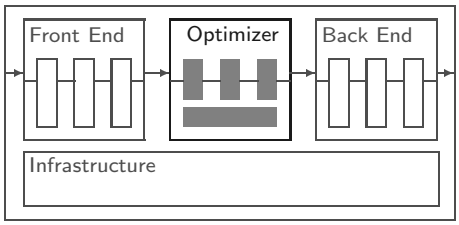


Chapter 10

Scalar Optimizations

10.1 Introduction

Code optimization in a compiler consists of analyses and transformations intended to improve the quality of the code that the compiler produces. Data-flow analysis, discussed in detail in Chapter 9, lets the compiler discover opportunities for transformation and lets the compiler prove the safety of applying the transformations. However, analysis is just the prelude to transformation: the compiler improve the code's performance by rewriting it.

Data-flow analysis serves as a unifying conceptual framework for the classic problems in static analysis. Problems are posed as data-flow frameworks. Instances of these problems, exhibited by programs, are solved using some general purpose solver. The results are produced as sets that annotate some form of the code. Sometimes, the insights of analysis can be directly encoded into the IR form of the code, as with SSA form.

Unfortunately, no unifying framework exists for optimizations—which combine a specific analysis with the rewriting necessary to achieve a desired improvement. Optimizations consume and produce the compiler's IR; they might be viewed as complex rewriting engines. Some optimizations are specified with detailed algorithms; for example, dominator-based value numbering builds up a collection of facts from low-level details (§8.5.2). Others are specified with high-level descriptions; for example, global redundancy elimination operates from a set of data-flow equations (§8.6), and inline substitution is usually described as replacing a call site with the text of the called procedure with appropriate substitution of actual arguments for formal parameters (§8.7.2). The techniques used to describe and implement transformations vary widely.

Optimization as Software Engineering

Having a separate optimizer can simplify the design and implementation of a compiler. The optimizer simplifies the front end; it can generate general-purpose code and ignore special cases. The optimizer simplifies the back end; it can focus on mapping the IR version of the program to the target machine. Without an optimizer, both the front end and back end must be concerned with finding opportunities for improvement and exploiting them.

In a pass-structured optimizer, each pass contains a transformation and the analysis required to support it. In principle, each task that the optimizer performs can be implemented once. This provides a single point of control and lets the compiler writer implement complex functions once, rather than many times. For example, deleting an operation from the IR can be complicated. If the deleted operation leaves a basic block empty, except for the block-ending branch or jump, then the transformation should also delete the block, and reconnect the block's predecessors to its successors, as appropriate. Keeping this functionality in one place simplifies implementation, understanding, and maintenance.

From a software engineering perspective, the pass structure, with a clear separation of concerns, makes sense. It lets each pass focus on a single task. It provides a clear separation of concerns—value numbering ignores register pressure and the register allocator ignores common subexpressions. It lets the compiler writer test passes independently and thoroughly, and it simplifies fault isolation.

The optimizer in a modern compiler is typically structured as a series of filters. Each filter, or pass, takes as its input the IR form of the code. Each pass produces as its output a rewritten version of the code, in IR form. This structure has evolved for several practical reasons. It breaks the implementation into smaller pieces, avoiding some of the complexity that arises in large, monolithic programs. It allows independent implementation and testing of the passes, which simplifies development, testing, and maintenance. It allows the compiler to provide different levels of optimization by activating a different set of passes for each level. Some passes execute once; others may execute several times in the sequence.

One of the critical issues in the design of an optimizer, then, is selecting a set of passes to implement and an order in which to run them. The selection of passes determines what specific inefficiencies in the IR program are discovered and improved. The order of execution determines how the passes interact.

For example, in the appropriate context ($\texttt{r}_2 \geq 0$ and $\texttt{r}_5 = 4$), the optimizer might rewrite `mult` $\texttt{r}_2$,$\texttt{r}_5 \Rightarrow \texttt{r}_{17}$ as `lshiftI` $\texttt{r}_2$,`2` $\Rightarrow \texttt{r}_{17}$. This improves the code by reducing demand for registers and replacing a potentially expensive operation, `mult`, with a cheap operation, `lshiftI`. In most cases, this is profitable. If, however, the next pass relies on commutativity to rearrange expressions, then replacing a multiply with a shift forecloses an opportunity (multiply is commuta-

tive; shift is not). To the extent that it makes later passes less effective, it may hurt overall code quality. Deferring the replacement of multiplies with shifts may avoid this problem; the context needed to prove safety and profitability for this rewrite is likely to survive the intervening passes.

Sometimes, a pass should be repeated multiple times in the sequence. For example, eliminating dead, or useless, code benefits the compiler in several ways. It shrinks the IR program, so later passes have less code to process. It eliminates some definitions and uses, so it may make the results of data-flow analysis sharper. Finally, it improves the resulting code—its actual purpose—by removing operations whose execution cannot be noticed. Because of the first two effects, dead-code elimination is often run early in compilation. For the final effect, it should run late in the compilation. Some passes are known to make code useless, so might also be run after such a pass. Thus, compilers often run dead code elimination several times during a compilation.

This chapter presents a selected set of transformations. The material is organized around a taxonomy of transformations, presented in §10.2. Section 10.3 presents example optimizations for those parts of the taxonomy that are not well covered in other chapters. The advanced topics section briefly discusses three subjects: combining optimizations for better results, optimizing for objective functions other than speed, and choosing an optimization sequence.

10.2 A Taxonomy for Transformations

The first hurdle in building an optimizer is conceptual. The literature on optimization describes hundreds, if not thousands, of distinct algorithms for improving IR programs. The compiler writer must select a subset of these transformations to apply. Reading the literature provides little help in the decision process, since most of the authors recommend using their own transformations.

To organize the space of optimizations, we use a simple taxonomy that categorizes transformations by the effect that they have on the code. The taxonomy is, of necessity, approximate. For example, some transformations have more than one effect. At a high level, we divide the transformations into two categories: machine independent transformations and machine dependent transformations.

Machine-independent transformations are those that largely ignore the details of the target machine. In many cases, the profitability of a transformation actually depends on detailed machine-dependent issues, but the implementation of the transformation ignores them.

For example, when local value numbering finds a redundant computation, it replaces it with a reference. This eliminates a computation, but it may increase the demand for registers. If the increased demand forces the register allocator to spill some value to memory, the cost of the memory operations probably exceeds the savings from eliminating the operation. However, value numbering deliberately ignores this effect because it cannot accurately determine whether a value must be spilled.

Machine-dependent transformations are those that explicitly consider details

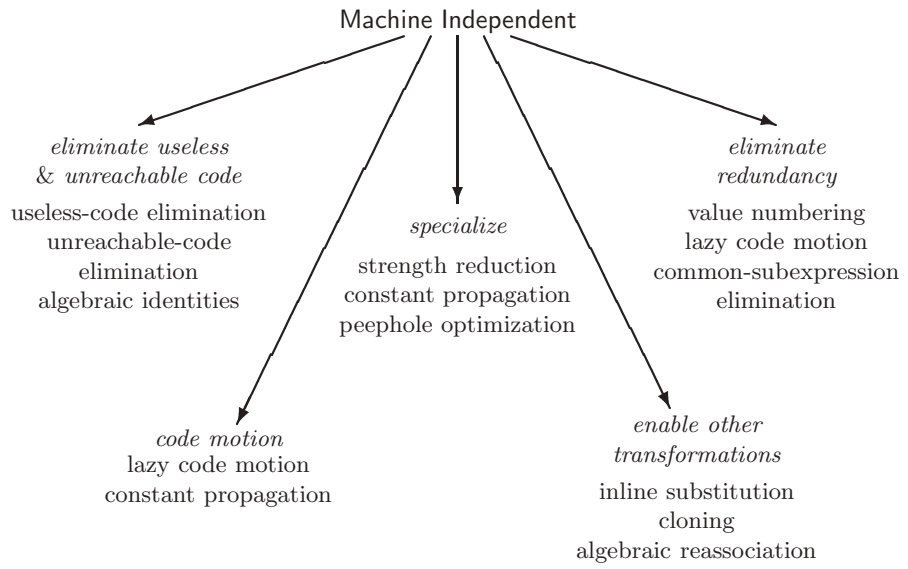


Figure 10.1: Machine-Independent Transformations

of the target machine. Many of these transformations fall into the realm of code generation, where the compiler maps the IR form of the code onto the target machine. However, some machine-dependent transformations fall in the realm of the optimizer. (Most are beyond the scope of this chapter, however.) Examples include transformations that rearrange the code to improve its behavior with regard to cache memory or that attempt to expose instruction-level parallelism.

While the distinction between these two categories is somewhat artificial, it has long been used as a first cut at classifying transformations.

10.2.1 Machine-Independent Transformations

In truth, there are a limited number of "machine-independent" ways that the compiler can improve the program. We will concern ourselves with five effects, shown in Figure 10.1. They are

- *eliminate useless or unreachable code:* If the compiler can prove that an operation is either useless or unreachable, it can eliminate the operation. Methods include useless-code elimination and unreachable-code elimination (§10.3.1), simplification of algebraic identities (part of local value numbering in §8.3.2), and sparse conditional constant propagation (§10.4.1) which discovers and removes some kinds of unreachable code.

- *move an operation to a place where it executes less frequently:* If the compiler can find a place where the operation will execute less frequently and produce the same answer, it can move the operation there. Methods in-

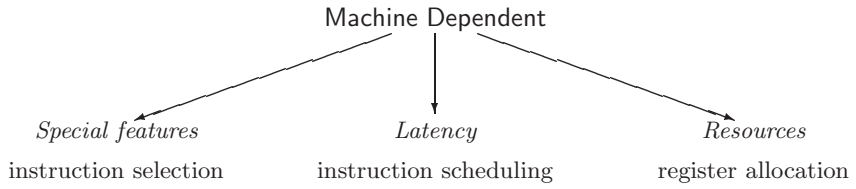


Figure 10.2: Machine Dependent Transformations

clude lazy code motion (§10.3.2) and constant propagation (§9.2.4, 10.3.3, and 10.4.1) which moves a computation from run time to compile time.

- *specialize a computation:* If the compiler can understand the specific context in which an operation will execute, it can often specialize the code to that context. Methods include operator strength reduction (§10.3.3), constant propagation (§9.2.4, 10.3.3, and 10.4.1), and peephole optimization (§11.4.1).

- *enable other transformations:* If the compiler can rearrange the code in a way that exposes more opportunities for other transformations, it can improve the overall effectiveness of optimization. Methods include inline substitution (§8.7.2), cloning (§8.7.1 and §12.4.2), and algebraic reassociation (§10.3.4).

- *eliminate a redundant computation:* If the compiler can prove that a computation is redundant, it can replace the computation with a reference to the previously computed value. Methods include local value numbering (§8.3.2), superlocal value numbering (§8.5.1), dominator-based value numbering (§8.5.2), and global common-subexpression elimination (§8.6).

We have already seen optimizations in each of these categories. Section 10.3 fills in the machine-independent part of the taxonomy more fully by presenting additional optimizations in each category, except redundancy elimination, an area already explored to a reasonable depth in Chapter 8.

Some techniques fit into several categories. Lazy code motion achieves both code motion and redundancy elimination. Constant propagation achieves code motion—by moving operations from run time to compile time—and specialization. In at least one form (see §10.4.1), it identifies and eliminates certain kinds of unreachable code.

10.2.2 Machine-Dependent Transformations

In "machine-dependent" transformations, the effects that the compiler can exploit are more limited. It can

- *take advantage of special hardware features:* Often, processor architects include features that they believe will help program execution. Such features include specialized operations—like a load operation that bypasses

the cache hierarchy, a branch operation that tells the prediction hardware not to track its results, or an advisory prefetch operation. If the compiler can make effective use of these features, they will, indeed, speed program execution. Recognizing opportunities to use these features sometimes takes additional work. The compiler writer might add new transformations to the optimizer or use a more complex instruction selection process. Some of this work falls into the realm of instruction selection, described in depth in Chapter 11.

- *manage or hide latency:* In some cases, the compiler can arrange the final code in a way that hides the latency of some operations. For example, memory operations can have latencies in the tens or hundreds of cycles. If the target machine supports either a prefetch operation or a non-blocking load, the compiler may find a schedule that issues a memory operation far enough in advance of its use to hide the latency. Rearranging the iteration space of a loop, or changing the packing of values into memory can improve the run-time cache locality and help to manage latency problems by reducing the number of memory operations that miss in the cache. Instruction scheduling, described in detail in Chapter 12, addresses some of these problems. The compiler writer may, however, need to add transformations that directly address these issues.

- *manage bounded machine resources:* Another source of complexity in compilation comes from the fact that the target machine has bounded resources—registers, functional units, cache memory, and main memory. The compiler must map the needs of the computation onto the bounded resources that the machine provides and, when the computation needs more resources than are available, the compiler must rewrite the code in a way that reduces its resource needs. For example, register allocation maps a computation's values to the target machine's register set; Chapter 13 describes this problem in detail.

Figure 10.2 shows the machine-dependent portion of the taxonomy. Because an example of each of these effects is the subject of a separate chapter, we omit them from this chapter.

10.3 Example Optimizations

This section describes additional optimizations in four of the five categories that we identified as machine-independent transformations. Each is intended to improve performance on a scalar uniprocessor machine. The selection of optimizations is illustrative, rather than exhaustive. However, each transformations is a good example of how to achieve that specific effect.

10.3.1 Eliminating Useless and Unreachable Code

Sometimes, a program contains computations that have no externally visible effect. If the compiler can determine that a given operation has this property,

it can remove the operation from the program. In general, programmers do not intentionally create such code. However, it arises in most programs as the direct result of analysis and optimization in the compiler, and even from macro expansion in the compiler's front end.

