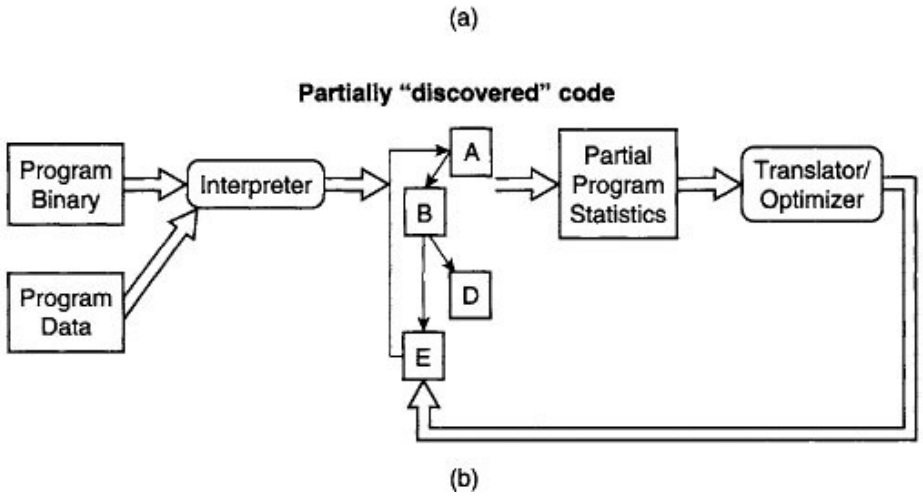


Figure 4.11 Using Program Profiling. *(a) Traditional profiling applied across multiple program runs; (b) on-the-fly profiling in a dynamic optimizing VM.*

When dynamic optimization is being used, there are overhead costs every time a guest program is run. These costs are in the form of program analysis in order to place profiling probes as well as the cost of collecting the profile data after the probes have been placed. And of course, in a dynamic optimization environment, the overhead of optimization must be outweighed by the benefits of executing the optimized code. An interesting compromise solution is used in the FX!32 system (described in **Section 3.10**), where all code translation and optimization is done between program runs and optimized code is saved on disk between runs.

Because of the tradeoffs involved, different VM applications, and the variety of optimizations that may be performed, there are a number of possible profiling techniques that are of interest when performing dynamic optimization in a VM environment. In this section we will overview the spectrum of profiling techniques. Some of these will be useful for supporting dynamic optimizing process VMs, described later in this chapter. Others will be more useful with HLL VM implementations, discussed in **Chapter 6**.

4.2.2 Types of Profiles

There are several types of profile data that can be used in a dynamically optimizing VM. The first type simply indicates how frequently different code regions are being executed. This information can be used to decide the level of optimization that should be performed for a given region. The heavily used code regions, or *hotspots*, should be optimized more, and lightly used code can be optimized less (or not at all) because the optimization process itself is time consuming and potentially performance degrading. For example, in the framework of **Figure 4.4**, a code sequence may be interpreted until it is determined to be "hot" and then binary translated to speed up future emulation.

A second important type of profile data is based on control flow (branch and jump) predictability, where the profiler collects control flow statistics that can be used for determining aspects of a program's dynamic execution behavior. This information may be used as the basis for gathering and rearranging basic blocks into larger units of frequently executed code, such as the superblocks of **Figure 4.3**. These larger code units provide opportunities for optimizations over a scope larger than single basic blocks and may also lead to better performance due to instruction cache locality effects.

Other types of profile data may be used to guide specific optimizations. This profile data may focus on data or address values. For example, if an optimization depends on whether load and store addresses are aligned on natural boundaries (see **Section 4.6.4**), then profiling of load and store addresses can determine whether this condition holds (at least most of the time) and the optimization should be applied. Or if a program variable is found to hold the same value most (or all) of the time, then program code can be streamlined (or "specialized") for the very common case. These more specific types of profile data will not be discussed further in this section but will be described when required by a particular optimization technique.

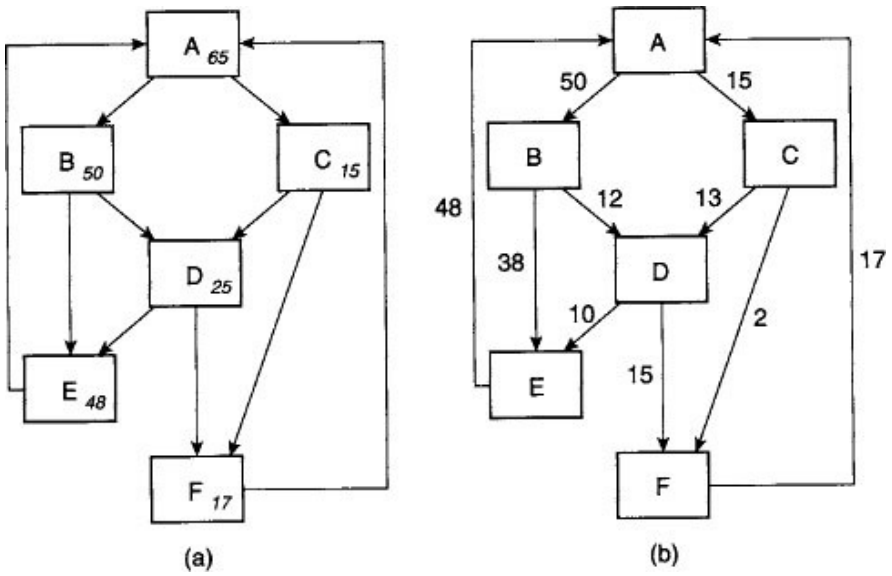


Figure 4.12 Profiling Applied to a Control Flow Graph. *Boxes (nodes) represent basic blocks, and arrows (edges) indicate control transfers. (a) A basic block profile (or node profile); (b) an edge profile.*

The first two types of profile information described earlier are almost universally used by dynamic translation/optimization systems and deserve additional explanation. **Figure 4.12** shows a program's control flow graph. The control flow graph contains *nodes,* which correspond to the program's basic blocks (as defined in **Section 2.6**), and *edges,* which connect the basic blocks and represent flow of control (due to branches and jumps). In the figure, basic block A ends with a conditional branch; one path of the branch, i.e., the fall-through path, goes to basic block B, while the taken path of the branch goes to basic block C.

To determine the hot regions of code, one should profile each basic block and count the number of times each is executed during some time interval; the blocks most frequently executed are the hot ones. A *basic block profile* (or *node profile* in graph terminology) is illustrated in **Figure 4.12**a. In this example, the numbers contained in the nodes are the counts of the number of times the corresponding basic block has been executed. The blocks A, B, and E are the hottest blocks.

Alternatively, the *edge profile* is shown in **Figure 4.12**b. The edge

Alternatively, the *edge profile* is shown in Figure 4.12b. The edge profile contains counts of the number of times each control flow edge is followed. In the figure, the edge profile is shown by labeling each of the edges with the number of times it is followed.

In general, there are more edges than nodes, so it may appear slightly more expensive to profile all the edges. However, algorithms have been developed for significantly reducing the number of edges that must be probed in order to compute a complete edge profile (**Ball and Larus 1994**). Furthermore, the edge profile data provides a more precise view of program execution than the basic block profile; i.e., the basic block profile can be derived from the edge profile by summing the number of incoming edges to each block (or by summing outgoing edge counts). We see in Figure 4.12b, for example, that the basic block D must be executed 25 times—found by adding the profile counts on the incoming edges to basic block D (12 plus 13). Note that the converse property is not necessarily true; i.e., the edge profile cannot always be derived from a basic block (node) profile.

Another type of profile based on control flow is the *path profile* (**Ball and Larus 1996**). A path profile subsumes an edge profile by counting paths containing multiple edges rather than individual edges. For some optimizations, such as superblock formation, the path profile (at least in theory) is the most appropriate type of profile because the basic blocks arranged to form a superblock should be contained in a commonly followed path. Although it has been shown that collecting path profile data is not necessarily more complex than gathering edge profile data, more up-front program analysis is required in order to decide where to place the profile probes. This up-front analysis probably makes it impractical for most dynamic optimization systems because not enough of the program's structure is known *a priori.* A heuristic for finding frequent paths is simply to follow the sequence of the frequent edges. Although this heuristic does not always find the most frequent paths, significant counterex-

amples are relatively rare (**Ball, Mataga, and Sagiv 1998**), and most optimization algorithms approximate the path profile by using a heuristic based on the edge profile.

4.2.3 Collecting Profiles

There are two ways of collecting a profile: with instrumentation and with sampling. *Instrumentation-based profiling* typically targets specific program-related events and counts all instances of the events being profiled, for example, the number of times a basic block is entered or the number of times a conditional branch is taken versus not taken. Generally speaking many different events can be monitored simultaneously. Monitoring can be implemented by inserting probe instructions into the code being profiled or by using underlying profiling hardware to collect data. Using software instrumentation slows down the program being profiled considerably but can be performed on any hardware platform. Hardware instrumentation adds much less overhead but is not very well supported by current hardware platforms. The Intel Itanium platforms are a notable exception (**Choi et al. 2002**). Furthermore, hardware instrumentation generally does not have the flexibility of software instrumentation; the types of program events to be profiled are built into the hardware.

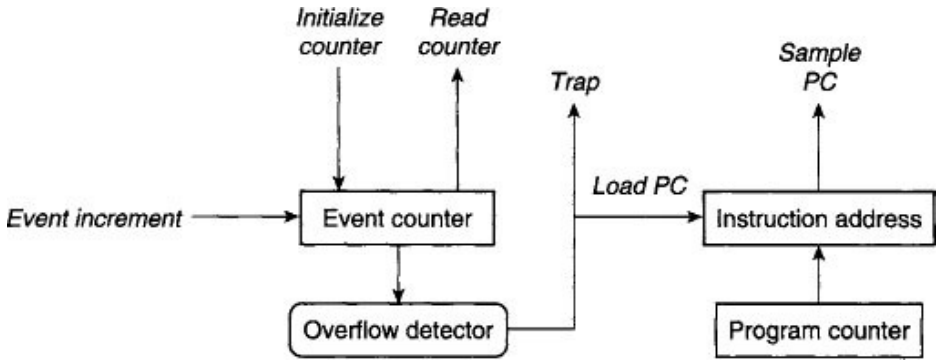


Figure 4.13 Hardware Used for Sampling. *An event counter triggers a trap; the PC captured at the time of the trap produces the sample.*

With *sampling-based profiling,* the program runs in its unmodified

form, and, at either fixed or random intervals, the program is interrupted and an instance of a program-related event is captured (see **Figure 4.13**). For example, program counter values at taken branches may be sampled. After a number of such samples have been taken, a statistical picture of the program emerges; for example, where the program hotspots or hot edges lie.

An important tradeoff between the two profiling methods is that instrumentation can collect a given number of profile data points over a much shorter period of time, but commonly used software instrumentation slows down the program during the time the profile is being taken. Sampling slows down a program much less, at least as perceived by the user, but requires a longer time interval for collecting the same amount of profile information. Of course, the slowdown depends on the sampling interval (typical sampling intervals may be many thousands of instructions).

On a per data point basis, sampling generally has a higher absolute overhead than instrumentation because it causes traps to the profile collecting software. But this overhead is spread out over a much longer total time interval, so the perceived slowdown during the profiling process is less. In addition, multiple samples can be collected and buffered before signaling the user-level profiling software (**Dean et al. 1997**). With instrumentation, a profile is quickly taken, optimizations are performed, and instrumentation software is either completely or partially removed.

The preferred profiling technique depends on where in the emulation spectrum (**Figure 4.5**) the optimization process is taking place. For interpretation, software instrumentation is about the only choice, and even there the options are rather limited, as we shall see. For optimizing translated code that may benefit from further optimization, or in dynamic optimization systems where no initial translation is required, there is a much wider variety of options. In general, dynamic optimization systems use instrumentation because

general, dynamic optimization systems use instrumentation because it allows the overall optimization process to move much more quickly from slower, less optimized code to highly optimized versions. Sampling is more useful when a longer-running program is already well optimized but further higher-level optimizations may be beneficial. In the following subsections, we will overview profiling methods for both interpreted and translated code.

