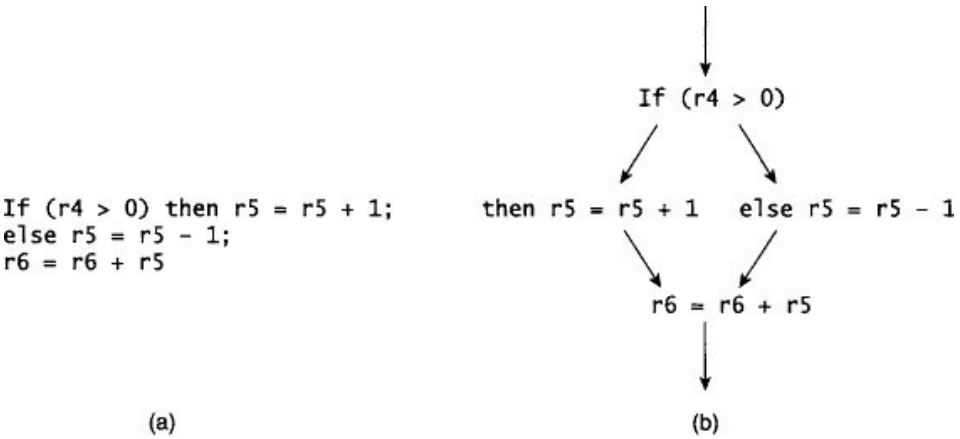
normal (aligned) target load/store instruction is used. If, on the other hand, unaligned accesses did occur for a given load or store, then the binary translator inserts the appropriate multi-instruction sequences. Of course, there must still be a trap handler to cover those cases where the profile information does not correctly identify a memory access instruction that later uses an unaligned address.

The second example of an instruction-set-specific optimization is known as *if conversion* (**Mahlke et al. 1992**). Some instruction sets implement special features to allow removal of certain conditional branch instructions. These special features include predication (**Hsu and Davidson 1986**), nullification (**Kane 1996**), and conditional move instructions. An example of code that might benefit from this optimization is shown in **Figure 4.44**a. In this example, there is a small if-then-else piece of code that leads to a small "hammock" region (**Figure 4.44**b). An assembly language version of this code is in **Figure 4.44**c. To execute the example code, a conditional branch and possibly an unconditional branch are required. Although branch prediction can often mitigate the performance-degrading effects of conditional branches, performance losses can be large when the prediction is incorrect.

For the purpose of this example we have "enhanced" the PowerPC instruction set by adding an integer conditional move instruction (the PowerPC instruction set has conditional move instructions for floating-point data but not integer data). This conditional move instruction (cmovgt in the example) moves the first source register into the destination register if the specified condition register indicates "greater than." With the conditional move instruction, the hammock region can be implemented as shown in **Figure 4.44**d. Here, both r5+1 and r5—1 are computed. Then the conditional move essentially selects the correct one. The result is a code sequence free of conditional branches (or any other branch, for that matter).

4.7 Same-ISA Optimization Systems: Special-Case Process Virtual Machines

Some process virtual machines perform dynamic optimization *without* performing binary translation. That is, these dynamic optimizers have the same source and target ISAs. Although not usually referred to as virtual machines, they do provide a process with a performance-enhanced execution environment, and they use many of the same techniques as in binary-translating process VMs. Unlike other process VMs, however, the objective of same-ISA dynamic binary optimizers is not compatibility. For example, the objective might be performance, as in the Mojo system (**Chen et al. 2000**), the Wiggins/Redstone system (**Deaver, Gorton, and Rubin 1999**) and the original HP Dynamo system (**Bala, Duesterwald, and Banerjia 2000**), or it might be increased security, as in the later DynamoRIO project (**Bruening, Garnett, and Amarasinghe 2003**). Although it does not provide the same level of transparency as the systems just mentioned, **Kistler and Franz (2001)** also developed an advanced dynamic optimization framework that employs many features of the process VMs discussed in this section.



```
        cmpi    cr0,r4,0        ;compare r4 with zero
        bgt     cr0, skip       ;branch to skip if r4 > 0
        addi    r5,r5,1         ;add 1 to r5
        b       next            ;branch to next
skip:   subi    r5,r5,1         ;sub 1 from r5
next:   add     r6,r6,r5        ;accumulate r5 values in r6
```

(c)

```
        cmpi    cr0,r4,0        ;compare r4 with zero
        addi    r30,r5,1        ;add 1 to r5
        subi    r5,r5,1         ;sub 1 from r5
        cmovgt  r5,r30,cr0      ;conditional move r30 to r5 if r4 > 0
next:   add     r6,r6,r5        ;accumulate r5 values in r6
```

(d)

Figure 4.44 Example of If-Conversion. *(a) Original code; (b) illustration of "hammock" formed by control flow; (c) assembly code with branches; (d) assembly code after if-conversion.*

Because of their similarities, many techniques proposed for translating process VMs are also used in same-ISA dynamic binary optimizers, albeit in a simpler form. On the other hand, because the source and target ISAs are exactly the same, these dynamic optimizers also have some special opportunities and advantages for overall performance improvement.

- It is easier to perform fast initial "emulation" of the nonoptimized source binary because the source binary can be copied directly into a basic block cache, with profiling code being added. Then as hot code sections are found, they can be placed in an optimized superblock cache. This approach is illustrated in **Figure 4.45**, and it is used in the Mojo system as well as the IA-32-based DynamoRIO. Note that the original HP Dynamo system uses interpretation during start-up, not basic block caching. An argument for using interpretation is that it avoids saving and restoring a large number of registers (as in the PA-RISC ISA) every time there is a switch between the cached "translated" code and the runtime. On the other hand, in the IA-32 ISA, there are only a few registers, so the method of using a basic block cache has more reasonable overhead. Carrying this concept still further, a technique known as *patching* (described in more detail in the next subsection) can be used to avoid interpretation and/or basic block caching of unoptimized code altogether. Patching, in effect, uses the original source code (not a cached copy) for initial emulation prior to optimization. Consequently, patching can be used only because the source and target ISAs are the same.

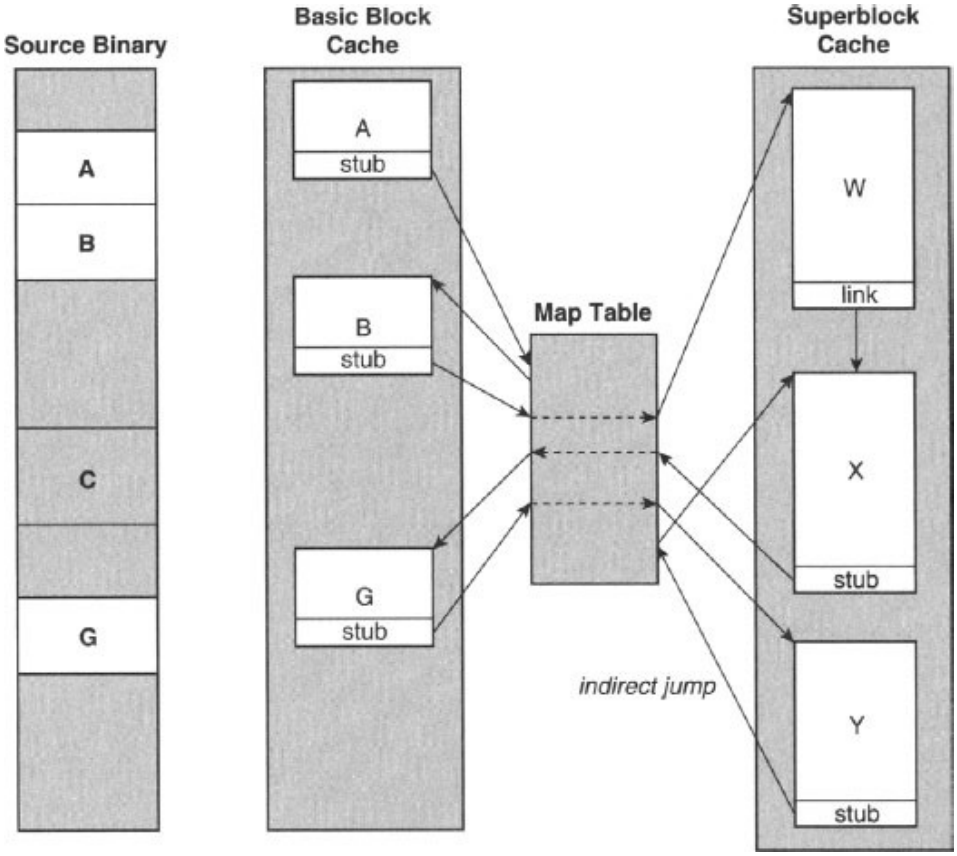


Figure 4.45 Dynamic Binary Optimization Using a Basic Block Cache. *Source basic blocks are copied into a code cache for initial emulation. Hot blocks are later collected into optimized superblocks. In this example, there is no chaining in the basic block cache because these blocks are executed infrequently or only during start-up. Also, in this example it is assumed that the basic block cache and superblock cache share a PC map table, but an implementation could also use separate tables.*