Two distinct effects can make an operation eligible for removal. The operation can be *dead* or *useless*—that is, no operation uses its value, or any use of its value cannot be detected. The operation can be *unreachable*—that is, no valid control-flow path contains the operation. If a computation falls into either category, it can be eliminated.

Removing useless or unreachable code shrinks the program. This produces a smaller IR program, which leads to a smaller executable program and to faster compilation. It may also increase the compiler's ability to improve the code. For example, unreachable code may have effects that show up in the results of static analysis and prevent the application of some transformation. In this case, removing the unreachable block may change the analysis results and allow further transformations (see, for example §10.20).

Some forms of redundancy elimination also remove useless code. For example, local value numbering applies algebraic identities to simplify the code. Examples include $x + 0 \Rightarrow x$, $y \times 1 \Rightarrow y$, and $max(z,z) \Rightarrow z$. Each of these eliminates a useless operation—by definition, an operation that, when removed, makes no difference in the program's externally visible behavior.

The algorithms in this section modify the control-flow graph (CFG). Thus, they distinguish branches (`cbr`) from jumps (`jmp`). Close attention to this distinction will help the reader understand the algorithms.

Eliminating Useless Code

The classic algorithms for eliminating useless code operate in a manner similar to mark-sweep garbage collectors (See §6.7.3). Like mark-sweep collectors, they need two passes over the code. The first pass begins by clearing all the mark fields and marking "critical" operations—such as input/output statements, code in the procedure's entry and exit blocks, calls to other procedures, and code that sets return values.[1] Next, it traces the operands of critical operations back to their definitions, and marks those operations as useful. This process continues, in a simple worklist iterative scheme, until no more operations can be marked as useful. The second pass walks the code and removes any operation not marked as useful.

Figure 10.3 makes these ideas concrete. The algorithm, which we call *Dead*, assumes that the code is in SSA form. This simplifies the process because each use refers to a single definition. *Dead* consists of two passes. The first, called *MarkPass*, discovers the set of useful operations. The second, called *SweepPass*, removes useless operations. *MarkPass* relies on reverse dominance frontiers, defined below.

The treatment of operations other than branches or jumps is straightforward.

[1] This can happen in any of several ways, including an assignment to a call-by-reference parameter, assignment through an unknown pointer, or an actual return statement.

```
Dead()
  MarkPass()
  SweepPass()


SweepPass()
  for each operation i
      if i is unmarked then
        if i is a branch then
          rewrite i with a jump
            to i's nearest marked
            postdominator
        if i is not a jump then
          delete i
```

```
MarkPass()
  WorkList ← ∅
  for each operation i
      clear i's mark
      if i is critical then
        mark operation i
        WorkList ← WorkList ∪ {i}
  while (WorkList ≠ ∅)
      remove i from WorkList
        (assume i is x ← y op z)
      if def(y) is not marked then
        mark def(y)
        WorkList ← WorkList ∪ {def(y)}
      if def(z) is not marked then
        mark def(z)
        WorkList ← WorkList ∪ {def(z)}
      for each b ∈ RDF(block(i))
        let j be the branch that ends b
        if j is unmarked then
          mark j
          WorkList ← WorkList ∪ {j}
```

Figure 10.3: Useless code elimination

The marking phase determines whether an operation is useful. The sweep phase removes operations that have not been marked as useful.

The treatment of control-flow operations is more complex. Every jump is considered useful. Branches are considered useful only if the execution of a useful operation depends on their presence. As the marking phase discovers useful operations, it also marks the appropriate branches as useful. To map from a marked operation to the branches that it makes useful, the algorithm relies on the notion of *control dependence*.

The definition of control dependence relies on postdominance. In a CFG, node j postdominates node i if every path from i to the CFG's exit node passes through i. Using postdominance, we can define control dependence as follows: in a CFG, node j is control-dependent on node i if and only if

1. There exists a nonnull path from i to j where j postdominates every node on the path after i. Once execution begins on this path, it must flow through j to reach the exit of the CFG (from the definition of postdominance), and

2. j does not strictly postdominate i. Another edge leaves i and takes control to a node outside the path to j. There must be a path beginning with this edge that leads to the CFG's exit without passing through j.

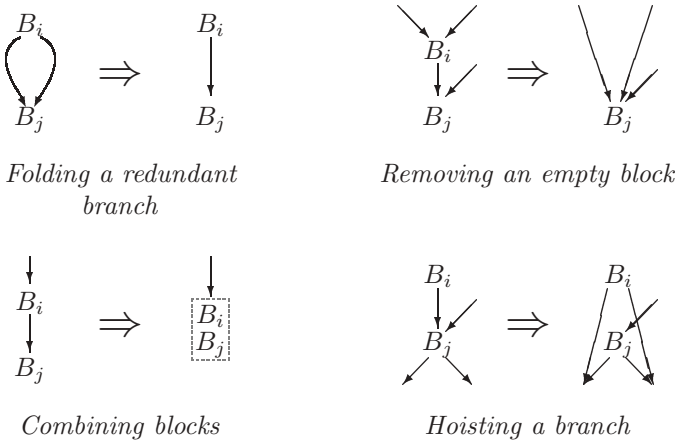


Figure 10.4: Transformations used in CLEAN

In other words, two or more edges leave i. One edge leads to j, and one or more of the other edges do not. Thus, the decision made at the branch ending block i can determine whether or not j executes. If an operation in j is useful, then the branch ending i is also useful.

This notion of control dependence is captured precisely by the *reverse dominance frontier* of j, denoted RDF(j). Reverse dominance frontiers are simply dominance frontiers computed on the reverse CFG. When *MarkPass* marks an operation useful, it visits every block in the reverse dominance frontier of the block that contains this useful operation, and marks its block-ending branch as useful. As it marks these branches, it adds them to the worklist.

SweepPass replaces any unmarked branch with a jump to its first postdominator that contains a marked operation or branch. If the branch is unmarked, then its successors, down to its immediate postdominator, contain no useful operations. (Otherwise, when those operations were marked, the branch would have been marked.) A similar argument applies if the immediate postdominator contains no marked operations or branch. To find the nearest useful postdominator, the algorithm can walk up the postdominator tree until it finds a block that contains a useful operation. Since, by definition, the exit block is useful, the search must terminate.

After *Dead* runs, the code contains no useless computations. It may contain empty blocks, which can be removed by the next algorithm.

Eliminating Useless Control Flow

Optimization can change the IR form of the program so that it has extraneous, or useless, control flow. If this happens, the compiler should include a pass that simplifies the CFG and eliminates useless control flow. This section presents a simple algorithm called *Clean* that handles this task.

Clean uses four transformations on the CFG, shown in Figure 10.4. They are

applied in the following order:

Folding a redundant branch: If *Clean* finds a block that ends in a branch, and both sides of the branch target the same block, it should replace the branch with a jump to the target block. This situation arises as the result of other simplifications. For example, B_i might have had two successors, each with a jump to B_j. If another transformation removed all the computations from those blocks, then removing the empty blocks might produce the initial graph shown in the figure.

Removing an empty block: If *Clean* finds a block that contains only a jump, it can merge the block into its successor. This situation arises when other passes remove all of the operations from a block B_i. Since B_i has only one successor, the transformation retargets the edges that enter B_i to B_j and deletes B_i from B_j's set of predecessors. This simplifies the graph. It should also speed up execution. In the original graph, the paths through B_i needed two control-flow operations to reach B_j. In the transformed graph, those paths use one operation to reach B_j.

Combining blocks: If *Clean* finds a block B_i that ends in a jump to B_j and B_j has only one predecessor, it can combine the two blocks. This situation can arise in several ways. Another transformation might eliminate other edges that entered B_j, or B_i and B_j might be the result of folding a redundant branch (described above). In either case, the two blocks can be combined into a single block. This eliminates the jump at the end of B_i.

Hoisting a branch: If *Clean* finds a block B_i that ends with a jump to an empty block B_j, and B_j ends with a branch, *Clean* can replace the block-ending jump in B_i with a copy of the branch from B_j. In effect, this hoists the branch into B_i. This situation arises when other passes eliminate the operations in B_j, leaving a jump to a branch. The transformed code achieves the same effect with just a branch. This adds an edge to the CFG. Notice that B_i cannot be empty, or else empty block removal would have eliminated it. Similarly, B_i cannot be B_j's sole predecessor, or else *Clean* would have combined the two blocks. (After hoisting, B_j still has at least one predecessor.)

Some bookkeeping is required to implement these transformations. Some of the modifications are trivial. To fold a redundant branch in a program represented with ILOC and a graphical CFG, *Clean* simply overwrites the block-ending branch with a jump and adjusts the successor and predecessor lists of the blocks. Others are more difficult. Merging two blocks may involve allocating space for the merged block, copying the operations into the new block, adjusting the predecessor and successor lists of the new block and its neighbors in the CFG, and discarding the two original blocks.

Clean applies these four transformations in a systematic fashion. It traverses the graph in postorder, so that B_i's successors are simplified before B_i, unless the

```
OnePass()
  for each block i, in postorder
    if i ends in a conditional branch then
      if both targets are identical then
        replace the branch with a jump
    if i ends in a jump to j then
      if i is empty then
        replace transfers to i with transfers to j
      if j has only one predecessor then
        coalesce i and j
      if j is empty and ends in a conditional branch then
        overwrite i's jump with a copy of j's branch

Clean()
  while the CFG keeps changing
    compute postorder
    OnePass()
```

Figure 10.5: The algorithm for CLEAN

successor lies along a back edge with respect to the postorder numbering. In that case, *Clean* will visit the predecessor before the successor. This is unavoidable in a cyclic graph. Simplifying successors before predecessors reduces the number of times that the implementation must move some edges.

In some situations, more than one of the transformations may apply. Careful analysis of the various cases leads to the order shown in Figure 10.5. The algorithm uses a series of *if* statements rather than an *if-then-else* to let it apply multiple transformations in a single visit to a block.

If the CFG contains back edges, then a pass of *Clean* may create additional opportunities—unprocessed successors along the back edges. These, in turn may create other opportunities. To handle this problem, *Clean* can iterate. It must compute a new postorder numbering between calls to *OnePass* because each pass changes the underlying graph. Figure 10.5 shows pseudocode for *Clean*.

Clean does not handle all the cases that can arise. For example, it cannot, by itself, eliminate an empty loop. Consider the CFG shown on the left side of Figure 10.6. Assume that block B_2 is empty. None of *Clean*'s transformations can eliminate B_2. The branch that ends B_2 is not redundant. B_2 does not end with a jump, so *Clean* cannot combined it with B_3. Its predecessor ends with a branch rather than a jump, so *Clean* can neither combine it with B_1 nor its branch into B_1.

However, cooperation between *Clean* and *Dead* can eliminate the empty loop. *Dead* used control dependence to mark useful branches. If B_1 and B_3 contain useful operations, but B_2 does not, then the marking pass in *Dead* will not mark the branch ending B_2 as useful because $B_2 \notin \text{RDF}(B_3)$. Because the branch is

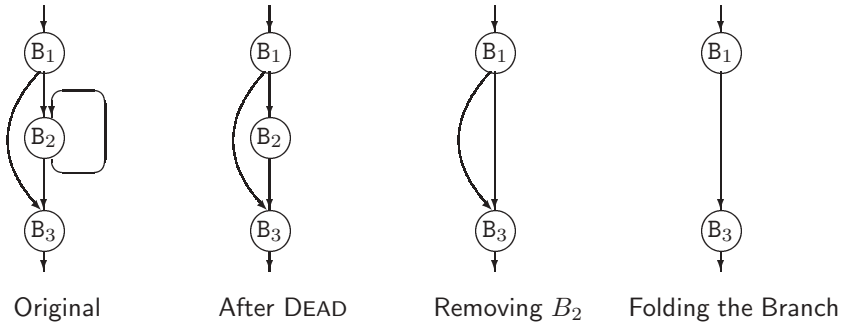


Figure 10.6: Removing an empty loop

useless, the code that computes the branch condition is also useless. Thus, *Dead* eliminates all of the operations in B_2 and converts the branch that ends it into a jump to its closest useful postdominator, B_3. This eliminates the original loop and produces the second CFG shown in the figure.

In this form, *Clean* folds B_2 into B_3, shown in the third CFG in the figure. This also makes the branch at the end of B_1 redundant. *Clean* rewrites it with a jump, producing the final CFG shown in the figure. At this point, if B_1 is B_3's sole remaining predecessor, *Clean* coalesces the two blocks into a single one.

This cooperation is simpler and more effective than adding another transformation to *Clean* that handles empty loops. Such a transformation might recognize a branch from B_i to itself and, for an empty B_i, rewrite it with a jump to the branch's other target. The problem lies in determining when B_i is truly empty. If B_i contains no operations other than the branch, then the code that computes the branch condition must lie outside the loop. Thus, the transformation is only safe if the self-loop never executes. Reasoning about the number of executions of the self-loop requires knowledge about the runtime value of the comparison, a task that is, in general, beyond the compiler's ability. If the block contains operations, but only operations that control the branch, then the transformation would need to recognize the situation with pattern matching. In either case, this new transformation would be more complex than the four included in *Clean*. Relying on the combination of *Dead* and *Clean* achieves the appropriate result in a simpler, more modular fashion.

Eliminating Unreachable Code

Sometimes the CFG contains code that is unreachable. The compiler should find unreachable blocks and remove them. A block can be unreachable for two distinct reasons: there may be no path through the CFG that leads to the block, or the paths that reach the block may not be executable—for example, guarded by a condition that always evaluates to false.