4.2.4 Profiling During Interpretation

There are two key points to consider when profiling during interpretation: The source instructions are actually accessed as data, and the interpreter routines are the code that is being executed. Consequently, any profiling code must be added to the interpreter routines. This implies that profiling is most easily done if it is applied to specific instruction *types* (e.g., opcodes) rather than specific source instructions. It can also be applied for certain classes of instructions that are easily identified during the interpretation process, for example, the destinations of backward branches—as we will see, these are often used as the starting points for forming large translation blocks.

To perform basic block profiling while interpreting, profile code should be added to all control transfer (branch and jump) instructions (after the PC has been updated) because the instructions immediately following these instructions, by definition, begin new basic blocks. Counting the number of times each of these destination instructions is executed constitutes a basic block profile. For edge profiling, the same control transfer instructions can be profiled, but both the PC of the control transfer instruction and the target are used to define a specific edge.

The profile data is kept in a table that is accessed via source PC values of control transfer destinations (basic block profiling) or the PC values that define an edge. This table is similar to the PC map table

used during emulation (described in **Section 2.6**), in that it can be hashed into by the source PC. The entries contain the basic block or edge counts. For conditional branches, counts can be maintained for cases both taken and not taken. **Figures 4.14** and **4.15** illustrate the interpreter-based profiling method for conditional branches. **Figure 4.14** is the interpreter routine for a PowerPC conditional branch instruction. A data structure for collecting edge profiles is given in **Figure 4.15**.

```
Instruction function list
.
.
branch_conditional(inst) {
     BO = extract(inst,25,5);
     BI = extract(inst,20,5);
     displacement = extract(inst,15,14) * 4;
     .
     .
     // code to compute whether branch should be taken
     .
     .
     profile_addr = lookup(PC);
     if (branch_taken)
          profile_cnt(profile_addr, taken);
          PC = PC + displacement;
     Else
          profile_cnt(profile_addr, nottaken);
          PC = PC + 4;
}
```

Figure 4.14 PowerPC Branch Conditional Interpreter Routine with Profiling Code Added (in italics).

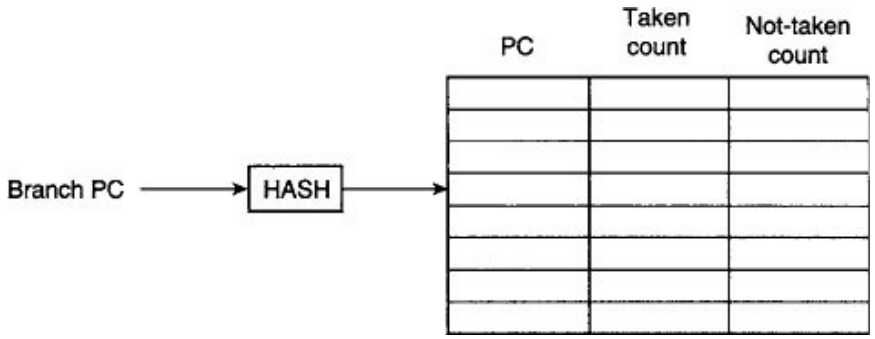


Figure 4.15 Profile Table for Collecting an Edge Profile During Interpretation. *A branch PC accesses the table via hashing; a taken or not-taken counter is incremented, depending on the branch outcome.*

In the branch interpreter routine, the PC of the branch instruction is

used for performing a profile table lookup via a hash function. Then entries in the profile table are updated appropriately. If the branch is taken, the taken count is incremented; otherwise the not-taken count is incremented. Alternatively, for basic block profiling, the source PC at the end of the branch interpreter routine can be used as the key to the profile table, and only one count per PC is maintained.

Profile Counter Decaying

As just described, the profile counts held in the profile table increment indefinitely. In practice this leads to an obvious problem—a count may eventually overflow the size of its count field. An immediate solution is to use *saturating* counts; i.e., when a counter reaches its maximum value, it stays there. An additional enhancement lies in the observation that many optimization methods focus on relative frequency of events, e.g., which basic blocks are more heavily used, not the absolute counts. Furthermore, when it comes to making dynamic optimization decisions, recent program event history is probably more valuable than history in the distant past.

This leads to a counter "decay" process. To implement counter decay, the profile management software periodically divides all the profile counts by 2 (using a right shift). After decay, the counts maintain their relative values, and, as the program phase changes and a profiled event becomes inactive, counts carried over from the previous phase will eventually decay to zero. Consequently, at any given time the profile counts reflect the relative activity over a relatively recent period of time.

Profiling Jump Instructions

Up to this point, when profiling control flow, e.g., edge profiles, we have been concerned primarily with conditional branch instructions. Profiling indirect jumps is slightly complicated because there

may be a number of target addresses, and in some cases they may change frequently. This is the case, for example, in return jumps from procedures that have many call sites. This means that a complete profile mechanism that tracks all the targets would consume both more space and more time than conditional branch edge profiling. The process can be simplified, however, by noting that profile-driven optimizations of indirect jumps tend to be focused on those jumps that very frequently have the same target (or a small number of targets). Consequently, it is only necessary to maintain a table with a small number of target addresses (say one or two) and track only the more recently used targets. If the target changes frequently among a large number of addresses, then these counts will always be low, and optimizations based on frequent jump targets will likely provide little benefit anyway.

4.2.5 Profiling Translated Code

When profiling translated code, each individual instruction can have its own custom profiling code. Among other things, instrumenting individual instructions means that profiling can be selectively applied. It also means that profile counters can be assigned to each static instruction at the time the instrumentation is inserted so that profile counters are directly addressed and no hashing is needed. Profile counters can be held in a simple array, with an index into the array hardcoded into the profile code for a specific instruction.

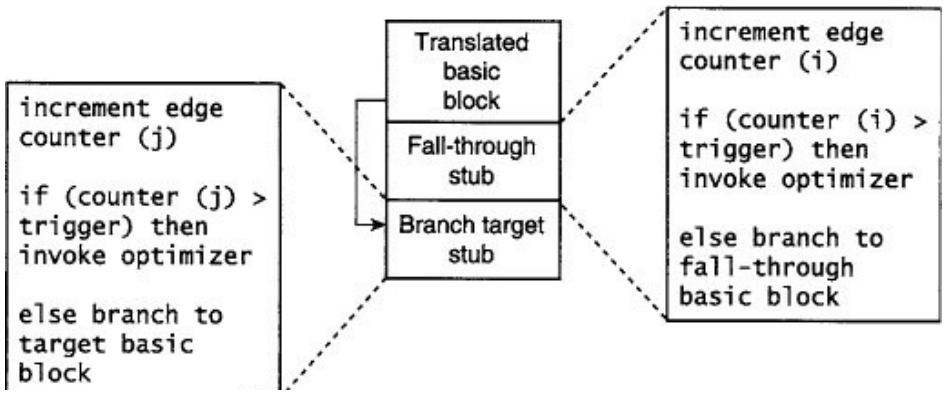


Figure 4.16 Edge Profiling Code Inserted into Stubs of a Binary Translated Basic Block.

Figure 4.16 illustrates the insertion of instrumentation for edge profiling into translated basic blocks (as in **Section 2.6**). Here, instrumentation is placed into stub code at the end of each translated basic block. In this example, the optimization algorithm is invoked if a profile value exceeds a specified threshold value, or *trigger*. An advantage of placing profile code in the stub region is that profile code can be easily inserted and removed as needed.

4.2.6 Profiling Overhead

Software profiling adds instructions to the overall emulation process. For example, in **Figure 4.15**, instructions must be added to (1) access the hash table (this will require the hash function plus at least one load and compare) and (2) increment the proper count (two more memory accesses plus an add instruction). For profiling during interpretation, this may add an additional overhead of 10–20% because interpretation already requires executing a number of target instructions per interpreted source instruction. For profiling translated code, fewer instructions may be needed if the profile table lookup is replaced with directly addressable counters, but the *relative* overhead is still significantly higher than with interpretation because translated code itself is much more efficient than interpreted code. In addition to performance overhead, profiling leads to memory overhead for maintaining tables. For example, a profile table such as in **Figure 4.15** would require perhaps four words of data for each instruction being profiled.

Profiling overheads can be reduced in a number of ways. One way is to reduce the number of instrumentation points by selecting a smaller set of key points, perhaps using heuristics, and to use the collected data at these points to calculate the profile counts at the others. Another method, based on code duplication, is especially attractive for same-ISA optimization and is described in **Section 4.7**

tractive for same-ISA optimization and is described in **Section 4.7**.

4.3 Optimizing Translation Blocks

As was pointed out in the introduction to this chapter, there is often a two-part strategy for optimizing translated code blocks. The first part of the strategy is to use knowledge of dominant control flow for enhancing memory locality by placing frequently followed sequences of basic blocks into contiguous memory locations. In practice these are usually superblocks. In the second part of the strategy, the enlarged translation blocks are optimized, with emphasis on optimizations that are skewed to favor the commonly followed sequence. That is, the optimization increases performance if the common sequence is followed, with a possible loss of performance in the uncommon case. Although the two parts of the strategy are actually relatively independent, both are useful and are often applied in the same system. A common approach is first to form a localized block and then to optimize it as a unit. In the following discussion we will follow the same order.

4.3.1 Improving Locality

There are two kinds of memory localities. The first is *spatial locality,* where an access to a memory location is soon followed by a memory access to an adjacent memory location. The second is *temporal locality,* where an accessed memory location is accessed again in the near future. Instruction fetches naturally have a high degree of both types of localities, and instruction-fetch hardware tends to perform better when locality is greater. By rearranging the way code is placed in memory, moreover, both types of localities can be further enhanced.

The methods and benefits of improving code locality are best explained via an example. **Figure 4.17**a shows a region of code as orig-

inally placed in memory. Basic blocks are labeled with uppercase letters, and branch instructions are shown. Other instructions are represented with horizontal lines. **Figure 4.17**b is the control flow graph for the same code sequence, with edge profile information given. The most common path through the code is A, D, E, G. Block F is rarely executed.

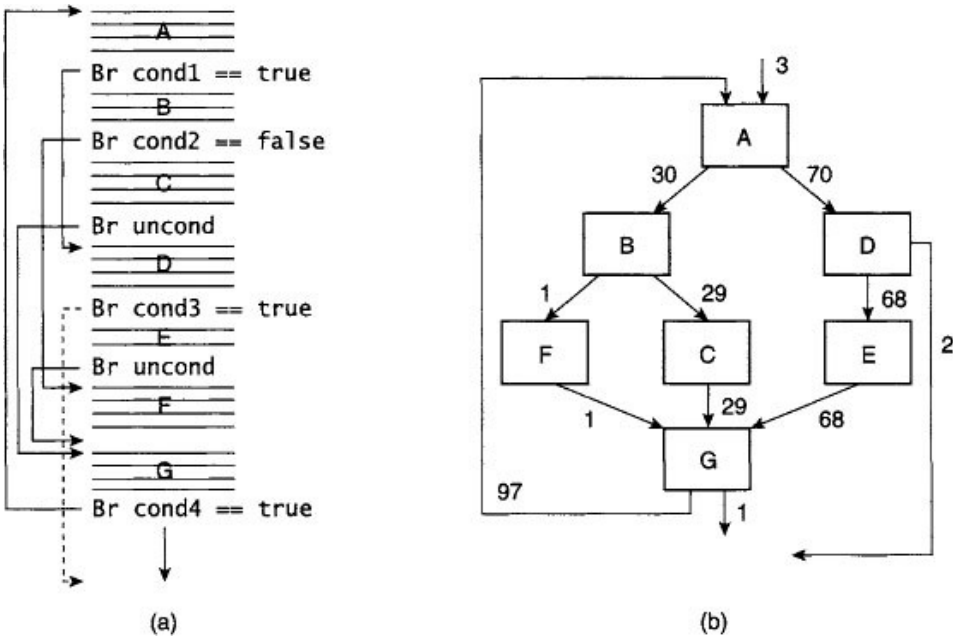


Figure 4.17 Example Code Sequence. *(a) Basic blocks as initially arranged in memory; (b) edge profile for example code.*

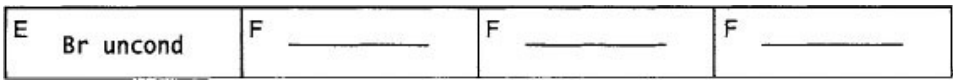


Figure 4.18 Cache Line Containing Instructions from Basic Blocks E and F.