- When the same ISA is being used, dynamic optimization is not a necessity. In particular, if dynamic optimization is found to provide no performance benefit (or even a loss), then the dynamic optimizer can "bail out" and simply run the original code. For example, if the instruction working set is quite large and there is thrashing in the code cache due to repeated retranslations, then a bailout may be triggered.
- Sample-based profiling may be more attractive, because running the original code does not result in any apparent performance loss to the user. Hence, the original binary can be sampled over a relatively long period of time before optimization begins.
- For software-instrumented profiling, an interesting technique that has been used successfully is to duplicate the code to be profiled, with one copy being fully instrumented and the other (original) copy containing branches into the instrumented version (**Ronsse and De Bosschere 2000**; **Arnold and Ryder 2001**). When profile information is to be collected (this can be controlled via the branches in the original copy), the original code copy branches

controlled via the branches in the original copy), the original code copy branches into the instrumented version. After profile information is collected, the instrumented version branches back into the original copy.

- There are no instruction semantic mismatch problems. For example, both the source and the target have exactly the same condition code semantics and hence eliminate the overhead of emulating special cases in software.

Problems with same-ISA optimization can occur, however, because the source and target memory spaces and register files are of the same size. This places tight constraints on the emulation process, especially with respect to compatibility. Because the register files of the source and the target are the same size, there are no scratch registers for use by the optimizer, and this may limit the types of optimizations that can be performed. It also may lead to problems when the runtime saves and restores the guest's registers, because it may be necessary already to have a free register in order to load/store registers to memory (**Bala, Duesterwald, and Banerjia 2000**). Also, for intrinsic compatibility, there must be provision for large binaries that consume most, or all, of the address space, as discussed in **Section 3.3.2**—of course, if the address space requirements of the guest become too large, the runtime can bail out, release its address space, and revert to normal execution.

4.7.1 Code Patching

Code patching is a time-honored technique (**Gill 1951**) that is widely used in debuggers and other code profiling and analysis applications (**Larus and Schnarr 1995**; **Tamches and Miller 1999**; **Hunt and Brubacher 1999**). As noted earlier, code patching can also be applied to same-ISA dynamic binary optimizers. With patching, code is scanned when it is first encountered, but it is not copied into a code cache; rather it is left in place, with certain instructions being replaced with other instructions, i.e., "patches" (usually branch instructions that exit the source code). These patches typically pass control to an emulation manager and/or profiling code. The original code at the point of a patch is maintained in a *patch table* by the emulation manager so that patches can be later removed, if necessary,

For example, if a superblock in the code cache is flushed, then the patch redirecting the flow of control to the superblock should be removed.

Figure 4.46 illustrates the same example as given in **Figure 4.45**, with patching used instead of basic block caching. Note that in this example, map table lookups have been eliminated. Control can be passed to optimized superblocks via patches made at the time superblocks are formed. A particularly interesting situation is illustrated at the bottom of the figure, where there is an indirect jump exiting superblock Y. In conventional emulation systems, this would typically result in a map table lookup (unless software jump prediction is used and is successful—see **Section 2.7.2**). However, with patching, the indirect jump at the end of superblock Y can use the source PC to transfer control directly back into the source code. If the destination source code has already been optimized, a patch in the source binary immediately redirects control back into the superblock cache. The branch at this patch point would likely be a highly predictable branch, so the map table lookup is avoided, and the performance loss is negligible.

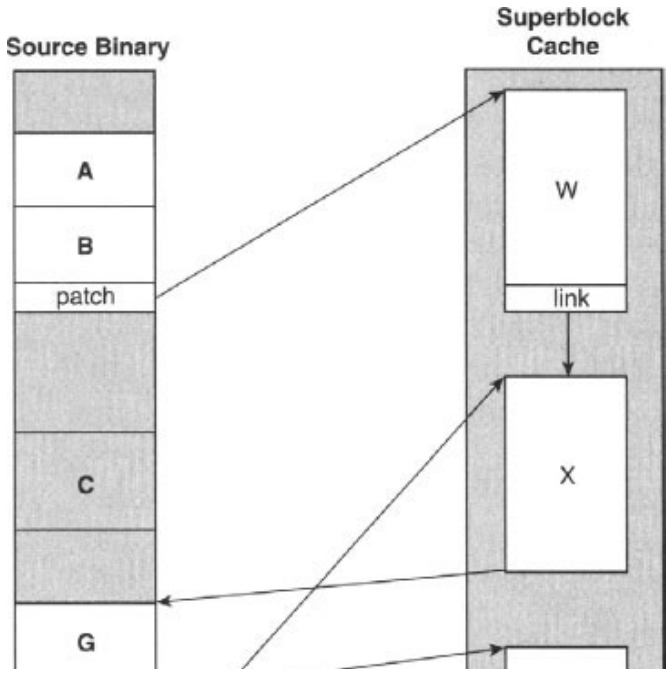




Figure 4.46 Example from **Figure 4.45** with Code Patching. *Map table lookups are no longer needed, because control can be transferred directly into the original source binary.*

Besides this performance advantage, another, more obvious advantage is that there is less replicated code; there doesn't have to be a source code copy in the basic block cache. Patching also has some disadvantages; the most significant is that it makes the handling of self-referencing code about as difficult as self-modifying code. With conventional code caching, the source binary is left intact, so self-referencing code automatically reads the correct data (see **Section 3.4.2**). With code patching, this will not be the case. Consequently, the source code has to be protected by making it execute-only. Then an attempt to read the source code as data will result in a trap, which can be handled by the runtime. The runtime can use its patch table to construct the correct data to be returned to the reading instruction.

The ADORE system (**Lu et al. 2004**) is an example of a system that supports dynamic binary optimization via code patching. The ADORE system targets the Intel IPF instruction set binaries and uses hardware-profiling information to support optimizations, including insertion of data-prefetch operations (**Lu et al. 2003**).

4.7.2 Case Study: HP Dynamo

The Dynamo system started as a research project at HP Labs (**Bala, Duesterwald, and Banerjia 1999**). Originally developed for a platform running UNIX and the HP PA8000 ISA, the Dynamo infrastructure has been retargeted toward the IA-32 ISA with both Windows and Linux operating systems (**Bruening, Duesterwald, and Amarasinghe 2001**). Subsequently the project was transferred to

the RIO research project at MIT as the DynamoRIO system. The DynamoRIO system has been extensively studied and is a valuable resource for both its innovations in process VM implementation and its well-documented performance features. Many of the techniques described in this chapter are Dynamo innovations or are at least used in the Dynamo system.

The basic design of Dynamo is very much along the lines of **Figure 4.4**. It starts out interpreting, and then after an execution threshold is reached it optimizes superblocks. As pointed out in the previous section, in the original HP PA8000 implementation, interpretation is chosen over basic block caching because of the large number of registers that would have to be saved and restored when a cached basic block is entered and exited (from the runtime). In the later DynamoRIO IA-32 implementation, basic block caching is used.

For superblock formation, Dynamo profiles backward branches and exit points from existing superblocks. When the threshold is reached (i.e., a hotspot is found), it uses a most-recently-executed heuristic (*most recently executed tail* in Dynamo terminology) to build superblocks. **Figure 4.47** shows data from the Dynamo Technical Report illustrating the relationship between the hotspot threshold and overall speedup for a set of benchmark programs (note that for programs where performance is lost, Dynamo "bails out," so little performance is actually lost). Overall, a threshold of 50 was deemed the best choice.

Dynamo uses many of the optimization techniques described earlier, but it tends not to reorder code (it is intended to be run on an out-of-order superscalar processor). The Dynamo study found that of the safe optimizations (i.e., those where trap compatibility can be maintained), strength reduction was the most commonly applicable, and redundant branch removal was second. Of the more aggressive, potentially unsafe optimizations, redundant loads and dead-assignment elimination were the most common, with code sinking

also providing a significant number of optimization opportunities. A load instruction is redundant if there is a preceding load (or store) instruction to the same address and the value previously loaded or stored remains in a register. The redundant load is then replaced with a register copy. Of course, if there is any possibility that the load or store is to a volatile memory location, then it cannot be considered redundant.

