Chapter 8

Introduction to Optimization

CHAPTER OVERVIEW

To improve the quality of the code that it generates, an optimizing compiler analyzes the code and rewrites it into a more efficient form. This chapter introduces the problems and techniques of code optimization and presents key concepts through a series of example optimizations. Chapter 9 expands on this material with a deeper exploration of program analysis. Chapter 10 provides a broader coverage of optimizing transformations.

Keywords: Optimization, Safety, Profitability, Scope of Optimization, Analysis, Transformation

8.1 INTRODUCTION

The compiler's front end translates the source-code program into some intermediate representation (IR). The back end translates the IR program into a form where it can execute directly on the target machine, either a hardware platform such as a commodity microprocessor or a virtual machine as in Java. Between these processes sits the compiler's middle section, its optimizer. The task of the optimizer is to transform the IR program produced by the front end in a way that will improve the quality of the code produced by the back end. "Improvement" can take on many meanings. Often, it implies faster execution for the compiled code. It can also mean an executable that uses less energy when it runs or that occupies less space in memory. All of these goals fall into the realm of optimization.

This chapter introduces the subject of code optimization and provides examples of several different techniques that attack different kinds of inefficiencies and operate on different regions in the code. Chapter 9 provides a deeper treatment of some of the techniques of program analysis that are used

Engineering a Compiler. DOI: 10.1016/B978-0-12-088478-0.00008-6

to support optimization. Chapter 10 describes additional code-improvement transformations.

Conceptual Roadmap

The goal of code optimization is to discover, at compile time, information about the runtime behavior of the program and to use that information to improve the code generated by the compiler. Improvement can take many forms. The most common goal of optimization is to make the compiled code run faster. For some applications, however, the size of the compiled code outweighs its execution speed; consider, for example, an application that will be committed to read-only memory, where code size affects the cost of the system. Other objectives for optimization include reducing the energy cost of execution, improving the code's response to real-time events, or reducing total memory traffic.

Optimizers use many different techniques to improve code. A proper discussion of optimization must consider the inefficiencies that can be improved and the techniques proposed for doing so. For each source of inefficiency, the compiler writer must choose from multiple techniques that claim to improve efficiency. The remainder of this section illustrates some of the problems that arise in optimization by looking at two examples that involve inefficiencies in array-address calculations.

Safety
A transformation is *safe* when it does not change the results of running the program.

Profit
A transformation is *profitable* to apply at some point when the result is an actual improvement.

Before implementing a transformation, the compiler writer must understand when it can be *safely* applied and when to expect *profit* from its application. Section 8.2 explores safety and profitability. Section 8.3 lays out the different granularities, or scopes, over which optimization occurs. The remainder of the chapter uses select examples to illustrate different sources of improvement and different scopes of optimization. This chapter has no "Advanced Topics" section; Chapters 9 and 10 serve that purpose.

Overview

Opportunities for optimization arise from many sources. A major source of inefficiency arises from the implementation of source-language abstractions. Because the translation from source code into IR is a local process—it occurs without extensive analysis of the surrounding context—it typically generates IR to handle the most general case of each construct. With contextual knowledge, the optimizer can often determine that the code does not need that full generality; when that happens, the optimizer can rewrite the code in a more restricted and more efficient way.

A second significant source of opportunity for the optimizer lies with the target machine. The compiler must understand, in detail, the properties

of the target that affect performance. Issues such as the number of functional units and their capabilities, the latency and bandwidth to various levels of the memory hierarchy, the various addressing modes supported in the instruction set, and the availability of unusual or complex operations all affect the kind of code that the compiler should generate for a given application.

Historically, most optimizing compilers have focused on improving the runtime speed of the compiled code. Improvement can, however, take other forms. In some applications, the size of the compiled code is as important as its speed. Examples include code that will be committed to read-only memory, where size is an economic constraint, or code that will be transmitted over a limited-bandwidth communications channel before it executes, where size has a direct impact on time to completion. Optimization for these applications should produce code that occupies less space. In other cases, the user may want to optimize for criteria such as register use, memory use, energy consumption, or response to real-time events.

Optimization is a large and detailed subject whose study could fill one or more complete courses (and books). This chapter introduces the subject and some of the critical ideas from optimization that play a role in Chapters 11, 12, and 13. The next two chapters delve more deeply into the analysis and transformation of programs. Chapter 9 presents an overview of static analysis. It describes some of the analysis problems that an optimizing compiler must solve and presents practical techniques that have been used to solve them. Chapter 10 examines so-called scalar optimizations—those intended for a uniprocessor—in a more systematic way.

8.2 BACKGROUND

Until the early 1980s, many compiler writers considered optimization as a feature that should be added to the compiler only after its other parts were working well. This led to a distinction between *debugging compilers* and *optimizing compilers.* A debugging compiler emphasized quick compilation at the expense of code quality. These compilers did not significantly rearrange the code, so a strong correspondence remained between the source code and the executable code. This simplified the task of mapping a runtime error to a specific line of source code; hence the term *debugging* compiler. In contrast, an optimizing compiler focuses on improving the running time of the executable code at the expense of compile time. Spending more time in compilation often produces better code. Because the optimizer often moves operations around, the mapping from source code to executable code is less transparent, and debugging is, accordingly, harder.

As RISC processors have moved into the marketplace (and as RISC implementation techniques were applied to CISC architectures), more of the burden for runtime performance has fallen on compilers. To increase performance, processor architects have turned to features that require more support from the compiler. These include delay slots following branches, nonblocking memory operations, increased use of pipelines, and increased numbers of functional units. These features make processors more performance sensitive to both high-level issues of program layout and structure and to low-level details of scheduling and resource allocation. As the gap between processor speed and application performance has grown, the demand for optimization has grown to the point where users expect every compiler to perform optimization.

The routine inclusion of an optimizer, in turn, changes the environment in which both the front end and the back end operate. Optimization further insulates the front end from performance concerns. To an extent, this simplifies the task of IR generation in the front end. At the same time, optimization changes the code that the back end processes. Modern optimizers assume that the back end will handle resource allocation; thus, they typically target an idealized machine that has an unlimited supply of registers, memory, and functional units. This, in turn, places more pressure on the techniques used in the compiler's back end.

If compilers are to shoulder their share of responsibility for runtime performance, they must include optimizers. As we shall see, the tools of optimization also play a large role in the compiler's back end. For these reasons, it is important to introduce optimization and explore some of the issues that it raises before discussing the techniques used in a compiler's back end.

8.2.1 Examples

To provide a focus for this discussion, we will begin by examining two examples in depth. The first, a simple two-dimensional array-address calculation, shows the role that knowledge and context play in the kind of code that the compiler can produce. The second, a loop nest from the routine `dmxpy` in the widely-used LINPACK numerical library, provides insight into the transformation process itself and into the challenges that transformed code can present to the compiler.

Improving an Array-Address Calculation

Consider the IR that a compiler's front end might generate for an array reference, such as `m(i,j)` in FORTRAN. Without specific knowledge about `m`, `i`, and `j`, or the surrounding context, the compiler must generate the full

expression for addressing a two-dimensional array stored in column-major order. In Chapter 7, we saw the calculation for row-major order; FORTRAN's column-major order is similar:

$$@\mathtt{m} + (\mathtt{j} - low_2(\mathtt{m})) \times (high_1(\mathtt{m}) - low_1(\mathtt{m}) + 1) \times w + (\mathtt{i} - low_1(\mathtt{m})) \times w$$

where @m is the runtime address of the first element of m, $low_i(\mathtt{m})$ and $high_i(\mathtt{m})$ are the lower and upper bounds, respectively, of m's i^{th} dimension, and w is the size of an element of m. The compiler's ability to reduce the cost of that computation depends directly on its analysis of the code and the surrounding context.

If m is a local array with lower bounds of one in each dimension and known upper bounds, then the compiler can simplify the calculation to

$$@\mathtt{m} + (\mathtt{j} - 1) \times hw + (\mathtt{i} - 1) \times w$$

where hw is $high_1(\mathtt{m}) \times w$. If the reference occurs inside a loop where j runs from 1 to k, the compiler might use *operator strength reduction* to replace the term $(\mathtt{j} - 1) \times hw$ with a sequence $\mathtt{j}'_1$, $\mathtt{j}'_2$, $\mathtt{j}'_3$, ... $\mathtt{j}'_k$, where $\mathtt{j}'_1 = (1 - 1) \times hw = 0$ and $\mathtt{j}'_i = \mathtt{j}'_{i-1} + hw$. If i is also the induction variable of a loop running from 1 to l, then strength reduction can replace $(\mathtt{i} - 1) \times w$ with the sequence $\mathtt{i}'_1$, $\mathtt{i}'_2$, $\mathtt{i}'_3$, ... $\mathtt{i}'_l$, where $\mathtt{i}'_1 = 0$ and $\mathtt{i}'_j = \mathtt{i}'_{j-1} + w$. After these changes, the address calculation is just

$$@\mathtt{m} + \mathtt{j}' + \mathtt{i}'$$

Strength reduction
a transformation that rewrites a series of operations, for example

$$i \cdot c, (i+1) \cdot c, \ldots, (i+k) \cdot c$$

with an equivalent series

$$i'_1, i'_2, \ldots, i'_k,$$

where $i'_1 = i \cdot c$ and $i'_j = i'_{j-1} + c$

See Section 10.7.2.

The j loop must increment j′ by hw and the i loop must increment i′ by w. If the j loop is the outer loop, then the computation of @m + j′ can be moved out of the inner loop. At this point, the address computation in the inner loop contains an add and the increment for i', while the outer loop contains an add and the increment for j'. Knowing the context around the reference to m(i,j) allows the compiler to significantly reduce the cost of array addressing.

If m is an actual parameter to the procedure, then the compiler may not know these facts at compile time. In fact, the upper and lower bounds for m might change on each call to the procedure. In such cases, the compiler may be unable to simplify the address calculation as shown.

Improving a Loop Nest in LINPACK

As a more dramatic example of context, consider the loop nest shown in Figure 8.1. It is the central loop nest of the FORTRAN version of the routine dmxpy from the LINPACK numerical library. The code wraps two loops around a single long assignment. The loop nest forms the core of a

```
      subroutine dmxpy (n1, y, n2, ldm, x, m)
      double precision y(*), x(*), m(ldm,*)
        ...
      jmin = j+16
      do 60 j = jmin, n2, 16
          do 50 i = 1, n1
           y(i) = ((((((((((((((( (y(i))
     $          + x(j-15)*m(i,j-15)) + x(j-14)*m(i,j-14))
     $          + x(j-13)*m(i,j-13)) + x(j-12)*m(i,j-12))
     $          + x(j-11)*m(i,j-11)) + x(j-10)*m(i,j-10))
     $          + x(j- 9)*m(i,j- 9)) + x(j- 8)*m(i,j- 8))
     $          + x(j- 7)*m(i,j- 7)) + x(j- 6)*m(i,j- 6))
     $          + x(j- 5)*m(i,j- 5)) + x(j- 4)*m(i,j- 4))
     $          + x(j- 3)*m(i,j- 3)) + x(j- 2)*m(i,j- 2))
     $          + x(j- 1)*m(i,j- 1)) + x(j) *m(i,j)
50        continue
60    continue
        ...
      end
```

■ **FIGURE 8.1** Excerpt from `dmxpy` in LINPACK.

routine to compute $y + x \times m$, for vectors x and y and matrix m. We will consider the code from two different perspectives: first, the transformations that the author hand-applied to improve performance, and second, the challenges that the compiler faces in translating this loop nest to run efficiently on a specific processor.

Before the author hand-transformed the code, the loop nest performed the following simpler version of the same computation:

```
      do 60 j = 1, n2
         do 50 i = 1, n1
            y(i) = y(i) + x(j) * m(i,j)
50    continue
60 continue
```

Loop unrolling
This replicates the loop body for distinct iterations and adjusts the index calculations to match.

To improve performance, the author *unrolled* the outer loop, the `j` loop, 16 times. That rewrite created 16 copies of the assignment statement with distinct values for `j`, ranging from `j` through `j-15`. It also changed the increment on the outer loop from 1 to 16. Next, the author merged the 16 assignments into a single statement, eliminating 15 occurrences of `y(i) = y(i) + ···`; that eliminates 15 additions and most of the loads and

stores of y(i). Unrolling the loop eliminates some scalar operations. It often improves cache locality, as well.

To handle the cases where the the array bounds are not integral multiples of 16, the full procedure has four versions of the loop nest that precede the one shown in Figure 8.1. These "setup loops" process up to 15 columns of m, leaving j set to a value for which n2 - j is an integral multiple of 16. The first loop handles a single column of m, corresponding to an odd n2. The other three loop nests handle two, four and eight columns of m. This guarantees that the final loop nest, shown in Figure 8.1, can process the columns 16 at a time.

Ideally, the compiler would automatically transform the original loop nest into this more efficient version, or into whatever form is most appropriate for a given target machine. However, few compilers include all of the optimizations needed to accomplish that goal. In the case of dmxpy, the author performed the optimizations by hand to produce good performance across a wide range of target machines and compilers.

From the compiler's perspective, mapping the loop nest shown in Figure 8.1 onto the target machine presents some hard challenges. The loop nest contains 33 distinct array-address expressions, 16 for m, 16 for x, and one for y that it uses twice. Unless the compiler can simplify those address calculations, the loop will be awash in integer arithmetic.

Consider the references to x. They do not change during execution of the inner loop, which varies i. The optimizer can move the address calculations and the loads for x out of the inner loop. If it can keep the x values in registers, it can eliminate a large part of the overhead from the inner loop. For a reference such as x(j-12), the address calculation is just @x + (j − 12) × w. To further simplify matters, the compiler can refactor all 16 references to x into the form @x + jw − c_k, where jw is j · w and c_k is $k \cdot w$ for each $0 \leq k \leq 15$. In this form, each load uses the same base address, @x + jw, with a different constant offset, c_k.

To map this efficiently onto the target machine requires knowledge of the available addressing modes. If the target has the equivalent of ILOC's loadAI operation (a register base address plus a small constant offset), then all the accesses to x can be written to use a single induction variable. Its initial value is @x + jmin · w. Each iteration of the j loop increments it by w.