First consider how the example code will map onto cache lines; this reveals two ways in which performance may be lost in a typical high-performance processor. **Figure 4.18** shows a cache line that contains a total of four instructions at the boundary between basic blocks E and F. Because block F is rarely used, there is relatively little useful locality (spatial or temporal) within this line; nevertheless, the whole line is placed in the cache just to hold the single instruction from block E. It would be better to use the cache space consumed by instructions from block F for other instructions that are

more frequently used. Also, when the line is fetched from the cache, only one of the four instructions is useful. Performance may be lost because of relatively low instruction-fetch bandwidth.

To improve performance, profile information can be used to rearrange the layout of the blocks in memory (**Pettis and Hansen 1990**). A rearranged layout is shown in **Figure 4.19**. To accomplish the new layout, some of the conditional branch tests are reversed. For example, the test at the end of block A is changed from true to false. Furthermore, some unconditional branches may be removed altogether, for example, the branch that ends block E. Also, as part of the relayout process, block F, which is rarely used, can be placed in a region of memory reserved for such seldom-used blocks. These blocks will generally not consume instruction cache space. Finally, instruction-fetch efficiency is improved, for example, when executing the code sequence in blocks E and G, which are always executed in sequence.

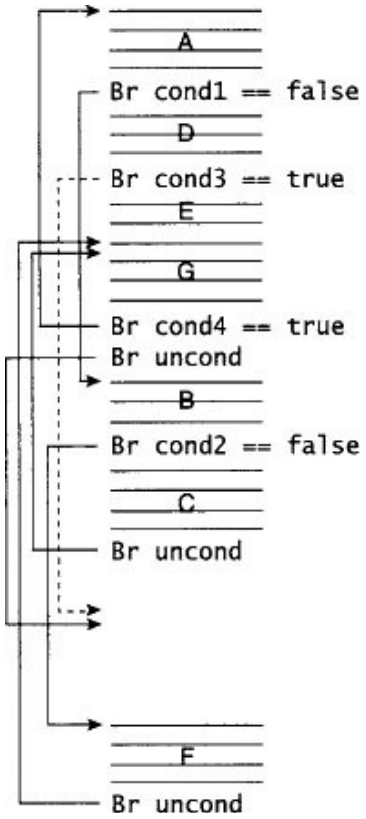


Figure 4.19 Example Code After Relayout to Improve Locality.

Another method for improving spatial locality (and to provide other optimization opportunities) is to perform *procedure inlining* (**Scheifler 1977**). With conventional procedure inlining, the body of a procedure is duplicated and placed at its call/return site. Procedure inlining is illustrated in **Figure 4.20**, where procedure xyz is called from two locations (**Figure 4.20**a). After inlining has been performed, the code belonging to procedure xyz is duplicated and placed at the two call sites, as shown in **Figure 4.20**b. The call and return instructions for procedure xyz are removed. It may also be possible to remove some register save/restores that surround the call and return in the original code. Hence, inlining not only increases spatial locality in the instruction stream but also eliminates much of the overhead of calls and returns.

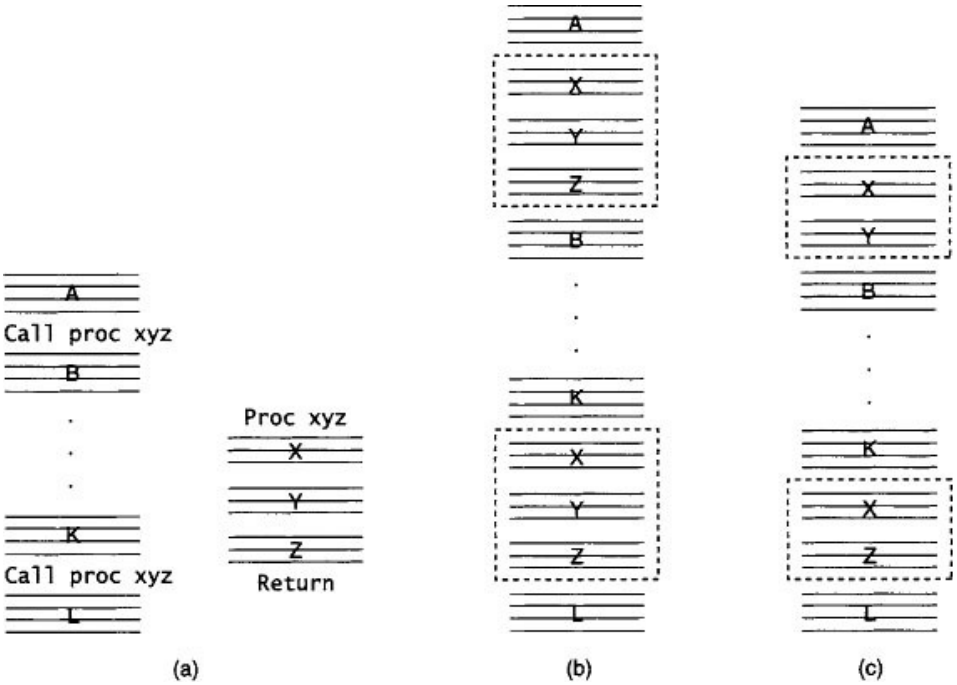


Figure 4.20 Procedure Inlining. *(a) A code sequence calling a procedure from two call sites. (b) With conventional procedure inlining, procedure B is duplicated and placed at the original call sites. (c) With partial procedure inlining, only a specific control flow path is inlined at each call site.*

In dynamic optimization systems, procedure inlining is often implemented in a dynamic form, based on commonly followed paths dis-

covered at runtime. The leads to dynamic or *partial procedure inlining*, which is based on similar concepts as dynamic basic blocks, discussed in **Section 2.6.3**. The important point is that because code is discovered incrementally at runtime, it is often not practical to discover and inline a full procedure containing multiple control flow paths. For example, in **Figure 4.20**a, when procedure xyz is called from the first call site, the path of execution may flow through blocks X and Y; when it is called from the second call site, the path of execution may flow through blocks X and Z. When partial procedure inlining is applied, as shown in **Figure 4.20**c, only the specific dynamic paths are inlined at the call sites.

Unlike simple code relayout, as described earlier, inlining may increase the total code size when the same procedure is called from several different sites. This may have a negative effect on instruction cache performance. It also may increase register "pressure," i.e., the demand for registers within the region of the inlined code. Consequently, partial procedure inlining is typically used only for those procedures that are very frequently called and that are relatively small—many library routines fit these criteria. Profile information can be used to identify procedures that are good candidates for inlining (**Chang et al. 1992**; **Ayers, Schooler, and Gottlieb 1997**).

Specific algorithms for implementing both code relayout and inlining have been used in conventional optimizing compilers. Rather than describe these conventional methods in detail, however, we will consider only the methods useful during dynamic translation and optimization.

We now describe three ways of rearranging basic blocks according to control flow. The first way, *trace* formation, follows naturally from the discussion just given. However, the second way, *superblock* formation, is more widely used in virtual machine implementations, because superblocks are more amenable to interbasic block opti-

mizations, as we shall see. The third way of arranging code is to use *tree groups,* a generalization of superblocks, which are useful when control flow is difficult to predict, and provides a wider scope for optimization. In the following subsections, we discuss traces, superblocks, and tree groups and then turn to optimization methods based on traces and superblocks.

4.3.2 Traces

This code-relayout optimization essentially divides the program into chunks of contiguous instructions containing multiple basic blocks (**Fisher 1981**; **Lowney et al. 1993**). In the example of **Figure 4.17**, the sequence ADEG is the most commonly executed path and is placed in memory as a single contiguous block of code. This sequence forms a *trace*[1]—a contiguous sequence of basic blocks. A second, lesser used trace in the example is BC. Block F alone forms a rarely used trace containing only one basic block.

It should be clear that edge profile data can be useful for determining the frequently followed paths. Classically, with an offline profile method (see **Figure 4.11**a) traces are formed via the following steps.

1. A profile is collected during one or more executions of the program using test data.
2. Using the profile data, begin with the most frequently executed basic block that is not already part of a trace; this is a trace *start point.*
3. Using edge or path profile data, begin at the start point and follow the most common control path, collecting basic blocks along the path until a *stopping condition* is met. One stopping condition is that a block already belonging to another trace is reached. Another example stopping condition is the arrival at a procedure call/return boundary (if the procedure is not being inlined). A more complete set of stopping conditions is given in **Section 4.3.4**.
4. Collect the basic blocks into a trace, reversing branch tests and removing/adding unconditional branches as needed.
5. If all instructions have been included as part of some trace, stop; otherwise go to step 2.

If we follow this algorithm with the example in **Figure 4.17**, then we

start with block A and follow it to D and E and stop at G. In this case, we use loop-closing branches as a stopping condition (as is often done). Then the next trace starts at B and stops at C. Finally, F is left as the last trace. **Figure 4.21** illustrates the traces with respect to the control flow graph; when these traces are laid out in memory, they will appear as in **Figure 4.19**.

In a dynamic environment (as in **Figure 4.11**b), the foregoing process of trace formation can be modified appropriately and would build up traces in an incremental manner as hot regions of code are discovered. In practice, however, traces as just described are not commonly used as translation blocks in today's dynamic translation/optimization systems; superblocks and tree groups are more commonly used, so we defer discussion of dynamic trace formation to the next section.

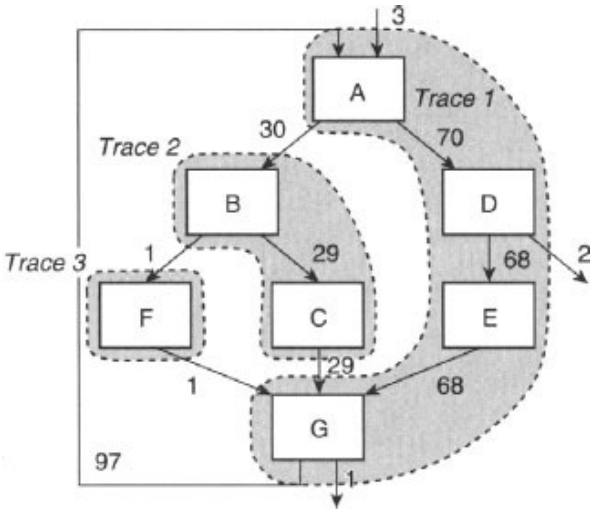


Figure 4.21 Using Edge Profiles or Collect Basic Blocks into Traces.

4.3.3 Superblocks

A widely used alternative to the trace is the superblock (**Hwu et al. 1993**); by construction, superblocks have only one entrance at the top and no side entrances. In contrast, a trace may have both side entrances and side exits. For example, in **Figure 4.21**, the trace

ADEG contains a side exit from block D and two side entrances into block G. As we shall see in **Section 4.5.3**, disallowing side entrances simplifies later code optimizations.

It might first appear that if superblocks are formed using a method like the one described for traces, the result is relatively small superblocks; for example, in **Figure 4.22**a ADE, BC, F, and G form a complete set of superblocks. These blocks are smaller than the traces and in some cases may be too small to provide many opportunities for optimizations. However, larger superblocks can be formed by allowing some basic blocks to appear more than once. This is illustrated in **Figure 4.22**b, where larger superblocks have been formed. Here, the superblock ADEG contains the most common sequence of basic blocks (according to the profile information given in **Figure 4.12**). Now, because block G in superblock ADEG can only be reached via a side entrance, block G is replicated for the superblocks that contain BCG and FG. The process of replicating code that appears at the end of a superblock in order to form other superblocks is referred to as *tail duplication.*

4.3.4 Dynamic Superblock Formation

Superblocks can be formed via a profile-driven static algorithm just as traces are; i.e., by first using test input data to collect a profile and then as part of a static compilation step forming all the superblocks. However, because superblocks are a common choice for virtual machine implementations and are formed at run time, we will consider only their *dynamic* formation in detail. The key point is that they are formed incrementally as the source code is being emulated.

A complication that comes from using superblocks is that basic block replication leads to more choices, especially the choice of when to terminate a superblock, so a number of heuristics are followed. There are three key questions regarding dynamic superblock formation: (1) At what point in the code should a superblock be

started? (2) As a superblock is being built, what should the next basic block be? (3) At what point should a superblock be terminated? We will discuss each of these in turn, focusing on heuristics that appear to work well in practice.