The former case is easy to handle. The compiler can perform a simple mark/sweep style reachability analysis on the CFG. Starting with the entry, it marks every reachable node in the CFG. If all branches and jumps are unam-

biguous, then all unmarked blocks can be deleted. With ambiguous branches or jumps, the compiler must preserve any block whose address can reach such a jump.[2] This analysis is simple and inexpensive. It can be done during traversals of the CFG for other purposes, or during CFG construction itself.

Handling the second case is harder. It requires the compiler to reason about the values of expressions that control branches. Section 10.4.1 presents an algorithm that finds some blocks that are unreachable because the paths leading to them are not executable.

10.3.2 Code Motion

Moving a computation to a point where it executes less frequently should reduce the total operation count for the running program. The first transformation presented in this section, *lazy code motion*, uses code motion to speed up execution. Because loops tend to execute many more times than the code that surrounds them, much of the work in this area has focused on moving loop-invariant expressions out of loops. Lazy code motion performs loop-invariant code motion. It extends the notions originally formulated in the available expressions problem to include operations that are a are redundant along some, but not all, paths. It inserts code to make them redundant on all paths; and removes the newly redundant expression.

Some compilers, however, optimize for other criteria. If the compiler is concerned about the size of the executable code, it may consider code motion to reduce the number of copies of a specific operation. The second transformation presented in this section, *hoisting*, uses code motion to reduce duplication of instructions. It discovers cases where inserting an operation makes several copies of the same operation redundant without changing the values computed by the program.

Lazy Code Motion

Lazy code motion (LCM) uses data-flow analysis to discover both operations that are candidates for code motion and locations where it can place those operations. The algorithm operates on the IR form of the program and its CFG, rather than SSA form. The algorithm consists of six sets of data-flow equations and a simple strategy for interpreting the results as directions for modifying the code.

LCM combines code motion with elimination of redundant and partially-redundant computations. Redundancy was introduced in Chapter 8. A computation is partially-redundant at point p if it occurs on some, but not all, paths that reach p. Figure 10.7 shows two ways that an expression can be partially redundant. In the first example, $a \leftarrow b \times c$ occurs on one path leading to the merge point, but not on the other. To make the second computation redundant, LCM inserts an evaluation of $a \leftarrow b \times c$ on the other path. In the second

[2] If the source language includes the ability to perform arithmetic on pointers or labels, every block must be preserved. Otherwise, the compiler should be able to limit the preserved set to those blocks whose labels are referenced.

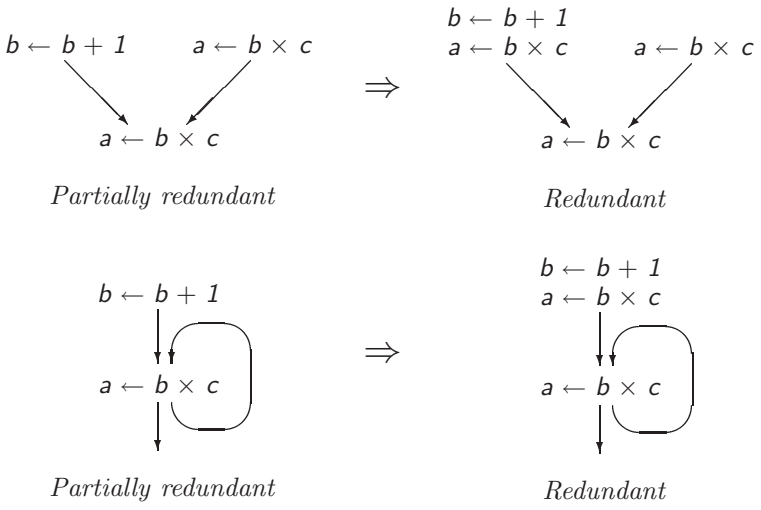


Figure 10.7: Converting partial-redundancy into redundancy

example, $a \leftarrow b \times c$ is redundant along the loop's back edge, but not along the edge entering the loop. Inserting an evaluation of $a \leftarrow b \times c$ before the loop makes the occurrence inside the loop redundant. By making the loop invariant computation redundant and eliminating it, LCM moves it out of the loop.

To accomplish this, the optimizer solves a series of data-flow problems. It computes information about *availability*, a forward data-flow problem familiar from §8.6 and about *anticipability*, a related notion solved as a backward data-flow problem. The next step uses the results of these analyses to compute an *earliest placement*; this annotates each edge with the a set describing the expressions for which this edge is a legal placement and no earlier placement in the graph is possible. The algorithm then looks for possible *later placements*—situations where moving an expression forward in the CFG or later in execution from its earliest placement achieves the same effect as the earliest placement. Finally, it computes an *insertion set* for each edge that specifies which expressions to insert along that edge and a *deletion set* for each node that contains the expressions to remove from the corresponding block. A simple follow-on pass interprets the insertion and deletion sets and rewrites the IR.

Background The first step in solving the data-flow equations is to compute local predicates for each block. LCM uses three kinds of local information.

DEEXPR(b) is the set of expressions e in b that are downward exposed. If $e \in$ DEEXPR(b), evaluating e at the end of block b produces the same value as evaluating it in its original position.

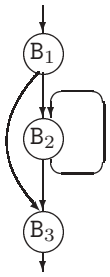
UEEXPR(b) is the set of upward exposed expressions e in b. If $e \in$ UEEXPR(b), evaluating e at the entry to block b produces the same value as evaluating it in its original position.

```
B1: loadI  1        ⇒ r1
    i2i    r1       ⇒ r2
    loadAI r0,@m    ⇒ r3
    i2i    r3       ⇒ r4
    cmp_LT r2,r4    ⇒ r5
    cbr    r5       → B2,B3
B2: mult   r17,r18  ⇒ r20
    add    r19,r20  ⇒ r21
    i2i    r21      ⇒ r8
    addI   r2,1     ⇒ r6
    i2i    r6       ⇒ r2
    cmp_GT r2,r4    ⇒ r7
    cbr    r7       → B3,B2
B3: ...
```

A Simple Loop

$\left\{ \begin{array}{c} r_1, r_3, r_5, r_6, \\ r_7, r_{20}, r_{21} \end{array} \right\}$

Set of Expressions



Its CFG

	B_1	B_2	B_3
DEExpr	$\{r_1, r_3, r_5\}$	$\{r_7, r_{20}, r_{21}\}$	$\cdots$
UEExpr	$\{r_1, r_3\}$	$\{r_6, r_{20}, r_{21}\}$	$\cdots$
ExprKill	$\{r_5, r_6, r_7\}$	$\{r_5, r_6, r_7\}$	$\cdots$

Figure 10.8: Example for lazy code motion

ExprKill(b) is the set of expressions e that are killed in b. If $e \in$ ExprKill(b), then b contains a redefinition of one or more operands of e. As a consequence, evaluating e at the entry to b may produce a different value than evaluating it at the end of b.

We have used all these sets in other data-flow problems.

The equations for LCM rely on several implicit assumptions about the shape of the code. They assume that textually identical expressions always define the same name, which suggests an unlimited set of names—one name for each unique textual expression. (This is rule one from the register-naming rules described in §5.6.) Since every definition of r_k comes from the expression $r_i + r_j$ and no other expression defines r_k, the optimizer does not need to find each definition of $r_i + r_j$ and copy the result into a temporary location for later reuse. Instead, it can simply use r_k.

LCM moves expressions, not assignments. Requiring that textually identical expressions define the same virtual register implies that program variables are set by register-to-register copy operations. The code in Figure 10.8 has this property. By dividing the name space into variables and expressions, we can limit the domain of LCM to expressions. In that example, the variables are r_2, r_4, and r_8, each of which is defined by a copy operation. All the other names, r_1, r_3, r_5, r_6, r_7, r_{20}, and r_{21}, are expressions. The lower part of Figure 10.8 shows the local information for the blocks in the example.

Available Expressions The first step in LCM computes available expressions.

$$\text{AVAILIN}(n) = \bigcap_{m \in preds(n)} \text{AVAILOUT}(m), \quad n \neq n_0$$

$$\text{AVAILOUT}(m) = \text{DEEXPR}(m) \cup (\text{AVAILIN}(m) \cap \overline{\text{EXPRKILL}(m)})$$

The form of the equations differs slightly from the one shown in §8.6. In this form, the equations compute a distinct set for each block's entry, AVAILIN, and its exit, AVAILOUT. The AVAIL sets in the earlier equations are identical to the AVAILIN sets here. The AVAILOUT sets represent the contribution of a block to its successors' AVAILIN sets. The solver must initialize AVAILIN(n_0) to $\emptyset$ and the remaining AVAILIN and AVAILOUT sets to contain the set of all expressions.

For the example in Figure 10.8, the equations for availability produce the following results:

	B_1	B_2	B_3
AVAILIN	$\emptyset$	$\{r_1, r_3\}$	$\{r_1, r_3\}$
AVAILOUT	$\{r_1, r_3, r_5\}$	$\{r_1, r_3, r_7, r_{20}, r_{21}\}$	$\cdots$

In global redundancy elimination, we interpreted $x \in \text{AVAILIN}(b)$ to mean that, along every path from n_0 to b, x is computed and none of its operands is redefined between the point where x is computed and b. An alternative view, useful in understanding LCM, is that $x \in \text{AVAILIN}(b)$ if and only if the compiler could place an evaluation of x at the entry to b and obtain the result produced by the most recent evaluation on any control-flow path from n_0 to b. In this light, the AVAILIN sets tell the compiler how far forward in the CFG it can move the evaluation of x, ignoring any uses of x.

Anticipable Expressions Availability provides LCM with information about moving evaluations forward in the CFG. LCM also needs information about moving evaluations backward in the CFG. To obtain this, it computes information about *anticipable expressions*, namely, expressions that can be evaluated earlier in the CFG than their current block.

The UEEXPR set captures this notion locally. If $e \in \text{UEEXPR}(b)$, then b contains at least one evaluation of e and that evaluation uses operands that are defined before entry to b. Thus, $e \in \text{UEEXPR}(b)$ implies that an evaluation of e at the entry to b must produce the same value as the first evaluation of e in b, so the compiler can safely move that first evaluation to the block's entry.

The second set of data-flow equations for LCM extends the notion of anticipability across multiple blocks. It is a backward data-flow problem.

$$\text{ANTOUT}(n) = \bigcap_{m \in succ(n)} \text{ANTIN}(m), \quad n \neq n_f$$

$$\text{ANTIN}(m) = \text{UEEXPR}(m) \cup (\text{ANTOUT}(m) \cap \overline{\text{EXPRKILL}(m)})$$

LCM must initialize the ANTOUT set for n_f to $\emptyset$ and the remaining ANTIN and ANTOUT sets to contain the set of all expressions.

ANTIN and ANTOUT provide the compiler with information about the backward motion of expressions in ANTOUT(b). If $x \in$ ANTOUT(b), then the compiler can place an evaluation of x at the end of block b and produce the same value as the next evaluation on any path leaving b.

For the ongoing example, the equations for anticipability produce the following results.

	B_1	B_2	B_3
ANTIN	$\{r_1, r_3\}$	$\{r_{20}, r_{21}\}$	$\emptyset$
ANTOUT	$\emptyset$	$\emptyset$	$\emptyset$

Earliest Placement Given availability, which encodes information about forward movement in the CFG, and anticipability, which encodes information about backward movement in the CFG, the compiler can compute an *earliest placement* for each expression. To keep the equations simple, it is easier to place computations on edges in the CFG rather than in the nodes. This lets the equations compute a placement without having to choose a block. The compiler can defer until later the decision to place the operation at the end of the edge's source, at the start of its destination, or in a new block in mid-edge. (See the discussion of critical edges in §9.3.4.)

For an edge $\langle i, j \rangle$ in the CFG, an expression e is in EARLIEST(i, j) if and only if the computation can legally move to $\langle i, j \rangle$ and cannot move to any earlier edge in the CFG. The equation for EARLIEST reflects this condition:

$$\text{EARLIEST}(i, j) = \text{ANTIN}(j) \cap \overline{\text{AVAILOUT}(i)} \cap (\text{EXPRKILL}(i) \cup \overline{\text{ANTOUT}(i)})$$

These conditions all have simple explanations. For e to be legal on edge $\langle i, j \rangle$ and not be movable further up in the CFG, the following three conditions must hold:

1. $e \in$ ANTIN(j) proves that the compiler can move e to the head of j.
2. $e \notin$ AVAILOUT(i) proves that no previous computation of e is available on exit from i. If $e \in$ AVAILOUT(i), then a computation of e on $\langle i, j \rangle$ would be redundant and unnecessary.
3. $e \in (\text{EXPRKILL}(i) \cup \overline{\text{ANTOUT}(i)})$ proves that e cannot move upward, through i, to an earlier edge. If $e \in$ EXPRKILL(i), then e would produce a different result at the head of block i, so it cannot move through i. If $e \in \overline{\text{ANTOUT}(i)}$, then e cannot move into block i. If either of these conditions holds, then $\langle i, j \rangle$ is the earliest edge where e can be placed.

Since LCM cannot move an expression earlier than n_0, LCM should ignore the final term, (EXPRKILL$(i) \cup \overline{\text{ANTOUT}(i)}$). This simplification marginally reduces the cost of computing EARLIEST.

Computing EARLIEST requires no iteration; it relies solely on previously computed values. EARLIEST is used, in turn, in the computation of the LATER

sets. Because the LATER computation iterates, it may be faster to precompute the EARLIEST sets for each edge. Alternatively, the compiler writer can forward substitute the right-hand side directly into the next set of equations.