The 16 values of m used in the inner loop change on each iteration. Thus, the inner loop must compute addresses and load 16 elements of m on each iteration. Careful refactoring of the address expressions, combined with strength reduction, can reduce the overhead of accessing m. The value

@m + j · $high_1$(m) · w can be computed in the j loop. (Notice that $high_1$(m) is the only concrete dimension declared in dmxpy's header.) The inner loop can produce a base address by adding it to (i − 1) · w. Then, the 16 loads can use distinct constants, $c_k \cdot high_1$(m), where c_k is $k \cdot w$ for each $0 \leq k \leq 15$.

To achieve this code shape, the compiler must refactor the address expressions, perform strength reduction, recognize loop-invariant calculations and move them out of inner loops, and choose the appropriate addressing mode for the loads. Even with these improvements, the inner loop must perform 16 loads, 16 floating-point multiplies, and 16 floating-point adds, plus one store. The resulting block will present a challenge to the instruction scheduler.

If the compiler fails in some part of this transformation sequence, the resulting code might be substantially worse than the original. For example, if it cannot refactor the address expressions around a common base address for x and one for m, the code might maintain 33 distinct induction variables—one for each distinct address expression for x, m, and y. If the resulting demand for registers forces the register allocator to spill, it will insert additional loads and stores into the loop (which is already likely to be memory bound). In cases such as this one, the quality of code produced by the compiler depends on an orchestrated series of transformations that all must work; when one fails to achieve its purpose, the overall sequence may produce lower quality code than the user expects.

8.2.2 Considerations for Optimization

In the previous example, the programmer applied the transformations in the belief that they would make the program run faster. The programmer had to believe that they would preserve the meaning of the program. (After all, if transformations need not preserve meaning, why not replace the entire procedure with a single nop?)

Two issues, safety and profitability, lie at the heart of every optimization. The compiler must have a mechanism to prove that each application of the transformation is safe—that is, it preserves the program's meaning. The compiler must have a reason to believe that applying the transformation is profitable—that is, it improves the program's performance. If either of these is not true—that is, applying the transformation will change the program's meaning or will make its performance worse—the compiler should not apply the transformation.

Safety

How did the programmer know that this transformation was safe? That is, why did the programmer believe that the transformed code would produce the same results as the original code? Close examination of the loop nest

DEFINING SAFETY

Correctness is the single most important criterion that a compiler must meet—the code that the compiler produces must have the same meaning as the input program. Each time the optimizer applies a transformation, that action must preserve the correctness of the translation.

Typically, *meaning* is defined as the observable behavior of the program. For a batch program, this is the memory state after it halts, along with any output it generates. If the program terminates, the values of all visible variables immediately before it halts should be the same under any translation scheme. For an interactive program, behavior is more complex and difficult to capture.

Plotkin formalized this notion as *observational equivalence*.

For two expressions, M and N, we say that M and N are observationally equivalent if and only if, in any context C where both M and N are closed (that is, have no free variables), evaluating C[M] and C[N] either produces identical results or neither terminates [286].

Thus, two expressions are observationally equivalent if their impacts on the visible, external environment are identical.

In practice, compilers use a simpler and looser notion of equivalence than Plotkin's, namely, that if, in their actual program context, two different expressions e and e' produce identical results, then the compiler can substitute e' for e. This standard deals only with contexts that actually arise in the program; tailoring code to context is the essence of optimization. It does not mention what happens when a computation goes awry, or diverges.

In practice, compilers take care not to introduce divergence—the original code would work correctly, but the optimized code tries to divide by zero, or loops indefinitely. The opposite case, where the original code would diverge, but the optimized code does not, is rarely mentioned.

shows that the only interaction between successive iterations occurs through the elements of `y`.

- A value computed as `y(i)` is not reused until the next iteration of the outer loop. The iterations of the inner loop are independent of each other, because each iteration defines precisely one value and no other iteration references that value. Thus, the iterations can execute in any order. (For example, if we run the inner loop from `n1` to `1` it produces the same results.)
- The interaction through `y` is limited in its effect. The i^{th} element of `y` accumulates the sum of all the i^{th} iterations of the inner loop. This pattern of accumulation is safely reproduced in the unrolled loop.

A large part of the analysis done in optimization goes toward proving the safety of transformations.

Profitability

Why did the programmer think that loop unrolling would improve performance? That is, why is the transformation profitable? Several different effects of unrolling might speed up the code.

- The total number of loop iterations is reduced by a factor of 16. This reduces the overhead operations due to loop control: adds, compares, jumps, and branches. If the loop executes frequently, these savings become significant.
  This effect might suggest unrolling by an even larger factor. Finite resource limits probably dictated the choice of 16. For example, the inner loop uses the same 16 values of `x` for all the iterations of the inner loop. Many processors have only 32 registers that can hold a floating-point number. Unrolling by 32, the next power of two, would create enough of these "loop-invariant" values that they could not fit in the register set. Spilling them to memory would add `loads` and `stores` to the inner loop and undo the benefits of unrolling.
- The array-address computations contain duplicated work. Consider the use of `y(i)`. The original code computed `y(i)`'s address once per multiplication of `x` and `m`; the transformed code computes it once per 16 multiplications. The unrolled code does $\frac{1}{16}$ as much work to address `y(i)`. The 16 references to `m`, and to a lesser extent `x`, should include common portions that the loop can compute once and reuse, as well.
- The transformed loop performs more work per memory operation, where "work" excludes the overhead of implementing the array and loop abstractions. The original loop performed two arithmetic operations for three memory operations, while the unrolled loop performs 32 arithmetic operations for 18 memory operations, assuming that all the `x` values stay in registers. Thus, the unrolled loop is less likely to be *memory bound*. It has enough independent arithmetic to overlap the loads and hide some of their latencies.

Memory bound
A loop where loads and stores take more cycles than does computation is considered *memory bound*.

To determine if a loop is memory bound requires detailed knowledge about both the loop and the target machine.

Unrolling can help with other machine-dependent effects. It increases the amount of code in the inner loop, which may provide the instruction scheduler with more opportunities to hide latencies. If the end-of-loop branch has a long latency, the longer loop body may let the compiler fill more of that branch's delay slots. On some processors, unused delay slots must be filled with `nops`, in which case loop unrolling can decrease the number of `nops` fetched, reduce memory traffic and, perhaps, reduce the energy used to execute the program.

Risk

If transformations intended to improve performance make it harder for the compiler to generate good code for the program, those potential problems should be considered as profitability issues. The hand transformations performed on `dmxpy` create new challenges for a compiler, including the following:

- *Demand for registers* The original loop needs only a handful of registers to hold its active values. Only `x(j)`, some part of the address calculations for `x`, `y`, and `m`, and the loop index variables need registers across loop iterations, while `y(i)` and `m(i,j)` need registers briefly. In contrast, the transformed loop has 16 elements of `x` to keep in registers across the loop, along with the 16 values of `m` and `y(i)` that need registers briefly.
- *Form of address calculation* The original loop deals with three addresses, one each for `y`, `x`, and `m`. Because the transformed loop references many more distinct locations in each iteration, the compiler must shape the address calculations carefully to avoid repeated calculations and excessive demand for registers. In the worst case, the code might use independent calculations for all 16 elements of `x`, all 16 elements of `m`, and the one element of `y`.
  If the compiler shapes the address calculations appropriately, it can use a single pointer for `m` and another for `x`, each with 16 constant-valued offsets. It can rewrite the loop to use that pointer in the end-of-loop test, obviating the need for another register and eliminating another update. Planning and optimization make the difference.

Other problems of a machine-specific nature arise as well. For example, the 17 loads and one store, the 16 multiplies, the 16 adds, plus the address calculations and loop-overhead operations in each iteration must be scheduled with care. The compiler may need to issue some of the load operations in a previous iteration so that it can schedule the initial floating-point operations in a timely fashion.

8.2.3 Opportunities for Optimization

As we have seen, the task of optimizing a simple loop can involve complex considerations. In general, optimizing compilers capitalize on opportunities that arise from several distinct sources.

1. *Reducing the overhead of abstraction* As we saw for the array-address calculation at the beginning of the chapter, the data structures and types introduced by programming languages require runtime support. Optimizers use analysis and transformation to reduce this overhead.

2. *Taking advantage of special cases* Often, the compiler can use knowledge about the context in which an operation executes to specialize that operation. As an example, a C++ compiler can sometimes determine that a call to a virtual function always uses the same implementation. In that case, it can remap the call and reduce the cost of each invocation.
3. *Matching the code to system resources* If the resource requirements of a program differ from the processor's capacities, the compiler may transform the program to align its needs more closely with available resources. The transformations applied to `dmxpy` have this effect; they decrease the number of memory accesses per floating-point operation.

These are broad areas, described in sweeping generality. As we discuss specific analysis and transformation techniques, in Chapters 9 and 10, we will fill in these areas with more detailed examples.

SECTION REVIEW

Most compiler-based optimization works by specializing general purpose code to its specific context. For some code transformations, the benefits accrue from local effects, as with the improvements in the array-address calculations. Other transformations require broad knowledge of larger regions in the code and accrue their benefits from effects that occur over larger swaths of the code.

In considering any optimization, the compiler writer must worry about the following:

1. Safety, for example, does the transformation not change the meaning of the code?
2. Profitability, for example, how will the transformation improve the code?
3. Finding opportunities, for example, how can the compiler quickly locate places in the code where applying the given transformation is both safe and profitable?

Review Questions

1. In the code fragment from `dmxpy` in LINPACK, why did the programmer choose to unroll the outer loop rather than the inner loop? How would you expect the results to differ had she unrolled the inner loop?
2. In the `c` fragment shown below, what facts would the compiler need to discover before it could improve the code beyond a simple byte-oriented, load/store implementation?

```
MemCopy(char *source, char *dest, int length) {
   int i;
   for (i=1; i≤length; i++)
      { *dest++ = *source++; }
}
```

8.3 SCOPE OF OPTIMIZATION

Scope of optimization
The region of code where an optimization operates is its *scope of optimization*.

Optimizations operate at different granularities or scopes. In the previous section, we looked at optimization of a single array reference and of an entire loop nest. The different scopes of these optimizations presented different opportunities to the optimizer. Reformulating the array reference improved performance for the execution of that array reference. Rewriting the loop improved performance across a larger region. In general, transformations and the analyses that support them operate on one of four distinct scopes: local, regional, global, or whole program.

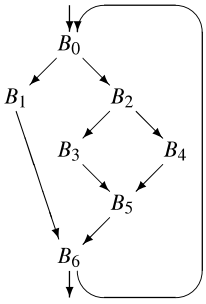
Local Methods

Local methods operate over a single basic block: a maximal-length sequence of branch-free code. In an ILOC program, a basic block begins with a labelled operation and ends with a branch or a jump. In ILOC, the operation after a branch or jump must be labelled or else it cannot be reached; other notations allow a "fall-through" branch so that the operation after a branch or jump need not be labelled. The behavior of straight-line code is easier to analyze and understand than is code that contains branches and cycles.

Inside a basic block, two important properties hold. First, statements are executed sequentially. Second, if any statement executes, the entire block executes, unless a runtime exception occurs. These two properties let the compiler prove, with relatively simple analyses, facts that may be stronger than those provable for larger scopes. Thus, local methods sometimes make improvements that simply cannot be obtained for larger scopes. At the same time, local methods are limited to improvements that involve operations that all occur in the same block.

Regional Methods

Regional methods operate over scopes larger than a single block but smaller than a full procedure. In the example control-flow graph (CFG) in the margin, the compiler might consider the entire loop, $\{B_0, B_1, B_2, B_3, B_4, B_5, B_6\}$, as a single region. In some cases, considering a subset of the code for the full procedure produces sharper analysis and better transformation results



than would occur with information from the full procedure. For example, inside a loop nest, the compiler may be able to prove that a heavily used pointer is invariant (single-valued), even though it is modified elsewhere in the procedure. Such knowledge can enable optimizations such as keeping in a register the value referenced through that pointer.

The compiler can choose regions in many different ways. A region might be defined by some source-code control structure, such as a loop nest. The compiler might look at the subset of blocks in the region that form an *extended basic block* (EBB). The example CFG contains three EBBs: $\{B_0, B_1, B_2, B_3, B_4\}$, $\{B_5\}$, and $\{B_6\}$. While the two single-block EBBs provide no advantage over a purely local view, the large EBB may offer opportunities for optimization (see Section 8.5.1). Finally, the compiler might consider a subset of the CFG defined by some graph-theoretic property, such as a *dominator relation* or one of the strongly connected components in the CFG.

Extended basic block
a set of blocks $\beta_1, \beta_2, \ldots, \beta_n$ where β_1 has multiple CFG predecessors and each other β_i has just one, which is some β_j in the set

Dominator
In a CFG, *x dominates y* if and only if every path from the root to *y* includes *x*.

Regional methods have several strengths. Limiting the scope of a transformation to a region smaller than the entire procedure allows the compiler to focus its efforts on heavily executed regions—for example, the body of a loop typically executes much more frequently than the surrounding code. The compiler can apply different optimization strategies to distinct regions. Finally, the focus on a limited area in the code often allows the compiler to derive sharper information about program behavior which, in turn, exposes opportunities for improvement.

Global Methods

These methods, also called *intraprocedural methods*, use an entire procedure as context. The motivation for global methods is simple: decisions that are locally optimal may have bad consequences in some larger context. The procedure provides the compiler with a natural boundary for both analysis and transformation. Procedures are abstractions that encapsulate and insulate runtime environments. At the same time, they serve as units of separate compilation in many systems.

Global methods typically operate by building a representation of the procedure, such as a CFG, analyzing that representation, and transforming the underlying code. If the CFG can have cycles, the compiler must analyze the entire procedure before it understands what facts hold on entrance to any specific block. Thus, most global transformations have separate analysis and transformation phases. The analytical phase gathers facts and reasons about them. The transformation phase uses those facts to determine the safety and profitability of a specific transformation. By virtue of their global view,

INTRAPROCEDURAL VERSUS INTERPROCEDURAL

Few terms in compilation create as much confusion as the word *global*. Global analysis and optimization operate on an entire procedure. The modern English connotation, however, suggests an all-encompassing scope, as does the use of global in discussions of lexical scoping rules. In analysis and optimization, however, global means pertaining to a single procedure.