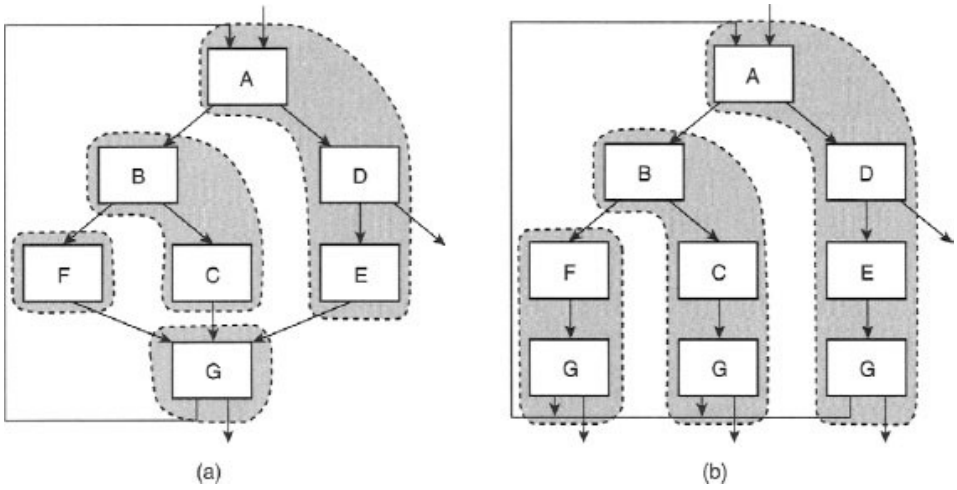


Figure 4.22 Superblocks. *Superblocks are regions of code with only one entry point and one or more exit points. (a) Superblocks containing no replicated code; (b) superblocks with replicated code (e.g., block G) are larger and provide more optimization opportunities.*

Starting Points

In general, a superblock should start at a heavily used basic block. Consequently, as code is initially being emulated, either through interpretation or simple basic block translation, profile information is collected in order to determine those heavily used basic blocks where superblock formation should be started. For this purpose, there are two methods for determining profile points. One is simply to profile all basic blocks. Another is to use heuristics based on program structure to select a narrower set of good candidate start points and then to profile only at those points. One such heuristic is to use the targets of backward branches as candidate start points. Heavily used code will very likely be part of a loop, and the top of every loop is the target of a backward branch. A second heuristic is to use an exit arc from an existing superblock. These arcs are good candidates because, by definition, the existing superblocks are known to be hot, and some exit points will also be hot (although per-

haps somewhat less so than the original basic block). In addition, exit points are often not the targets of backward branches and would otherwise be overlooked by the first heuristic. Using heuristics for determining candidate start points reduces the initial number of points to be profiled significantly, and it reduces those that must be monitored for "hotness" as the incremental formation process proceeds.

Regardless of how candidate start points are selected for profiling, most superblock formation methods define a *start threshold* value. When a profiled basic block's execution frequency reaches this threshold, a new superblock is started. In IA-32 EL (**Baraz et al. 2003**), profiling continues until a certain number of hot basic blocks have reached the threshold (or until one reaches twice the threshold); at that point, superblock formation is started. The value of the start threshold depends on the staged emulation tradeoff involving interpretation versus translation overhead, as discussed earlier; a threshold of a few tens to hundreds of executions is typical.

Continuation

After a superblock is begun at an initial basic block, the next consideration is which subsequent blocks should be collected and added as the superblock is grown. This can be done using either node or edge information. There are two basic heuristics for using this information: One is *most frequently used* and the other is *most recently used.*

In an example of the most-frequently-used approach (**Cifuentes and Emmerik 2000**), node profile information is used to identify the most likely successor basic block(s). For this purpose, a second threshold, a *continuation threshold,* is set to determine the possible candidates. This continuation threshold will typically be lower than the start point threshold; for example, a typical continuation threshold might be half the start threshold. Because blocks other than start points must be profiled, a relatively complete set of profile data

must be collected for all basic blocks.

At the time the start threshold is reached and superblock formation is to begin, the set of all basic blocks that have reached the continuation threshold is collected. This is the *continuation set.* Then the superblock formation algorithm starts with the hottest basic block and builds a superblock, following control flow edges and including only blocks that are in the continuation set. When a superblock is complete (stopping points are discussed shortly), the emulation process can resume with profiling until another basic block achieves the start threshold. An alternative, however, is to go ahead and form additional superblocks using the members of the continuation set. That is, of the blocks remaining in the continuation set, take the hottest as a new start point and build a second superblock, removing blocks from the continuation set as they are used. This process continues until all the blocks in the continuation set have been exhausted. This method of building multiple superblocks at once has the advantage of amortizing some of the superblock formation overhead over multiple superblocks.

A most-recently-used method (**Duesterwald and Bala 2000**) relies on edge-based information. The superblock formation algorithm simply follows the actual dynamic control flow path one edge at a time, beginning at the time the start point is triggered. That is, it is assumed that the very next sequence of blocks following a start point is also likely to be a common path—a reasonably good assumption, given the observations on program behavior in **Section 4.1**. With this method, only candidate start points need to be profiled—there is no need to use profiling for continuation blocks. Hence, one benefit of this approach is that the profile overhead can be substantially reduced versus a most-frequently-used algorithm. A similar approach has been advocated for procedure layout (**Chen and Leupen 1997**).

Finally, an alternative and more sophisticated approach combines

Finally, an alternative and more sophisticated approach combines edge profiling and the most-frequently-used heuristic (**Berndl and Hendren 2003**). This approach relies on correlating sequences of conditional branches (edges in an edge profile) to detect the most likely paths that connect basic blocks into superblocks.

Stopping Points

At some point the superblock formation process must stop. Again, heuristics are typically used, and the following are possible choices. Any given system may use some or all of these heuristics (or possibly others).

1. **The start point of the same superblock is reached.** This indicates the closing of a loop that was started with this superblock. In some systems, superblock formation can continue even after a loop is closed, which in effect leads to dynamic loop unrolling.
2. **A start point of some other superblock is reached.** When this occurs, superblock formation stops and the two superblocks can be linked together (**Section 2.7**).
3. **A superblock has reached some maximum length.** This maximum length may vary from a few tens to hundreds of instructions. A reason for having a maximum length is that it will keep code expansion in check. Because a basic block can be used in more than one superblock, there may be multiple copies of a given basic block. The longer superblocks grow, the more basic block replication there will be.
4. When using the most-frequently-used heuristic, **there are no more candidate basic blocks that have reached the candidate threshold.**
5. **An indirect jump is reached, or there is a procedure call.** The use of this stopping heuristic depends on whether partial procedure inlining is enabled, and, if enabled, whether the procedure satisfies criteria for inlining. Tradeoffs involving inlining are discussed further in the context of HLL VMs (see **Section 6.6.2**).

Example

As an example, we first use the most-frequently-executed method for superblock formation (**Cifuentes and Emmerik 2000**). Consider the control flow graph shown in **Figure 4.17**. Say the start point

threshold is set at 100 and the continuation threshold at 50. Then, as shown in the figure, basic block A has just reached the start point threshold. Blocks D, E, and G have reached the continuation threshold. Because A has reached the start threshold, superblock formation begins, and superblock ADEG is formed. In this example, formation stops because of the branch from G back to A. At this point, the superblock can be optimized as a unit, with any needed compensation code being added at the exit points. Then, as execution proceeds, block B may next reach the threshold and superblock BCG will most likely be formed. Again, superblock formation stops because G leads to the start point of existing superblock ADEG. Finally, block F might eventually reach the threshold, causing superblock FG to be formed. If F never reaches the threshold, then it is never placed in a superblock. **Figure 4.23** shows the final code layout for the example superblocks. Each superblock is a contiguous-code region, and basic block G is replicated three times.

If the most recently executed method is used (**Duesterwald and Bala 2000**), then only block A is profiled, because it is the only target of a backward branch in this example. When the profile count for block A reaches a threshold value, say, 100, then superblock formation immediately begins and follows the execution flow as it happens. This will most likely lead to superblock ADEG (if the branches are decided in their most probable direction). Then block B is initialized for profiling, because it is now the destination of a superblock exit. After the profile count of B reaches 100, then another superblock will be formed, again following the “natural” flow as it occurs. In this case, it will most likely be BCG. Hence, both the most-frequently-executed and most-recently-executed methods seem to have the same result. However, note that at the time A reaches the threshold, there is about a 30% chance that the branch exiting A will go to B rather than D, and superblock ABCG will be formed rather than ADEG. A subsequent superblock would then be DEG. This illustrates that there are cases where a most-recently-executed method may not select superblocks quite as well as a most-frequently-

executed method. Tree groups, described in the next subsection, are aimed at just those situations.

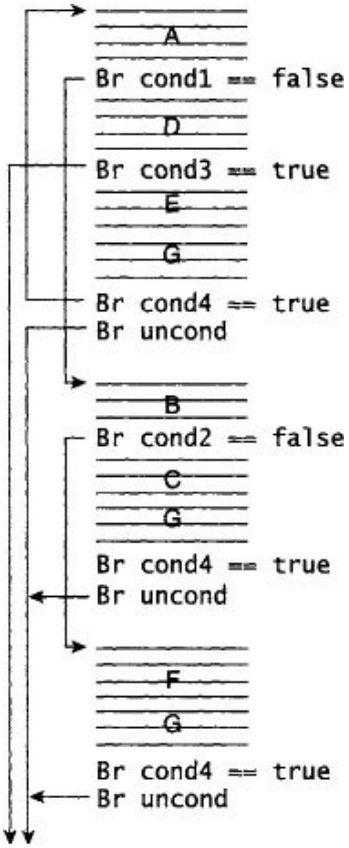


Figure 4.23 Code Layout Following Superblock Formation for Example CFG.

4.3.5 Tree Groups

Although traces and superblocks (as well as dynamic basic blocks) are the most commonly used units for translation and optimization, there are other possibilities. Traces and superblocks are based on the principle that conditional branches are predominantly decided one way (**Figure 4.7**). However, there are some branches for which this is not the case. For example, in **Figure 4.7**, almost 20% of the branches range between 30–70 and 70–30, taken versus not-taken. Almost 10% of the branches are about 50–50. For branches that tend to split their decisions, a superblock or trace side exit is frequently taken. When this happens, there is often overhead involved in compensation code (**Figure 4.2**). Furthermore, optimizations are typi

pensation code (Figure 4.2). Furthermore, optimizations are typically not done along the side exit path, thereby losing performance opportunities.

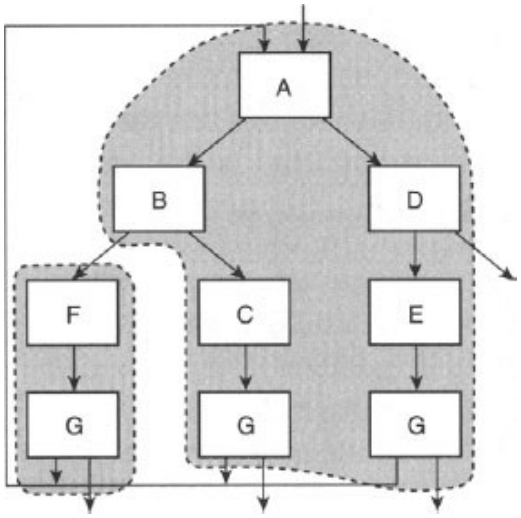


Figure 4.24 Tree Group. *A tree group encompasses multiple commonly followed control flow paths.*

For those situations where conditional branch outcomes are more evenly balanced, it may be better to use *tree regions* or *tree groups* rather than superblocks (**Banerjia, Havanki, and Conte 1997**). Tree groups are essentially a generalization of superblocks, and they offer more optimization opportunities. They have one entrance at the top and may have multiple exits, but they can incorporate multiple flows of control rather than a single flow of control as with a superblock. That is, as the name suggests, they form trees of connected basic blocks. A tree group for the code illustrated in **Figure 4.17** is given in **Figure 4.24**. A tree group is formed by using tail duplication, as is done with superblocks. The branch at the end of block A in **Figure 4.17** is a 30–70 branch, so we include both exit paths in the tree group. On the other hand, the edge from F to G is rarely taken, so it is not included in the major tree region. However, it may later be used as part of a smaller tree region, as shown in the figure.