For the continuing example, the EARLIEST computation produces the following sets

	$\langle B_1, B_2 \rangle$	$\langle B_1, B_3 \rangle$	$\langle B_2, B_2 \rangle$	$\langle B_2, B_3 \rangle$
EARLIEST	$\{r_{20}, r_{21}\}$	$\emptyset$	$\emptyset$	$\emptyset$

Later Placement The final data-flow problem in LCM determines whether an earliest placement can be moved forward or later in the CFG, while achieving the same effect.

$$\text{LATERIN}(j) = \bigcap_{i \in pred(j)} \text{LATER}(i, j), \quad j \neq n_0$$

$$\text{LATER}(i, j) = \text{EARLIEST}(i, j) \cup \text{LATERIN}(i) \cap \overline{\text{UEEXPR}(i)}$$

The solver must initialize $\text{LATERIN}(n_0)$ to $\emptyset$.

An expression $e \in \text{LATERIN}(k)$ if and only if every path that reaches k has an edge $\langle p, q \rangle$ where $e \in \text{EARLIEST}(\langle p, q \rangle)$, and the path from q to k neither redefines e's operand, nor contains an evaluation of e (that earlier placement would anticipate). The EARLIEST term in the equation for LATER ensures that $\text{LATER}(i, j)$ includes $\text{EARLIEST}(i, j)$. The rest of that equation puts e into $\text{LATER}(i, j)$ if e can be moved forward from i ($e \in \text{LATERIN}(i)$) and a placement at the entry to i does not anticipate a use in i ($e \notin \text{UEEXPR}(i)$).

Once the equations have been solved, $e \in \text{LATERIN}(i)$ implies that the compiler could move the evaluation of e forward through i without losing any benefit—that is, there is no evaluation of e in i that an earlier evaluation would anticipate; and $e \in \text{LATER}(i, j)$ implies that the compiler could move an evaluation of e in i forward to j.

For the ongoing example, these equations produce the following sets.

	B_1	B_2	B_3
LATERIN	$\emptyset$	$\emptyset$	$\emptyset$

	$\langle B_1, B_2 \rangle$	$\langle B_1, B_3 \rangle$	$\langle B_2, B_2 \rangle$	$\langle B_2, B_3 \rangle$
LATER	$\{r_{20}, r_{21}\}$	$\emptyset$	$\emptyset$	$\emptyset$

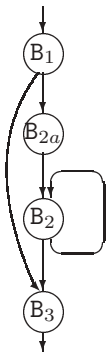
Rewriting the Code The final step in performing lazy code motion is to rewrite the code so that it capitalizes on the knowledge derived by the earlier data-flow computations. To simplify the process, LCM computes two additional sets, INSERT and DELETE.

The INSERT set specifies, for each edge, the computations that LCM should insert on that edge.

$$\text{INSERT}(i, j) = \text{LATER}(i, j) \cap \overline{\text{LATERIN}(j)}$$

```
B1:  loadI  1        ⇒ r1
     loadAI r0,@m    ⇒ r2
     cmp_LT r1,r2    ⇒ r3
     cbr    r3       → B2a,B3
B2a: mult   r17,r18  ⇒ r20
     add    r19,r20  ⇒ r21
     jump            → B2
B2:  i2i    r21      ⇒r8
     addI   r1,1     ⇒r4
     i2i    r4       ⇒ r1
     cmp_GT r1,r2    ⇒ r5
     cbr    r5       → B3,B2
B3:  ...
```

The Transformed Code



Its CFG

Figure 10.9: Example after lazy code motion

If i has only one successor, LCM can insert the computations at the end of i. If j has only one predecessor, it can insert the computations at the entry of j. If neither condition applies, the edge $\langle i, j\rangle$ is a critical edge and the compiler should split $\langle i, j\rangle$ by inserting a block in the middle of the edge to hold the computations specified in INSERT(i, j).

The DELETE set specifies for a block which computations LCM should delete from the block.

$$\text{DELETE}(i) = \text{UEEXPR}(i) \cap \overline{\text{LATERIN}(i)}, \quad i \neq n_0$$

DELETE(n_0) is empty, of course, since no block precedes it. If $e \in$ DELETE(i), then the first computation of e in i is redundant, after all the insertions have been made. Any subsequent evaluations of e in i that have upwards-exposed uses—that is, its operands are not defined between the start of i and the evaluation can also be deleted. Because all evaluations of e define the same name, the compiler need not rewrite subsequent references to the deleted evaluation. Those references will simply refer to earlier evaluations of e that LCM has proven to produce the same result.

For our example, the INSERT and DELETE sets are simple.

	$\langle$B1,B2$\rangle$	$\langle$B1,B3$\rangle$	$\langle$B2,B2$\rangle$	$\langle$B2,B3$\rangle$
INSERT	{r20,r21}	∅	∅	∅

	B1	B2	B3
DELETE	∅	{r20,r21}	∅

Interpreting the INSERT and DELETE sets rewrites the code as shown on the left side of Figure 10.9. LCM deletes the expressions that define $\texttt{r}_{20}$ and $\texttt{r}_2$1 from $\texttt{B}_2$ and inserts them on the edge from $\texttt{B}_1$ to $\texttt{B}_2$.

Since B_1 has two successors and B_2 has two predecessors, $\langle B_1, B_2 \rangle$ is a critical edge. Thus, LCM must split the edge, creating a new block B_{2a} to hold the inserted computation of r_{20}. Splitting $\langle B_1, B_2 \rangle$ adds an extra `jump` to the code. Subsequent work in code generation will almost certainly implement the `jump` in B_{2a} as a fall-through, eliminating any cost associated with it.

Notice that LCM leaves the copy defining r_8 in B_2. LCM moves expressions, not assignments. (Recall that r_8 is a variable, not an expression.) If the copy is unnecessary, the register allocator will discover that fact and coalesce it away.

Hoisting

The compiler can also use code motion to shrink the size of the compiled code. Section 9.2.4 introduced the notion of a very busy expression. The compiler can use the results of computing VERYBUSY sets to perform *code hoisting*.

The idea is simple. An expression $e \in$ VERYBUSY(b), for some block b, if and only if e is evaluated along every path leaving b and evaluating e at the end of b would produce the same result as each of those evaluations. (That is, none of e's operands is redefined between the end of b and the next evaluation of e along every path leaving b.) To shrink the code, the compiler can insert an evaluation of e at the end of b and replace the first occurrence of e on each path leaving b with a reference.

To replace those expressions directly, the compiler would need to locate them. It could insert e, then solve another data-flow problem, proving that the path from b to some evaluation of e is e-clear. Alternatively, it could traverse each of the paths leaving b to find the first block where e is defined—by looking in the block's UEEXPR set. Each of these approaches seems complicated.

A simpler approach has the compiler visit each block b and insert an evaluation of e at the end of b, for every expression $e \in$ VERYBUSY(b). If the compiler uses a uniform discipline for naming, as suggested in the discussion of LCM, then each evaluation will define the appropriate name. Subsequent application of LCM or redundancy elimination will then remove the newly redundant expressions.

10.3.3 Specialization

In most compilers, the shape of the IR program is determined by the front end, before any detailed analysis of the code. Of necessity, this produces general code that works in any context that the running program might encounter. With analysis, however, the compiler can often learn enough to narrow the contexts in which the code must operate. This creates the opportunity for the compiler to specialize the sequence of operations in ways that capitalize on its knowledge of the context in which the code will execute.

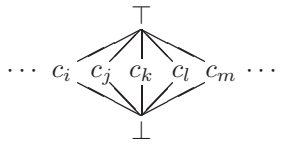
As an example, consider constant propagation. Constant propagation tries to discover specific values taken on by the arguments to an operation. For an operation such as $x \leftarrow y \times z$, If the compiler discovers that y always has the value 4 and z is nonnegative, it can replace the multiply with a shift operation, which is often less expensive. If it discovers that z has the value 17, it can replace

```
Worklist ← ∅
for each ssa name n
  initialize Value(n) by rule
  if Value(n) ≠ ⊤ then
    Worklist ← Worklist ∪ {n}

while (Worklist ≠ ∅)
  remove n from Worklist
  for each operation o that uses n
    let m be the ssa name that o defines
    if Value(m) ≠ ⊥ then
      t ← result of modeling m
      if t ≠ Value(m) then
        Value(m) ← t
        Worklist ← Worklist ∪ {m}
```

$$\begin{array}{ll} \top \wedge x = x & \forall x \\ c_i \wedge c_j = c_i & \text{if } c_i = c_j \\ c_i \wedge c_j = \bot & \text{if } c_i \neq c_j \\ \bot \wedge x = \bot & \forall x \end{array}$$

Meet rule



The lattice

Figure 10.10: Sparse Simple Constant Propagation

the operation with an immediate load of 68. These operations form a hierarchy. The multiply is general; it works for any values of y and z (although it may raise an exception for some of them). The shift is less general: it produces the correct result if and only if y has the value 4 and z is nonnegative. Of course if either y or z is zero, x is also zero. The load immediate is least general: it works only when the arguments have the property that the value of $y \times z$ is known.

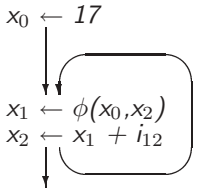
Other examples of specialization include *peephole optimization* and *tail-recursion elimination*. Peephole optimization slides a small "window" (the peephole) over the code and looks for simplifications within the window. It originated as an efficient way to perform some final local optimization, after code generation (see §11.4).

Tail-recursion elimination recognizes when the final operation in a procedure is a self-recursive call. It replaces the call with a jump back to the procedure's entry, allowing reuse of the activation record and avoiding the expense of the full procedure linkage convention. This important case arises in programs that use recursion to traverse data structures or to perform iterative calculations. As detailed examples of specialization, the next two sections describe SSA-based algorithms for constant propagation and for operator strength reduction with linear-function test replacement.

Constant Propagation

Using SSA form, we can reformulate constant propagation in a much more intuitive way than the equations in §9.2.4. The algorithm, called *sparse simple constant propagation* (SSCP), is shown in Figure 10.10.

SSCP consists of an initialization phase and a propagation phase. The ini-



Time Step	Lattice Values					
	Pessimistic			Optimistic		
	x_0	x_1	x_2	x_0	x_1	x_2
0	17	$\perp$	$\perp$	17	$\top$	$\top$
1	17	$\perp$	$\perp$	17	17	$17 \wedge i_{12}$

Figure 10.11: Optimistic constant example

tialization phase iterates over the SSA names. For each SSA name *n*, it examines the operation that defines *n* and sets *Value(n)* according to a simple set of rules. If *n* is defined by a ϕ-function, SSCP sets *Value(n)* to $\top$. If *n*'s value is a known constant *c*, SSCP sets *Value(n)* to *c*. If *n*'s value cannot be known—for example, it is defined by reading a value from some external media—SSCP sets *Value(n)* to $\perp$. Finally, if *n*'s value is not known, SSCP sets *Value(n)* to $\top$. If *Value(n)* is not $\top$, the algorithm adds *n* to the worklist.

The propagation phase is straightforward. It removes an SSA name *n* from the worklist. The algorithm examines each operation *o* that uses *n*, where *o* defines some SSA name *m*. If *Value(m)* has already reached $\perp$, then no further evaluation is needed. Otherwise, it models the evaluation of *o* by interpreting the operation over the lattice values of its arguments. If the result is lower than *Value(m)*, it lowers *Value(m)* accordingly and adds *m* to the worklist. The algorithm halts when the worklist is empty.

Interpreting an operation over its lattice values requires some care. For a ϕ-function, the result is simply the meet of the lattice values of all the ϕ-function's arguments—even if one or more operand has the value $\top$. For a more general operation, the compiler must apply operator-specific knowledge. If any operand has the lattice value $\top$, the evaluation should return $\top$. If none of the operands have the values $\top$, the model should produce an appropriate value. For the operation $x \leftarrow y \times z$, with *Value(y)* = *3* and *Value(z)* = *17*, the model should produce the value 51. If *Value(y)* = $\perp$, the model should produce zero for *Value(z)* = *0* and $\perp$ for any other lattice value. SSCP needs similar interpretations for each value-producing operation in the IR.

Complexity The propagation phase of SSCP is a classic fixed-point scheme. The arguments for termination and complexity follow from the length of descending chains through the lattice that it uses to represent values, shown in the bottom right part of the figure. The *Value* associated with any SSA name can have one of three initial values—$\top$, some constant c_i, and $\perp$. The propagation phase can only lower its value. For a given SSA name, this can happen at most twice—from $\top$ to c_i to $\perp$. SSCP adds an SSA name to the worklist only when its value changes, so each SSA name appears on the worklist at most two times. SSCP evaluates an operation when one of its operands is pulled from the worklist. Thus, the total number of evaluations is two times the number of uses in the program.

The SSA Graph

In some algorithms, viewing the SSA form of the code as a graph simplifies either the discussion or the implementation. The algorithm for strength reduction interprets the SSA-form of the code as a graph.

In SSA form, each name has a unique definition, so that name specifies a particular operation in the code that computed its value. Each use of a name occurs in a specific operation, so the use can be interpreted as a chain from the use to its definition. Thus, a simple lookup table that maps names to the operations that define them creates a chain from each use to the corresponding definition. Mapping a def to the operations that use it is slightly more complex. However, this map can easily be constructed during the renaming phase of the SSA construction.

We draw SSA graphs with edges that run from a use to a definition. This indicates the relationship implied by the SSA names. The compiler will need to traverse the edges in both directions. Strength reduction moves, primarily, from uses to definitions. The sparse conditional constant propagation algorithm can be viewed as moving from definitions to uses on the SSA graph. The compiler writer can easily add the data structures needed to allow traversal in both directions.