Interest in analysis and optimization across procedure boundaries necessitated terminology to differentiate between *global* analysis and analysis over larger scopes. The term *interprocedural* was introduced to describe analysis that ranged from two procedures to a whole program. Accordingly, authors began to use the term *intraprocedural* for single-procedure techniques. Since these words are so close in spelling and pronunciation, they are easy to confuse and awkward to use.

Perkin-Elmer Corporation tried to remedy this confusion when it introduced its "universal" FORTRAN VIIZ optimizing compiler for the PE 3200; the system performed extensive inlining followed by aggressive global optimization on the resulting code. Universal did not stick. We prefer the term *whole program* and use it whenever possible. It conveys the right distinction and reminds the reader and listener that "global" is not "universal."

these methods can discover opportunities that neither local nor regional methods can.

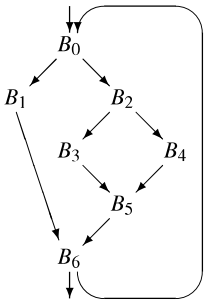
Interprocedural Methods

These methods, sometimes called *whole-program methods*, consider scopes larger than a single procedure. We consider any transformation that involves more than one procedure to be an interprocedural transformation. Just as moving from a local scope to a global scope exposes new opportunities, so moving from single procedures to the multiple procedures can expose new opportunities. It also raises new challenges. For example, parameter-binding rules introduce significant complications into the analysis that supports optimization.

Interprocedural analysis and optimization occurs, at least conceptually, on the program's call graph. In some cases, these techniques analyze the entire program; in other cases the compiler may examine just a subset of the source code. Two classic examples of interprocedural optimizations are inline substitution, which replaces a procedure call with a copy of the body of the callee, and interprocedural constant propagation, which propagates and folds information about constants throughout the entireprogram.

SECTION REVIEW

Compilers perform both analysis and transformation over a variety of scopes, ranging from single basic blocks (local methods) to entire programs (whole-program methods). In general, the number of opportunities for improvement grows with the scope of optimization. However, analyzing larger scopes often results in less precise knowledge about the code's behavior. Thus, no simple relationship exits between scope of optimization and quality of the resulting code. It would be intellectually pleasing if a larger scope of optimization led, in general, to better code quality. Unfortunately, that relationship does not necessarily hold true.



Review Questions

1. Basic blocks have the property that if one instruction executes, every instruction in the block executes, in a specified order (unless an exception occurs). State the weaker property that holds for a block in an extended basic block, other than the entry block, such as block B_2 in the EBB $\{B_0, B_1, B_2, B_3, B_4\}$, for the control-flow graph shown in the margin.
2. What kinds of improvement might the compiler find with whole-program compilation? Name several inefficiencies that can only be addressed by examining code across procedure boundaries. How does interprocedural optimization interact with the desire to compile procedures separately?

8.4 LOCAL OPTIMIZATION

Optimizations that operate over a local scope—a single basic block—are among the simplest techniques that the compiler can use. The simple execution model of a basic block leads to reasonably precise analysis in support of optimization. Thus, these methods are surprisingly effective.

Redundant
An expression e is *redundant* at p if it has already been evaluated on every path that leads to p.

This section presents two local methods as examples. The first, *value numbering*, finds redundant expressions in a basic block and replaces the redundant evaluations with reuse of a previously computed value. The second, *tree-height balancing*, reorganizes expression trees to expose more instruction-level parallelism.

8.4.1 Local Value Numbering

Consider the four-statement basic block shown in the margin. We will refer to the block as B. An expression, such as `b + c` or `a - d`, is redundant in B if and only if it has been previously computed in B and no intervening

operation redefines one of its constituent arguments. In *B*, the occurrence of `b + c` in the third operation is not redundant because the second operation redefines `b`. The occurrence of `a - d` in the fourth operation is redundant because *B* does not redefine `a` or `d` between the second and fourth operations.

```
a ← b + c
b ← a - d
c ← b + c
d ← a + d
```
Original Block

The compiler can rewrite this block so that it computes `a - d` once, as shown in the margin. The second evaluation of `a - d` is replaced with a copy from `b`. An alternative strategy would replace subsequent uses of `d` with uses of `b`. However, that approach requires analysis to determine whether or not `b` is redefined before some use of `d`. In practice, it is simpler to have the optimizer insert a copy and let a subsequent pass determine which copy operations are, in fact, necessary and which ones can have their source and destination names combined.

```
a ← b + c
b ← a - d
c ← b + c
d ← b
```
Rewritten Block

In general, replacing redundant evaluations with references to previously computed values is profitable—that is, the resulting code runs more quickly than the original. However, profitability is not guaranteed. Replacing `d ← a - d` with `d ← b` has the potential to extend the lifetime of `b` and to shorten the lifetimes of either `a` or `d` or both—depending, in each case, on where the last use of the value lies. Depending on the precise details, each rewrite can increase demand for registers, decrease demand for registers, or leave it unchanged. Replacing a redundant computation with a reference is likely to be unprofitable if the rewrite causes the register allocator to spill a value in the block.

Lifetime
The lifetime of a name is the region of code between its definitions and its uses. Here, definition means assignment.

In practice, the optimizer cannot consistently predict the behavior of the register allocator, in part because the code will be further transformed before it reaches the allocator. Therefore, most algorithms for removing redundancy assume that rewriting to avoid redundancy is profitable.

In the previous example, the redundant expression was textually identical to the earlier instance. Assignment can, of course, produce a redundant expression that differs textually from its predecessor. Consider the block shown in the margin. The assignment of `b` to `d` makes the expression `d × c` produce the same value as `b × c`. To recognize this case, the compiler must track the flow of values through names. Techniques that rely on textual identity do not detect such cases.

```
a ← b × c
d ← b
e ← d × c
```
Effect of Assignment

Programmers will protest that they do not write code that contains redundant expressions like those in the example. In practice, redundancy elimination finds many opportunities. Translation from source code to IR elaborates many details, such as address calculations, and introduces redundant expressions.

Many techniques that find and eliminate redundancies have been developed. *Local value numbering* is one of the oldest and most powerful of

these transformations. It discovers such redundancies within a basic block and rewrites the block to avoid them. It provides a simple and efficient framework for other local optimizations, such as constant folding and simplification using algebraic identities.

The Algorithm

The idea behind value numbering is simple. The algorithm traverses a basic block and assigns a distinct number to each value that the block computes. It chooses the numbers so that two expressions, e_i and e_j, have the same value number if and only e_i and e_j have provably equal values for all possible operands of the expressions.

Figure 8.2 shows the basic local value numbering algorithm (LVN). LVN takes as input a block with n binary operations, each of the form $T_i \leftarrow L_i\ Op_i\ R_i$. LVN examines each operation, in order. LVN uses a hash table to map names, constants, and expressions into distinct value numbers. The hash table is initially empty.

To process the i^{th} operation, LVN obtains value numbers for L_i and R_i by looking for them in the hash table. If it finds an entry, LVN uses the value number of that entry. If not, it creates one and assigns a new value number.

Given value numbers for L_i and R_i, called $VN(L_i)$ and $VN(R_i)$, LVN constructs a hash key from $\langle VN(L_i), Op_i, VN(R_i)\rangle$ and looks up that key in the table. If an entry exists, the expression is redundant and can be replaced by a reference to the previously computed value. If not, operation i is the first computation of the expression in this block, so LVN creates an entry for its hash key and assigns that entry a new value number. It also assigns the hash key's value number, whether new or pre-existing, to the table entry for T_i. Because LVN uses value numbers to construct the expression's hash

```
for i ← 0 to n-1, where the block has n operations    "Ti ← Li Opi Ri"
    1. get the value numbers for Li and Ri
    2. construct a hash key from Opi and the value numbers for Li and Ri
    3. if the hash key is already present in the table then
           replace operation i with a copy of the value into Ti and
           associate the value number with Ti
       else
           insert a new value number into the table at the hash key location
           record that new value number for Ti
```

■ **FIGURE 8.2** Value Numbering a Single Block.

THE IMPORTANCE OF ORDER

The specific order in which expressions are written has a direct impact on the ability of optimizations to analyze and transform them. Consider the following distinct encodings of $v \leftarrow a \times b \times c$:

$$\begin{array}{ll} t_0 \leftarrow a \times b & t_0 \leftarrow b \times c \\ v \leftarrow t_0 \times c & v \leftarrow a \times t_0 \end{array}$$

The encoding on the left assigns value numbers to $a \times b$, to $(a \times b) \times c$ and to v, while the encoding on the right assigns value numbers to $b \times c$, to $a \times (b \times c)$ and to v. Depending on the surrounding context, one or the other encoding may be preferable. For example, if $b \times c$ occurs later in the block but $a \times b$ does not, then the right-hand encoding produces redundancy while the left does not.

In general, using commutativity, associativity, and distributivity to reorder expressions can change the results of optimization. Similar effects can be seen with constant folding; if we replace a with 3 and c with 5, neither ordering produces the constant operation 3×5, which can be folded.

Because the number of ways to reorder expressions is prohibitively large, compilers use heuristic techniques to find good orderings for expressions. For example, the IBM FORTRAN H compiler generated array-address computations in an order that tended to improve other optimizations. Other compilers have sorted the operands of commutative and associative operations into an order that corresponds to the loop nesting level at which they are defined. Because so many solutions are possible, heuristic solutions for this problem often require experimentation and tuning to discover what is appropriate for a specific language, compiler, and coding style.

key, rather than names, it can effectively track the flow of values through copy and assignment operations, such as the small example labelled "Effect of Assignment" on the previous page. Extending LVN to expressions of arbitrary arity is straightforward.

To see how LVN works, consider our original example block, shown on page 421. The version in the margin shows the value numbers that LVN assigns as superscripts. In the first operation, with an empty value table, b and c get new value numbers, 0 and 1 respectively. LVN constructs the textual string "0 + 1" as a hash key for the expression $a + b$ and performs a lookup. It does not find an entry for that key, so the lookup fails. Accordingly, LVN creates a new entry for "0 + 1" and assigns it value number 2. LVN then creates an entry for a and assigns it the value number of the expression, namely 2. Repeating this process for each operation, in sequential order, produces the rest of the value numbers shown in the margin.

$$\begin{array}{l} a^2 \leftarrow b^0 + c^1 \\ b^4 \leftarrow a^2 - d^3 \\ c^5 \leftarrow b^4 + c^1 \\ d^4 \leftarrow a^2 - d^3 \end{array}$$

```
a ← b + c
b ← a - d
c ← b + c
d ← b
```

The value numbers reveal, correctly, that the two occurrences of `b+c` produce different values, due to the intervening redefinition of `b`. On the other hand, the two occurrences of `a - d` produce the same value, since they have the same input value numbers and the same operator. LVN discovers this and records it by assigning `b` and `d` the same value number, namely 4. That knowledge lets LVN rewrite the fourth operation with a `d ← b` as shown in the margin. Subsequent passes may eliminate the copy.

Extending the Algorithm

LVN provides a natural framework to perform several other local optimizations.

- *Commutative operations* Commutative operations that differ only in the order of their operands, such as $a \times b$ and $b \times a$, should receive the same value numbers. As LVN constructs a hash key for the right-hand side of the current operation, it can sort the operands using some convenient scheme, such as ordering them by value number. This simple action will ensure that commutative variants receive the same value number.
- *Constant folding* If all the operands of an operation have known constant values, LVN can perform the operation and fold the answer directly into the code. LVN can store information about constants in the hash table, including their value. Before hash-key formation, it can test the operands and, if possible, evaluate them. If LVN discovers a constant expression, it can replace the operation with an immediate load of the result. Subsequent copy folding will clean up the code.
- *Algebraic identities* LVN can apply algebraic identities to simplify the code. For example, $x + 0$ and x should receive the same value number. Unfortunately, LVN needs special-case code for each identity. A series of tests, one per identity, can easily become long enough to produce an unacceptable slowdown in the algorithm. To ameliorate this problem, LVN should organize the tests into operator-specific decision trees. Since each operator has just a few identities, this approach keeps the overhead low. Figure 8.3 shows some of the identities that can be handled in this way.

$a + 0 = a$	$a - 0 = a$	$a - a = 0$	$2 \times a = a + a$
$a \times 1 = a$	$a \times 0 = 0$	$a \div 1 = a$	$a \div a = 1, a \neq 0$
$a^1 = a$	$a^2 = a \times a$	$a \gg 0 = a$	$a \ll 0 = a$
a AND a = a	a OR a = a	MAX (a,a) = a	MIN (a,a) = a

FIGURE 8.3 Algebraic Identities for Value Numbering.

```
for i ← 0 to n-1, where the block has n operations    "Ti ← Li Opi Ri"
   1.  get the value numbers for Li and Ri
   2.  if Li and Ri are both constant then evaluate Li Opi Ri,
           assign the result to Ti, and mark Ti as constant
   3.  if Li Opi Ri matches an identity in Figure 8.3, then replace it with
           a copy operation or an assignment
   4.  construct a hash key from Opi and the value numbers for Li and Ri,
          using the value numbers in ascending order, if Opi commutes
   5.  if the hash key is already present in the table then
           replace operation i with a copy into Ti and
           associate the value number with Ti
       else
          insert a new value number into the table at the hash key location
          record that new value number for Ti
```

■ **FIGURE 8.4** Local Value Numbering with Extensions.

A clever implementor will discover other identities, including some that are type specific. The exclusive-or of two identical values should yield a zero of the appropriate type. Numbers in IEEE floating-point format have their own special cases introduced by the explicit representations of ∞ and NaN; for example, $\infty - \infty = \text{NaN}$, $\infty - \text{NaN} = \text{NaN}$, and $\infty \div \text{NaN} = \text{NaN}$.