Just as with traces and superblocks, tree groups can be constructed using collected profile information. For example, a variation of the

most-frequently-executed heuristic might set a start threshold at 100 and a continuation threshold at 25 (the continuation threshold is lower than with superblocks, to "encourage" the formation of larger trees). Then for our example code, an initial tree group would consist of blocks ADEGBCG, as show in **Figure 4.24**; a later tree region might be the smaller FG. Tree groups can also be constructed incrementally. For example, in **Figure 4.24** one might first form the tree group ADEG (actually a superblock). Then after additional program emulation and profiling, the path BCG can be added to form the final tree group. The problem with forming tree groups incrementally, however, is that reoptimization may be needed each time a new path is added to the tree group.

4.4 Optimization Framework

In the following sections, we begin with traces and superblocks and consider ways of optimizing code within these large translation blocks. In this section, we first discuss some of the overarching issues that are common to all the optimization methods.

In general, dynamic optimization, as discussed in this chapter, differs from static compiler-based optimization in that the optimizer operates only over well-delineated, "straight-line" code regions, e.g., traces or superblocks. Another difference is that high-level semantic information from the original program is not available; information must be extracted from the executable binary. For example, data structure declarations are unavailable. The overall philosophy is that fast, low-overhead optimizations should be used to gather the "low-hanging fruit." In contrast, dynamic optimization for HLL virtual machines (**Chapter 6**) is much less restricted. In HLL VMs, there is more semantic information available to the optimizer, and the scope of optimization may be much larger than with the optimizations discussed in this chapter.

Dynamic optimizations are performed in addition to any optimizations the original compiler may have done. Because optimization is being performed at run time, however, there are new optimization opportunities that may not have been available to the static compiler. In general, these new opportunities involve optimizations along frequently followed paths that cross basic block boundaries. For example, when consecutive blocks are considered, redundant instructions may be found and removed, or instructions may be reordered across basic block boundaries.

4.4.1 Approach

Figure 4.25 illustrates the overall approach to optimization. Based on profile information, basic blocks are first collected to produce a straight-line code sequence that forms a trace or superblock. These instructions are converted into an intermediate form and placed in a scheduling/optimization buffer by the translation software. The intermediate form contains essential dependence information but would typically use a single assignment format (Cytron et al. 1991) so that nonessential dependences do not appear. In order to simplify eventual target code generation, the intermediate form is similar to the one that a compiler might use for the target instruction set. Then code scheduling and optimization take place, followed by register assignment. As the code is optimized, side table information is also generated. For example, a side table may be used for tracking the relationships between source register values and target register values in order to implement precise state recovery in the event of a trap or interrupt; this is explained later. In some cases, compensation code is added at the intermediate entry and exit points of the scheduled, optimized trace (Figure 4.26). In practice, these entries and exits will be infrequently used, but, as we shall see, the compensation code is still required for correctness in the event they are used.



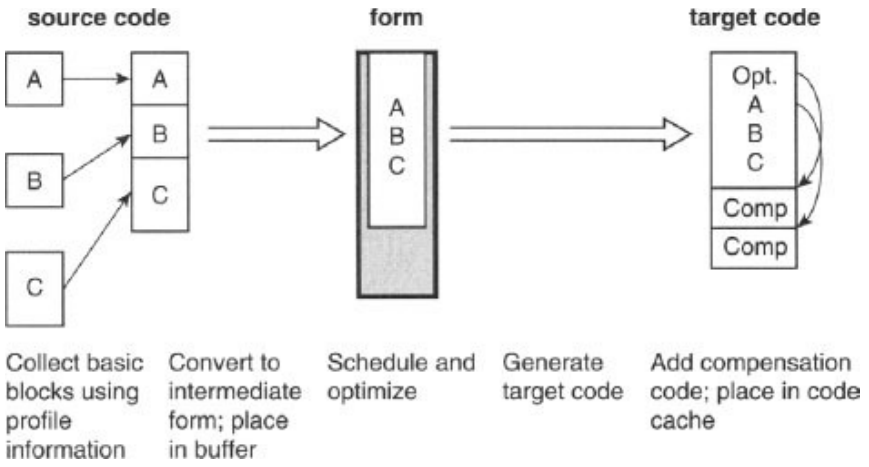


Figure 4.25 Major Steps in the Scheduling/Optimization Process.

An example of an interbasic block optimization that uses dynamic information was given at the beginning of this chapter (**Figure 4.2**), where the knowledge that the most common control path went from block A to block C allowed formation of a superblock containing basic blocks A and C. Then as an optimization, the assignment to R1 in block A was removed. This resulted in compensation code being added at the exit to block B.

4.4.2 Optimization and Compatibility

An important consideration during the optimization process is maintaining compatibility. In **Section 3.2** we discussed compatibility issues in general. To reiterate, for practical process VMs, we maintain an isomorphic relationship between the guest VM and the native platform (**Figure 3.4**). As part of this relationship, we first assume that all control transfer points (a system call, return, trap, or interrupt) between the guest process and the OS on its native platform will map to a corresponding transfer of control between the guest process and the virtual machine runtime. Furthermore, at the time of one of these transfers to the runtime, it should be possible to reconstruct the entire guest process's state, both registers and memory. With respect to optimizations, it is the traps that we are most concerned with, both the control transfer aspect and the state reconstruction aspect, i.e., forming the precise state. We address both of

struction aspect, i.e., forming the precise state. We address both of these in the following paragraphs.

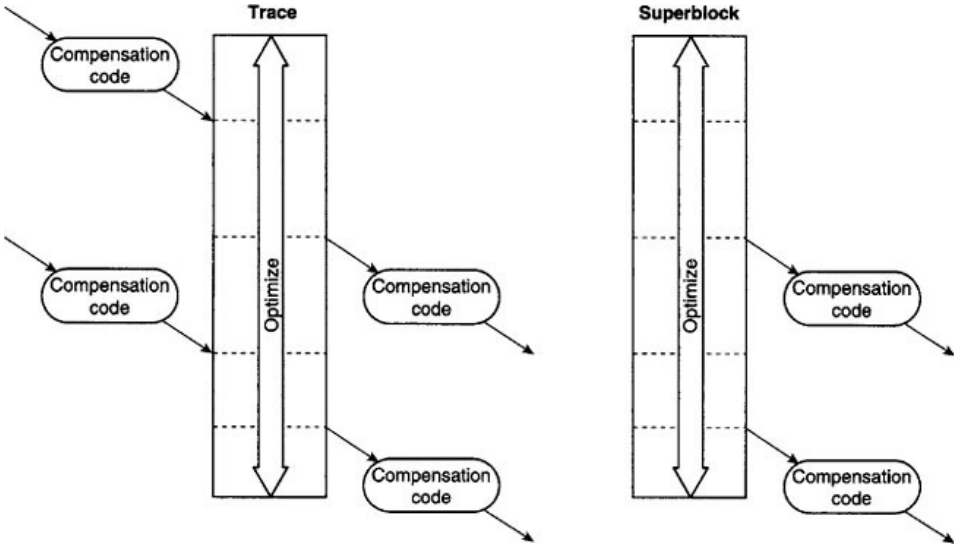


Figure 4.26 Optimization of Traces and Superblocks. *A trace (left) and a superblock (right) are collections of basic blocks (shown with dotted lines) that are combined using commonly executed program paths. Optimizations can be performed across the entire trace or superblock, with compensation code being added for the uncommon intermediate entrances and exits.*

We define a process VM implementation to be *trap compatible* if any trap (except a page fault) that would occur during the native execution of a source instruction is also observed during the emulation of the corresponding translated target instruction(s). And the converse is also true: Any trap that is *observed* during execution of a target instruction should also occur in the corresponding source instruction. We exclude page faults because they are the result not of an action of the running process but of the allocation of resources by the host OS. Page fault compatibility is treated in **Chapter 7** in the context of codesigned VMs that are system VMs.

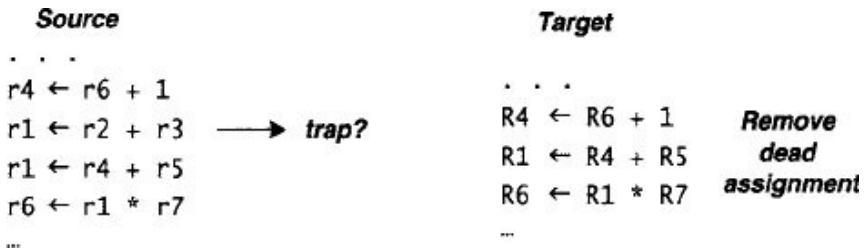


Figure 4.27 A Code Optimization That Removes an Apparently Redundant Instruction. *If the removed instruction would have trapped due to overflow, however, no trap will occur in optimized version*

optimized version.

Consider the example on the left side of **Figure 4.27**. In this example, the first instruction of the source code (on the left) computes r1, which turns out to be a "dead" assignment, because the next instruction does not read r1 but overwrites it. This apparent redundancy may have occurred as a by-product of forming dynamic translation blocks. An obvious optimization would be to remove the instruction making the redundant assignment (r1 ← r2 + r3). However, this could violate trap compatibility as we have defined it because the removed instruction may trap due to overflow. That is, if the instruction is removed, an overflowing instruction in the original source code is skipped, and the program's execution after optimization is not exactly the same as before the optimization.

With respect to the converse condition for trap compatibility given earlier, it may sometimes be the case that a trap will occur due to the execution of a target instruction that would not have occurred during execution of the source program. However, if the runtime is able to detect these traps and not make them visible to the guest program, then the converse condition still holds. That is, the trap is not "observed" by the guest program because the runtime filters out any spurious traps.

At the time of a trap, the memory and register state corresponding to a particular point in the guest program's execution (as identified by the trap/interrupt PC) becomes visible. *Memory* and *register state compatibility* are maintained if the runtime can reconstruct them to the same values they would have had on a native platform. The requirement of supporting state compatibility can have a significant effect on the optimizations that can be performed. For example, in **Figure 4.28** (on the left), it might be beneficial to reorder the instruction that computes r6 to a point higher in the code sequence to allow its execution to be overlapped with independent instructions (because multiplications typically take several cycles to execute). In

this example, the multiply instruction can be overlapped with the add instruction that writes to r9. The rescheduled code is shown in the middle of the figure. However, if the add that produces R9 should overflow and trap, then the value in R6, as seen by the trap handler, will not be the same as in the original code sequence.

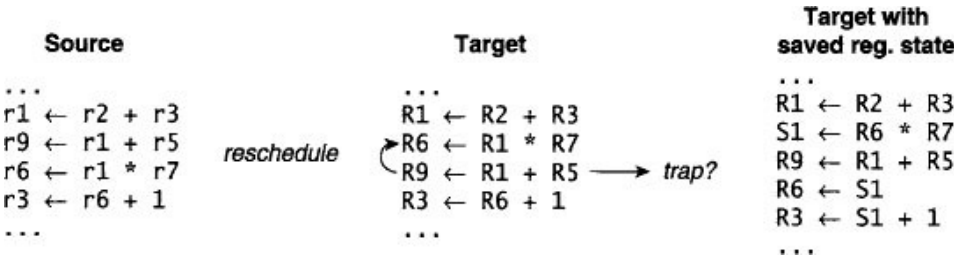


Figure 4.28 Code That Is Reordered to Overlap Execution of a Multiplication with an Independent Instruction. *If the multiplication overflows, the state observed by trap handler is changed.*

Register compatibility is easier to maintain than memory state compatibility because register values can be backed up (or checkpointed) more easily. This is illustrated in the rightmost code sequence of **Figure 4.28**. Here, the code is reordered, but the result of the multiplication is initially placed into a scratch register, S1. The value in S1 is then copied into register R6 at the same place R6 would have been updated in the original code. If an overflow trap should occur, the runtime can recover the original state from S2, using techniques to be described in **Section 4.5**.

In the case where *intrinsic compatibility* (see **Section 3.2.1**) is required, then optimizations should be restricted to those that make all register and memory state recoverable at the time any potentially trapping instruction is emulated. However, if the developer of the VM software can make some assumptions about the producer of a binary—for example, that certain types of bugs are not present, that a particular compiler will used for the source binary, that only certain trap handlers will be used, or that some traps will never be enabled—then additional optimizations can be performed while still providing *extrinsic compatibility* under the given assumptions. We will return to the topic of compatibility in **Section 4.6.2**, after spe-

cific optimizations have been described.