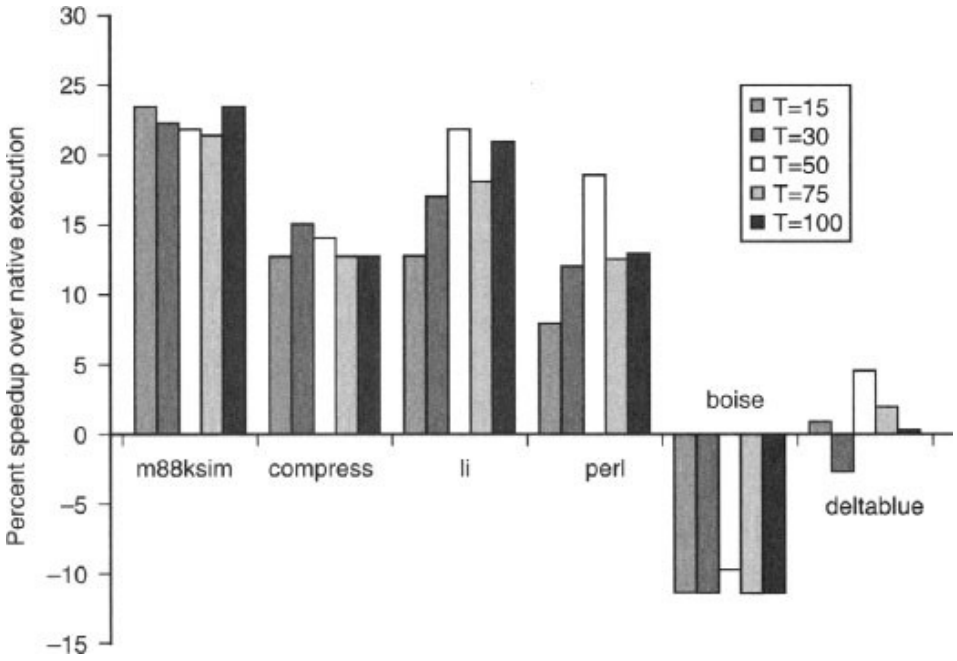


Figure 4.47 Relative Speedup as a Function of the Threshold Value for Superblock Formation. *The Y-axis gives percent speedup over native execution. The T values are the "hot" thresholds for initiating superblock formation.*

For code cache management, Dynamo uses the preemptive flush method. That is, it monitors the rate at which new superblocks are formed. If the rate begins to increase substantially, it assumes a new program phase is being entered and flushes the code cache at that point. Repeated flushing, however, indicates an instruction working set too large to fit in the code cache, so Dynamo bails out and executes the original code without optimizing it.

As noted earlier, Dynamo researchers were faced with the lack of scratch registers when performing same-ISA optimization. Because the source and target ISAs necessarily have the same number of reg-

isters, there is none left over for other operations. For example, software jump prediction, as in **Section 2.7.2**, requires an extra register for comparisons against an indirect jump's source register. Also, most operations that involve code motion result in extended register live ranges, and, consequently, the need to hold more register values at the same time. Finally, when there is a switch from the code cache to the runtime software, at least some registers must be saved so that the runtime can use them for its execution.

In conventional systems, the solution is to spill registers to a save area. However, in the HP ISA (and many other ISAs), a register pointer (to the register save area) is required in order to save the registers (in other words, a register must be freed in order to free registers); this leads to a quandary. The Dynamo solution is to perform a second register allocation pass after an optimized superblock has been formed. During this second pass, it looks for registers containing dead values within a superblock and uses one of them for the register save area pointer. If such a dead register cannot be found, then it temporarily spills a live register to memory in order to maintain the save area pointer. This general approach raises a couple of questions that are not directly addressed in the Dynamo report. The first is the matter of strict register state compatibility following a trap; i.e., reusing an apparently dead register destroys its contents, which might lead to register state incompatibility in a trap handler. The second is the issue of protecting the register save area (Section 3.4.3). A "rogue" load or store that happens to access the register save area could lead to incompatible, erroneous results. These problems indicate that it is difficult to achieve intrinsic compatibility in a dynamic optimizer. If there is some (external) assurance that applications being optimized are free of these problems (and many debugged applications would not exhibit these problems), then the dead-register approach would work well.

In order to integrate the Dynamo process VM into the host platform,

It is assumed that the kernel loader is used for loading the guest application. The kernel loader links the execution start-up code, crt0, into the guest application. When the loader is finished, it transfers execution to crt0 to initialize the process. In Dynamo, a custom version of crt0 is used. This new version of crt0 maps the Dynamo code as a linked library and calls Dynamo's entry point. Then the Dynamo runtime takes control of execution of the guest process.

Dynamo performance is shown in **Figure 4.48**. Dynamo is applied to source binaries that have been optimized originally at different levels by the HP compiler. The figure shows performance for three increasing levels of optimization (from +O2 to +O4). The O4 level is also enhanced with conventional code profiling that is fed back into the compilation process (**Figure 4.11**a). Performance is given as total run time (execution time), so a lower number is better. We see that Dynamo improves overall performance for all three optimization levels when applied alone, but it does not improve performance for +O4+profiling.

The reasons are illustrated in **Figure 4.49**, which breaks down improvements for Dynamo with O2 optimization. This figure shows that the biggest part of the gain occurs before any of the optimizations is performed. In other words, most of the performance gain comes from partial procedure inlining and superblock formation. This is the same gain that can be provided by feedback profiling (in +O4+profiling). That is, the profile information is used for relaying out the code to enhance locality. We also see from **Figure 4.49** that the "conservative" (safe) and "aggressive" optimizations provide similar benefits (although on different benchmarks). This indicates that a significant amount of the overall performance gain remains if the aggressive unsafe optimizations are not performed. Finally, note that the cases where performance is lost with Dynamo (e.g., benchmarks go, jpeg, vortex, and boise), there will be a bailout, so that performance losses no longer occur.

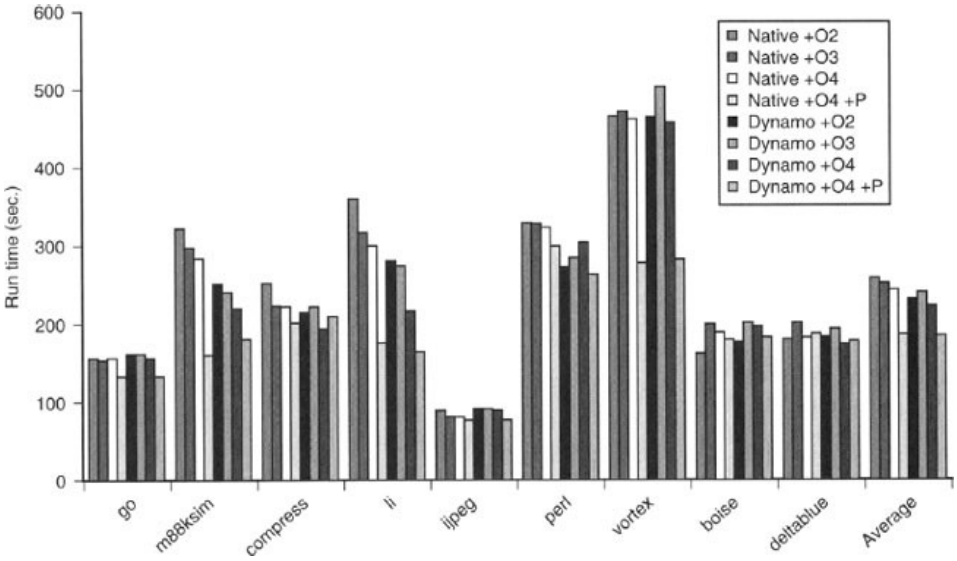
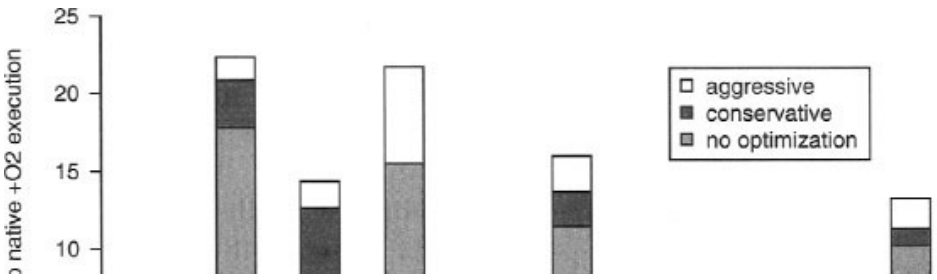


Figure 4.48 Dynamo Performance for Nine Benchmark Programs.