NaN
Not a Number, a defined constant that represents an invalid or meaningless result in the IEEE standard for floating-point arithmetic

Figure 8.4 shows LVN with these extensions. Steps 1 and 5 appeared in the original algorithm. Step 2 evaluates and folds constant-valued operations. Step 3 checks for algebraic identities using the decision trees mentioned earlier. Step 4 reorders the operands of commutative operations. Even with these extensions, the cost per IR operation remains extremely low. Each step has an efficient implementation.

The Role of Naming

The choice of names for variables and values can limit the effectiveness of value numbering. Consider what happens when LVN is applied to the block shown in the margin. Again, the superscripts indicate the value numbers assigned to each name and value.

$$\begin{aligned} a^3 &\leftarrow x^1 + y^2 \\ b^3 &\leftarrow x^1 + y^2 \\ a^4 &\leftarrow 17^4 \\ c^3 &\leftarrow x^1 + y^2 \end{aligned}$$

In the first operation, LVN assigns 1 to x, 2 to y and 3 to both x + y and to a. At the second operation, it discovers that x + y is redundant, with value number 3. Accordingly, it rewrites b ← x + y with b ← a. The third operation

is both straightforward and nonredundant. At the fourth operation, it again discovers that x+y is redundant, with value number 3. It cannot, however, rewrite the operation as c ← a because a no longer has value number 3.

We can cure this problem in two distinct ways. We can modify LVN so that it keeps a mapping from value numbers to names. At an assignment to some name, say a, it must remove a from the list for its old value number and add a to the list for its new value number. Then, at a replacement, it can use any name that currently contains that value number. This approach adds some cost to the processing of each assignment and clutters up the code for the basic algorithm.

As an alternative, the compiler can rewrite the code in a way that gives each assignment a distinct name. Adding a subscript to each name for uniqueness, as shown in the margin, is sufficient. With these new names, the code defines each value exactly once. Thus, no value is ever redefined and lost, or *killed*. If we apply LVN to this block, it produces the desired result. It proves that the second and fourth operations are redundant; each can be replaced with a copy from a_0.

$a_0^3 \leftarrow x_0^1 + y_0^2$
$b_0^3 \leftarrow x_0^1 + y_0^2$
$a_1^4 \leftarrow 17^4$
$c_0^3 \leftarrow x_0^1 + y_0^2$

However, the compiler must now reconcile these subscripted names with the names in surrounding blocks to preserve the meaning of the original code. In our example, the original name a should refer to the value from the subscripted name a_1 in the rewritten code. A clever implementation would map the new a_1 to the original a, b_0 to the original b, c_0 to the original c, and rename a_0 to a new temporary name. That solution reconciles the name space of the transformed block with the surrounding context without introducing copies.

This naming scheme approximates one property of the name space created for static single-assignment form, or SSA, introduced in Section 5.4.2. Section 9.3 explores translation from linear code into SSA form and from SSA form back into linear code. The algorithms that it presents for name-space translation are more general than needed for a single block, but will certainly handle the single-block case and will attempt to minimize the number of copy operations that must be inserted.

The Impact of Indirect Assignments

The previous discussion assumes that assignments are direct and obvious, as in a ← b × c. Many programs contain indirect assignments, where the compiler may not know which values or locations are modified. Examples include assignment through a pointer, such as *p=0; in C, or assignment to a structure element or an array element, such as a(i,j)=0 in FORTRAN. Indirect assignments complicate value numbering and other optimizations

RUNTIME EXCEPTIONS AND OPTIMIZATION

Some abnormal runtime conditions can raise exceptions. Examples include out-of-bounds memory references, undefined arithmetic operations such as division by zero, and ill-formed operations. (One way for a debugger to trigger a breakpoint is to replace the instruction with an ill-formed one and to catch the exception.) Some languages include features for handling exceptions, for both predefined and programmer-defined situations.

Typically, a runtime exception causes a transfer of control to an exception handler. The handler may cure the problem, re-execute the offending operation, and return control to the block. Alternatively, it may transfer control elsewhere or terminate execution.

The optimizer must understand which operations can raise an exception and must consider the impact of an exception on program execution. Because an exception handler might modify the values of variables or transfer control, the compiler must treat exception-raising operations conservatively. For example, every exception-raising operation might force termination of the current basic block. Such treatment can severely limit the optimizer's ability to improve the code.

To optimize exception-laden code, the compiler needs to understand and model the effects of exception handlers. To do so, it needs access to the code for the exception handlers and it needs a model of overall execution to understand which handlers might be in place when a specific exception-raising operation executes.

because they create imprecisions in the compiler's understanding of the flow of values.

Consider value numbering with the subscripted naming scheme presented in the previous section. To manage the subscripts, the compiler maintains a map from the base variable name, say `a`, to its current subscript. On an assignment, such as `a` ← `b` + `c`, the compiler simply increments the current subscript for `a`. Entries in the value table for the previous subscript remain intact. On an indirect assignment, such as `*p` ← `0`, the compiler may not know which base-name subscripts to increment. Without specific knowledge of the memory locations to which `p` can refer, the compiler must increment the subscript of every variable that the assignment could possibly modify—potentially, the set of all variables. Similarly, an assignment such as `a(i,j)=0`, where the value of either `i` or `j` is unknown, must be treated as if it changes the value of every element of `a`.

Hint: The hash table of value numbers must reflect subscripted names. The compiler can use a second, smaller table to map base names to subscripts.

While this sounds drastic, it shows the true impact of an ambiguous indirect assignment on the set of facts that the compiler can derive. The compiler can perform analysis to disambiguate pointer references—that is, to narrow the

Ambiguous reference
A reference is *ambiguous* if the compiler cannot isolate it to a single memory location.

set of variables that the compiler believes a pointer can address. Similarly, it can use a variety of techniques to understand the patterns of element access in an array—again, to shrink the set of locations that it must assume are modified by an assignment to one element.

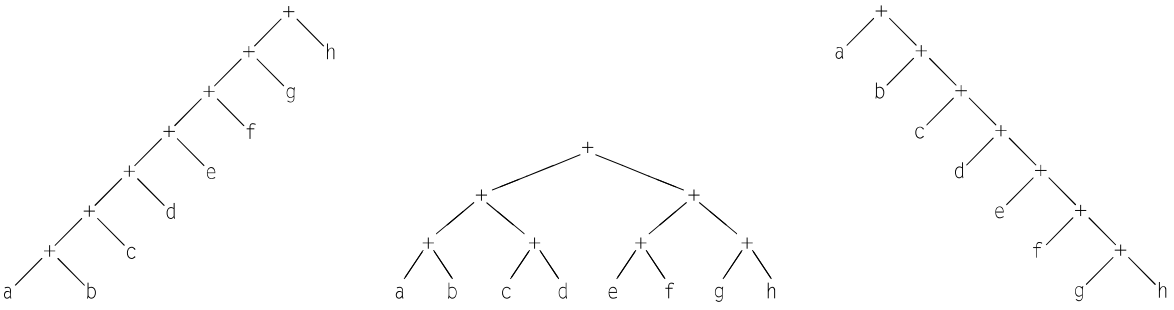
8.4.2 Tree-Height Balancing

As we saw in Chapter 7, the specific details of how the compiler encodes a computation can affect the compiler's ability to optimize that computation. Many modern processors have multiple functional units so that they can execute multiple independent operations in each cycle. If the compiler can arrange the instruction stream so that it contains independent operations, encoded in the appropriate, machine-specific way, then the application will run more quickly.

```
t1 ← a + b
t2 ← t1 + c
t3 ← t2 + d
t4 ← t3 + e
t5 ← t4 + f
t6 ← t5 + g
t7 ← t6 + h
```

Consider the code for a + b + c + d + e + f + g + h shown in the margin. A left-to-right evaluation would produce the left-associative tree in Figure 8.5a. Other permissible trees include those in Figure 8.5b and c. Each distinct tree implies constraints on the execution order that are not required by the rules of addition. The left-associative tree implies that the program must evaluate a + b before it can perform the additions involving either g or h. The corresponding right-associative tree, created by a right-recursive grammar, implies that g + h must precede additions involving a or b. The balanced tree imposes fewer constraints, but it still implies an evaluation order with more constraints than the actual arithmetic.

If the processor can perform more than one addition at a time, then the balanced tree should let the compiler produce a shorter schedule for the computation. Figure 8.6 shows possible schedules for the balanced tree and the left-associative tree on a computer with two single-cycle adders. The balanced tree can execute in four cycles, with one unit idle in the fourth cycle.



■ **FIGURE 8.5** Potential Tree Shapes for a + b + c + d + e + f + g + h.

	Balanced Tree		Left-Associative Tree	
	Unit 0	**Unit 1**	**Unit 0**	**Unit 1**
1	$t_1 \leftarrow a+b$	$t_2 \leftarrow c+d$	$t_1 \leftarrow a+b$	—
2	$t_3 \leftarrow e+f$	$t_4 \leftarrow g+h$	$t_2 \leftarrow t_1+c$	—
3	$t_5 \leftarrow t_1+t_2$	$t_6 \leftarrow t_3+t_4$	$t_3 \leftarrow t_2+d$	—
4	$t_7 \leftarrow t_5+t_6$	—	$t_4 \leftarrow t_3+e$	—
5	—	—	$t_5 \leftarrow t_4+f$	—
6	—	—	$t_6 \leftarrow t_5+g$	—
7	—	—	$t_7 \leftarrow t_6+h$	—

■ **FIGURE 8.6** Schedules from Different Tree Shapes for a + b + c + d + e + f + g + h.

In contrast, the left-associative tree requires seven cycles, leaving the second adder idle throughout the computation. The shape of the left-associative tree forces the compiler to serialize the additions. The right-associative tree will produce a similar effect.

This small example suggests an important optimization: using the commutative and associative laws of arithmetic to expose additional parallelism in expression evaluation. The remainder of this section presents an algorithm for rewriting code to create expressions whose tree form approximates a balanced tree. This particular transformation aims to improve execution time by exposing more concurrent operations, or *instruction-level parallelism*, to the compiler's instruction scheduler.

To formalize these notions into an algorithm, we will follow a simple scheme.

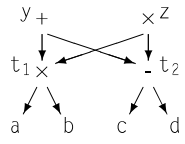
1. The algorithm identifies candidate expression trees in the block. All of the operators in a candidate tree must be identical; they must also be commutative and associative. Equally important, each name that labels an interior node of the candidate tree must be used exactly once.
2. For each candidate tree, the algorithm finds all its operands, assigns them a rank, and enters them into a priority queue, ordered by ascending rank. From this queue, the algorithm then reconstructs a tree that approximates a balanced binary tree.

This two phase scheme, analysis followed by transformation, is common in optimization.

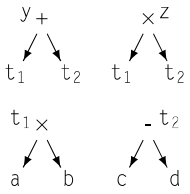
```
t1 ← a × b
t2 ← c - d
y  ← t1 + t2
z  ← t1 × t2
```
Short Basic Block

Finding Candidate Trees

A basic block consists of one or more intermixed computations. The compiler can interpret a block, in linear code, as a dependence graph (see Section 5.2.2); the graph captures both the flow of values and the ordering



Its Dependence Graph



Trees in the Graph

Observable value
A value is *observable*, with respect to a code fragment (block, loop, etc.), if it is read outside that fragment.

constraints on the operations. In the short block shown in the margin, the code must compute $a \times b$ before it can compute either $t_1 + t_2$ or $t_1 \times t_2$.

The dependence graph does not, in general, form a single tree. Instead, it consists of multiple, intertwined, connected trees. The candidate expression trees that the balancing algorithm needs each contain a subset of the nodes in the block's dependence graph. Our example block is too short to have nontrivial trees, but it has four distinct trees—one for each operation, as shown in the margin.

When the algorithm rearranges operands, larger candidate trees provide more opportunities for rearrangement. Thus, the algorithm tries to construct maximal-sized candidate trees. Conceptually, the algorithm finds candidate trees that can be considered as a single n-ary operator, for as large a value of n as possible. Several factors limit the size of a candidate tree.

1. The tree can be no larger than the block that it represents. Other transformations can increase the size of a basic block (see Section 10.6.1).
2. The rewritten code cannot change the *observable values* of the block—that is, any value used outside the block must be computed and preserved as it was in the original code. Similarly, any value used multiple times in the block must be preserved; in the example, both t_1 and t_2 have this property.
3. The tree cannot extend backward past the start of the block. In our marginal example, a, b, c, and d all receive their values before the start of the block. Thus, they become leaves in the tree.

The tree-finding phase also needs to know, for each name T_i defined in the block, where T_i is referenced. It assumes a set USES(T_i) that contains the index in the block of each use of T_i. If T_i is used after the block, then USES(T_i) should contain two additional entries—arbitrary integers greater than the number of operations in the block. This trick ensures that $|\text{USES}(x)| = 1$ if and only if x is used as a local temporary variable. We leave the construction of the USES sets as an exercise for the reader (see Problem 8.8 on page 473); it relies on LIVEOUT sets (see Section 8.6.1).