4.4.3 Consistent Register Mapping

Initially, we assume there are always sufficient target registers available to map source register values to target registers and to accommodate any optimizations that may be performed. This may require that there be more available target registers than original source registers. This is not an unreasonable assumption, however, because the IA-32 is a commonly used source ISA, and the target ISA is often a RISC ISA, with many more registers than an IA-32. The relative numbers of source and target registers are an important consideration, however, and this issue was discussed in **Section 3.3.1**.

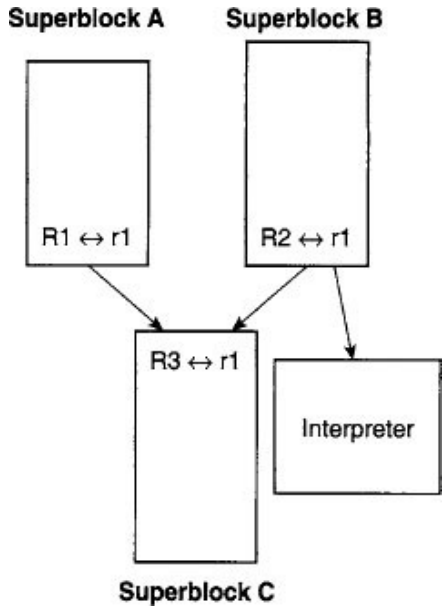


Figure 4.29 An Inconsistent Register Mapping. *When one superblock branches to another, the source-to-target register mapping must be correctly managed. Furthermore, when jumping to the interpreter, the interpreter must know the specific mapping being used so that it can correctly update its register context block in memory.*

Given a sufficient number of target registers, we assume that individual source registers can be permanently mapped to target registers. Although in theory a less permanent mapping could be done on a per-translation-block basis, such a flexible mapping method leads to complications when a transition is made from one translation

block to another or from a translation block to the interpreter. This is illustrated in the following example.

Figure 4.29 shows a situation where two different translated superblocks (A and B) branch to the same translated superblock, C. If superblocks A and B use different register mappings (e.g., one uses R1 for source register r1 while the other uses R2), then one (or possibly both) of them may be inconsistent with the mapping in the common destination superblock, C. Consequently, extra register copy instructions must be inserted at the end of superblocks A and B to make their mappings consistent with the destination superblock, C. A similar situation occurs if superblock B jumps to the interpreter because the conditional branch at its end is not taken. When interpreter is entered, then the runtime must copy target register values into the register context block the interpreter keeps in memory. If the register mapping is not consistent among translation blocks, then a side table must be provided to assist the runtime with the copying. The table would indicate the specific mapping used by each translated superblock.

To simplify the process, often a fixed source-to-target register mapping is maintained at all translation block boundaries, with the mapping being more flexible within the superblock boundaries. In some cases, this may cause additional register copy instructions at the time a translation block is exited. It must be noted, however, that such a fixed mapping is not necessary—through careful bookkeeping at the time translation is performed, the translator (with the assistance of side tables) can make sure that source and destination mappings always match when a transition is made; this process will be explored in **Section 4.6.3**.

4.5 Code Reordering

An important optimization performed in a number of virtual ma-

chine applications is code reordering. In many microarchitectures, performance is affected by the order in which instructions are issued and executed. The most significant examples are simple pipelined microarchitectures that execute instructions strictly in program order. This was done in many of the early RISC processors and is still done in a number of embedded processors. Another example is VLIW processors that expect the compiler to reorder instructions so that several independent instructions can be packed together within the same long instruction word. The Intel IPF (formerly IA-64) implementations (i.e., Itanium) are modern high-performance processors that take this approach. Finally, in many cases, performance of dynamic out-of-order superscalar processors can benefit from code reordering because of variations in functional unit latencies, especially cache memory latencies.

Because of its importance, we will consider code reordering before looking at other optimizations. Code reordering is a fairly easy optimization to understand, and several important issues can first be discussed with respect to reordering and later extended to other types of optimizations.

4.5.1 Primitive Instruction Reordering

The key issues related to code reordering are easier to understand if we first consider reordering pairs of instructions. This will enable us to see the steps that are required for implementing precise traps or for adding compensation code at side entry and exit points to ensure correctness. This pairwise-reordering approach may seem a little pedestrian, but it is conceptually useful for understanding the important issues. By considering reordering in this manner, we are reducing the problem to “primitives” from which more complex algorithms are built.

The types of reordering allowed and any required compensation code, i.e., the rules for reordering, depend on the types of instruc-

code, i.e., the rules for reordering, depend on the types of instructions involved. We therefore divide instructions into four categories.

Register updates, denoted as "reg"—instructions that produce a register result, i.e., that change the register portion of the architected state. This category typically includes load instructions as well as ALU/shift-type operations that write to registers. These instructions have the property that their state modification(s) can be "undone," e.g., by saving about-to-be-overwritten values in other target registers or spilling them to a region of memory. The saved state may be required in the event that a reordered instruction traps and the runtime needs to recover the precise source ISA state.

Memory update instructions, "mem"—instructions that place a value in memory; i.e., that change the portion of architected state residing in main memory. In most instruction sets, only memory store instructions are in this category. A key property is that the updating of the memory state cannot be easily undone. In general, any instruction whose modification to the process state cannot be undone should be put in this category. This includes memory accesses to volatile locations (described below).

Branch instructions, "br"—a branch instruction that determines control flow but does not produce any register or memory result. Branches cannot trap.

Join points, "join"—the points where a jump or branch target enters the code sequence (i.e., the trace). Although these are not instructions strictly speaking, they can affect scheduling when instructions are moved above or beyond join points.

As an example, the instructions in **Figure 4.30** have been designated with the category to which they belong. In this and other examples, we use a register transfer notation to make the various instruction dependences easier to see.

Accesses to volatile memory locations—A *volatile* memory location is one that may be accessed by an entity other than the process at hand, for example, by another process or thread in a multiprocessing system or by the I/O system if memory-mapped I/O is being used. Very often, there are severe constraints on optimizations when volatile memory locations are involved. For example, instructions accessing volatile locations cannot be removed, and in many cases they may not be reordered. More discussion of memory reordering in multiprocessor VMs is in **Chapter 9**. For our analysis of code reordering, it is best to categorize volatile memory accesses as "mem"—even if they are not stores. For example, in **Figure 4.30** if the first two load instructions are to volatile memory locations, then they should each be categorized as "mem" rather than "reg." A final comment is that in some VM applications, the emulation system does not know *a priori* whether memory locations are volatile and must then make very conservative assumptions about optimizations involving any memory operations.

```
...
R1 ← mem(R6)          reg
R2 ← mem(R6 +4)       reg
R3 ← R1 + 1           reg
R4 ← R1 << 2          reg
Br exit if R7 == 0    br
R7 ← R7 + 1           reg
mem (r6) ← R3         mem
```

Figure 4.30 Code Sequence with Instructions Denoted According to Scheduling Category.

We now consider scheduling situations in terms of the foregoing instruction categories. There are 16 different cases; i.e., both instructions in a pair are one of the four types. A few of the 16 cases are degenerate, however; e.g., reordering two join points has no real effect. The focus of this discussion will be the requirements of maintaining a consistent register assignment and implementation of precise traps (to be discussed in greater detail in **Section 4.5.2**).

To provide some overall organization, we consider first code movement involving branches, then code movement around join points,

and finally code movement in straight-line code.

The first two cases we consider are illustrated in **Figure 4.31**a. These involve moving instructions from above to below a branch exit point. Instructions that change the architected state, whether writing to a register or to memory, can be moved below a conditional branch that exits a trace or superblock. When this is done, compensation code is added at the exit point to duplicate the update performed by the instruction being moved. This ensures that if an early exit is taken, the state is the same as in the original code sequence. For example, in **Figure 4.31**b, a shift instruction is moved after a conditional branch. In the event that the branch is taken, a duplicate of the reordered instruction is placed on the target path as compensation code.

The cases where instructions are moved above a conditional branch are illustrated in **Figure 4.32**a. A register update instruction can be moved ahead of a conditional branch as shown, but special provisions must be made for supporting precise traps. If a register instruction writes to register R in the original sequence, the old value in R must be maintained, at least until the flow of execution reaches the original position of the register update instruction (i.e., after the branch). One way to do this is to place the new value into a temporary register, T (thus saving the old value of R) and then later copying the new value into R. This is illustrated in the first two code sequences of **Figure 4.32**b. Here, a multiply instruction is moved above the branch. The result of the multiply instruction is held in register T1, and then later (after the branch) it is copied into the original destination register, R6. In many cases, the explicit copy instruction may not be needed. For example, in the rightmost column of **Figure 4.32**b, the explicit copy is removed. Temporary register T1 is still needed to prevent overwriting the value in R6 in the event that the branch is taken. The key point is that at every instruction in the sequence (including at the branch exit), the correct value of R6 is available, either in R6 itself or in some other register. In other

is available, either in R6 itself or in some other register. In other words, in the target code, the *live range* of the original value in register R6 is extended at least to the point where it would have been overwritten in the original code sequence.

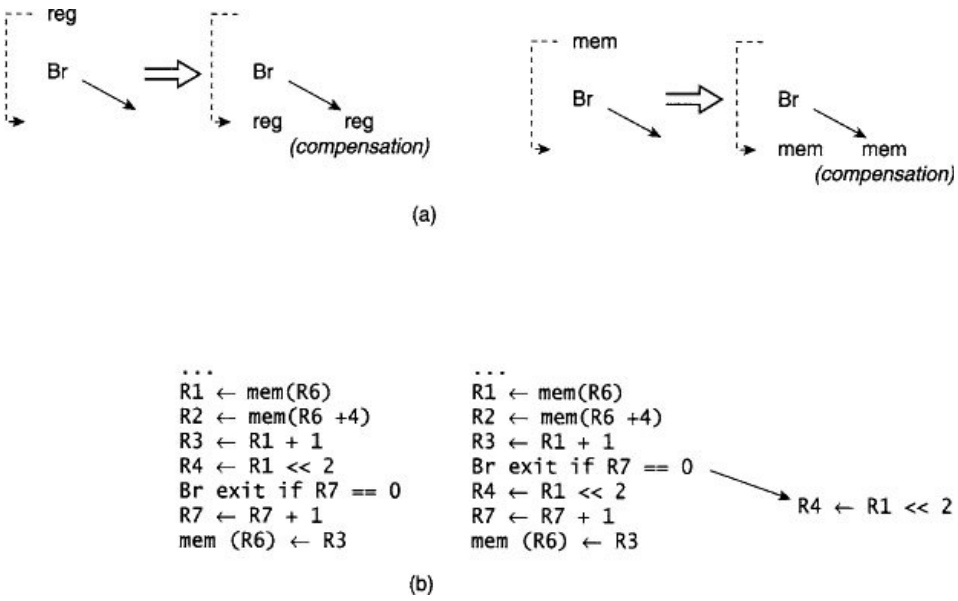


Figure 4.31 Moving Instructions Around a Conditional Branch Instruction. *(a) Two cases are shown: moving a register update instruction below a branch and moving a store below a branch. In both cases, compensation code is required. (b) An example of a register update instruction (R4 ← R1 < 2) being moved below a conditional branch; a duplicate is used as compensation code on the taken branch path.*

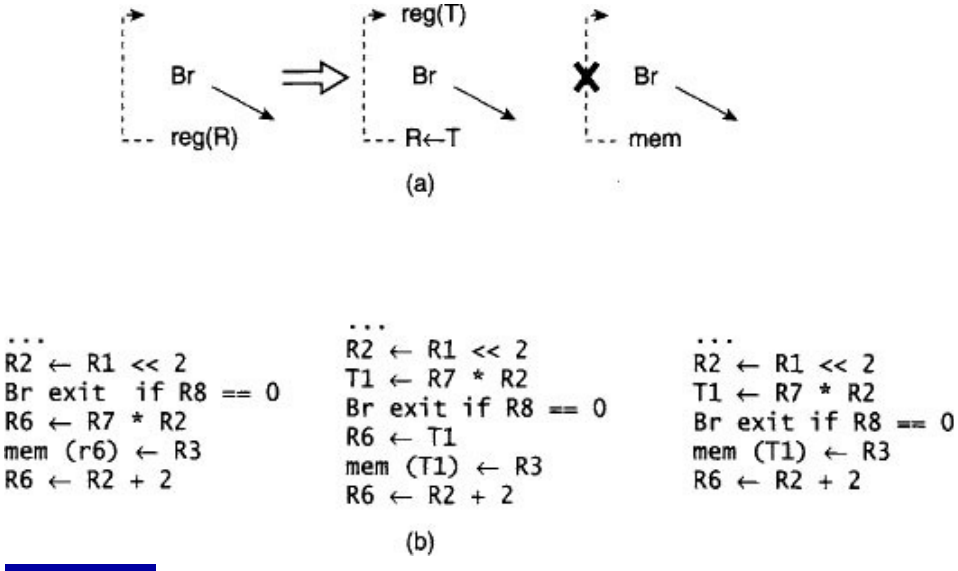


Figure 4.32 Moving Instructions from Below a Conditional Branch to Above. *(a) A register update instruction can be moved above a branch instruction as long as the old value remains available until after the branch. This may require the update value to be placed in a temporary register in case the branch exit is taken. A mem update (store) instruction cannot be moved above a branch. (b) An example where a multiply instruction (R6 ← R7 * R2) is moved above a conditional branch.*

Also, as illustrated in the rightmost code sequence of **Figure 4.32**a, a mem instruction cannot be moved ahead of the branch because we are assuming that once a mem instruction writes to memory, the previous memory value is unrecoverable. Hence, it would not be possible to back up the old value as was done with the register instruction.