4.7.3 Discussion

When compared with traditional off-line feedback-directed optimization (see **Section 4.2.1**), as is done with +O4+profiling in **Figure 4.48**, it appears that dynamic binary optimization holds few or no performance advantages. To a large extent, off-line optimization uses similar profile-based information to perform similar optimizations, although one significant difference is that with dynamic optimization, the optimizations applied to a code region can change if program behavior changes over the course of its execution. Also, in practice many binaries have not received an intensive optimization effort; i.e., for various reasons, binaries are sometimes “shipped” with little optimization. In these cases dynamic binary optimization may provide performance advantages by allowing optimization to occur in the field.



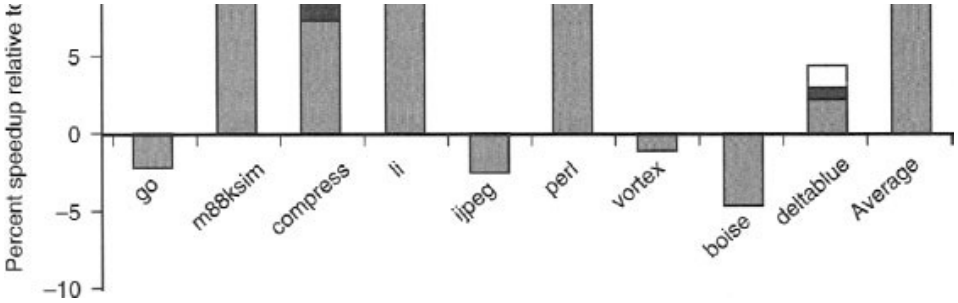


Figure 4.49 HP Dynamo Performance Improvements for Superblock Formation Only (No Optimization), Aggressive Optimization, and Conservative Optimization.

An additional matter to consider is that the underlying HP hardware platform used in the initial Dynamo study does not predict indirect jumps; rather, instruction fetching temporarily stalls when an indirect jump is encountered. Consequently, replacing indirect jumps with inline branches and partial procedure inlining, which eliminates some indirect jumps, results in significant performance gains on the HP hardware. When using hardware with indirect jump prediction (as in most current microprocessors), the performance advantages of dynamic optimization are significantly reduced. In the IA-32 version, DynamoRIO (**Bruening, Duesterwald, and Amarasinghe 2001**), indirect-jump handling is cited as a significant cause of overall performance losses. Much of the performance loss caused by indirect jumps results from the map table lookup that occurs in order to find the target PC value in the code cache. As we have pointed out, however, the code-patching method avoids these hash table lookups; in fact, this feature alone may be sufficient reason to favor code patching over complete code caching in dynamic binary optimizers.

4.8 Summary

Although optimization is not necessary for functionally correct VM implementations, it is important for many practical VM implementations. Because virtualization adds performance overhead, it is important to minimize performance losses, and optimization techniques can mitigate performance losses

niques can mitigate performance losses.

The most common structure for a VM implementation is to use staged emulation with interpretation or simple translation, followed by optimized binary translation. Profiling is typically enabled during the earlier stages, and sometimes for optimized code, in order to guide the optimization process. Overall, the largest performance gains usually come from the formation of translation blocks with good locality characteristics, e.g., inlined procedures and superblocks. After translated superblocks are linked together, performance can often approach native host execution.

The largest complication to the overall optimization process is the need to recover the correct state at the time of a trap or interrupt. How strict this requirement is depends on the type of VM being implemented and on the user's requirements (and applications). If very strict requirements must be met, e.g., if intrinsic compatibility is required, then in most VMs optimization is probably limited to code relayout and simple optimizations that do not remove instructions.

Another complicating factor is the relative sizes of the guest and host register and memory spaces. If the host has larger register files and memory space, then optimization is simplified and is often straightforward. On the other hand, if the host has no more registers and address space (or, worse yet, less), then optimization becomes more complex, and in some cases there can be significant slowdowns.

Finally, we often think of dynamic optimization as being applied to process VMs, either in conjunction with ISA translation or purely for optimization. However, these techniques are not restricted to process VMs. For example, optimization is a very important part of codesigned VMs (**Chapter 7**). Furthermore, as we shall see in **Section 8.2.3**, code caching and/or code patching are important

parts of some conventional system VM implementations.

---

1. Sometimes the term *trace* is also applied to superblocks; for example, the *traces* held in a trace cache (**Rotenberg, Bennett, and Smith 1996**) can be more precisely described as superblocks (all superblocks are traces, but not all traces are superblocks).

Previous chapter
Chapter 3: Process Virtual Machines

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

Chapter Five

High-Level Language Virtual Machine Architecture

A computer system implements a particular ISA and runs a specific OS only as a means to an end. It is the real work done by application programs that is important. Nevertheless, in a conventional computing environment, a compiled application is firmly tied to a particular OS and ISA. To move an application program to a platform with a different OS and/or ISA, it must be *ported,* which involves recompilation at the least but may also involve rewriting libraries and/or translation of OS calls. Furthermore, porting must be done on a per-program basis, and, after porting is done, any later upgrades or bug fixes must be applied to all the ported versions, followed by recompilation.

The process VMs described in **Chapter 3** are an approach for making applications portable in a more transparent and general way. In a process VM, a virtual execution environment is provided to a guest process. The process VM has to be developed once, and then, at least in theory, any application program that is developed for the VM's source ISA and OS will run without any additional effort. This is done by emulating the application's ISA and the OS calls. However, as discussed in **Chapter 3**, the OS interface is difficult to emulate (or may be impossible to emulate exactly if the host and guest operating systems are different). Furthermore, because conventional ISAs all have their quirks, writing a high-performance, accurate emulator to map from one conventional ISA to another is a painstaking task.

Thus, a conventional process virtual machine is a pragmatic "after-

the-fact" solution to the application portability problem. It is after the fact in the sense that neither the guest program nor the host platform were designed with the thought that the guest program would be run on the host platform. In contrast, by taking a step back, one can *design* a special guest-ISA/system interface, with VM-based portability to a variety of host platforms as a primary goal. Therefore, to a large extent, the new ISA is designed to be free of quirks and requirements for implementation-specific computational resources. Also, the system interface supported by the VM is raised to a higher level of abstraction than that found in a typical OS interface by consolidating common OS features and defining an abstract interface (a set of libraries) that can be supported by all conventional operating systems.

Just as important as portability, the guest ISA can be designed so that it reflects important features of a specific high-level language (HLL) or a class of HLLs (e.g., object-oriented languages). This leads to efficient implementations of the targeted HLL(s), and it simplifies the compilation process by separating machine-independent and machine-dependent portions of the compiler. Because efficient support for specific HLLs is one of the goals of these VMs, we refer to them as *high-level language virtual machines* (HLL VMs).

An HLL VM is similar to a conventional process VM, but the ISA is usually defined for user-mode programs only and is generally not designed for a real hardware processor—it will only be executed on a virtual processor.[1] Consequently, we refer to it as a *virtual ISA,* or V-ISA. The system interface is a set of standardized libraries (called "APIs") that can access files, perform network operations, and perform graphics operations, for example. The HLL VM is designed to be supported on a number of operating systems and hardware platforms (ideally, all of them—but at least the commonly used ones).

Here it is important to interject that a virtual instruction set architecture usually contains much more than just instructions. In a

modern V-ISA, the data aspects are at least as important as the instructions. Consequently, the specification for a modern V-ISA includes lengthy definitions of *metadata,* and it is the metadata that often dominates the overall specification. In contrast, the instruction definitions are quite simple, by design. In fact, one could reasonably argue that the term *data set architecture* would be more fitting than *instruction set architecture.* In any event, we will use the acronym V-ISA because it is the counterpart of the ISA in a conventional computer system.