Coverage SSCP discovers all of the constants found by the data-flow framework in §9.2.4. Because it initializes unknown values to $\top$, rather than $\bot$, it can propagate some values into cycles in the graph—loops in the CFG. Algorithms that begin with the value $\top$, rather than $\bot$, are often called *optimistic* algorithms. The intuition behind this term is that initialization to $\top$ allows the algorithm to propagate information into a cyclic region, optimistically assuming that the value along the back edge will confirm this initial propagation. An initialization to $\bot$, called *pessimistic*, cuts off that possibility.

To see this, consider the SSA fragment in Figure 10.11. If the algorithm pessimistically initializes x_1 and x_2 to $\bot$, it will not propagate the value 17 into the loop. When it evaluates the ϕ-function for x_1, it computes $17 \wedge \bot$ to yield $\bot$. With x_1 set to $\bot$, x_2 also gets $\bot$, even if i_{17} has a known value, such as 0.

If, on the other hand, the algorithm optimistically initializes unknown values to $\top$, it can propagate the value of x_0 into the loop. When it computes a value for x_1, it evaluates $17 \wedge \top$ and assigns the result, 17, to x_1. Since x_1's value changes, the algorithm places x_1 on the worklist. The algorithm then re-evaluates the definition of x_2 If, for example, i_{12} has the value 0, then this assigns x_2 the value 17 and adds x_2 to the worklist. When it re-evaluates the ϕ-function, it computes $17 \wedge 17$ and proves that x_1 is 17.

Consider what would happen if i_{12} has the value 2, instead. Then, when SSCP evaluates $x_1 + i_{12}$, it assigns x_2 the value 19. Now, x_1 gets the value $17 \wedge 19$, or $\bot$. This, in turn, propagates back to x_2, producing the same final result as the pessimistic algorithm.

Strength Reduction

Strength reduction is a transformation that replaces a repeated series of expensive (strong) operations with a series of cheap (weak) operations that compute the same values. The classic example replaces integer multiplications based on a loop index with equivalent additions. This particular case arises routinely from the expansion of array and structure addresses in loops. The left side of Figure 10.12 shows the ILOC code that might be generated for the following simple loop:

```
sum ← 0
for i ← 1 to 100
  sum ← sum + a(i)
```

The code is in semi-pruned SSA form; the purely local values (r_1, r_2, r_3, and r_4) have neither subscripts nor ϕ-functions. Notice how the reference to *a(i)* expands into four operations—the `subI`, `multI`, and `addI` that compute `(i-1)` × `4` + `@a` and the `load` that defines r_4.

For each iteration, this sequence of operations computes the address of *a(i)* from scratch as a function of the loop index variable *i*. Consider the sequence of values taken on by r_{i1}, r_1, r_2, and r_3.

r_{i_1}: { 1, 2, 3, ..., 100 }
r_1: { 0, 1, 2, ..., 99 }
r_2: { 0, 4, 8, ..., 396 }
r_3: { *@a*, *@a*+4, *@a*+8, ..., *@a*+396 }

The values in r_1, r_2, and r_3 exist solely to compute the address for the `load` operation. If the program computed each value of r_3 from the preceding one, it could eliminate the operations that define r_1 and r_2. Of course, r_3 would then need an initialization and an update. This would make it a nonlocal name, so it would also need a ϕ-function at both l_1 and l_2.

The right side of Figure 10.12 shows the code after strength reduction, linear-function test replacement, and dead-code elimination. It computes those values formerly in r_3 directly into r_{t_7} and uses r_{t_7} in the `load` operation. The end-of-loop test, which used r_1 in the original code, has been modified to use r_{t_8}. This makes the computation of r_1, r_2, r_3, r_{i_0}, r_{i_1}, and r_{i_2} all dead. They have been removed to produce the final code. Now, the loop contains just five operations, ignoring ϕ-functions, where the original code contained eight. (In translating from SSA-form back to executable code, the ϕ-functions will become copy operations that the register allocator can usually remove.)

If the `multI` operation is more expensive than an `addI`, the savings will be larger. Historically, the high cost of multiplication justified strength reduction. However, even if multiplication and addition have equal costs, the strength-reduced form of the loop may be preferred because it creates a better code shape for later transformations and for code generation. In particular, if the target machine has an auto-increment addressing mode, then the `addI` operation in the loop can be folded into the memory operation. This option simply does

```
    loadI  0          ⇒ r_s0
    loadI  1          ⇒ r_i0
    loadI  100        ⇒ r_100
l1: phi    r_i0,r_i2  ⇒ r_i1
    phi    r_s0,r_s2  ⇒ r_s1
    subI   r_i1,1     ⇒ r_1
    multI  r_1,4      ⇒ r_2
    addI   r_2,@a     ⇒ r_3
    load   r_3        ⇒ r_4
    add    r_s1,r_4   ⇒ r_s2
    addI   r_i1,1     ⇒ r_i2
    cmp_LE r_i2,r_100 ⇒ r_5
    cbr    r_5        → l1,l2
l2: ...
```

Original code

```
    loadI  0          ⇒ r_s0
    loadI  @a         ⇒ r_t6
    addI   r_t6,396   ⇒ r_lim
l1: phi    r_t6,r_t8  ⇒ r_t7
    phi    r_s0,r_s2  ⇒ r_s1
    load   r_t7       ⇒ r_4
    add    r_s1,r_4   ⇒ r_s2
    addI   r_t7,4     ⇒ r_t8
    cmp_LE r_t8,r_lim ⇒ r_5
    cbr    r_5        → l1,l2
l2: ...
```

Strength-reduced code

Figure 10.12: Strength reduction example

not exist for the original multiply.

The rest of this section presents a simple algorithm for strength reduction, called *OSR*, followed by a scheme for linear function test replacement that works with *OSR* to move the end-of-loop test from r_{i_2} to r_{t_8}. *OSR* operates on the SSA graph for the code; Figure 10.13 shows the relationship between the ILOC SSA form for the example and its SSA graph.

Background Strength reduction looks for contexts in which an operation, such as multiply, executes inside a loop and its arguments are (1) a value that does not vary in that loop, called a *region constant*, and (2) a value that varies systematically from iteration to iteration, called an *induction variable*. When it finds this situation, it creates a new induction variable that computes the same sequence of values as the original multiplication in a more efficient way. The restrictions on the form of the multiply operation's arguments ensures that this new induction variable can be computed using additions, rather than multiplications.

We call an operation that can be reduced in this way a *candidate operation*. To simplify the presentation of *OSR*, we only consider candidate operations that have one of the following forms:

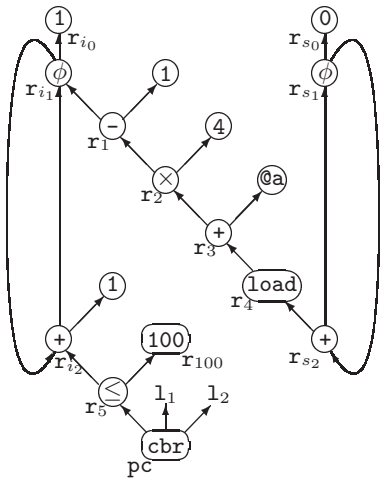
$$x \leftarrow c \times i \qquad x \leftarrow i \times c \qquad x \leftarrow i \pm c \qquad x \leftarrow c + i$$

where c is a region constant and i is an induction variable. The key to finding and reducing candidate operations is efficient identification of region constants and induction variables. An operation is a candidate if and only if it has one of these forms, including the restrictions on arguments.

A region constant can either be a literal constant, such as `10`, or a loop-invariant value—one not modified inside the loop. With the code in SSA form,

```
     loadI  0          ⇒ r_s0
     loadI  1          ⇒ r_i0
     loadI  100        ⇒ r100
l1:  phi    r_i0,r_i2  ⇒ r_i1
     phi    r_s0,r_s2  ⇒ r_s1
     subI   r_i1,1     ⇒ r1
     multI  r1,4       ⇒ r2
     addI   r2,@a      ⇒ r3
     load   r3         ⇒ r4
     add    r4,r_s1    ⇒ r_s2
     addI   r_i1,1     ⇒ r_i2
     cmp_LE r_i2,r100  ⇒ r5
     cbr    r5         → l1,l2
l2:  ...
```

Example in ILOC SSA form



Corresponding SSA graph

Figure 10.13: Relating SSA in ILOC to the SSA graph

the compiler can determine if an argument is loop invariant by checking the location of its sole definition—its definition must dominate the entry of the loop that defines the induction variable. *OSR* can check both of these conditions in constant time. Performing lazy code motion and constant propagation before strength reduction may expose more values as region constants.

Intuitively, an induction variable is a variable whose values in the loop form an arithmetic progression. For the purposes of this algorithm, we can use a much more specific and restricted definition: an induction variable is a strongly-connected component (SCC) of the SSA graph where each operation that updates its value is one of (1) an induction variable plus a region constant, (2) an induction variable minus a region constant, (3) a ϕ-function, or (4) a register-to-register copy from another induction variable. While this definition is much less general than conventional definitions, it is sufficient to let the *OSR* algorithm find and reduce candidate operations. To identify induction variables, *OSR* finds SCCs in the SSA graph and iterates over them to determine if each operation in the SCC is one of these four.

Because *OSR* defines induction variables in the SSA graph and region constants relative to a loop in the CFG, the test to determine if a value is constant relative to the loop containing a specific induction variable is complicated. Consider an operation o of the form $x \leftarrow i \times c$ where i is an induction variable. For o to be a candidate for strength reduction, c must be a region constant with respect to the outermost loop where i varies. To test c, *OSR* must relate the SCC for i in the SSA graph back to a loop in the CFG.

OSR finds the SSA-graph node with the lowest reverse postorder number in the SCC defining i. It considers this node to be the header of the SCC and

records that fact in the header field of each node of the SCC. (Any node in the SSA graph that is not part of an induction variable has its header field set to NULL.) In SSA form, the induction variable's header is the ϕ-function at the start of the outermost loop in which i varies. In an operation $x \leftarrow i \times c$ where i is an induction variable, c is a region constant if the CFG block that contains its definition dominates the CFG block that contains i's header. This condition ensures that c is invariant in the outermost loop where i varies. To perform this test, the SSA construction must produce a map from each SSA node to the CFG block where it originated.

The header field plays a critical role in determining whether or not an operation can be strength reduced. When *OSR* encounters an operation $x \leftarrow y \times z$, it can determine if y is an induction variable by following the SSA graph edge to y's definition and inspecting its header field. A NULL header field indicates that y is not an induction variable. If both y and z have a NULL header field, the operation cannot be strength reduced.

If one of y or z has a non-NULL header field, then *OSR* uses that header field to determine if the other operand is a region constant. Assume y's header is not NULL. To find the CFG block for the entry of the outermost loop where y varies, *OSR* consults the SSA to CFG map, indexed by y's header. If the CFG block containing z's definition dominates the CFG block of y's header, then z is a region constant relative to the induction variable y.

The Algorithm To perform strength reduction, *OSR* must find each candidate operation and determine if one of its arguments is an induction variable and the other is a region constant. If the candidate meets these criteria, *OSR* can reduce it by creating a new induction variable that computes the needed values and replacing the candidate operation with a register-to-register copy from this new induction variable. (It should avoid creating duplicate induction variables.)

Based on the discussion above, we know that *OSR* can identify induction variables by finding SCCs in the SSA graph. It can discover region constants by examining the value's definition. If the definition results from an immediate operation, or its CFG block dominates the CFG block of the induction variable's header, then the value is a region constant. The key is putting these ideas together into an efficient algorithm.

OSR uses Tarjan's strongly connected region finder to drive the entire process. As shown in Figure 10.14, *OSR* takes an SSA graph as its argument and repeatedly applies the strongly connected region finder, *DFS* to it. (This process stops when *DFS* has visited every node in *G*.)

DFS performs a depth-first search of the SSA graph. It assigns each node a number, corresponding to the order in which *DFS* visits the node. It pushes each node onto an explicit stack, and labels the node with the lowest depth-first number that can be reached from its children. When it returns from processing the children, if the lowest node reachable from *n* has *n*'s number, then *n* is the header of an SCC. *DFS* pops nodes off the stack until it reaches its own node; all of those nodes are members of the SCC.

DFS removes SCCs from the stack in an order that simplifies the rest of

```
OSR(G)
  nextNum ← 0
  while there is an unvisited n ∈ G
    DFS(n)

DFS(n)
  n.Num ← nextNum++
  n.Visited ← true
  n.Low ← n.Num
  push(n)
  for each operand o of n
    if o.Visited = false then
      DFS(o)
      n.Low ← min(n.Low,o.Low)
    if o.Num < n.Num and
        o is on the stack
      then n.Low ← min(n.Low,o.Num)
  if n.Low = n.Num then
    SCC ← ∅
    until x = n do
      x ← pop()
      SCC ← SCC ∪ { x }
    Process(SCC)

Process(N)
  if N has only one member n
    then if n is a candidate operation
        then Replace(n,iv,rc)
        else n.Header ← NULL
    else ClassifyIV(N)

ClassifyIV(N)
  IsIV ← true
  for each node n ∈ N
    if n is not a valid update for
        an induction variable
      then IsIV ← false
  if IsIV then
    header ← n ∈ N with the
          lowest RPO number
    for each node n ∈ N
      n.Header ← header
  else
    for each node n ∈ N
      if n is a candidate operation
        then Replace(n,iv,rc)
        else n.Header ← NULL
```

Figure 10.14: Operator Strength Reduction Algorithm

OSR. When an SCC is popped from the stack and passed to *Process*, *DFS* has already visited all of its children in the SSA graph. If we interpret the SSA graph so that its edges run from uses to definitions, as shown in Figure 10.13, then candidate operations are encountered only after their arguments have been passed to *Process*. When *Process* encounters an operation that is a candidate for strength reduction, its arguments have already been classified. Thus, *Process* can examine operations, identify candidates, and invoke *Replace* to rewrite them in strength-reduced form during the depth-first search.