Figures 8.7 and 8.8 present the algorithm for balancing a basic block. Phase 1 of the algorithm, in Figure 8.7, is deceptively simple. It iterates over the operations in the block. It tests each operation to see if that operation must be the root of its own tree. When it finds a root, it adds the name defined by that operation to a priority queue of names, ordered by precedence of the root's operator.

```
// Rebalance a block b of n operations, each of form "Ti ← Li Opi Ri"
// Phase 1: build a queue, Roots, of the candidate trees
Roots ← new queue of names
for i ← 0 to n-1
    Rank(Ti) ← -1;
    if Opi is commutative and associative and
       (|Uses(Ti)| > 1 or (|Uses(Ti)| = 1 and Op_Uses(Ti) ≠ Opi)) then
          mark Ti as a root
          Enqueue(Roots,Ti,precedence of Opi)
// Phase 2: remove a tree from Roots and rebalance it
while (Roots is not empty)
    var ← Dequeue(Roots)
    Balance(var)

Balance(root)    // Create balanced tree from its root, Ti in "Ti ← Li Opi Ri"
    if Rank(root) ≥ 0
       then return    // have already processed this tree
    q ← new queue of names                    // First, flatten the tree
    Rank(root) ← Flatten(Li,q) + Flatten(Ri,q)
    Rebuild(q,Opi)                            //Then, rebuild a balanced tree

Flatten(var,q)    // Flatten computes a rank for var & builds the queue
    if var is a constant                      // Cannot recur further
       then
          Rank(var) ← 0
          Enqueue(q,var,Rank(var))
       else if var∈UEVar(b)                   // Cannot recur past top of block
              then
                  Rank(var) ← 1
                  Enqueue(q,var,Rank(var))
              else if var is a root
                     then                     // New queue for new root
                        Balance(var)          // Recur to find its rank
                        Enqueue(q,var,Rank(var))
                     else                     // var is Tj in jth op in block
                        Flatten(Lj,q)         // Recur on left operand
                        Flatten(Rj,q)         // Recur on right operand
    return Rank(var)
```

■ **FIGURE 8.7** Tree-Height Balancing Algorithm, Part I.

```
Rebuild(q,op)                                  // Build a balanced expression
    while (q is not empty)
       NL ← Dequeue(q)                         // Get a left operand
       NR ← Dequeue(q)                         // Get a right operand

       if NL and NR are both constants then    // Fold expression if constant
          NT ← Fold(op,NL,NR)
          if q is empty
             then
                Emit("root ← NT")
                Rank(root) = 0;
             else
                Enqueue(q,NT,0)
                Rank(NT) = 0;
       else                                    // op is not a constant expression
          if q is empty                        // Get a name for result
             then NT ← root
             else NT ← new name
          Emit("NT ← NL op NR")
          Rank(NT) ← Rank(NL) + Rank(NR)       // Compute its rank
          if q is not empty                    // More ops in q ⇒ add NT to q
             then Enqueue(q,NT,r)
```

■ **FIGURE 8.8** Tree-Height Balancing Algorithm, Part II.

The test to identify a root has two parts. Assume that operation i has the form $T_i \leftarrow L_i\ Op_i\ R_i$. First, Op_i must be both commutative and associative. Second, one of the following two conditions must hold:

1. If T_i is used more than once, then operation i must be marked as a root to ensure that T_i is available for all of its uses. Multiple uses make T_i observable.
2. If T_i is used just once, in operation j, but $Op_i \neq Op_j$, then operation i must be a root, because it cannot be part of the tree that contains Op_j.

In either case, phase 1 marks Op_i as a root and enqueues it.

Rebuilding the Block in Balanced Form

Phase 2 takes the queue of candidate-tree roots and builds, from each root, an approximately balanced tree. Phase 2 starts with a while loop that calls `Balance` on each candidate tree root. `Balance`, `Flatten`, and `Rebuild` implement phase two.

`Balance` is invoked on a candidate-tree root. Working with `Flatten`, it creates a priority queue that holds all the operands of the current tree. `Balance` allocates a new queue and then invokes `Flatten` to recursively walk the tree, assign ranks to each operand, and enqueue them. Once the candidate tree has been flattened and ranked, `Balance` invokes `Rebuild` (see Figure 8.8) to reconstruct the code.

`Rebuild` uses a simple algorithm to construct the new code sequence. It repeatedly removes the two lowest ranked items from the tree. It emits an operation to combine them. It ranks the result and inserts the ranked result back into the priority queue. This process continues until the queue is empty.

Several details of this scheme are important.

1. When traversing a candidate tree, `Flatten` can encounter the root of another tree. At that point, it recurs on `Balance` rather than on `Flatten`, to create a new priority queue for the root's candidate tree and to ensure that it emits the code for the higher precedence subtree before the code that references the subtree's value. Recall that phase 1 ranked the `Roots` queue in increasing precedence order, which forces the correct order of evaluation here.
2. The block contains three kinds of references: constants, names defined in the block before their use in the block, and *upward-exposed names*. The routine `Flatten` handles each case separately. It relies on the set UEVAR(b) that contains all the upwards-exposed names in block b. The computation of UEVAR is described in Section 8.6.1 and shown in Figure 8.14a.
3. Phase 2 ranks operands in a careful way. Constants receive rank zero, which forces them to the front of the queue, where `Fold` evaluates constant-valued operations, creates new names for the results, and works the results into the tree. Leaves receive rank one. Interior nodes receive the sum of their subtree ranks, which is equal to the number of nonconstant operands in the subtree. This ranking produces an approximation to a balanced binary tree.

Upward exposed
A name x is *upward exposed* in block b if the first use of x in b refers to a value computed before entering b.

Examples

Consider what happens when we apply the algorithm to our original example in Figure 8.5. Assume that t_7 is live on exit from the block, that t_1 through t_6 are not, and that `Enqueue` inserts before the first equal-priority element. In that case, phase 1 finds a single root, t_7, and phase 2 invokes `Balance` on t_7. `Balance`, in turn, invokes `Flatten` followed by `Rebuild`. `Flatten` builds the queue:

$$\{ \langle h,1\rangle, \langle g,1\rangle, \langle f,1\rangle, \langle e,1\rangle, \langle d,1\rangle, \langle c,1\rangle, \langle b,1\rangle, \langle a,1\rangle \}.$$

Rebuild dequeues ⟨h,1⟩ and ⟨g,1⟩, emits "$n_0 \leftarrow h + g$", and enqueues ⟨n_0,2⟩. Next, it dequeues ⟨f,1⟩ and ⟨e,1⟩, emits "$n_1 \leftarrow f + e$", and enqueues ⟨n_1,2⟩. It dequeues ⟨d,1⟩ and ⟨c,1⟩, emits "$n_2 \leftarrow d + c$", and enqueues ⟨n_2,2⟩. It then dequeues ⟨b,1⟩ and ⟨a,1⟩, emits "$n_3 \leftarrow b + a$", and enqueues ⟨n_3,2⟩.

$n_0 \leftarrow h + g$
$n_1 \leftarrow f + e$
$n_2 \leftarrow d + c$
$n_3 \leftarrow b + a$
$n_4 \leftarrow n_3 + n_2$
$n_5 \leftarrow n_1 + n_0$
$t_7 \leftarrow n_5 + n_4$

At this point, *Rebuild* has produced partial sums with all eight of the original values. The queue now contains {⟨n_3,2⟩, ⟨n_2,2⟩, ⟨n_1,2⟩, ⟨n_0,2⟩}. The next iteration dequeues ⟨n_3,2⟩ and ⟨n_2,2⟩, emits "$n_4 \leftarrow n_3 + n_2$" and enqueues ⟨n_4,4⟩. Next, it dequeues ⟨n_1,2⟩ and ⟨n_0,2⟩, emits "$n_5 \leftarrow n_1 + n_0$" and enqueues ⟨n_5,4⟩. The final iteration dequeues ⟨n_5,4⟩ and ⟨n_4,4⟩, and emits "$t_7 \leftarrow n_5 + n_4$". The complete code sequence, shown the margin, matches to the balanced tree shown in Figure 8.5c; the resulting code can be scheduled as in the left side of Figure 8.6.

As a second example, consider the basic block shown in Figure 8.9a. This code might result from local value numbering; constants have been folded and redundant computations eliminated. The block contains several intertwined computations. Figure 8.9b shows the expression trees in the block. Note that t_3 and t_7 are reused by name. The longest path chain of computation is the tree headed by t_6 which has six operations.

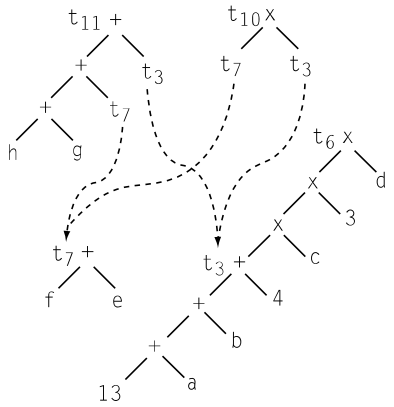
When we apply phase 1 of the tree-height balancing algorithm to the block in Figure 8.9, it finds five roots, shown boxed in Figure 8.9c. It marks t_3 and t_7 because they have multiple uses. It marks t_6, t_{10}, and t_{11} because they are in LIVEOUT(*b*). At the end of phase 1, the priority queue *Roots*

UEVAR is
{a,c,e,f,g,h,m,n}

LIVEOUT is
{t_6,t_{10},t_{11}}

$t_1 \leftarrow 13 + a$
$t_2 \leftarrow t_1 + b$
$t_3 \leftarrow t_2 + 4$
$t_4 \leftarrow t_3 \times c$
$t_5 \leftarrow 3 \times t_4$
$t_6 \leftarrow d \times t_5$
$t_7 \leftarrow e + f$
$t_8 \leftarrow t_7 + g$
$t_9 \leftarrow t_8 + h$
$t_{10} \leftarrow t_3 \times t_7$
$t_{11} \leftarrow t_3 + t_9$

(a) Original Code



(b) Trees in the Code

$t_1 \leftarrow 13 + a$
$t_2 \leftarrow t_1 + b$
$\boxed{t_3} \leftarrow t_2 + 4$
$t_4 \leftarrow t_3 \times c$
$t_5 \leftarrow 3 \times t_4$
$\boxed{t_6} \leftarrow d \times t_5$
$\boxed{t_7} \leftarrow e + f$
$t_8 \leftarrow t_7 + g$
$t_9 \leftarrow t_8 + h$
$\boxed{t_{10}} \leftarrow t_3 \times t_7$
$\boxed{t_{11}} \leftarrow t_3 + t_9$

(c) Finding Roots

■ **FIGURE 8.9** Example of Tree-Height Balancing.

contains:

$$\{ \langle t_{11},1\rangle, \langle t_7,1\rangle, \langle t_3,1\rangle, \langle t_{10},2\rangle, \langle t_6,2\rangle \},$$

assuming that the precedence of + is 1, the precedence of × is 2.

Phase 2 of the algorithm repeatedly removes a node from the *Roots* queue and calls *Balance* to process it. *Balance*, in turn, uses *Flatten* to create a priority queue of operands and then uses *Rebuild* to create a balanced computation from the operands. (Remember that each tree contains just one kind of operation.)

Phase 2 begins by calling *Balance* on t_{11}. Recall from Figure 8.9 that t_{11} is the sum of t_3 and t_7. *Balance* calls *Flatten* on each of those nodes, which are, themselves, roots of other trees. Thus, the call to *Flatten*(t_3, q) invokes *Balance* on t_3 and then invokes it on t_7.

Balance(t_3) flattens that tree into the queue { ⟨4,0⟩, ⟨13,0⟩, ⟨b,1⟩, ⟨a,1⟩ } and invokes *Rebuild* on that queue. *Rebuild* dequeues ⟨4,0⟩ and ⟨13,0⟩, combines them, and enqueues ⟨17,0⟩. Next, it dequeues ⟨17,0⟩ and ⟨b,1⟩, emits "$n_0 \leftarrow 17 + b$", and adds ⟨n_0,1⟩ to the queue. On the final iteration for the t_3 tree, it dequeues ⟨n_0,1⟩ and ⟨a,1⟩, and emits "$t_3 \leftarrow n_0 + a$". It marks t_3 with rank 2 and returns.

$n_0 \leftarrow 17 + b$
$t_3 \leftarrow n_0 + a$

Invoking *Balance* on t_7 builds a trivial queue, { ⟨e,1⟩, ⟨f,1⟩ } and emits the operation "$t_7 \leftarrow e + f$". That completes the first iteration of the while loop in phase 2.

$t_7 \leftarrow e + f$

Next, phase 2 invokes *Balance* on the tree at t_{11}. It calls *Flatten*, which builds the queue { ⟨h,1⟩, ⟨g,1⟩, ⟨t_7,2⟩, ⟨t_3,2⟩ }. Then, *Rebuild* emits the code "$n_1 \leftarrow h + g$" and enqueues n_1 with rank 2. Next, it emits the code "$n_2 \leftarrow n_1 + t_7$" and enqueues n_2 with rank 4. Finally, it emits the code "$t_{11} \leftarrow n_2 + t_3$" and marks t_{11} with rank 6.

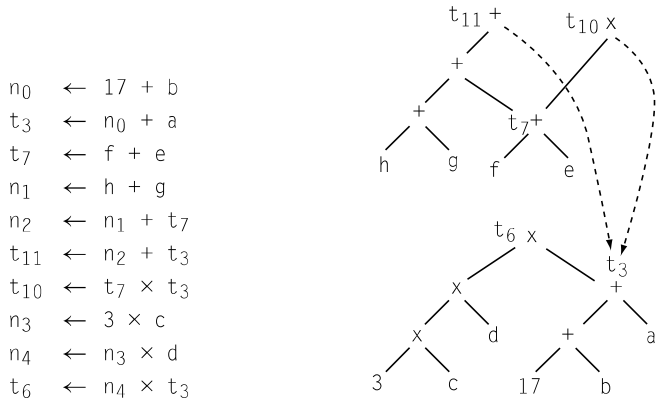
$n_1 \leftarrow h + g$
$n_2 \leftarrow n_1 + t_7$
$t_{11} \leftarrow n_2 + t_3$

The next two items that phase 2 dequeues from the *Roots* queue, t_7 and t_3, have already been processed, so they have nonzero ranks. Thus, *Balance* returns immediately on each of them.

The final call to *Balance* from phase 2 passes it the root t_6. For t_6, *Flatten* constructs the queue: { ⟨3,0⟩, ⟨d,1⟩, ⟨c,1⟩, ⟨t_3,2⟩ }. *Rebuild* emits the code "$n_3 \leftarrow 3 + d$" and enqueues n_3 with rank 1. Next, it emits "$n_4 \leftarrow n_3 + c$" and enqueues n_4 with rank 2. Finally, it emits "$t_6 \leftarrow n_4 + t_3$" and marks t_3 with rank 4.

$n_3 \leftarrow 3 + d$
$n_4 \leftarrow n_3 + c$
$t_6 \leftarrow n_4 + t_3$

The resulting tree is shown in Figure 8.10. Note that the tree rooted at t_6 now has a height of three operations, instead of six.



(a) Transformed Code (b) Trees in the Code

■ **FIGURE 8.10** Code Structure after Balancing.

SECTION REVIEW

Local optimization operates on the code for a single basic block. The techniques rely on the information available in the block to rewrite that block. In the process, they must maintain the block's interactions with the surrounding execution context. In particular, they must preserve any observable values computed in the block.