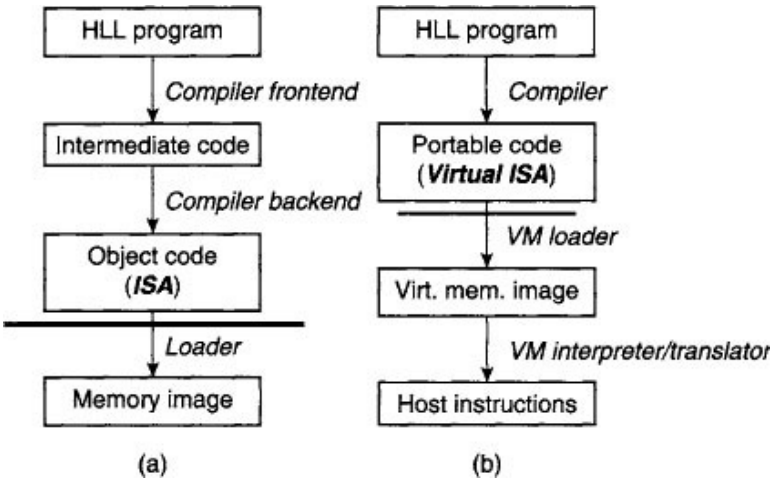


Figure 5.1 Relationship Between (a) a Conventional Compiler and Loader and (b) a Compiler and Loader in an HLL VM.

One can view HLL VMs from a languages/compiler perspective; see **Figure 5.1**. The conventional steps in getting from a high-level language program to a binary executing on hardware is shown in **Figure 5.1**a. The compiler frontend first parses the program and converts it to an intermediate form. Then a compiler backend takes the intermediate form, perhaps performs optimizations on it, and generates object code. The object code is in an ISA-like format, except for some symbolic information that must be resolved at load time. Finally, a loader converts the object code into a memory image that is executed by the underlying hardware processor. The program is generally distributed in the object code (“binary”) form, and, because this code is specific to an ISA and OS, it can be run only on

compatible platforms. With an HLL VM (**Figure 5.1**b), the compiler frontend parses and converts the program into a virtual ISA form, which is in some respects similar to a conventional intermediate form. Then the program is distributed in this form. At the time it is ready for execution on an HLL VM, a VM loader is invoked and converts the program into a form that is dependent on the virtual machine implementation. Additional optimizations may be performed as part of the emulation process. Emulation involves interpretation and/or binary translation from the guest V-ISA to the host ISA. The VM loader and interpreter/translator are major components of the HLL VM implementation.

Over the years, language designers have developed a number of HLL VMs targeted at specific HLLs or families of HLLs. An HLL VM that popularized the approach shown in **Figure 5.1**b was implemented for the Pascal programming language, using a V-ISA known as *P-code*. The best-known current example of an HLL VM was designed to support the Java programming language (**Gosling, Joy, and Steele 1996**), although other languages have been successfully compiled to it. Java programs are first compiled to Java *binary classes*,[2] which contain both a rich collection of metadata and low-level instructions encoded as "bytecode" sequences. The Java virtual machine (JVM) then loads the binary classes and executes them. A more recent HLL VM, the common language infrastructure (CLI), is part of the Microsoft .NET framework. The CLI is designed to be a target for a relatively broad class of commonly used HLLs, with object-oriented languages being of primary interest; however, non-object-oriented languages are supported as well.

Because of the importance of the Java and CLI VMs and because these two VMs have many similarities, the next two chapters will be structured somewhat differently than the other chapters. Rather than describing a variety of general features and then concluding with relatively brief case studies, we instead develop much of the discussion around Java and the JVM. The Java virtual machine is

widely used and has been the focus of a large amount of well-documented study. The Microsoft CLI is also discussed, primarily to highlight the differences with the JVM as well as the features that are more clearly delineated in the CLI than in the JVM.

This chapter concentrates on HLL VM architecture, i.e., the functional specification and a description of important properties. The next chapter concentrates on HLL VM implementation; i.e., the ways that one actually constructs an HLL VM, including techniques for enhancing performance of HLL implementations. In some cases, this division is rather arbitrary—it is difficult to discuss pure architecture without delving into some aspects of implementation. In these two chapters, we heavily emphasize HLL VM support for object-oriented languages. However, it is not practical for us to discuss object-oriented programming languages and concepts in any significant detail. We provide only a brief overview of the more important Java language features in this chapter. Consequently, for a good understanding of this material, it is essential that the reader have some familiarity with an object-oriented language such as Java or C#. There are a number of good books that can be consulted for learning or reviewing this material; for example, the Java and C# "in a nutshell" books (**Flanagan 1999**; **Drayton, Albahari, and Neward 2002**) contain fairly concise descriptions.

Before describing the important object-oriented HLL VMs, we first describe an important historical HLL VM—which will also provide an opportunity to illustrate important platform-independence properties of many HLL VMs. As we shall see, however, platform independence is only the tip of the iceberg when it comes to the features of modern component-based, network-oriented HLL VMs.

5.1 The Pascal P-Code Virtual Machine

The Pascal P-code virtual machine (**Nori et al. 1975**) was one of the keys to the widespread popularity of the Pascal high-level programming language because it greatly simplified the porting of a Pascal compiler. Because of its historical importance and its simplicity, a brief case study of the P-code virtual machine is a good way to introduce some of the important HLL VM concepts. **Figure 5.1** illustrates the basic idea. A compiler frontend parses a Pascal program and generates a P-code program. P-code is a simple, stack-oriented instruction set, summarized shortly. The combination of the Pascal compiler and the P-code virtual machine yields a complete Pascal compiler and execution system for the given host platform. Only one compiler has to be developed (distributed as P-code), and porting Pascal to a new host platform is reduced to implementing the virtual machine, which is much easier than writing a complete Pascal compiler from scratch.

There are actually two major parts of a P-code virtual machine; one is the P-code emulator (often an interpreter), and the other is a set of standard library routines that interface with the host operating system to implement I/O. For example, the library routines readln() and writeln() read a line of input and write a line of output. These routines can be called via P-code but may themselves be written in (or compiled to) the host platform's native assembly code. Writing these standard library routines in native code is part of the VM development process, along with writing an interpreter.

5.1.1 Memory Architecture

The P-code V-ISA uses a memory model that consists of a program memory area and a constant area, a stack, and a memory heap for data. A program counter (PC) fetches instructions from the program area; otherwise, program memory cannot be read or written. **Figure 5.2** illustrates the data areas of memory. All the data areas are divided into *cells*, and each cell can hold a single value. The ac-

are divided into cells, and each cell can hold a single value. The actual size of a cell is implementation dependent but clearly must be large enough to hold the largest value that can be specified in P-code. The compiler generates values held in the constant area; these are typically immediate operands used by the program. A single stack acts as both a procedure stack, e.g., for holding procedure linkage information, parameters, and local variables, and as an operand (or evaluation) stack for instruction execution. At any given time, the mark pointer (MP) points to the beginning of the stack frame for the currently active procedure, and the extreme pointer (EP) points to the maximum extent the current frame can reach (in a valid Pascal program, this extreme extent can be determined at compile time).

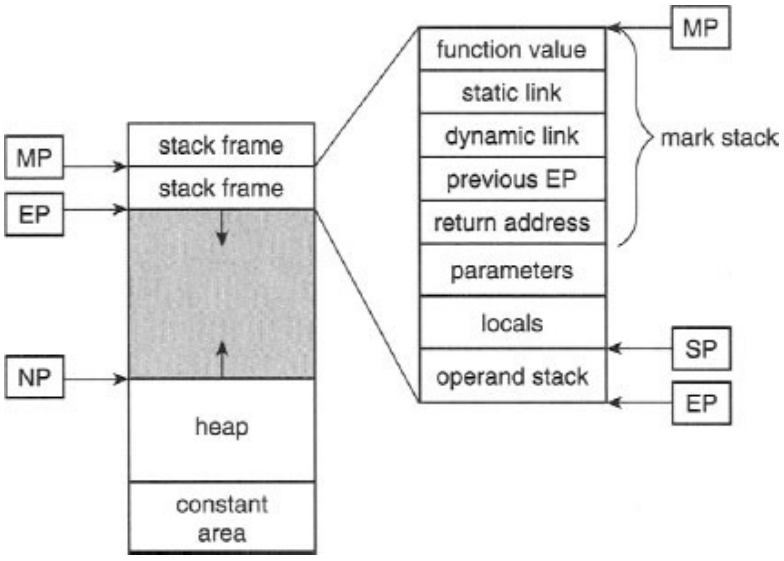


Figure 5.2 P-Code Memory Architecture.