DFS passes each SCC to *Process*. If the SCC consists of a single node *n* that has one of the candidate forms ($x \leftarrow c \times i$, $x \leftarrow i \times c$, $x \leftarrow i \pm c$, or $x \leftarrow c + i$), *Process* passes *n* to *Replace*, which rewrites the code, as described below.[3] If the SCC contains multiple nodes, *Process* passes the SCC to *ClassifyIV* to determine whether or not it is an induction variable.

ClassifyIV examines each node in the SCC to check it against the set of valid updates for an induction variable. If all the updates are valid, the SCC is an induction variable and *Process* sets each node's header field to contain

[3]The process of identifying *n* as a candidate necessarily identifies one operand as an induction variable, iv, and the other as a region candidate, rc.

```
Replace(n,iv,rc)
  result ← Reduce(n.op,iv,rc)
  replace n with a copy from result
  n.header ← iv.header

Reduce(op,iv,rc)
  result ← lookup(op,iv,rc)
  if result is "not found" then
    result ← NewName()
    insert(op,iv,rc,result)
    newDef ← Clone(iv,result)
    newDef.header ← iv.header
    for each operand o of newDef
        if o.header = iv.header
          then rewrite o with
                Reduce(op,o,rc)
        else if op is × or
              newDef.op is φ
          then replace o with
                Apply(op,o,rc)
  return result

Apply(op,o1,o2)
  result ← lookup(op,o1,o2)
  if result is "not found" then
    if o1 is an induction variable
        and o2 is a region constant
      then result ← Reduce(op,o1,o2)
    else if o2 is an induction variable
        and o1 is a region constant
      then result ← Reduce(op,o2,o1)
    else
      result ← NewName()
      insert(op,o1,o2,result)
      Find block b dominated by the
        definitions of o1 and o2
      Create "op o1,o2 ⇒ result"
        at the end of b and set its
        header to NULL
  return result
```

Figure 10.15: Algorithm for the Rewriting Step

the node in the SCC with lowest reverse postorder number. If the SCC is not an induction variable, *ClassifyIV* revisits each node in the SCC to test it as a candidate operation, either passing it to *Replace* or setting its header to show that it is not an induction variable.

Rewriting the Code The remaining piece of *OSR* implements the rewriting step. Both *Process* and *ClassifyIV* call *Replace* to perform the rewrite. Figure 10.15 shows the code for *Replace* and its support functions *Reduce* and *Apply*.

Replace takes three arguments, an SSA-graph node *n*, an induction variable, and a region constant. The latter two are operands to *n*. *Replace* calls *Reduce* to rewrite the operation represented by *n*. Next, it replaces *n* with a copy operation from the result produced by *Replace*. It sets *n*'s header field, and returns.

Reduce and *Apply* do most of the work. They use a hash table to avoid inserting duplicate operations. Since *OSR* works on SSA names, a single global hash table suffices. It can be initialized in *OSR* before the first call to *DFS*.

The plan for *Reduce* is simple. It takes an opcode and its two operands and either creates a new induction variable to replace the computation or returns the name of an induction variable previously created for the same combination of opcode and arguments. It consults the hash table to avoid duplicate work. If the desired induction variable is not in the hash table, it creates the induction variable in a two-step process. First, it calls *Clone* to copy the definition for *iv*,

the induction variable in the operation being reduced. Next, it recurs on the operands of this new definition.

These operands fall into two categories. If the operand is defined inside the SCC, it is part of *iv* so *Reduce* recurs on that operand. This forms the new induction variable by cloning its way around the SCC of the original induction variable *iv*. An operand defined outside the SCC must be either the initial value of *iv* or a value by which *iv* is incremented. The initial value must be a ϕ-function argument from outside the SCC; *Reduce* calls *Apply* on each such argument. *Reduce* can leave an induction-variable increment alone, unless the candidate operation is a multiply. For a multiply, *Reduce* must compute a new increment as the product of the old increment and the original region constant *rc*. It invokes *Apply* to generate this computation.

Apply takes an opcode and two operands, locates an appropriate point in the code, and inserts that operation. It returns the new SSA name for the result of that operation. A few details need further explanation. If this new operation is, itself, a candidate, *Apply* invokes *Reduce* to handle it. Otherwise, *Apply* gets a new name, inserts the operation and returns the result. (If both *o1* and *o2* are constant, *Apply* can evaluate the operation and insert an immediate load.) It locates an appropriate block for the new operation using dominance information. Intuitively, the new operation must go into a block dominated by the blocks that define its operands. If one operand is a constant, *Apply* can duplicate the constant in the block that defines the other operand. Otherwise, both operands must have definitions that dominate the header block, and one must dominate the other. *Apply* can insert the operation immediately after this later definition.

Back to the Example Consider what happens when OSR encounters the example from Figure 10.13. Assume that it begins with the node labelled $\mathtt{r}_{s_2}$ and that it visits left children before right children. It recurs down the chain of operations that define $\mathtt{r}_4$, $\mathtt{r}_3$, $\mathtt{r}_2$, $\mathtt{r}_1$, and $\mathtt{r}_{i_1}$. At $\mathtt{r}_{i_1}$, it recurs on $\mathtt{r}_{i_2}$ and then $\mathtt{r}_{i_0}$. It finds the two single-node SCCs that contain the literal constant one. Neither is a candidate, so *Process* marks them as non-induction variables by setting their headers to NULL.

The first non-trivial SCC that *DFS* discovers contains $\mathtt{r}_{i_1}$ and $\mathtt{r}_{i_2}$. All the operations are valid updates for an induction variable, so *ClassifyIV* marks each node as an induction variable by setting its header field to point to the node with the lowest depth-first number in the SCC—the node for $\mathtt{r}_{i_1}$.

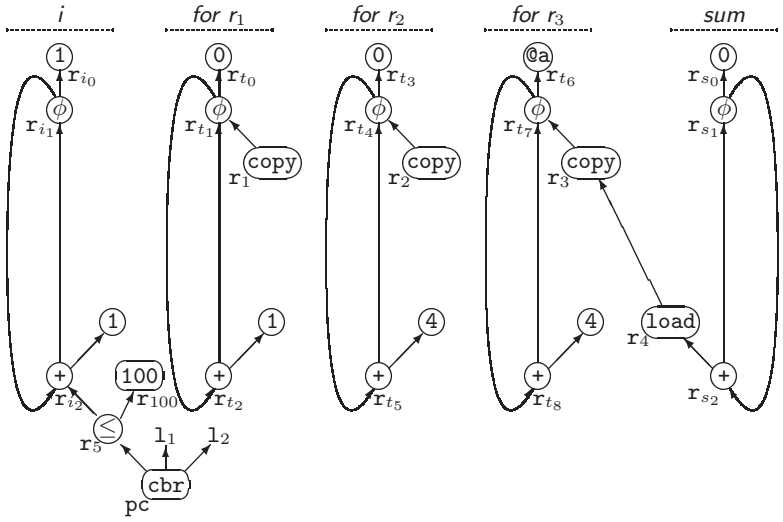


Figure 10.16: Transformed SSA Graph for the Example

Now, *DFS* returns to the node for r_1. Its left child is an induction variable and its right child is a region constant, so it invokes *Reduce* to create an induction variable. In this case, r_1 is $r_{i_1} - 1$, so the induction variable has an initial value equal one less than the initial value of the old induction variable, or zero. The increment is the same. Figure 10.16 shows the SCC that *Reduce* and *Apply* create, under the label "*for* r_1." Finally, the definition of r_1 is replaced with a copy operation, $r_1 \leftarrow r_{t_1}$. The copy operation is marked as an induction variable.

Next, *DFS* discovers the SCC that consists of the node labelled r_2. *Process* discovers that it is a candidate because its left operand (the copy that now defines r_1) is an induction variable and its right operand is a region constant. *Process* invokes *Replace* to create an induction variable that is $r_1 \times 4$. *Reduce* and *Apply* clone the induction variable for r_1, adjust the increment, since the operation is a multiply, and add a copy to r_2.

DFS next passes the node for r_3 to *Process*. This creates another induction variable with `@a` as its initial value, and copies its value to r_3.

Process handles the `load`, followed by the SCC that computes the sum. It finds that none of these operations are candidates.

Finally, *OSR* invokes *DFS* on the unvisited node for the `cbr`. *DFS* visits the comparison, the previously-marked induction variable, and the constant 100. No further reductions occur.

The SSA graph in Figure 10.16 shows all of the induction variables created by this process. The induction variables labelled "*for* r_1" and "*for* r_2" are dead. The induction variable for *i* would be dead, except that the end-of-loop test still uses it. To eliminate this induction variable, the compiler can apply linear-function test replacement to transfer the test onto the induction variable for r_3.

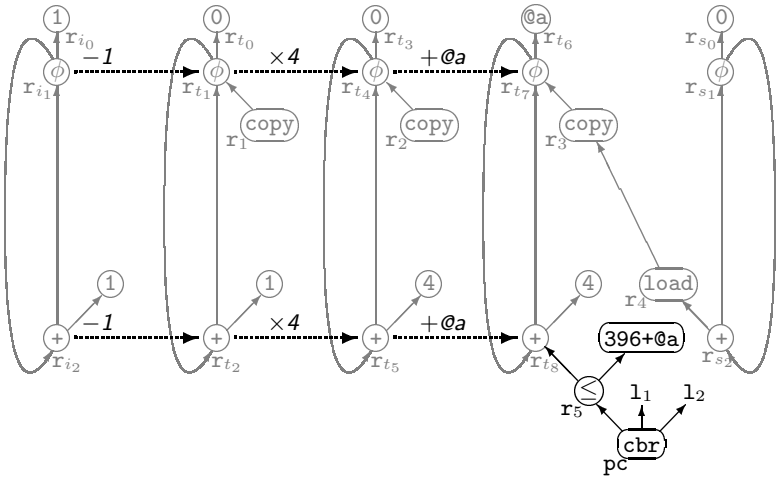


Figure 10.17: Example after LFTR

Linear-Function Test Replacement

Strength reduction often eliminates all uses of an induction variable, except for an end-of-loop test. In that case, the compiler may be able to rewrite the end-of-loop test to use another induction variable in the loop. If the compiler can remove this last use, it can eliminate the original induction variable as dead code. This transformation is called linear-function test replacement (LFTR).

To perform LFTR, the compiler must (1) locate comparisons that rely on otherwise unneeded induction variables, (2) locate an appropriate new induction variable that the comparison should use, (3) compute the correct region constant for the rewritten test, and (4) rewrite the code. Having LFTR cooperate with *OSR* can simplify all of these tasks to produce a fast, effective transformation.

The operations that LFTR targets compare the value of an induction variable against a region constant. *OSR* examines each operation in the program to determine if it is a candidate for strength reduction. It can easily and inexpensively build a list of all the comparison operations that involve induction variables. After *OSR* finishes its work, LFTR should revisit each of these comparisons. If the induction-variable argument of a comparison was reduced by *OSR*, LFTR should retarget the comparison to use the new induction variable.

To facilitate this process, *Reduce* can record the mathematical relationship used to derive each new induction variable that it creates. It can insert a special LFTR edge from each node in the original induction variable to the corresponding node in its reduced counterpart and label it with the opcode and region constant of the candidate operation responsible for creating the new induction variable. Figure 10.17 shows these edges added to the SSA graph for the example. The example involved a series of reductions; these create a chain of edges with the appropriate labels. Starting from the original induction variable, we find the

labels *-1*, *x4*, and *+@a*.

When LFTR finds a comparison that should be replaced, it can follow the edges from its induction-variable argument to the final induction variable that resulted from a chain of one or more reductions. The comparison should use this induction variable with an appropriate new region constant.

The labels on the edges that LFTR traverses describe the transformation that must be applied to the original region constant to derive the new region constant. In the example, the trail of edges leads from $\mathtt{r}_{i_2}$ leads to $\mathtt{r}_{t_8}$, and produces the value `(100 - 1)` $\times$ `4 + @a` for the transformed test. Figure 10.17 shows the edges and the rewritten test.

This version of LFTR is simple, efficient, and effective. It relies on close collaboration with *OSR* to identify comparisons that might be retargeted, and to leave behind a record of the reductions that it performed. Using these two data structures, LFTR can find comparisons to retarget, find the appropriate place to retarget them, and find the necessary transformation for the comparison's constant argument.

10.3.4 Enabling Other Techniques

Often, an optimizer includes passes whose primary purpose is to create or expose opportunities for other transformations. In some cases, the transformation changes the shape of the code to make it more amenable to optimization. In other cases, the transformation creates a point in the code where specific conditions hold that make another transformation safe. By directly creating the necessary code shape, these enabling transformations reduce the sensitivity of the optimizer to the shape of the input code.

Several enabling transformations are described in other parts of the book. Inline substitution (§8.7.2) merges two procedures to eliminate the overhead of a procedure call and to create a larger context for specialization. Block cloning (§8.7.1) replicates individual blocks to eliminate branches and to create situations where the compiler can derive more precise knowledge of the context that a block inherits from its CFG predecessors. For example, §12.4.2 describes how block cloning can improve the results of instruction scheduling. This section presents three simple enabling transformations: *loop unrolling*, *loop unswitching*, and *renaming*.