Because they limit their scope to a single block, local optimizations can rely on properties that only hold true in straightline code. For example, local value numbering relies on the fact that all the operations in the block execute in an order that is consistent with straightline execution. Thus, it can build a model of prior context that exposes redundancies and constant-valued expressions. Similarly, tree-height balancing relies on the fact that a block has just one exit to determine which subexpressions in the block it must preserve and which ones it can rearrange.

Review Questions

1. Sketch an algorithm to find the basic blocks in a procedure expressed in ILOC. What data structures might you use to represent the basic block?
2. The tree-height balancing algorithm given in Figures 8.7 and 8.8 ranks a node *n* in the final expression tree with the number of nonconstant leaves below it in the final tree. How would you modify the algorithm to produce ranks that correspond to the height of *n* in the tree? Would that change the code that the algorithm produces?

8.5 REGIONAL OPTIMIZATION

Inefficiencies are not limited to single blocks. Code that executes in one block may provide the context for improving the code in another block. Thus, most optimizations examine a larger context than a single block.

This section examines two techniques that operate over regions of code that include multiple blocks but do not, typically, extend to an entire procedure. The primary complication that arises in the shift from local optimization to regional optimization is the need to handle more than one possibility for the flow of control. An if-then-else can take one of two paths. The branch at the end of a loop can jump back to another iteration or it can jump to the code that follows the loop.

To illustrate regional techniques, we present two of them. The first, superlocal value numbering, is an extension of local value numbering to larger regions. The second is a loop optimization that appeared in our discussion of the `dmxpy` loop nest: loop unrolling.

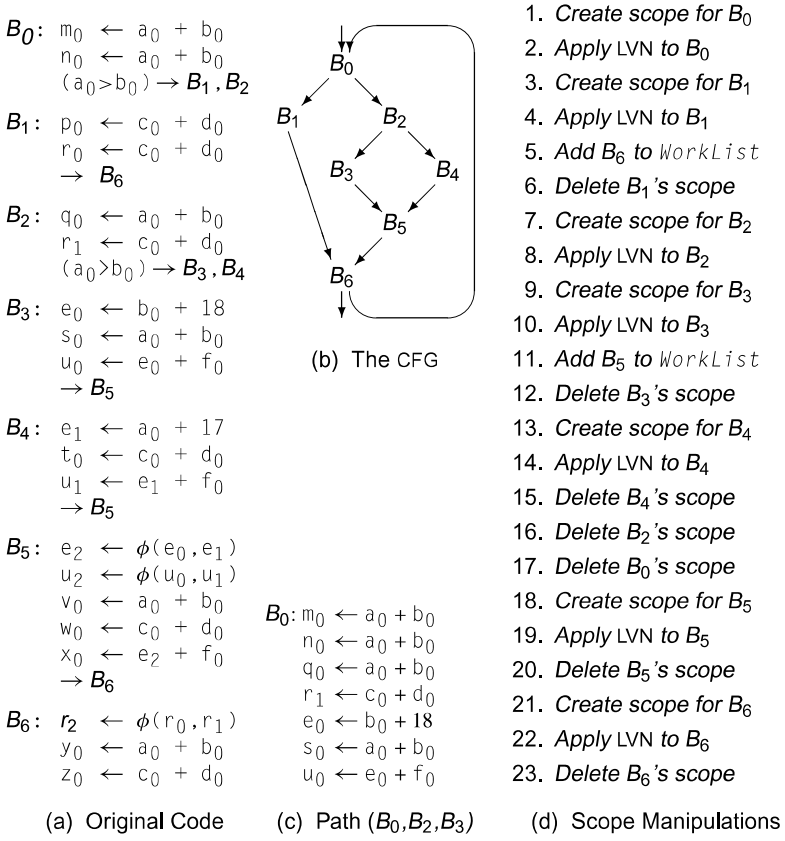
8.5.1 Superlocal Value Numbering

To improve the results of local value numbering, the compiler can extend its scope from a single basic block to an extended basic block, or EBB. To process an EBB, the algorithm should value number each path through the EBB. Consider, for example, the code shown in Figure 8.11a. Its CFG, shown in Figure 8.11b, contains one nontrivial EBB, $(B_0, B_1, B_2, B_3, B_4)$, and two trivial EBBs, (B_5) and (B_6). We call the resulting algorithm *superlocal value numbering* (SVN).

In the large EBB, SVN could treat each of the three paths as if it were a single block. That is, it could behave as if each of (B_0, B_1), (B_0, B_2, B_3), and (B_0, B_2, B_4) were straightline code. To process (B_0, B_1), the compiler can apply LVN to B_0 and use the resulting hash table as a starting point when it applies LVN to B_1. The same approach would handle (B_0, B_2, B_3) and (B_0, B_2, B_4) by processing the blocks for each in order and carrying the hash tables forward. The effect of this scheme is to treat a path as if it were a single block. For example, it would optimize (B_0, B_2, B_3) as if it had the code as shown in Figure 8.11c. Any block with multiple predecessors, such as B_5 and B_6, must be handled as in local value numbering—without context from any predecessors.

This approach can find redundancies and constant-valued expressions that a strictly local value numbering algorithm would miss.

- In (B_0, B_1), LVN discovers that the assignments to n_0 and r_0 are redundant. SVN discovers the same redundancies.



■ **FIGURE 8.11** Superlocal Value Numbering Example.

- In (B_0, B_2, B_3), LVN finds that the assignment to n_0 is redundant. SVN also finds that the assignments to q_0 and s_0 are redundant.
- In (B_0, B_2, B_4), LVN finds that the assignment to n_0 is redundant. SVN also finds that the assignments to q_0 and t_0 are redundant.
- In B_5 and B_6, SVN degenerates to LVN.

The difficulty in this approach lies in making the process efficient. The obvious approach is to treat each path as if it were a single block, pretending, for example, that the code for (B_0, B_2, B_3) looks like the code in Figure 8.11c. Unfortunately, this approach analyzes a block once for each path that includes it. In the example, this approach would analyze B_0 three times and B_2 twice. While we want the optimization benefits that come from examining increased context, we also want to minimize compile-time costs.

For this reason, superlocal algorithms often capitalize on the tree structure of the EBB.

To make SVN efficient, the compiler must reuse the results of blocks that occur as prefixes on multiple paths through the EBB. It needs a way to undo the effects of processing a block. After processing (B_0, B_2, B_3) it must recreate the state for the end of (B_0, B_2) so that it can reuse that state to process B_4.

Among the many ways that the compiler can accomplish this effect are:

- It can record the state of the table at each block boundary and restore that state when needed.
- It can unwind the effects of a block by walking the block backward and, at each operation, undoing the work of the forward pass.
- It can implement the value table using the mechanisms developed for lexically scoped hash tables. As it enters a block, it creates a new scope. To retract the block's effects, it deletes that block's scope.

While all three schemes will work, using a scoped value table can produce the simplest and fastest implementation, particularly if the compiler can reuse an implementation from the front end (see Section 5.5.3).

Figure 8.12 shows a high-level sketch of the SVN algorithm, using a scoped value table. It assumes that the LVN algorithm has been parameterized to accept a block and a scoped value table. At each block b, it allocates a value table for b, links the value tables of the predecessor block as if it were a surrounding scope, and invokes LVN on block b with this new table. When LVN returns, SVN must decide what to do with each of b's successors.

The "sheaf-of-tables" implementation shown in Section 5.5.3 has the right properties for SVN. SVN can easily estimate the size of each table. The deletion mechanism is both simple and fast.

For a successor s of b, two cases arise. If s has exactly one predecessor, b, then it should be processed with the accumulated context from b. Accordingly, SVN recurs on s with the table containing b's context. If s has multiple predecessors, then s must start with an empty context. Thus, SVN adds s to the `WorkList` where the outer loop will later find it and invoke SVN on it and the empty table.

One complication remains. A name's value number is recorded in the value table associated with the first operation in the EBB that defines it. This effect can defeat our use of the scoping mechanism. In our example CFG, if a name x were defined in each of B_0, B_3, and B_4, its value number would be recorded in the scoped table for B_0. When SVN processed B_3, it would record x's new value number from B_3 in the table for B_0. When SVN deleted the table for B_3 and created a new table for B_4, the value number from the definition in B_3 would remain.

```
// Start the process
WorkList ← { entry block }
Empty ← new table
while (WorkList is not empty)
   remove b from WorkList
   SVN(b, Empty)

// Superlocal value numbering algorithm
SVN(Block, Table)
   t ← new table for Block
   link Table as the surrounding scope for t
   LVN(Block, t)
   for each successor s of Block do
      if s has only 1 predecessor
         then SVN(s, t)
      else if s has not been processed
         then add s to WorkList
   deallocate t
```

■ **FIGURE 8.12** Superlocal Value Numbering Algorithm.

To avoid this complication, the compiler can run SVN on a representation that defines each name once. As we saw in Section 5.4.2, SSA form has the requisite property; each name is defined at exactly one point in the code. Using SSA form ensures that SVN records the value number for a definition in the table that corresponds to the block containing the definition. With SSA form, deleting the table for a block undoes all of its effects and reverts the value table to its state at the exit of the block's CFG predecessor. As discussed in Section 8.4.1, using SSA form can also make LVN more effective.

Applying the algorithm from Figure 8.12 to the code from Figure 8.11a produces the sequence of actions shown in Figure 8.11d. It begins with B_0 and proceeds down to B_1. At the end of B_1, it visits B_6, realizes that B_6 has multiple predecessors, and adds it to the worklist. Next, it backs up and processes B_2 and then B_3. At the end of B_3, it adds B_5 to the worklist. It then backs up to B_2 and processes B_4. At that point, control returns to the while loop, which invokes SVN on the two singleton blocks from the worklist, B_5 and B_6.

In terms of effectiveness, SVN discovers and removes redundant computations that LVN cannot. As mentioned earlier in the section, it finds that the

assignments to q_0, s_0, and t_0 are redundant because of definitions in earlier blocks. LVN, with its purely local scope, cannot find these redundancies.

On the other hand, SVN has its own limitations. It fails to find redundancies in B_5 and B_6. The reader can tell, by inspection, that each assignment in these two blocks is redundant. Because those blocks have multiple predecessors, SVN cannot carry context into them. Thus, it misses those opportunities; to catch them, we need an algorithm that can consider a larger amount of context.

8.5.2 Loop Unrolling

Loop unrolling is, perhaps, the oldest and best-known loop transformation. To unroll a loop, the compiler replicates the loop's body and adjusts the logic that controls the number of iterations performed. To see this, consider the loop nest from `dmxpy` used as an example in Section 8.2.

```
      do 60 j = 1, n2
         do 50 i = 1, n1
            y(i) = y(i) + x(j) * m(i,j)
50       continue
60    continue
```

Loop fusion
The process of combining two loop bodies into one is called *fusion*.

Fusion is safe when each definition and each use in the resulting loop has the same value that it did in the original loops.

The compiler can unroll either the inner loop or the outer loop. The result of inner-loop unrolling is shown in Figure 8.13a. Unrolling the outer loop produces four inner loops; if the compiler then combines those inner-loop bodies—a transformation called *loop fusion*—it will produce code similar to that shown in Figure 8.13b. The combination of outer-loop unrolling and subsequent fusion of the inner loops is often called *unroll-and-jam*.

In each case, the transformed code needs a short prologue loop that peels off enough iterations to ensure that the unrolled loop processes an integral multiple of four iterations. If the respective loop bounds are all known at compile time, the compiler can determine whether or not the prologue is necessary.

These two distinct strategies, inner-loop unrolling and outer-loop unrolling, produce different results for this particular loop nest. Inner loop unrolling produces code that executes many fewer test-and-branch sequences than did the original code. In contrast, outer-loop unrolling followed by fusion of the inner loops not only reduces the number of test-and-branch sequences, but also produces reuse of `y(i)` and sequential access to both `x` and `m`. The increased reuse fundamentally changes the ratio of arithmetic operations to

Access to `m` is sequential because FORTRAN stores arrays in column-major order.

```
      do 60 j = 1, n2
         nextra = mod(n1,4)
         if (nextra .ge. 1) then
            do 49 i = 1, nextra
               y(i) = y(i) + x(j) * m(i,j)
49          continue

         do 50 i = nextra + 1, n1, 4
            y(i)   = y(i) + x(j) * m(i,j)
            y(i+1) = y(i+1) + x(j) * m(i+1,j)
            y(i+2) = y(i+2) + x(j) * m(i+2,j)
            y(i+3) = y(i+3) + x(j) * m(i+3,j)
50          continue
60       continue
```

(a) Unroll Inner Loop by Four

```
      nextra = mod(n2,4)
      if (nextra .ge. 1) then
         do 59 j = 1, nextra
            do 49 i = 1, n1
               y(i) = y(i) + x(j) * m(i,j)
49          continue
59       continue

      do 60 j = nextra+1, n2, 4
         do 50 i = 1, n1
            y(i) = y(i) + x(j) * m(i,j)
            y(i) = y(i) + x(j+1) * m(i,j+1)
            y(i) = y(i) + x(j+2) * m(i,j+2)
            y(i) = y(i) + x(j+3) * m(i,j+3)
50          continue
60       continue
```

(b) Unroll Outer Loop by Four, Fuse Inner Loops

■ **FIGURE 8.13** Unrolling `dmxpy`'s Loop Nest.

memory operations in the loop; undoubtedly, the author of `dmxpy` had that effect in mind when he hand-optimized the code. As discussed below, each approach may also accrue indirect benefits.

Sources of Improvement and Degradation

Loop unrolling has both direct and indirect effects on the code that the compiler can produce for a given loop. The final performance of the loop depends on all of the effects, direct and indirect.

In terms of direct benefits, unrolling should reduce the number of operations required to complete the loop. The control-flow changes reduce the total number of test-and-branch sequences. Unrolling can create reuse within the loop body, reducing memory traffic. Finally, if the loop contains a cyclic chain of copy operations, unrolling can eliminate the copies (see Exercise 5 in this chapter).