The heap space sits on top of the constant area and is for dynamically allocated data structures. At any given time the new pointer (NP) delineates the maximum extent of the heap. In Pascal, the user code is responsible for releasing heap-allocated data structures when they are no longer needed, and the virtual machine software is responsible for overall management of the heap area. Conceptually, the stack grows down from the top and the heap (sitting on top of the constant area) grows up from the bottom. Before

the NP is adjusted upward, it is checked against the EP to make sure there is enough memory remaining; otherwise there is an exception.

When a procedure is called, it is given a new stack frame. The stack is divided into several sections as shown on the right side of **Figure 5.2**. The return function value, static link, dynamic link, previous EP, and return address are a fixed-size region at the beginning of the stack frame, collectively called the *mark stack*. The MP points to the base of the mark stack (as well as the base of the entire frame).

The *function value* returns a result value from the function (if there is any); the *static link* is used for linking together statically nested procedures and for accessing local variables within the scope of the active procedure. The Pascal language supports nested procedures, with the scope of local variables determined by the static procedure nesting. That is, the local variables declared within a given source code procedure can be accessed by any procedures nested within that procedure. The *dynamic link* is the MP value for the previous frame, and it allows the stack to be popped back to the previous frame. The *previous EP* is similarly used to establish the EP value when the active procedure returns. The *return address* is the PC value to which the active procedure should return.

Following the mark stack on a stack frame are parameters passed to the procedure and the local variable declared within the given procedure. Finally, the operand stack is used for holding operands and intermediate values when stack instructions are executed. The stack pointer, SP, points to the current top of the operand stack and moves up and down as computation proceeds (unlike the EP, which marks the farthest extent of SP).

5.1.2 Instruction Set

The basic instruction set includes instructions that push or pop

stack values from/to memory areas as well as arithmetic, logical, and shift instructions that operate on the stack. The instructions are typed; so, for example, there are both add integer (adi), and add real (adr) instructions. There are also Boolean, character, set, and pointer types. Most load and store instructions (that push and pop the stack) contain an opcode and an offset indicating the relative location of an operand. The arithmetic, logical, and shift instructions consist of only an opcode; they pop operands from the stack, perform the operation, and push the result back onto the stack. Following is an example of code that adds 1 to a local variable. The variable happens to be three cells above the mark stack. Because the variable is local to the procedure containing this code, it is at nesting level 0 with respect to this procedure.

```
lodi  0  3    // load variable from current frame (nest 0 depth),
              // offset 3 from top of mark stack.
ldci  1       // push constant 1
addi          // add
stri  0  3    // store variable back to location 3 of current frame
```

5.1.3 P-Code Summary

Overall the P-code V-ISA is quite simple, consisting of a small instruction set with elementary operations and a virtual machine implementation that is straightforward. In many respects, P-code set the standard for later HLL virtual machines, including the Java virtual machine; it shares the following features with the more recent HLL VMs.

- It uses a stack instruction set that requires a host platform to provide only a minimum amount of register support, making it easily translated to virtually any host ISA. Using a stack ISA for P-code also leads to small "binaries," an especially important feature when many desktop computers had small hard drives (or none at all) and supported only 64 KB of main memory.
- Its memory is divided into cells, whose implementation details, e.g., number of bits per cell, are hidden from the ISA.

- Main memory is partitioned into a stack and a heap, whose extents are not an architecture feature. Furthermore, instructions are never concerned with absolute memory addresses. This allows memory size to be an implementation feature of the host platform.
- The interface to the OS is through standard libraries. This, at least in theory, insulates programs from the underlying host OS. In the case of P-code, however, achieving OS independence tended to reduce its standard libraries to the least common denominator, which resulted in relatively weak I/O capabilities. Maintaining platform-independent, standard libraries remains a very important consideration/problem in modern HLL VMs. Not only is there the least-common-denominator problem, but there is also a temptation to add library "extensions," which compromise platform independence.

The major differences between the P-code virtual machine and modern HLL VMs are largely caused by the need to support a networked computing environment and the object-oriented programming paradigm. These differences are highlighted in the next section.

5.2 Object-Oriented High-Level Language Virtual Machines

The P-code virtual machine was developed to be part of a stand-alone environment where users could compile and run programs on their own local machines. The use of P-code greatly simplifies the porting of a Pascal compiler to a given host platform. After a P-code emulator is developed, programs can be compiled and run on the host platform, with the compiler being trusted to provide good code. This approach has proven to be an effective one, and a number of high-level language compilers have been developed with underlying virtual machine support.

In the modern network computing environment the situation is much more complex. In this environment, there are a number of interconnected platforms, incorporating a variety of processor architectures, operating systems, memory sizes, I/O devices, etc. In this

environment, it would be very difficult to compile, distribute, and maintain software for all the possible combinations (or even the more popular ones). The application of HLL VM technology can greatly simplify the task. A HLL VM is developed for each platform in the network, and any programs compiled to meet the specification of the VM can be sent across the network for execution on any of the various platform types. In this application environment, the HLL VM is not intended primarily for easing the porting of compilers, as was the case with P-code; rather, it enables the platform-independent distribution of application software.

The degree of platform independence in a modern HLL VM also goes beyond that provided by P-code, by encompassing data as well as instructions. P-code focused on the instruction set aspects of platform independence; the data representations are fairly basic, with layout of data structures implicitly encoded in the P-code programs themselves. In contrast, the popular HLL VMs use V-ISAs that refer to data in a more abstract manner (via metadata). Consequently, data structures and other resource-related information are encoded in a platform-independent fashion, just as the instructions are. This is illustrated in **Figure 5.3**. A program file consists of platform-independent code and platform-independent metadata. The metadata describes the data structures (typically objects), their attributes, and their relationships. As shown in the figure, the VM software consists of an emulation engine that can either interpret the code or translate it into native code. The VM infrastructure also consists of a loader that can convert the metadata into internal machine-dependent data structures, taking the host platform's word size and addressing features into account.

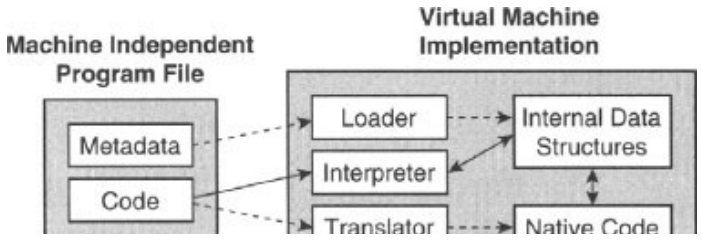




Figure 5.3 Transformation of a Machine-Independent Program File to Machine-Dependent Code and Data. *Dotted lines indicate transformation of code and data; solid lines indicate movement of code and data during emulation.*

Currently, there are two major HLL VMs being used and promoted for platform-independent network computing—the Sun Microsystems Java virtual machine (JVM) and the Microsoft common language infrastructure (CLI). Both JVM and CLI include the same basic techniques for HLL VM implementation, some of which are similar to P-code and to the process VMs of **Chapter 3**. The modern HLL VMs also include a number of other interesting and rather sophisticated features. Besides support for platform-independent software, key features of network-oriented HLL VMs include the following.

Security and Protection—It must be possible to load programs from untrusted sources (including one of the least trustworthy of sources, the Internet) yet execute them locally as part of a user process without compromising the security of the local system. A user's files and other local hardware resources must be secure from any program downloaded over a network. Also, as an element of a secure implementation, the VM implementation software must be protected from the application software, yet both application and VM software typically are part of the same process running on a host platform. The metaphor is that each guest program is executed within it own *sandbox,* i.e., an environment where it is confined to operate. The program can do what it wants within the sandbox but cannot disturb any resources outside the sandbox, unless given explicit permission. In .NET terminology, a sandboxed guest program is referred to as *managed code* because its execution is performed under the management of the VM runtime. This management includes not only security checking but other functions, such as automatic garbage collection. Because it is helpful and descriptive, we will use the term *managed* code, to distinguish it from *unmanaged* code that runs outside the sandbox. As will become apparent, the ability to

load an untrusted application and run it in a managed, secure fashion is a much bigger challenge than mere platform independence.

Robustness—The advantages of robust software are not restricted to the network computing environment, of course, but robust software becomes critical when dealing with the complexities of a platform-independent network environment. For developing large-scale software systems, the object-oriented programming model has a number of advantages and is widely used. Furthermore, the object model fits very well with distributed computing and dynamic linking. It also fits well with component-based programming, which can enhance programmer productivity considerably. Consequently, the object-oriented paradigm has become the model of choice for modern HLL VMs. Both Java and the CLI are designed to support object-oriented software. Other features that contribute significantly to robustness are VM support for strong type-checking and garbage collection.