We now turn to cases where an instruction is moved up past a join point (found in traces but not superblocks; there are no entry points in superblocks except at the top). This is the "dual" situation to the foregoing cases, where an instruction was moved past a branch exit point.

As shown in **Figure 4.33**a, both types of state-updating instructions, reg and mem instructions, can be moved above a join point. The compensation code is a duplicate of the moved instruction. The compensation guarantees that the state is correctly updated, regardless of the way the block is entered. Also, note that the moved instruction is actually the one at the join point, and the join point essentially moves down one instruction. Consequently, no actual code movement takes place; only the target of the join point is changed. This scheduling operation is useful only if additional upward code movements can be made. **Figure 4.33**b is an example where a load instruction at a join point is moved above the join point. The case where a mem (store) instruction is moved above a join point is similar.

Moving state-updating instructions (register updates or stores) below a join point will cause the state to be updated regardless of the entry path. In all but a few very special situations, this will result in an incorrect state update if the join path is used. Consequently, we will not consider rescheduling that involves this type of code motion.

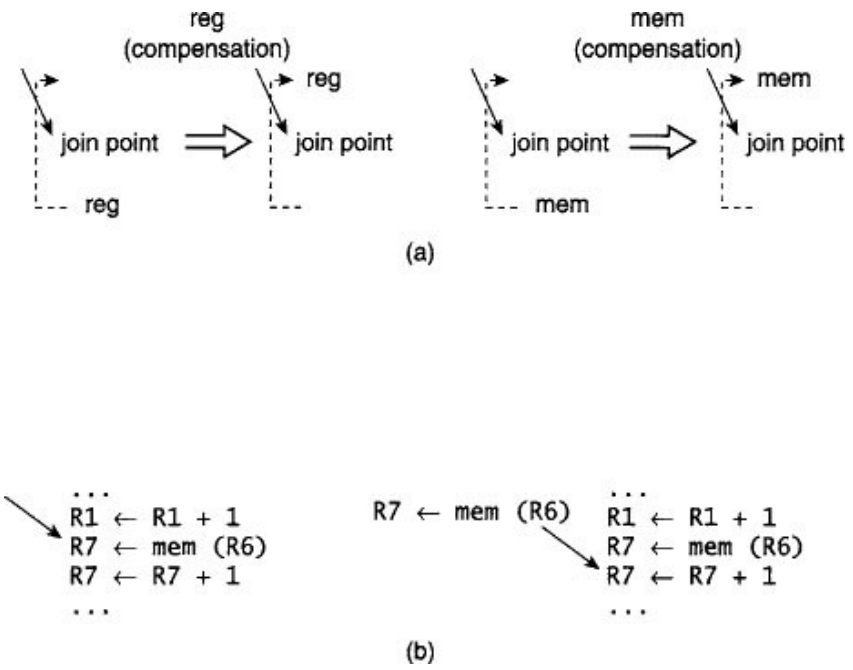
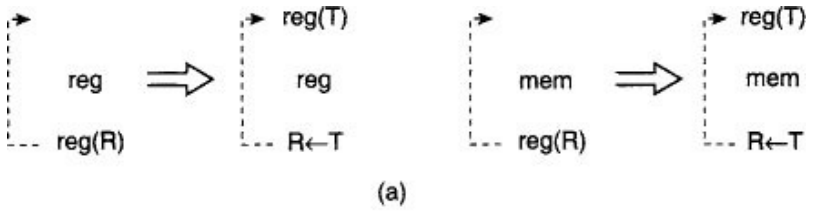


Figure 4.33 An Instruction Is Moved Above a Join Point. *(a) A register update or a mem instruction (a store) can be moved above a join point; if so, compensation is added by performing a duplicate instruction on the join path. (b) An example where a load instruction is moved above a join point.*

Now we turn to code movement in straight-line regions of code. First consider instructions that update registers as shown in **Figure 4.34**a. This case is similar to moving a register instruction above a branch point. When the instruction is moved, it initially updates a temporary register, T. Then later, at the point where it would have updated the register in the original sequence, the temporary register can be copied into the architected register. Of course, as was the case earlier, if the value is overwritten before the block of translated instructions ends, then the copy instruction is not needed. An example is in **Figure 4.34**b. If a memory-update instruction should cause a trap, then the state can be restored to that prior to the mem instruction in the original sequence, and interpretation can then be used to emulate execution in the original program order. In this case the trap will occur again and the state will be correct.

Other types of straight-line code reordering lead to problems, most notably moving a store above any other instruction that modifies the state. As noted earlier, the store cannot be “undone,” so if there

is a trap caused by the instruction that precedes it in the original schedule (but follows in the final schedule), the precise state cannot be restored.



(a)

```
...
R1 ← R1 * 3
mem(R6) ← R1
R7 ← R7 << 3
R9 ← R7 + R2
...
```

```
...
R1 ← R1 * 3
T1 ← R7 << 3
mem(R6) ← R1
R7 ← T1
R9 ← T1 + R2
...
```

(b)

Figure 4.34 Code Movement in Straight-Line Code Sequences. *Register updating instructions can be moved above potentially trapping instructions, but the computed value is held in a temporary register, in case a trap does occur.*

Summary

The various types of code reordering are given in **Table 4.1**. The columns of the table are the instruction types that are the first instruction of a pair to be reordered and the rows are the second. The table entries indicate the action that can be taken: whether reordering is allowed, and, if so, the action to be taken when reordering is performed. These actions permit precise state recovery when traps occur. We observe that reordering is always possible if a register instruction is the second of the pair; i.e., a register instruction can be moved "up" in a schedule arbitrarily (as long as data dependences are not violated). Other allowed code motions involve the movement of reg or store instructions from above to below branches, and the movement of a store from below to above a join point (with proper compensation).

4.5.2 Implementing a Scheduling Algorithm

Now that we have discussed scheduling primitives, we consider a complete code-scheduling algorithm. The algorithm is based on one given by Bich **Le (1998)** and schedules code that has been organized into superblocks. For handling precise traps, register assignments are performed to maintain extended live ranges so that if a trap occurs, the register state can always be backed up to some point prior to the trapping instruction. Then interpretation of the original source code can establish whether the trap is in fact real and will provide the correct state if it is. The example given here translates from an IA-32 instruction sequence to a PowerPC sequence. In the example, maintaining IA-32 condition codes is initially ignored; then at the end of this section is a discussion of condition code handling.

TABLE 4.1 Instruction Reordering by Type

	First			
Second	`reg`	`mem`	`br`	`join`
`reg`	Extend live range of reg instruction	Extend live range of reg instruction	Extend live range of reg instruction	Add compensation code at entrance
`mem`	Not allowed	Not allowed	Not allowed	Add compensation code at entrance
`br`	Add compensation code at branch exit	Add compensation code at branch exit	Not allowed (changes control flow)	Not allowed (changes control flow)
`join`	Not allowed (can only be done in rare cases)	Not allowed (can only be done in rare cases)	Not allowed (changes control flow)	No effect

Step 1: Translate to Single-Assignment Form

The instructions belonging to the superblock are translated to the target ISA and placed in a rescheduling buffer. As the instructions are placed in the instruction buffer, they are put in single-assignment form (a register is assigned a new value only once). To help with maintaining the consistent register mapping, incoming register values are mapped to their consistent target registers. Any new values generated in the superblock are placed in temporary

"registers," labeled ti, in order to maintain the single-assignment discipline.

Original Source Code		Translated in Scheduling Buffer	
add	%eax,%ebx	t5 ←	r1 + r2, set CR0
bz	L1	bz	CR0, L1
mov	%ebx,4(%eax)	t6 ←	mem(t5 + 4)
mul	%ebx,10	t7 ←	t6 * 10
add	%ebx,1	t8 ←	t7 + 1
add	%ecx,1	t9 ←	r3 + 1, set CR0
bz	L2	bz	CR0, L2
add	%ebx,%eax	t10←	t8 + t5
br	L3	b	L3

Step 2: Form Register Map

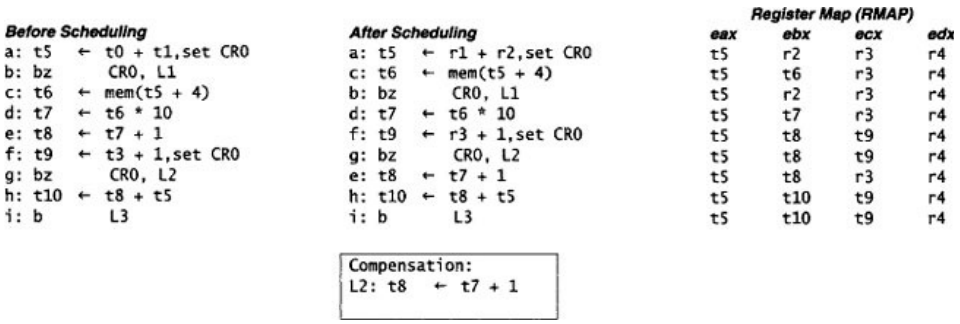
A register map (RMAP) is generated to track the values as assigned in the original source code. For each of the IA-32 source registers, eax, ebx, etc., the RMAP keeps track of the single-assignment register that holds the corresponding value at any given point in the code sequence.

Original Source Code		Single Assignment Form		Register Map (RMAP)			
				eax	ebx	ecx	edx
add	%eax,%ebx	t5 ←	r1 + r2, set CR0	t5	r2	r3	r4
bz	L1	bz	CR0, L1	t5	r2	r3	r4
mov	%ebx,4(%eax)	t6 ←	mem(t5 + 4)	t5	t6	r3	r4
mul	%ebx,10	t7 ←	t6 * 10	t5	t7	r3	r4
add	%ebx,1	t8 ←	t7 + 1	t5	t8	r3	r4
add	%ecx,1	t9 ←	r3 + 1, set CR0	t5	t8	t9	r4
bz	L2	bz	CR0, L2	t5	t8	t9	r4
add	%ebx,%eax	t10←	t8 + t5	t5	t10	t9	r4
br	L3	b	L3	t5	t10	t9	r4

Step 3: Reorder Code

The instructions in the intermediate form are reordered. As the code is reordered, the rows of the RMAP are reordered along with the corresponding instructions. Here, the instructions are denoted with labels a:, b:, c:, etc. to make their identification in the rescheduled code easier. In this example, the load instruction, *c,* is moved up above branch *b* to give the load a longer time to complete before its output value is needed. Similarly, the add instruction, *e,* which depends on the multiply, *d,* is moved downward, below the branch that follows it. As explained earlier, the movement of the add in-

struction below the branch requires compensation code to be added at exit label L2.