As a hazard, though, unrolling increases program size, both in its IR form and in its final form as executable code. Growth in IR increases compile time; growth in executable code has little effect until the loop overflows the instruction cache—at which time the degradation probably overwhelms any direct benefits.

The compiler can also unroll for indirect effects, which can affect performance. The key side effect of unrolling is to increase the number of

operations inside the loop body. Other optimizations can capitalize on this change in several ways:

- Increasing the number of independent operations in the loop body can lead to better instruction schedules. With more operations, the scheduler has a better chance to keep multiple functional units busy and to hide the latency of long-duration operations such as branches and memory accesses.
- Unrolling can move consecutive memory accesses into the same loop iteration, where the compiler can schedule them together. That may improve locality or allow the use of multiword operations.
- Unrolling can expose cross-iteration redundancies that are harder to discover in the original code. For example, both versions of the code shown in Figure 8.13 reuse address expressions across iterations of the original loop. In the unrolled loop, local value numbering would find and eliminate those redundancies. In the original, it would miss them.
- The unrolled loop may optimize in a different way than the original loop. For example, increasing the number of times that a variable occurs inside the loop can change the weights used in spill code selection within the register allocator (see Section 13.4). Changing the pattern of register spills can radically affect the speed of the final code for the loop.
- The unrolled loop body may have a greater demand for registers than the original loop body. If the increased demand for registers induces additional register spills (stores and reloads), then the resulting memory traffic may overwhelm the potential benefits of unrolling.

These indirect interactions are much harder to characterize and understand than the direct effects. They can produce significant performance improvements. They can also produce performance degradations. The difficulty of predicting such indirect effects has led some researchers to advocate an adaptive approach to choosing unroll factors; in such systems, the compiler tries several unroll factors and measures the performance of the resulting code.

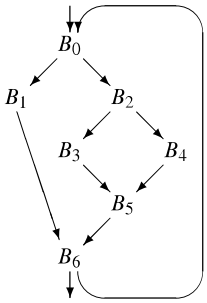
SECTION REVIEW

Optimizations that focus on regions larger than a block and smaller than a whole procedure can provide improved performance for a modest increase in compile-time cost. For some transformations, the analysis needed to support the transformation and the impact that it has on the compiled code are both limited in scope.

Superlocal transformations have a rich history in both the literature and the practice of code optimization. Many local transformations adapt

easily and efficiently to extended basic blocks. Superlocal extensions to instruction scheduling have been a staple of optimizing compilers for many years (see Section 12.4).

Loop-based optimizations, such as unrolling, can produce significant improvements, primarily because so many programs spend a significant fraction of their execution time inside loops. That simple fact makes loops and loop nests into rich targets for analysis and transformation. Improvements made inside a loop have a much larger impact than those made in code outside all loop nests. A regional approach to loop optimization makes sense because different loop nests can have radically different performance characteristics. Thus, loop optimization has been a major focus of optimization research for decades.



Review Questions

1. Superlocal value numbering extends local value numbering to extended basic blocks through clever use of a scoped hash table. Consider the issues that might arise in extending the tree-height balancing algorithm to a superlocal scope.
   a. How would you handle a single path through an EBB, such as (B_0, B_2, B_3) in the control-flow graph shown in the margin?
   b. What complications arise when the algorithm tries to process (B_0, B_2, B_4) after processing (B_0, B_2, B_3)?
2. The following code fragment computes a three-year trailing average:

```
TYTA(float *Series; float *TYTAvg; int count) {
   int i;
   float Minus2, Minus1;

   Minus2 = Series++;
   Minus1 = Series++;

   for (i=1; i ≤ count; i++) {
         Current = Series++;
         TYTAvg++ = (Current + Minus1 + Minus2)/3;
         Minus2 = Minus1;
         Minus1 = Current;
      }
}
```

   What improvements would accrue from unrolling the loop? How would the unroll factor affect the benefits?

Hint: Compare possible improvements with unroll factors of two and three.

8.6 GLOBAL OPTIMIZATION

Global optimizations operate on an entire procedure or method. Because their scope includes cyclic control-flow constructs such as loops, these methods typically perform an analysis phase before modifying the code.

This section presents two examples of global analysis and optimization. The first, finding uninitialized variables with live information, is not strictly an optimization. Rather, it uses global data-flow analysis to discover useful information about the flow of values in a procedure. We will use the discussion to introduce the computation of *live variables* information, which plays a role in many optimization techniques, including tree-height balancing (Section 8.4.2), the construction of SSA information (Section 9.3), and register allocation (Chapter 13). The second, global code placement, uses profile information gathered from running the compiled code to rearrange the layout of the executable code.

8.6.1 Finding Uninitialized Variables with Live Information

If a procedure p can use the value of some variable v before v has been assigned a value, we say that v is uninitialized at that use. Use of an uninitialized variable almost always indicates a logical error in the procedure being compiled. If the compiler can identify these situations, it should alert the programmer to their existence.

We can find potential uses of uninitialized variables by computing information about *liveness*. A variable v is live at point p if and only if there exists a path in the CFG from p to a use of v along which v is not redefined. We encode live information by computing, for each block b in the procedure, a set LIVEOUT(b) that contains all the variables that are live on exit from b. Given a LIVEOUT set for the CFG's entry node n_0, each variable in LIVEOUT(n_0) has a potentially uninitialized use.

Data-flow analysis
a form of compile-time analysis for reasoning about the flow of values at runtime

The computation of LIVEOUT sets is an example of *global data-flow analysis*, a family of techniques for reasoning, at compile time, about the flow of values at runtime. Problems in data-flow analysis are typically posed as a set of simultaneous equations over sets associated with the nodes and edges of a graph.

Defining the Data-Flow Problem

Computing LIVEOUT sets is a classic problem in global data-flow analysis. The compiler computes, for each node n in the procedure's CFG, a set LIVEOUT(n) that contains all the variables that are live on exit from the block

corresponding to n. For each node n in the procedure's CFG, LIVEOUT(n) is defined by an equation that uses the LIVEOUT sets of n's successors in the CFG, and two sets UEVAR(n) and VARKILL(n) that encode facts about the block associated with n. We can solve the equations using an iterative fixed-point method, similar to the fixed-point methods that we saw in earlier chapters such as the subset construction in Section 2.4.3.

The defining equation for LIVEOUT is:

$$\text{LIVEOUT}(n) = \bigcup_{m \in succ(n)} (\text{UEVAR}(m) \cup (\text{LIVEOUT}(m) \cap \overline{\text{VARKILL}(m)}))$$

UEVAR(m) contains the upward-exposed variables in m—those variables that are used in m before any redefinition in m. VARKILL(m) contains all the variables that are defined in m and the overline on VARKILL(m) indicates its logical complement, the set of all variables not defined in m. Because LIVEOUT(n) is defined in terms of n's successors, the equation describes a *backward data-flow problem.*

Backward data-flow problem
a problem in which information flows backward over graph edges

Forward data-flow problem
a problem in which information flows along the graph edges

The equation encodes the definition in an intuitive way. LIVEOUT(n) is just the union of those variables that are live at the head of some block m that immediately follows n in the CFG. The definition requires that a value be live on some path, not on all paths. Thus, the contributions of the successors of n in the CFG are unioned together to form LIVEOUT(n). The contribution of a specific successor m of n is:

$$\text{UEVAR}(m) \cup (\text{LIVEOUT}(m) \cap \overline{\text{VARKILL}(m)}).$$

A variable, v, is live on entry to m under one of two conditions. It can be referenced in m before it is redefined in m, in which case $v \in$ UEVAR(m). It can be live on exit from m and pass unscathed through m because m does not redefine it, in which case $v \in$ LIVEOUT(m) $\cap \overline{\text{VARKILL}(m)}$. Combining these two sets, with $\cup$, gives the necessary contribution of m to LIVEOUT(n). To compute LIVEOUT(n), the analyzer combines the contributions of all n's successors denoted $succ(n)$.

Solving the Data-Flow Problem

To compute the LIVEOUT sets for a procedure and its CFG, the compiler can use a three-step algorithm.

1. *Build a* CFG This step is conceptually simple, although language and architecture features can complicate the problem (see Section 5.3.4).
2. *Gather initial information* The analyzer computes a UEVAR and VARKILL set for each block b in a simple walk, as shown in Figure 8.14a.

3. *Solve the equations to produce* LIVEOUT*(b) for each block b* Figure 8.14b shows a simple iterative fixed-point algorithm that will solve the equations.

The following sections work through an example computation of LIVEOUT. Section 9.2 delves into data-flow computations in more depth.

Gathering Initial Information

To compute LIVEOUT, the analyzer needs UEVAR and VARKILL sets for each block. A single pass can compute both. For each block, the analyzer initializes these sets to ∅. Next, it walks the block, in order from top to bottom, and updates both UEVAR and VARKILL to reflect the impact of each operation. Figure 8.14a shows the details of this computation.

Consider the CFG with a simple loop that contains an `if-then` construct, shown in Figure 8.15a. The code abstracts away many details. Figure 8.15b shows the corresponding UEVAR and VARKILL sets.

Solving the Equations for LIVEOUT

Given the UEVAR and VARKILL sets, the compiler applies the algorithm from Figure 8.14b to compute LIVEOUT sets for each node in the CFG. It initializes all of the LIVEOUT sets to ∅. Next, it computes the LIVEOUT set for each block, in order from B_0 to B_4. It repeats the process, computing LIVEOUT for each node in order until the LIVEOUT sets no longer change.

```
// assume block b has k operations
// of form "x ← y op z"
for each block b
   Init(b)

Init(b)
   UEVAR(b) ← ∅
   VARKILL(b) ← ∅
   for i ← 1 to k
      if y ∉ VARKILL(b)
         then add y to UEVAR(b)
      if z ∉ VARKILL(b)
         then add z to UEVAR(b)
      add x to VARKILL(b)
```

(a) Gathering Initial Information

```
// assume CFG has N blocks
// numbered 0 to N-1
for i ← 0 to N-1
   LIVEOUT(i) ← ∅

changed ← true
while (changed)
   changed ← false
   for i ← 0 to N-1
      recompute LIVEOUT(i)
      if LIVEOUT(i) changed then
         changed ← true
```

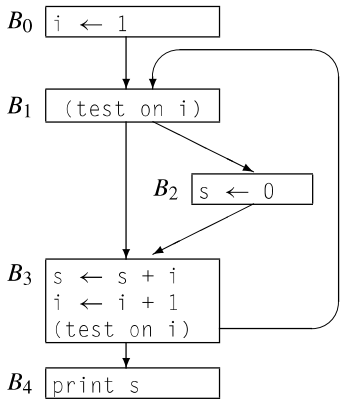
(b) Solving the Equations

■ **FIGURE 8.14** Iterative Live Analysis.

The table in Figure 8.15c shows the values of the LIVEOUT sets at each iteration of the solver. The row labelled *Initial* shows the initial values. The first iteration computes an initial approximation to the LIVEOUT sets. Because it processes the blocks in ascending order of their labels, B_0, B_1, and B_2 receive values based solely on the UEVAR sets of their CFG successors. When the algorithm reaches B_3, it has already computed an approximation for LIVEOUT(B_1), so the value that it computes for B_3 reflects the contribution of the new value for LIVEOUT(B_1). LIVEOUT(B_4) is empty, as befits the exit block.

In the second iteration, the value `s` is added to LIVEOUT(B_0) as a consequence of its presence in the approximation of LIVEOUT(B_1). No other changes occur. The third iteration does not change the values of the LIVEOUT sets and halts.

The order in which the algorithm processes the blocks affects the values of the intermediate sets. If the algorithm visited the blocks in descending



(a) Example Control-Flow Graph

	UEVAR	VARKILL
B_0	∅	{i}
B_1	{i}	∅
B_2	∅	{s}
B_3	{s,i}	{s,i}
B_4	{s}	∅

(b) Initial Information

	LIVEOUT(*n*)				
Iteration	B_0	B_1	B_2	B_3	B_4
Initial	∅	∅	∅	∅	∅
1	{i}	{s,i}	{s,i}	{s,i}	∅
2	{s,i}	{s,i}	{s,i}	{s,i}	∅
3	{s,i}	{s,i}	{s,i}	{s,i}	∅

(c) Progress of the Solution

■ **FIGURE 8.15** Example LIVEOUT Computation.

order of their labels, it would require one fewer pass. The final values of the LIVEOUT sets are independent of the evaluation order. The iterative solver in Figure 8.14 computes a fixed-point solution to the equations for LIVEOUT.

The algorithm will halt because the LIVEOUT sets are finite and the recomputation of the LIVEOUT set for a block can only increase the number of names in that set. The only mechanism in the equation for excluding a name is the intersection with $\overline{\text{VARKILL}}$. Since VARKILL does not change during the computation, the update to each LIVEOUT set increases monotonically and, thus, the algorithm must eventually halt.

Finding Uninitialized Variables

Once the compiler has computed LIVEOUT sets for each node in the procedure's CFG, finding uses of variables that may be uninitialized is straightforward. Consider some variable v. If $v \in \text{LIVEOUT}(n_0)$, where n_0 is the entry node of the procedure's CFG, then, by the construction of $\text{LIVEOUT}(n_0)$, there exists a path from n_0 to a use of v along which v is not defined. Thus, $v \in \text{LIVEOUT}(n_0)$ implies that v has a use that may receive an uninitialized value.

This approach will identify variables that have a potentially uninitialized use. The compiler should recognize that situation and report it to the programmer. However, this approach may yield false positives for several reasons.

- If v is accessible through another name and initialized through that name, live analysis will not connect the initialization and the use. This situation can arise when a pointer is set to the address of a local variable, as in the code fragment shown in the margin.
- If v exists before the current procedure is invoked, then it may have been previously initialized in a manner invisible to the analyzer. This case can arise with static variables of the current scope or with variables declared outside the current scope.
- The equations for live analysis may discover a path from the procedure's entry to a use of v along which v is not defined. If that path is not feasible at runtime, then v will appear in $\text{LIVEOUT}(n_0)$ even though no execution will ever use the uninitialized value. For example, the C program in the margin always initializes `s` before its use, yet `s` $\in$ $\text{LIVEOUT}(n_0)$.

```
  ...
p = &x;
*p = 0;
  ...
x = x + 1;
```

```
main() {
  int i, n, s;
  scanf("%d", &n);
  i = 1;
  while (i<=n) {
    if (i==1)
      s = 0;
    s = s + i++;
  }
}
```

If the procedure contains a procedure call and v is passed to that procedure in a way that allows modification, then the analyzer must account for possible side effects of the call. In the absence of specific information about the callee, the analyzer must assume that every variable that might be modified is

modified and that any variable that might be used is used. Such assumptions are safe, in that they represent the worst-case behavior.