Networking—In some environments, available networking hardware may only provide limited bandwidth. This limitation favors software that uses network bandwidth sparingly and efficiently. Consequently, application software should be loaded incrementally, on demand, via dynamic linking. This can save bandwidth by loading only software that is used, by spreading bandwidth usage over time, and by improving program start-up time because execution can begin as soon as the first software routines are loaded. There are also advantages to using dense instruction set encodings, both to reduce the bandwidth required for moving a program across a network and to reduce memory requirements on the local platform (which may also be limited in some applications).

Performance—While offering all the features just listed, it is also desirable to provide good performance. In general, one might be willing to sacrifice some performance—nothing comes for free after all

—but a good HLL VM should be part of an overall framework that provides the user with good performance. To achieve this, many of the techniques described in **Chapters 2** through **4** for conventional process VMs can be applied. In addition, there are techniques that are specific to (or more advantageous when applied in) the object-oriented, networked environment.

When all of the foregoing features are wound together into a single HLL VM, it is difficult to unravel the specific VM properties and associate them with specific features. The HLL V-ISAs and supporting VM implementations have become quite complex; many papers and thick books have been written on both the JVM and CLI architectures and implementations. We cannot begin to provide this level of detail; rather, our goal is to understand and illustrate the important principles and techniques.

In the following subsections, we discuss the basic techniques for supporting the four primary features just listed. In subsequent sections, we describe the Java virtual machine in some depth because it is conceptually simpler than the more recent Microsoft CLI. The JVM discussion will be followed by a discussion of the CLI, with its differences with JVM being highlighted.

Before proceeding, we pause briefly to discuss architecture/implementation terms again. First, there is some ambiguity when discussing Java virtual machine architecture and implementation; there is no widely used distinction between the two. That is, the term *Java virtual machine* may be used for the functional specification, e.g., the class file format, V-ISA, etc., and the same term is often applied to a specific implementation. Usually, the difference can be determined from the context. When there is any possibility of confusion, we will add *architecture* or *implementation* to *Java virtual machine.* The Microsoft common language infrastructure is an architecture, or functional specification. The Microsoft common language runtime (CLR) is a specific implementation of the CLI developed by

Microsoft. The Microsoft *shared-source CLI,* despite the name, is another example of a CLI implementation (*shared-source CLR* would be more appropriate). Sometimes, it is useful to discuss only the instruction set portion of a VM architecture. In the case of Java, it is generally referred to as *Java bytecodes* because, as we shall see, the instructions can be viewed as a stream of bytes. The instruction part of the CLI is known variously as the *Microsoft intermediate language* (MSIL), the *common intermediate language* (CIL), or simply the IL. The combination of a JVM implementation and a set of standard Java libraries (APIs) is called a *Java platform.* Finally, a CLI implementation, along with a set of standard libraries, implements the Microsoft .NET framework.

5.2.1 Security and Protection

A given application running on a local platform should have access to certain specified files (program and data) on both its local system and on remote systems. Meanwhile, there must be the ability to prevent the guest application from accessing other files on both the remote and local systems. In addition, the VM implementation software that is running the application should be protected from the application itself, even though both are running on the host platform as part of the same process. Therefore, the type of protection that must be provided is different from that supported by a conventional operating system, where users are protected from one another, but a given user's programs have access to all of that user's resources. In the HLL environment, untrusted programs are invited into the user's domain but must have strictly limited access within that domain. That is, a guest program is treated as managed code that can only operate within its protection sandbox. In modern HLL VMs such as Java, the VM implementation provides much of this additional protection "above" the operating system. As we shall see, a fundamental aspect of the popular HLL VMs is that the high-level languages (e.g., Java or C#) are strongly typed and legal programs must strictly define the scope of both data accesses (to objects) and

must strictly define the scope of both data accesses (to objects) and control flow. This information is then conveyed to the VM through the V-ISA.

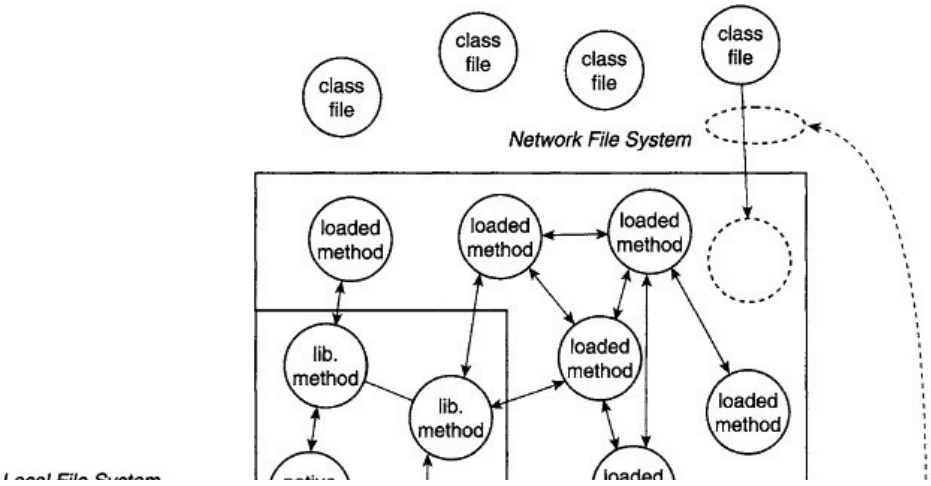
There are three main aspects to the overall security/protection sandbox. The first is gaining access to public data, e.g., stored in files that are on remote systems, while being prevented from accessing any other data on the remote system. Here, we include locating public files as part of the overall process. The second aspect is that similar properties must hold for local files. And the third aspect is that the managed application must be prevented from accessing memory or executing code that is outside the sandbox, even though it shares the process memory space with the VM runtime software. To simplify explanation of the basic principles, the following discussion is based on the original Java sandbox model. The original sandbox is relatively inflexible and was primarily targeted at *applets*—small application programs accessed via the Internet; it remains the default security model. The more recent security model (used in Java 2) provides greater flexibility, including finer-grained access control and significantly less reliance on trusted code. This model will receive more discussion in **Section 6.2**.

Remote files are found via agreed-upon naming conventions. A specific convention is not part of a standard *per se*, but most current operating systems, including Linux and Windows, use hierarchical file systems based on directories. Consequently, naming conventions typically rely on this hierarchical structure. Furthermore, the Internet connects these systems and also uses a hierarchical naming system. Hence, as part of an overall framework—whether a Java platform or the .NET framework—there is normally a naming convention based on a combination of Internet addresses and local file structure addresses. For example, in the Java platform a particular method might be denoted as

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

which is formed as a concatenation of an Internet address (in reverse order)—**ece.wisc.edu**—and a pathname—~jes/testpackage/shape/area. The critical thing is that a local system, given a name that follows the convention, must be able to find the stored representation of the requested item.

Protection of resources within the remote system's domain is not really a responsibility of the HLL VM running on the local system, even though data on the remote system may be part of an overall network computing framework. It is up to the user(s) of the remote system to determine which data files and/or directories should be made accessible over the network and which permissions (read or write) should be granted. Hence, even if the local system gives a correct path, if the file is not publicly accessible, then an attempt to access it will be denied permission. As noted earlier, this simple protection method is only the Java default. By using security, authentication, and cryptography APIs, a much more flexible security infrastructure can be built and supported by HLL VM platforms; for example, the Java 2 Standard Edition platform contains a number of APIs to support network-based security (**Gong, Ellison, and Dageforde 2003**). However, such VM platform security involves a number of higher-level issues that are well beyond our scope, although there is some additional discussion in the next chapter.



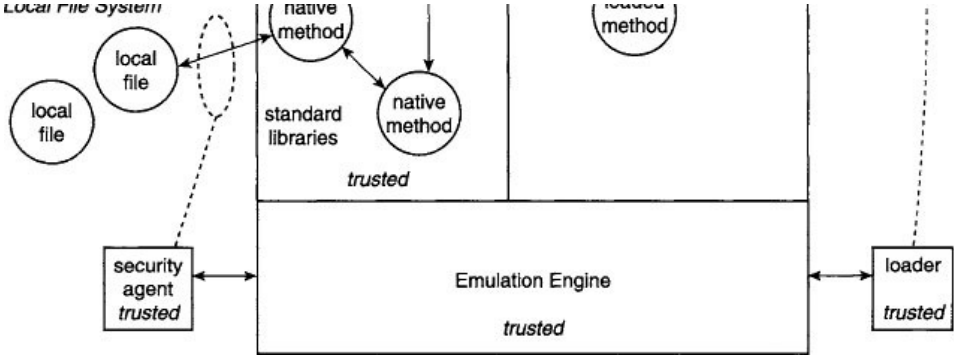


Figure 5.4 Components of the Java Protection Sandbox.