Loop Unrolling

Loop unrolling is one of the oldest enabling transformations. To unroll a loop, the compiler replicates the loop's body and adjusts the logic that controls the number of iterations performed. Consider the simple loop shown in the upper left portion of Figure 10.18.

If the compiler replaces the loop's body with four copies of itself, unrolling the loop by a factor of four, it can execute the same useful work with one quarter the number of comparisons and branches. If the compiler knows the value of `n`, say 100, and the unrolling factor evenly divides `n`, then the unrolled loop has the simple form shown in the upper right part of the figure.

```
do i = 1 to n by 1
   a(i) = a(i) + b(i)
   end
```

Original loop

```
do i = 1 to 100 by 4
   a(i)   = a(i) + b(i)
   a(i+1) = a(i+1) + b(i+1)
   a(i+2) = a(i+2) + b(i+2)
   a(i+3) = a(i+3) + b(i+3)
   end
```

Unrolled by 4, n = 100

```
i = 1
do while (i+3 ≤ n)
   a(i)   = a(i) + b(i)
   a(i+1) = a(i+1) + b(i+1)
   a(i+2) = a(i+2) + b(i+2)
   a(i+3) = a(i+3) + b(i+3)
   i = i + 4
   end
do while (i ≤ n)
   a(i) = a(i) + b(i)
   i = i + 1
   end
```

Unrolled by 4, arbitrary n

```
i = 1
if (mod(n,2) > 0) then
   a(i) = a(i) + b(i)
   i = i + 1
if (mod(n,4) > 1) then
   a(i)   = a(i) + b(i)
   a(i+1) = a(i+1) + b(i+1)
   i = i + 2
do j = i to n by 4
   a(j)   = a(j) + b(j)
   a(j+1) = a(j+1) + b(j+1)
   a(j+2) = a(j+2) + b(j+2)
   a(j+3) = a(j+3) + b(j+3)
   end
```

Unrolled by 4, arbitrary n

Figure 10.18: Unrolling a simple loop

When the loop bounds are unknown, unrolling requires some additional logic to support values of `n` for which `mod(n,4)` $\neq 0$. The version of the loop in the figure's lower left part shows a simple way to handle these cases. The version in the lower right portion achieves the same result with a little more code. For a more complex loop body, the lower right version may allow some improvements for the two iterations handled by the middle case.

The excerpt from `dmxpy` in the LINPACK library, shown in §8.2.1, uses the scheme from the lower right corner. The full code includes the cases for one, two, four, eight, and sixteen iterations. In each case, the unrolled loop contains another loop, so the amount of work in the inner loop justifies the extensive unrolling of the outer loop.

Loop unrolling reduces the total number of operations executed by the program. It also increases the program's size. (If the loop body grows too large for the instruction cache, the resulting cache misses can overcome any benefit from lower loop overhead.) However, the primary justification for unrolling loops is to create a better code shape for other optimizations.

Unrolling has two key effects that can create opportunities for other trans-

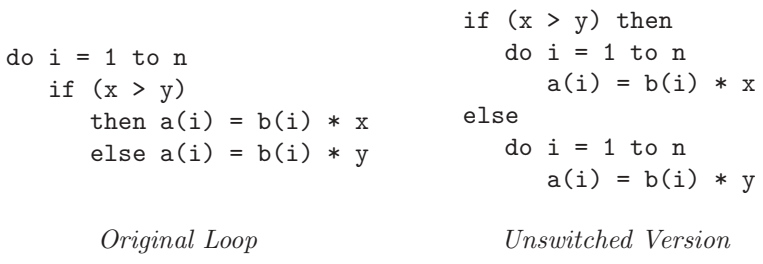


Figure 10.19: Unswitching a short loop

formations. It increases the number of operations in the loop body. For short loops and long branch latencies, this can produce a better schedule. In particular, it may give the scheduler more independent operations that it can execute in the same schedule. It may give the scheduler enough operations to fill branch delay slots. It can allow the scheduler to move consecutive memory accesses together. This improves locality and opens up the possibility of using memory operations that process more data at once.

As a final note, if the loop computes a value in one iteration that is used in a later iteration—a loop-carried data-dependence—and if copy operations are needed to preserve the value for later use, unrolling can eliminate the copy operations. With multiple cycles of copies, unrolling by the least common multiple of the various cycle lengths will eliminate all the copies.

Loop Unswitching

Loop unswitching hoists loop-invariant control-flow operations out of a loop. If the predicate in an *if-then-else* construct is loop invariant, then the compiler can rewrite the loop by pulling the *if-then-else* out of the loop and generating a tailored copy of the loop inside each half of this new *if-then-else*. Figure 10.19 shows this transformation for a short loop.

Unswitching is an enabling transformation; it allows the compiler to tailor loop bodies in ways that are otherwise hard to achieve. After unswitching, the remaining loops contain less control flow. They execute fewer branches and other operations to support those branches. This can lead to better scheduling, better register allocation, and faster execution. If the original loop contained loop-invariant code that was inside the *if-then-else*, then LCM could not move it out of the loop. After unswitching, LCM easily finds and removes such redundancies.

Unswitching also has a simple, direct effect that can improve a program—it moves the branching logic that governs the loop-invariant conditional out of the loop. Moving control flow out of loops is difficult. Techniques based on data-flow analysis, like LCM, have problems moving such constructs because the transformation modifies the CFG on which the analysis relies. Techniques based on value numbering can recognize some cases where the predicates controlling *if-then-else* constructs are identical, but they will not try to remove the construct from a loop.

Renaming

Most of the transformations presented in this chapter involve rewriting or reordering the operations in the program. Having the right code shape can expose opportunities for optimization. Similarly, having the right set of names can expose opportunities for optimization.

In local value numbering, for example, we saw the following example:

$a \leftarrow x + y$	$a_0 \leftarrow x_0 + y_0$
$b \leftarrow x + y$	$b_0 \leftarrow x_0 + y_0$
$a \leftarrow 17$	$a_1 \leftarrow 17$
$c \leftarrow x + y$	$c_0 \leftarrow x_0 + y_0$
Original code	SSA form

In the original code, local value numbering can recognize that all three computations of $x + y$ produce the same result. However, it cannot replace the final occurrence of $x + y$ because the intervening assignment to a has destroyed the copy of $x + y$ that it recognizes.

Converting the code to SSA form produces a name space like the one shown on the right. With an SSA name space, $x_0 + y_0$ remains available at the final operation, so local value numbering can replace the evaluation with a reference to a_0. (The alternative is to alter the local value-numbering algorithm so that it recognizes b as another copy of $x + y$. Renaming is the simpler and more general solution.)

In general, careful use of names can expose additional opportunities for optimization by making more facts visible to analysis and by avoiding some of the side effects that come with reuse of storage. For data-flow-based optimizations, like LCM, the analysis relies on lexical identity—redundant operations must have the same operation and their operands must have the same names.[4] A scheme that encodes some value identity, perhaps derived from value numbering, into the name space can expose more redundancies to LCM and let it eliminate them.

In instruction scheduling, names create the dependences that limit the scheduler's ability to rearrange operations. If the reuse of a name reflects the actual flow of values, these dependences are a critical part of correctness. If the reuse of a name occurs because the register allocator has placed two distinct values in the same register for efficiency, these dependences can unnecessarily restrict the schedule—leading, in some cases, to less efficient code.

Renaming is a subtle issue. The SSA construction renames all the values in the program according to a particular discipline. This name space helps in many optimizations. The naming conventions described in the discussion of lazy code motion, and in the digression on page 191, simplify the implementation of many transformations by creating a one-to-one mapping between the name used for a value and the textual form of the operation that computed the value. Compiler writers have long recognized that moving operations around in the control-flow

[4] LCM does not use SSA names because those names obscure lexical identity. Recreating that lexical identity incurs extra cost; as a result, an SSA-based LCM will run more slowly than the version described in §10.3.2.

graph (and, in fact, changing the CFG itself) can be beneficial. In the same way, they should recognize that the compiler need not be bound by the name space introduced by the programmer or by the translation from a source language to a specific IR. Renaming values into a name space appropriate to the task at hand can improve the effectiveness of many optimizations.

10.3.5 Redundancy Elimination

Chapter 8 uses redundancy elimination as its primary example to explore the different scopes of optimization. It describes local value numbering, superlocal value numbering, and dominator-based value numbering—all based on a bottom-up, detail-oriented approach that uses hashing to recognize values that must be equivalent. It shows the use of available expressions to perform global common-subexpression elimination as a way of introducing global analysis and transformation, and makes the point that global optimization typically requires a separation between analysis and transformation.

Earlier in this chapter, LCM appears as an example of code motion. It extends the data-flow approach pioneered with available expressions to a framework that unifies code motion and redundancy elimination. Hoisting eliminates identical operations to reduce the code size; this reduction does not, typically, decrease the number of operations that the program executes.

10.4 Advanced Topics

Most of the examples in this chapter have been chosen to illustrate a specific effect that the compiler can use to speed up the executable code. Sometimes, performing two optimizations together can produce results that cannot be obtained with any combination of applying them separately, The next subsection shows one such example—combining constant propagation with unreachable code elimination. Section 10.4.2 briefly describes some of the other objective functions that compilers consider. Finally, §10.4.3 discusses some of the issues that arise in choosing a specific application order for the optimizer's set of transformations.

10.4.1 Combining Optimizations

Sometimes, formulating two distinct optimizations together and solving them jointly can produce results that cannot be obtained by any combination of the optimizations run separately. As an example, consider the sparse simple constant propagation algorithm described in §10.3.3. It assigns a lattice value to the result of each operation in the SSA form of the program. When it halts, it has tagged every definition with a lattice value that is either $\top$, $\bot$ or a constant. A definition can only have the value $\top$ if it relies on an uninitialized variable, indicating a logical problem in the code being analyzed.

Sparse simple constant propagation assigns a lattice value to the argument used by a conditional branch. If the value is $\bot$, then either branch target is reachable. If the value is neither $\bot$ nor $\top$, then the operand must have a known

```
SSAWorkList ← ∅
CFGWorkList ← {n0}
for each block b
  mark b as unreachable
  for each operation o in b
    Value(o) ← ⊤

while (CFGWorkList ≠ ∅ or
       SSAWorkList ≠ ∅)
  if CFGWorkList ≠ ∅ then
    remove b from CFGWorkList
    mark b as reachable
    simultaneously model all the
        φ-functions in b
    model, in order, each
        operation o in b

  if SSAWorkList ≠ ∅ then
    remove s = ⟨u,v⟩ from SSAWorkList
    let o be the operation that uses v
    if Value(o) ≠ ⊥ then
      t ← result of modeling o
      if t ≠ Value(o) then
        Value(o) ← t
        for each SSA edge e = ⟨o,x⟩
          if block(x) is reachable
            then add e to SSAWorkList
```

The Algorithm

```
x ← c:                          (for constant c)
  Value(x) ← c

x ← φ(y,z):
  Value(x) ← Value(y) ∧ Value(z)

x ← y op z:
  if Value(y) ≠ ⊥ & Value(z) ≠ ⊥
    then Value(x) ← interpretation
                of Value(y) op Value(z)

cbr ri → l1,l2:
  if ri = ⊥ or ri = TRUE
    and block l1 is unreachable
    then add block l1 to CFGWorkList
  if ri = ⊥ or ri = FALSE
    and block l2 is unreachable
    then add block l2 to CFGWorkList

jump → l1:
  if block l1 is unreachable
    then add block l1 to CFGWorkList
```

Modeling Rules

Figure 10.20: Sparse Conditional Constant Propagation

value and the compiler can rewrite the branch with a jump to one of its two targets, simplifying the CFG. Since this removes an edge from the CFG, it may remove the last edge entering the block labelled with the removed branch target, making that block unreachable. In principle, constant propagation can ignore any effects of an unreachable block. Sparse simple constant propagation has no mechanism to take advantage of this knowledge.

We can extend the sparse simple constant algorithm to capitalize on these observations. The result, called *sparse conditional constant propagation* (SCCP), appears in Figure 10.20. The left side shows the algorithm. The right side sketches the modeling rules for the kinds of operations that SCCP must process.

To avoid including the effects of unreachable operations, SCCP handles both initialization and propagation differently than the sparse simple constant algorithm. First, SCCP marks each block with a reachability tag. Initially, each block's tag is set to indicate that the block is unreachable. Second, SCCP must

initialize the lattice value for each SSA name to $\top$. The earlier algorithm can handle assignments of known-constant values during initialization (*e.g.*, for $x_i \leftarrow 17$, it can initialize *Value*(x_i) to 17). SCCP cannot change the lattice value until it proves that the assignment is reachable. Third, the algorithm needs two worklists for propagation: one for blocks in the CFG and the other for edges in the SSA graph. Initially, the CFG worklist contains only the entry node n_0 while the SSA worklist is empty. The iterative propagation runs until it exhausts both worklists (a classic, albeit complex, fixed-point calculation).

To process a block b from the CFG worklist, SCCP first marks b as reachable. Next, it models the effect of all the ϕ-functions in b, taking care to read the lattice values of all the relevant arguments before redefining the lattice values of their outputs. (Recall that all the ϕ-functions in a block execute simultaneously.) Next, SCCP models the execution of each operation in b in a linear pass over the block. The right side of Figure 10.20 shows a set of typical modeling rules. These evaluations may change the lattice values of the SSA names defined by the operations.

Any time that SCCP changes the lattice value for a name, it must examine the SSA graph edges that connect the operation defining the changed value to subsequent uses. For each such edge, $s = \langle u,v \rangle$, if the block containing v is reachable, SCCP adds the edge s to the SSA worklist. Uses in an unreachable block are evaluated once SCCP discovers that the block is reachable.