Before Scheduling

```
a: t5  ← t0 + t1,set CR0
b: bz    CR0, L1
c: t6  ← mem(t5 + 4)
d: t7  ← t6 * 10
e: t8  ← t7 + 1
f: t9  ← t3 + 1,set CR0
g: bz    CR0, L2
h: t10 ← t8 + t5
i: b     L3
```

After Scheduling

```
a: t5  ← r1 + r2,set CR0
c: t6  ← mem(t5 + 4)
b: bz    CR0, L1
d: t7  ← t6 * 10
f: t9  ← r3 + 1,set CR0
g: bz    CR0, L2
e: t8  ← t7 + 1
h: t10 ← t8 + t5
i: b     L3
```

Register Map (RMAP)

eax	ebx	ecx	edx
t5	r2	r3	r4
t5	t6	r3	r4
t5	r2	r3	r4
t5	t7	r3	r4
t5	t8	t9	r4
t5	t8	t9	r4
t5	t8	r3	r4
t5	t10	t9	r4
t5	t10	t9	r4

Compensation:

```
L2: t8  ← t7 + 1
```

Step 4: Determine Checkpoints

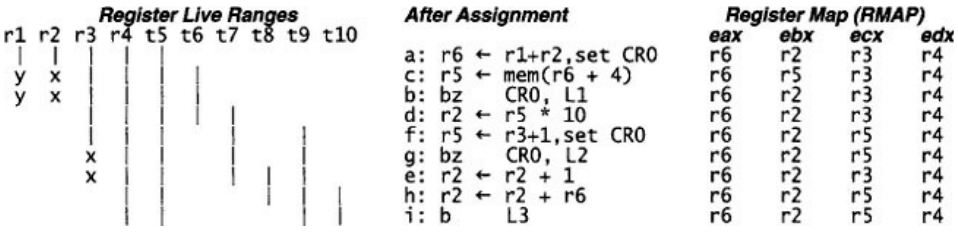
Instruction checkpoints are determined. These are used when an instruction traps or there is an early exit branch from the superblock. When a trap occurs, it must be possible to back up to some earlier point where a precise state can be recovered so that interpretation can proceed forward. These *checkpoints* are places where *all* instructions in the original sequence have been completed up to that point. To find these points, we consider an instruction to be "committed" if all preceding instructions in the original sequence have been completed. The points at which instructions commit are listed here. Then, for each instruction, the checkpoint (in other words, the backup point if it traps) is the closest instruction that has been committed. These are shown in the checkpoint column in the example. When the superblock is entered, the initial register mapping is the checkpoint, denoted as @.

After Scheduling	Register Map (RMAP) eax	ebx	ecx	edx	Commit	Checkpoint
a: t5 ← r1 + r2,set CR0	t5	r2	r3	r4	a	@
c: t6 ← mem(t5 + 4)	t5	t6	r3	r4		a
b: bz CR0, L1	t5	r2	r3	r4	b,c	a
d: t7 ← t6 * 10	t5	t7	r3	r4	d	c
f: t9 ← r3 + 1,set CR0	t5	t8	t9	r4		d
g: bz CR0, L2	t5	t8	t9	r4		d
e: t8 ← t7 + 1	t5	t8	r3	r4	e,f,g	d
h: t10 ← t8 + t5	t5	t10	t9	r4	h	g
i: b L3	t5	t10	t9	r4	i	h

Step 5: Assign Registers

Registers are assigned in a more or less conventional way, except for the way live ranges are determined. For determining live ranges, we need to consider the normal live ranges in the rescheduled instruction sequence and extended live ranges that are necessary for precise state recovery if there is a trap. The extended live ranges are based on the checkpoints. For each instruction that is a checkpoint, the live ranges of registers in its register mapping must be extended to the last point where it is used as a checkpoint. For example, instruction *d* serves as a checkpoint until instruction *e* appears in the reordered sequence. Registers t5, t7, r3, and r4 are in the register map for instruction *d*. Consequently, these registers must be kept live until instruction *e* appears in the reordered sequence. This means that if instruction *e* should trap, say, due to an overflow, then the register values at instruction *d* will be available and can be restored.

Actually, live range extension can be relaxed somewhat. We need to maintain checkpoints only for instructions that can trap or branch out of the superblock, because these are the only points where it is necessary that correct source state be available. In our example, the places where live ranges have been extended are marked with an "x."



Register Live Ranges										After Assignment	Register Map (RMAP)			
r1	r2	r3	r4	t5	t6	t7	t8	t9	t10		eax	ebx	ecx	edx
|	|	|	|	|						a: r6 ← r1+r2,set CR0	r6	r2	r3	r4
y	x	|	|	|	|					c: r5 ← mem(r6 + 4)	r6	r5	r3	r4
y	x	|	|	|	|					b: bz CR0, L1	r6	r2	r3	r4
		|	|	|	|	|				d: r2 ← r5 * 10	r6	r2	r3	r4
		|	|		|		|		f: r5 ← r3+1,set CR0	r6	r2	r5	r4
	x	|	|		|		|		g: bz CR0, L2	r6	r2	r5	r4
	x	|	|		|	|	|		e: r2 ← r2 + 1	r6	r2	r3	r4
		|	|			|	|	|	h: r2 ← r2 + r6	r6	r2	r5	r4
		|	|				|	|	i: b L3	r6	r2	r5	r4

After live ranges are determined, then registers are assigned as shown. Because the PowerPC has many more registers than the IA-32, there are plenty of registers available for the assignment. During assignment, it should be an objective for the consistent register mapping to be in place when the trace or superblock is exited.

Thus, the RMAP of the last instruction should, as much as possible, reflect the consistent register mapping. Copy instructions, if needed, are then added to force the consistent mapping. Similarly, if a side exit should be taken, it may be necessary to add compensation code in order to restore the RMAP.

Step 6: Add Compensation Code

Compensation code is added. In the example, compensation code is needed because instruction *e* was moved below branch *g*. Consequently, the same operation (r2 ← r2 + 1) must be added at the target of branch *g*. In addition, register r5 must be copied to register r3 in order to restore the consistent register mapping. The final translated and reordered PowerPC code is shown here on the right.

After Assignment

```
a: r1   ← r1+r2,set CR0
c: r5   ← mem(r1 + 4)
b: bz     CR0, L1
d: r2   ← r5 * 10
f: r5   ← r3+1,set CR0
g: bz     CR0, L2
e: r2   ← r2 + 1
h: r2   ← r2 + r1
i: b      L3
```

Compensation Code Added

```
a: r1   ← r1+r2,set CR0
c: r5   ← mem(r1 + 4)
b: bz     CR0, L1
d: r2   ← r5 * 10
f: r5   ← r3+1,set CR0
g: bz     CR0, L2'
e: r2   ← r2 + 1
h: r2   ← r2 + r1
i: b      L3
   r3   ← r5

L2':r3  ← r5
    r2  ← r2 + 1
```

PowerPC Code

```
a:  add.   r1,r1,r2
c:  lwz    r5, 4(r1)
b:  beq    CR0, L1
d:  muli   r2,r5,10
f:  addic. r5,r3,1
g:  beq    CR0, L2'
e:  addi   r2,r2,1
h:  add    r2,r2,r1
i:  b      L3
    mr     r3,r5

L2':mr     r3,r5
    addi   r2,r2,1
```

Precise State Recovery

Continuing with our example, when a trap occurs, the runtime first finds the trapping superblock and corresponding source basic blocks (possibly with the aid of side tables). It then reconstructs the RMAP at the checkpoint for the trapping instruction. It does this by retranslating the original source basic blocks and then performing rescheduling and register assignment. At that point, the runtime can use the reconstructed RMAP to reset the register values corresponding to the trapping instruction's checkpoint. The checkpoint algorithm guarantees that this can be done. Then the runtime can begin interpreting the original source code at the checkpoint.

For example, say that instruction *d* traps. Then the runtime needs to back up to a precise state with respect to the original source sequence. Hence, it backs up to instruction *d*'s checkpoint. In this case the checkpoint is instruction *c*. When the RMAP entry for *c* is reconstructed, the registers eax, ebx, ecx, and edx map to r1, r5, r3, r4, respectively. Therefore, the runtime restores these values to the mapped IA-32 registers and begins interpreting forward, beginning with instruction *c*. The trap will recur when instruction *d* is interpreted, and the state at that time will be correct.

As another example, assume that instruction *c* traps but the branch at *b* will be taken (and will exit the superblock prior to *c* in the original code). In the translated target code, instruction *c* executes and traps. The checkpoint for *c* is instruction *a*. The registers r1, r2, r3, and r4 hold the register state at instruction *a*. Consequently, if interpretation starts at instruction *a* with these values, the branch will be taken, and the trap does not occur. Eventually, as interpretation proceeds, a source PC will map to a translated superblock in the code cache, and the emulation will jump back into the code cache.

As an alternative to interpretation, one can also maintain a simple (not reordered or optimized) binary translation off to the side. Then rather than interpreting after the state has been restored, the runtime software can branch into the simple, in-order translation. This faster method for handling traps requires more memory and can be applied to those specific cases where a given instruction traps repeatedly.

Finally, a complementary method for precise state reconstruction is to add *repair code* (Gschwind and Altman 2000), executed by the runtime after a trap occurs. Repair code is similar to the compensation code placed at the side exits of a superblock. However, in the case of precise trap implementations, the "side exit" is actually a trap. The concept is illustrated in **Figure 4.35**, which is taken from

the example in **Figure 4.28**. Here, the code sequence is reordered, with repair code (the instruction R9 ← R1 + R5 in this example) being recorded in a side table. Then if the multiply instruction should trap, the repair code is executed by the runtime trap emulator to generate the correct precise value for register R9.

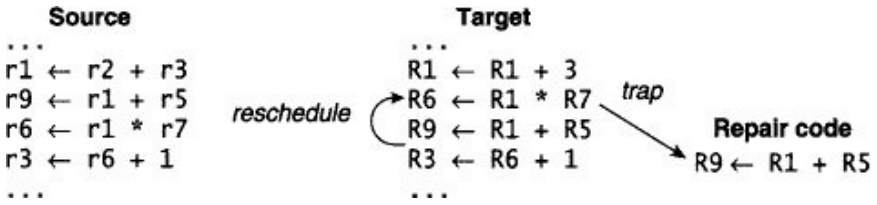


Figure 4.35 Using Repair Code to Reconstruct Precise State After a Trap. *The instruction sequence is reordered, with repair code being generated at a potential trap (exit) point. If the multiply instruction should happen to trap, repair code executed by the runtime trap handler recovers the precise value for register R9.*

Condition Code Handling

For handling condition codes, an efficient method is to use lazy evaluation, as described in **Section 2.8.2**. With lazy evaluation, condition codes are only evaluated when needed. For the purposes of conditional branch evaluation, data flow analysis can be performed within a translation block to determine which condition codes will actually be needed by the executing code. In the extended reordering example, only two instructions set condition codes that are potentially used within the translation block (instructions *a* and *f*), so these are the only instructions where condition codes are actually evaluated; but note that even then, the target ISA (PowerPC) versions of the condition codes are generated, not the source ISA (IA-32) versions.

A more difficult problem occurs when an interrupt or trap is triggered and condition codes must be materialized in order to provide the precise source ISA state. The basic method is similar to the one used for providing precise register state. More specifically, lazy evaluation can be implemented by extending the live ranges of the input operands to condition code-setting instructions. If needed, these op-

erands are used by the runtime's trap/interrupt routine to materialize the condition codes.

In the foregoing code-reordering algorithm, "condition code checkpoints," similar to the regular checkpoints, are maintained. The only difference is that these checkpoints are restricted to the committed *condition code-setting* instructions. Then the condition code checkpoints extend the live ranges of the operands that feed into a condition code-setting instruction. In the example, instructions *a, d, f, e,* and *h* correspond to IA-32 source code operations that set condition codes. In the example, the condition code checkpoints are similar to the regular checkpoints, except the condition code checkpoint for instruction *d* is *a* (not *c,* because *c* does not change condition codes). Consequently, the live ranges of the mapped registers at instruction *a* must be extended through instruction *d* in the reordered code. Taking this into account, the initial value of r1 must have its live range extended (denoted with a "y" in the following table). This further affects the register assignment because the original value in register r1 must be kept beyond instruction *a,* in case instruction *d* should trap and condition codes (generated by instruction *a*) must be materialized. In this example, the PowerPC versions of the condition codes happen to be generated by instruction *a.* However, these are not adequate for generating all the needed IA-32 condition codes. Extending the live range of register r1 means that register r6 must be used for holding the result of instruction *a.* This will eventually result in an additional register copy (r1 ← r6) when the translation block is exited, in order to maintain the consistent register mapping.

Step 5a: Assign Register with Condition Codes