As shown in **Figure 5.4**, a Java program consists of a number of binary classes (or *modules* in .NET terminology), in a standard format, consisting of both platform-independent code and metadata. These binary classes may be held in a local file system, or they may be accessible via a network. The VM implementation includes a loader (or set of loaders) able to take a binary class, verify that it is correct, and then translate it into an implementation-dependent format that is loaded into the VM's memory region. The verification process will be described in more detail later. A loader is *trusted* software; that is, the local user trusts it to work correctly and to be secure. The trusted loader may be supplied by the user or, more likely, by a software developer trusted by the user. The binary classes define the loaded methods and object types, which are operated on by the methods.

There are both user methods and system methods, some of which may be native methods. The application program contains user methods. System methods belong to standard libraries (APIs) that are part of the local VM implementation and are often handled as trusted software.[3]

Native methods are maintained as native binary code, as the name suggests. Native methods can be compiled from any of a number of high-level languages, including C, or assembly language. They are useful for interfacing with legacy code and features of the underlying host platform, such as OS calls. They can also be used by pro-

grams written in other languages to draw on the rich set of APIs implemented in Java. The HLL methods and native methods can interact by following established conventions at their respective interfaces; the Java/native interface (JNI) defines a convention for Java. Native methods are often incorporated as parts of trusted libraries, but a user may also write and install native methods (at his or her own risk because they are not subject to the same security checks as conventional methods).

The emulation engine, another trusted component, emulates the V-ISA code embodied in methods using techniques similar to those described in **Chapters 2** through **4**. That is, it may use interpretation or may execute code that has been translated (compiled) to native instructions. If a managed application program needs to access a local file or any other local resource, it can only do so through a security manager within which the user has either directly or indirectly provided permissions (**Section 6.2**). Whenever the managed application requests access to a local file, for example, the request is made through a method belonging to a standard library. The standard library method consults the security manager to see if permissions should be granted. If so, the file access proceeds.

The managed application code includes software in the form of actual binary classes and methods they contain. This untrusted application software and the VM software share the same address space and run at the same privilege level within the host system; consequently, the VM software must be protected from the application as it executes. For example, arbitrary access to the VM tables via application program load and store instructions must be prohibited. Similarly, the application code should not be allowed to jump to arbitrary locations in the VM code.

A combination of static and run-time checks implement address space protection. As the emulation engine executes a managed guest application, it can perform a memory bounds check on every load

application, it can perform a memory bounds check on every load, store, or jump instruction that static (load-time) code analysis cannot guarantee to be safe. These checks ensure that the guest only accesses its portion of the address space; i.e., it stays within the sandbox. Today's more popular HLL VMs, including both the JVM and CLI, are targeted at strongly typed HLLs and rely to a high degree on static (load-time) checking for protection, with minimal run-time checking. Because the strongly typed HLL approach is widely used, it is the one we focus on. In contrast, with conventional binaries, as would be generated by C or C++, static binary code analysis is rather limited, and many loads, stores, and jumps would require dynamic run-time checking. An especially efficient implementation of this full dynamic-checking approach simply forces all accesses to be within predefined, easily checked guest memory segments (**Wahbe et al. 1993**). This technique allows virtually any HLL to be used for generating guest programs. Such an approach is taken in the Omniware system (**Lucco, Sharp, and Wahbe 1995**).

We provide more detail in the next chapter when discussing Java virtual machine implementation, but the basic approach to checking is first to restrict the virtual ISA so that all the data structures are defined as part of the metadata and all memory accesses are to fixed fields within typed structures. The metadata, of course, reflects the object structure defined in the high-level language program. Similarly, all branch instructions are to fixed offsets within the code region. The only indirection allowed for control transfers is through explicit call and return instructions. As part of the loading process, the load and store instructions in the machine-independent code are checked for consistency with respect to the metadata specifications. The loader also checks all control flow (branch and jump) instructions for consistency. These load-time static checks provide most of the protection against out-of-bounds loads, stores, and control transfers. Exceptions are for those accesses that use a computed index value into a data structure, e.g., arrays. These accesses are checked dynamically at run time, along with checks for accesses us-

ing null pointers (actually *references,* in Java terminology).

In summary, we see that the sandbox around a managed execution environment consists of a number of interacting components. The files and other resources on a remote system are protected by the remote system itself, based on its conventional protection mechanisms. The files and other resources on the local machine are protected by trusted libraries and a security manager. Finally, the VM runtime software is protected from untrusted application software by a combination of static checking performed by a trusted loader and dynamic checks performed by a trusted emulation engine.

5.2.2 Robustness—Object-Oriented Programming

As noted earlier, the two most widely used HLL VMs are both based on an object-oriented model. We can't possibly cover all the aspects of object-oriented programming here. Rather, we provide a quick overview to introduce terminology and to make the following discussion of HLL VM implementations easier to understand.

In an object-oriented programming environment, the data-carrying entities are *objects.* The data contained in each object can only be accessed and manipulated via *methods,* or procedures, that are defined for that type of object. A *class* defines a type of object and its associated methods. At run time, objects may be created as *instances* of the class. In essence, the objects are a complex programmer-defined data type, and the methods are specific to the particular type for which they are defined. Objects can only be accessed via managed pointers, or *references,* that are generated at the time the object is created. In Java and CLI, *arrays* are defined as part of the language and are essentially an intrinsic form of object.

The data portion of an object consists of a number of fields. These fields may be individual data elements, or they may be references to

fields may be individual data elements, or they may be references to other objects and arrays. The data portion of an object can be either *static* or *dynamic.* If it is static, then only one copy of the associated data is created, and it is shared by all objects of the given type. If data is dynamic, then a new copy of the data is generated for each object that is created.

Through *inheritance,* new classes can be defined by extending an existing class. A derived class has the same members (e.g., fields) as the base class from which it is inheriting. When a class is *extended,* the function of an existing method for the base class can be overridden, by defining a new method having the same name and *signature.* A method's signature is an ordered list of its argument and return types. The new, overriding method is the one applied to the subclass. Furthermore, a subclass can have other members and methods that do not exist in the base class.

Inheritance, as just described, is a way of achieving *polymorphism* among subclasses. With inheritance and polymorphism, the exact code executed when a method is called depends on the specific type of the object given as an argument. Finally, an *interface* is similar to a class but has no associated object; it only contains method signatures. However, if a class (which does have an associated object) implements an interface, then the methods identified by the interface's signatures operate on the class's object. A subclass can only extend one class, but it can implement multiple interfaces.

Invoking a method that operates on an object is often likened to "sending a message" to the object, where the message indicates what should be done and may return a response value. The actual details of the object's implementation, for example, the way it is laid out in memory, are hidden and can therefore be implementation dependent. What is important is that methods always produce the same results regardless of the implementation.

There are also ways of restricting the scope over which an object

There are also ways of restricting the scope over which an object can be accessed by other parts of a program. In Java, for example, a class may be declared to be *public, private,* or *protected.* These designations indicate the visibility (and ability to use) an object outside of a given library of objects. The public and private designations are more or less self-explanatory: If a class is public, it can be used by code outside the library; if it is private, it can only be used by other classes inside the same library. Protected classes are similar to private ones, except any inheriting class is given access.

Example

Figure 5.5 is a Java program that defines a Rectangle class via two methods, one that finds the area and one that finds the perimeter of the Rectangle. The program also extends the Rectangle class with a Square class. The Square class has a somewhat simplified perimeter method that will override the Rectangle's perimeter method. The main program reads pairs of integers from the command line. These integers represent a rectangle's length and width. If the sides are the same, a Square object is instantiated; otherwise, a Rectangle is instantiated. Then the area and perimeter methods are called, with a reference to the Square or Rectangle as the argument. The particular perimeter method used depends on the type of object to which the reference points. That is, the specific perimeter method is *dynamically dispatched* at run time. This program is admittedly contrived—the sides array is somewhat superfluous—but it allows us to illustrate a number of JVM features in a small example.

Garbage Collection

One of the nice features of the object-oriented model is that, from the programmer's perspective, memory is a large, virtually unbounded space populated with objects. These objects essentially "float" in the memory space, with references tethering them to executing programs. Objects can be created dynamically, used for a