The last operation in b must be either a jump or a branch. If it is a jump to a block marked as unreachable, SCCP adds that block to the CFG worklist. If it is a branch, SCCP examines the lattice value of the controlling conditional expression. This indicates that one or both branch targets are reachable. If this selects a target that is not yet marked as reachable, SCCP adds it to the CFG worklist.

After the propagation step, a final pass is required to replace operations that have operands with *Value* tags other than $\bot$. It can specialize many of these operations. It should also rewrite branches that have known outcomes with the appropriate jump operations. (This lets later passes remove the code and simplify the control flow, as in §10.3.1). SCCP cannot rewrite the code until it knows the final lattice value for each definition, since a *Value* tag that indicates a constant value can later become $\bot$.

Subtleties in Evaluating and Rewriting Operations Some subtle issues arise in modeling individual operations. For example, if the algorithm encounters a multiply operation with operands $\top$ and $\bot$, it might conclude that the operation produces $\bot$. Doing so, however, is premature. Subsequent analysis might lower the $\top$ to zero, so that the multiply produces a value of zero. If SCCP uses the rule $\top \times \bot \rightarrow \bot$, it introduces the potential for non-monotonic behavior—the multiply's value might follow a sequence $\{\top, \bot, 0\}$. This can increase the running time of the algorithm, since the time bound depends on monotonic progress through a shallow lattice. Equally important, it can incorrectly cause other values to reach $\bot$.

To address this, SCCP should use three rules for multiplies that involve $\bot$,

as follows: $\top \times \bot \rightarrow \top$, $\alpha \times \bot \rightarrow \bot$, $\alpha \neq 0$, and $0 \times \bot \rightarrow 0$. This same effect occurs with any operation for which the value of one argument can completely determine the result. Other examples include a shift by more than the word length, a logical AND with zero, and a logical OR with all ones.

Some rewrites have unforeseen consequences. For example, replacing $4 \times x$, for nonnegative x, with a shift replaces a commutative operation with a noncommutative one. If the compiler subsequently tries to rearrange expressions using commutativity, this early rewrite forecloses an opportunity. This kind of interaction can have noticeable effects on code quality. To choose when the compiler should convert $4 \times x$ into a shift, the compiler writer must consider the order in which optimizations will be applied.

Effectiveness SCCP finds constants that the sparse simple constant algorithm cannot find. Similarly, it discovers unreachable code that cannot be found by any combination of the algorithms described in §10.3.1. It derives its power from combining reachability analysis with the propagation of lattice values. It can eliminate some CFG edges because the lattice values are sufficient to determine which path a branch takes. It can ignore SSA edges that arise from unreachable operations (by initializing those definitions to $\top$) because those operations will be evaluated if the block becomes reachable. The power of SCCP arises from the interplay between these ideas—constant propagation and reachability.

If reachability played no role in determining the lattice values, then the same effects could be achieved by performing constant propagation (and rewriting constant-valued branches as jumps) followed by unreachable-code elimination. If constant propagation played no role in reachability, then the same effects could be achieved by the other order—unreachable-code elimination followed by constant propagation. The power of SCCP to find simplifications beyond those combinations comes precisely from the fact that the two optimizations are interdependent.

10.4.2 Other Objectives for Optimization

Generating Smaller Code In some applications, the size of the compiled code is important. If the application is transmitted across a relatively slow communications link before it executes, then the perceived running time is the sum of the download time plus the running time. This places a premium on code size, since the user waits while the code is transmitted. Similarly, in many embedded applications, the code is stored in a permanent, read-only memory (ROM). Since larger ROMs cost more money, code size becomes an economic issue.

The compiler writer can attack this problem in several ways. It can apply transformations that directly shrink the code.

- *Hoisting* (described above in §9.2.4), shrinks the code by replacing multiple identical operations with a single, equivalent operation. As long as the expression is very busy at the point of insertion, hoisting should not lengthen any of the execution paths.
- *Sinking* moves common code sequences forward in the CFG to a point

where one copy of the sequence suffices. In *cross jumping*, a specialized form of sinking, the compiler examines all the branches that target the same label. If the same operation precedes each branch to the label, the compiler can move the operation to the label and keep just one copy of it. This eliminates duplication without lengthening the execution paths.

- *Procedure abstraction* uses pattern matching techniques to find repeated code sequences and replace them with calls to a single common implementation. If the common code sequence is longer than the sequence required for the call, this saves space. It makes the code slower, since every abstracted code sequence is replaced with a jump into the abstracted procedure and a jump back. Procedure abstraction can be applied across the whole program to find common sequences from different procedures.

As another approach, the compiler can avoid using transformations that enlarge the code. For example, loop unrolling typically expands the code. Similarly, LCM may enlarge the code by inserting new operations. By choosing algorithms carefully and avoiding those that cause significant code growth, the compiler writer build a compiler that always produces compact code.

Avoiding Page Faults and Instruction-Cache Misses In some environments, the overhead from page faults and instruction-cache misses make it worthwhile to transform the code in ways that improve the code's memory locality. The compiler can use several distinct but related effects to improve the paging and cache behavior of the instruction stream.

- *Procedure Placement:* If A calls B often, the compiler would like to ensure that A and B occupy adjacent locations in memory. If they fit on the same page, this can reduce the program's working set. Placing them in adjacent locations also reduces the likelihood of a conflict in the instruction cache.

- *Block Placement:* If block b_i ends in a branch and the compiler knows that the branch usually transfers control to b_j, then it can place b_i and b_j in contiguous memory. This makes b_j the fall-through case of the branch (and most processors both support and favor the fall-through case). It also increases the effectiveness of any hardware prefetching mechanism in the instruction cache.

- *Fluff removal:* If the compiler can determine that some code fragments execute rarely—that is, they are the (mostly) untaken targets of branches—then it can move them to distant locations. Such rarely executed code needlessly fills the cache and decreases the density of useful operations brought into the processor and executed. (For exception handlers, it may pay to keep the code on the same page, but move it out of line.)

To implement these transformations effectively, the compiler needs accurate information about how often each path through the code is taken. Typically, gathering this information requires a more complex compilation system that

collects execution-profile data and relates that data back to the code. The user compiles an application and runs it on "representative" inputs. The profile data from these runs is then used to optimize the code in a second compilation. An alternative, as old as compilation, is to build a model of the CFG and estimate execution frequencies using reasonable transition probabilities.

10.4.3 Choosing an Optimization Sequence

Choosing a specific set of transformations and an order for applying them is a critical task in the design of an optimizing compiler. For any particular problem, many distinct techniques exist. Each of them catches a different set of cases. Many of them address parts of several problems.

To make this concrete, recall the methods that we have presented for eliminating redundancies. These include three value numbering techniques (local value numbering, superlocal value numbering, and dominator-based value numbering) and two techniques based on data-flow analysis (global common-subexpression elimination based on available expressions and lazy code motion). The value-numbering techniques find and eliminate redundancies using a value-based notion of redundancy. They differ in the scope of optimization—covering single blocks (local value numbering), extended basic blocks (EBB value numbering), and the entire CFG minus back edges (dominator-based value numbering). They also perform constant propagation and use algebraic identities to remove some useless operations. The data-flow-based techniques use a lexical notion of redundancy—two operations are equivalent only if they have the same name. Global common-subexpression elimination only removes redundancies, while lazy code motion removes redundancies and partial redundancies and performs code motion.

In designing an optimizing compiler, the compiler writer must decide how to remove redundancies. Choosing among these five methods involves decisions about which cases are important, the relative difficulty of implementing the techniques, and how the compile-time costs compare to the run-time benefits. To complicate matters further, more techniques exist.

Equally difficult, optimizations that address different effects interact with each other. Optimizations can create opportunities for other optimizations, as constant propagation and lazy code motion improve strength reduction by revealing more values as region constants. Symmetrically, optimizations can make other optimizations less effective, as redundancy elimination can complicate register allocation by making values live over longer regions in the program. The effects of two optimizations can overlap. In a compiler that implements sparse conditional constant propagation, the constant-folding capabilities of the value-numbering techniques are less important.

After selecting a set of optimizations, the compiler writer must choose an order in which to apply them. Some constraints on the order are obvious; for example, it is worth running constant propagation and code motion before strength reduction. Others are less obvious; for example, should loop unswitching precede strength reduction or not? Should the constant-propagation pass

convert $x \leftarrow y \times 4$, for $y \geq 0$, to a shift operation? The decision may depend on which passes precede constant propagation and which passes follow it. Finally, some optimizations might be run multiple times. Dead-code elimination cleans up after strength reduction. After strength reduction, the compiler might run constant propagation again in an attempt to convert any remaining multiplies into shifts. Alternatively, it might use local value numbering to achieve the same effect, reasoning that any global effects have already been caught.

If the compiler writer intends to provide different levels of optimization, this entails designing several compilation sequences, each with its own set of rationales and its own set of passes. Higher levels of optimization typically add more transformations. They may also repeat some optimizations to capitalize on opportunities created by earlier transformations. See Muchnick [262] for suggested orders in which to perform optimizations.

10.5 Summary and Perspective

The design and implementation of an optimizing compiler is a complex undertaking. This chapter has introduced a conceptual framework for thinking about transformations—the taxonomy of effects. Each category in the taxonomy is represented by several examples—either in this chapter or elsewhere in the book.

The challenge for the compiler writer is to select a set of transformations that work well together to produce good code—code that meets the user's needs. The specific transformations implemented in a compiler determine, to a large extent, the kinds of programs for which it will produce good code.

Chapter Notes

While the algorithms presented in this chapter are modern, many of the basic ideas were well known in the 1960s and 1970s. Dead code elimination, code motion, strength reduction, and redundancy elimination are all described by Allen in 1969 [12] and in Cocke [86]. A number of survey papers from the 1970's provide overvies of the state of the field [15, 28, 308]. Modern books by Morgan [260] and Muchnick [262] both discuss the design, structure, and implementation of optimizing compilers. Wolfe [335] and Allen [19] focus on dependence-based analysis and transformations.

Dead algorithm implements a mark-sweep style of dead code elimination that was introduced by Kennedy [203, 206]. It is reminiscent of the Schorr-Waite marking algorithm [299]. *Dead* is specifically adapted from the work of Cytron *et al.* [104, Section 7.1]. *Clean* was developed and implemented in 1992 by Rob Shillner [245].

LCM improves on Morel and Renvoise's classic algorithm for partial redundancy elimination [259]. That original paper inspired many improvements, including [76, 121, 312, 124]. Knoop's lazy code motion [215] improved code placement; the formulation in §10.3.2 uses equations from Drechsler [125]. Bodik combined this approach with replication to find and remove all redundant

code [41].

Hoisting appears in the Allen-Cocke catalog as a technique for reducing code space [15]. The formulation using very busy expressions appears in several places, including Fischer [140]. Sinking or cross-jumping is described by Wulf *et al.* [339].

Both peephole optimization and tail-recursion elimination date to the early 1960s. Peephole optimization was first described by McKeeman [253]. Tail-recursion elimination is older; folklore tells McCarthy described it at the chalkboard during a talk in 1963. Steele's thesis [314] is a classic reference for tail-recursion elimination.

The sparse simple constant algorithm, SSCP, is due to Reif [285]. Wegman and Zadeck reformulate SSCP to use SSA form and present the SCCP algorithm from §10.4.1 [328, 329]. Their work clarified the distinction between optimistic and pessimistic algorithms; Click discusses the same issue from a set-building perspective [79].

Operator strength reduction has a rich history. One family of strength-reduction algorithms developed out of work by Allen, Cocke, and Kennedy [204, 83, 85, 18, 250]. The *OSR* algorithm fits in this family [101]. Another family of algorithms grew out of the data-flow approach to optimization exemplified by the LCM algorithm; techniques in this family include [118, 192, 198, 120, 216, 209, 169]. The version of *OSR* in §10.3.3 only reduces multiplications. Allen shows the reduction sequences for many other operators [18]; extending *OSR* to handle these cases is straightforward.

Loop optimizations have been studied extensively [27, 19]; Kennedy used unrolling to avoid copy operations at the end of a loop [202]. Cytron presents an interesting alternative to unswitching [105]. Wolf and Lam unified the treatment of a set of loop optimizations called the *unimodular* transformations [334]. McKinley gives practical insight into the impact of memory optimizations on performance [89, 254].

Combining optimizations, as in SCCP, often leads to improvements that cannot be obtained by independent application of the original optimizations. Value numbering combines redundancy elimination, constant propagation, and simplification of algebraic identities [50]. LCM combines elimination of redundancies & partial redundancies with code motion [215]. Click [81] combines Alpern's partitioning algorithm [20] with SCCP [329]. Many authors have combined register allocation and instruction scheduling [158, 267, 268, 261, 45, 274, 298].

Shrinking existing code or generating smaller programs has been a persistent theme in theliterature. Fabri [134] showed algorithms to reduce data-memory requirements by automatic generation of storage overlays. Marks [249] had the compiler synthesize both a compact, program-specific instruction set and an interpreter for that instruction set. Fraser [148] used procedure abstraction based on suffix-trees. Recent work has focused on compressing code for transmission [130, 145], and on directly generating small code—either with hardware assistance for decoding [233, 234] or without it [99].

Modern algorithms for code placement begin with Pettis and Hansen [273]. Later work includes work on branch alignment [59, 340] and code layout [88,

73, 156]. These optimizations improve performance by improving the behavior of code memory in a hierarchical memory system. They also eliminate branches and jumps.