The marginal example with the while loop illustrates one of the fundamental limits of data-flow analysis: it assumes that all paths through the CFG are feasible at runtime. That assumption can be overly conservative, as in the example. The only path in the CFG leading to an uninitialized use leads from entry of `main` into the loop, bypasses the initialization of `s`, and hits the increment of `s`. That path can never occur, because `i` must have the value `1` on the loop's first iteration. The equations for LIVEOUT cannot discover that fact.

The assumption that all paths in the CFG are feasible greatly reduces the cost of the analysis. At the same time, the assumption produces a loss of precision in the computed sets. To discover that `s` is initialized on the first iteration of the `for` loop, the compiler would need to combine an analysis that tracked individual paths with some form of constant propagation and with live analysis. To solve the problem in general would require symbolic evaluation of parts of the code during the analysis, a much more expensive prospect.

Other Uses for Live Variables

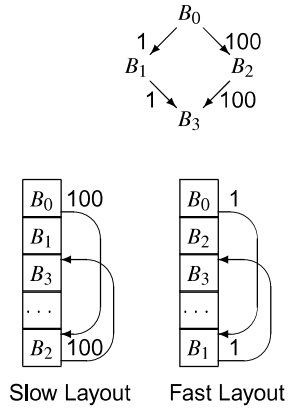
Compilers use liveness in many contexts other than finding uninitialized variables.

- Live-variable information plays a critical role in global register allocation (see Section 13.4). The register allocator need not keep values in registers unless they are live; when a value makes the transition from being live to being not live, the allocator can reuse its register for another purpose.
- Live-variable information is used to improve the SSA construction; a value does not need a ϕ-function in any block where it is not live. Using live information in this way can significantly reduce the number of ϕ-functions that the compiler must insert when building the SSA form of a program.
- The compiler can use live information to discover useless store operations. At an operation that stores v to memory, if v is not live then the store is useless. This simple technique works well for unambiguous scalar variables—that is, variables known by only one name.

In different contexts, liveness is calculated for different sets of names. We have discussed LIVEOUT with an implicit domain of variable names. In register allocation, the compiler will compute LIVEOUT sets over the domain of register names or over contiguous subranges of those register names.

8.6.2 Global Code Placement

Fall-through branch
A one-address branch is either *taken* or execution *falls through* to the next operation in sequence.



Many processors have asymmetric branch costs; the cost of a fall-through branch is less than the cost of a taken branch. Each branch has two successor basic blocks; the compiler can choose which block lies on the fall-through path and which lies on the taken path. The global code placement optimization relies, implicitly, on the observation that some branches have lopsided behavior—that the fall-through path has a lower cost than the taken path.

Consider the CFG shown in the margin. (B_0, B_2) executes 100 times more often than (B_0, B_1). With asymmetric branch costs, the compiler should use the less expensive branch for (B_0, B_2). If (B_0, B_1) and (B_0, B_2) had roughly equal execution frequencies, then block placement would have little impact for this code.

Two different layouts for this code are shown to the left. The "slow" layout uses the fall-through branch to implement (B_0, B_1) and the taken branch for (B_0, B_2). The "fast" layout reverses this decision. If the fall-through branch is faster than the taken branch, then the "fast" layout uses the faster branch 100 times more often.

The compiler can take advantage of asymmetric branch costs. If the compiler knows the expected relative execution frequencies of the branches in a procedure, it can select a code layout that improves runtime performance.

To perform global code placement, the compiler reorders the basic blocks of a procedure to optimize the use of fall-through branches. It follows two principles. First, the compiler should make the most likely execution paths use fall-through branches. Thus, whenever possible, block should be followed immediately by its most frequent successor. Second, the compiler should move code that executes infrequently to the end of the procedure. Taken together, these principles produce longer sequences that execute without a disruptive (e.g. taken) branch.

We expect two beneficial effects from this execution order. The code should execute a larger proportion of fall-through branches, which may directly improve performance. That pattern should lead to more efficient instruction cache use.

Code placement, like most optimizations at the global scope, has separate analysis and transformation phases. The analysis phase must gather estimates of each branch's relative execution frequency. The transformation uses those branch frequencies, expressed as weights on edges in the CFG, to build a model of the frequently executed paths. It then orders the basic blocks from that model.

GATHERING PROFILE DATA

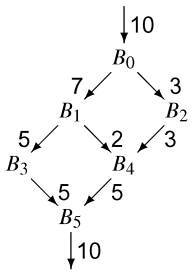
If the compiler understands the relative execution frequencies of the various parts of the program, it can use that information to improve the program's performance. Profile data can play an important role in optimizations such as global code placement (Section 8.6.2) or inline substitution (Section 8.7.1). Several approaches are used to gather profile data.

- *Instrumented executables* In this scheme, the compiler generates code to count specific events, such as procedure entries and exits or taken branches. At runtime, the data is written to an external file and processed offline by another tool.
- *Timer interrupts* Tools that use this approach interrupt program execution at frequent, regular intervals. The tool constructs a histogram of program counter locations where the interrupts occurred. Post-processing constructs a profile from the histogram.
- *Performance counters* Many processors offer some form of hardware counters to record hardware events, such as total cycles, cache misses, or taken branches. If counters are available, the runtime system can use them to construct highly accurate profile-like data.

These approaches produce somewhat different information and have distinct costs. An instrumented executable can measure almost any property of the execution; careful engineering can limit the overhead costs. A timer-interrupt system has lower overhead, but only pinpoints frequently executed statements (not the paths taken to reach them). Hardware counters are accurate and efficient, but depend in idiosyncratic ways on the specific processor architecture and implementation.

All of these approaches have proven successful at focusing optimization. Each of them requires cooperation between the compiler and the profiling tool on issues such as data formats, code layout, and methods for mapping runtime locations to program-based names.

Obtaining Profile Data



Example CFG

For global code placement, the compiler needs estimates of the relative execution frequency of each edge in the CFG. It can obtain that information from a profiling run of the code: compile the entire program, run it under a profiling tool on representative data, and give the compiler access to the resulting profile data. It can obtain that information from a model of program execution; such models range from simple to elaborate, with a range of accuracies.

Specifically, the compiler needs execution counts for the CFG edges. The CFG in the margin illustrates why edge counts are superior to block counts

for code placement. From the execution counts, shown as labels on the edges, we see that blocks B_0 and B_5 each execute ten times. The path (B_0, B_1, B_3, B_5) executes more than any other path in this CFG fragment. The edge counts suggest, for example, that making the branch (B_1, B_3) the fall-through case is better than making it the taken case. Relying on execution counts for blocks, however, the compiler would deduce that blocks B_3 and B_4 are of equal importance; it might well choose the less important edge, (B_1, B_4), as the fall-through case. The code-placement algorithm uses profile data to rank the CFG edges by frequency of execution. Thus, accurate edge data has a direct effect on the quality of the results.

Constructing Chains as Hot Paths in the CFG

To determine how it should lay out the code, the compiler constructs a set of CFG paths that include the most frequently executed edges—so-called *hot paths*. Each path is a chain of one or more blocks. Each path has a priority that will be used to construct the final code layout.

The compiler can use a greedy algorithm to find hot paths. Figure 8.16 shows one such algorithm. To begin, it creates a degenerate chain from each block that contains exactly that block. It sets the priority for each degenerate chain to a large number, such as the number of edges in the CFG or the largest available integer.

Next, the algorithm iterates over the edges in the CFG and builds up chains that model the hot paths. It takes the edges in order of execution frequency, with the most heavily used edges first. For an edge, $\langle x,y \rangle$, the algorithm merges the chain containing x with the chain containing y if and only if x is the last node in its chain and y is the first node in its chain. If either condition is not true, it leaves the chains that contain x and y alone.

The algorithm ignores self loops, $\langle x,x \rangle$, because they do not affect placement decisions.

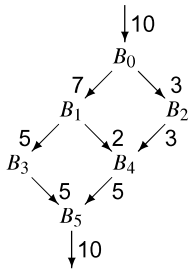
```
E ← |edges|
for each block b
   make a degenerate chain, d, for b
   priority(d) ← E
P ← 0
for each CFG edge ⟨x,y⟩, x ≠ y, in decreasing frequency order
   if x is the tail of chain a and y is the head of chain b then
      t ← priority(a)
      append b onto a
      priority(a) ← min(t, priority(b), P++)
```

■ **FIGURE 8.16** Building Hot Paths.

If it merges the chains for x and y, the algorithm must assign the new chain an appropriate priority. It computes that priority as the minimum of the priorities of the chains for x and y. If both x and y are degenerate chains with their initial high priority, it sets the priority of the new chain to the ordinal number of merges that the algorithm has considered, denoted as P. This value places the chain behind chains constructed from higher-frequency edges and ahead of those constructed from lower-frequency edges.

Forward branch
A branch whose target has a higher address than its source is called a *forward branch*. In some architectures, forward branches are less disruptive than backward branches.

The algorithm halts after it examines every edge. It produces a set of chains that model the hot paths in the CFG. Each node belongs to exactly one chain. Edges in chains execute more often than edges that cross from one chain to another. The priority values of each chain encode an order for relative layout of the chains that approximates the maximal number of executed forward branches.



Example CFG

To illustrate the algorithm's operation, consider its behavior when applied to the example CFG from the previous section, repeated in the margin. The algorithm proceeds as follows:

Edge	Set of Chains	P
—	$(B_0)_E$, $(B_1)_E$, $(B_2)_E$, $(B_3)_E$, $(B_4)_E$, $(B_5)_E$	0
(B_0, B_1)	$(B_0, B_1)_0$, $(B_2)_E$, $(B_3)_E$, $(B_4)_E$, $(B_5)_E$	1
(B_3, B_5)	$(B_0, B_1)_0$, $(B_2)_E$, $(B_3, B_5)_1$, $(B_4)_E$	2
(B_4, B_5)	$(B_0, B_1)_0$, $(B_2)_E$, $(B_3, B_5)_1$, $(B_4)_E$	2
(B_1, B_3)	$(B_0, B_1, B_3, B_5)_0$, $(B_2)_E$, $(B_4)_E$	3
(B_0, B_2)	$(B_0, B, B_3, B_5)_0$, $(B_2)_E$, $(B_4)_E$	3
(B_2, B_4)	$(B_0, B_1, B_3, B_5)_0$, $(B_2, B_4)_3$	4
(B_1, B_4)	$(B_0, B_1, B_3, B_5)_0$, $(B_2, B_4)_3$	4

Priorities are shown as subscripts on the chain and E is the number of edges in the CFG, as in Figure 8.16.

Breaking ties among equal-priority edges in a different way can produce a different set of chains. For example, if the algorithm considers (B_4, B_5) before (B_3, B_5), then it produces two chains: $(B_0, B_1, B_3)_0$ and $(B_2, B_4, B_5)_1$. Different chains may produce different code layouts. The layout algorithm still produces good results, even with a nonoptimal ordering for the equal-weight edges.

Performing Code Layout

The set of chains produced by the algorithm in Figure 8.16 constitutes a partial order on the set of basic blocks. To produce an executable image of

```
t ← chain headed by the CFG entry node
WorkList ← {(t, priority(t))}
while (Worklist ≠ ∅)
   remove a chain c of lowest priority from WorkList
   for each block x in c in chain order
      place x at the end of the executable code
   for each block x in c
      for each edge ⟨x, y⟩ where y is unplaced
         t ← chain containing ⟨x, y⟩
         if (t, priority(t)) ∉ WorkList
            then WorkList ← WorkList ∪ { (t, priority(t))}
```

■ **FIGURE 8.17** Code-Layout Algorithm.

the code, the compiler must place all of the blocks into a fixed linear order. Figure 8.17 shows an algorithm that computes a linear layout from the set of chains. It encodes two simple heuristics: (1) place the blocks of a chain in order, so that fall-through branches implement the chain's edges, and (2) chose among alternatives using the priority number recorded for the chains.

The algorithm represents a chain with a pair (c, p) where c is the chain's name and p is its priority. For the sake of efficiency, the test that avoids placing a chain on the worklist twice can be eliminated if we implement the worklist with a sparse set (see Appendix B.2.3). The following table shows the algorithm's behavior on the first set of chains produced for the example CFG:

Step	WorkList	Code Layout
—	$(B_0, B_1, B_3, B_5)_0$	
1	$(B_2, B_4)_3$	B_0, B_1, B_3, B_5
2	$\emptyset$	$B_0, B_1, B_3, B_5, B_2, B_4$

The first line shows the initial state. It puts the chain that contains B_0 on the worklist. The first iteration of the while loop places all the blocks in that chain. As it processes the edges leaving the placed blocks, it adds the other chain, (B_2, B_4) on the worklist. The second iteration places those two blocks; it adds nothing to the worklist, so the algorithm halts.

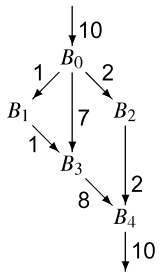
We noted that a change in tie breaking could produce a change in the set of chains produced for the example. Taking the edge (B_4, B_5) before (B_3, B_5)

produced the chains $(B_0, B_1, B_3)_0$ and $(B_2, B_4, B_5)_1$. Working from those chains, the code-layout algorithm behaves as follows:

Step	WorkList	Code Layout
—	$(B_0, B_1, B_3)_0$	
1	$(B_2, B_4, B_5)_1$	B_0, B_1, B_3
2		$B_0, B_1, B_3, B_2, B_4, B_5$

If we assume that the estimated execution frequences are correct, there is no reason to prefer one layout over the other.

A Final Example



Final Example

Consider how the global code-placement algorithm treats the CFG shown in the margin. The chain-construction algorithm proceeds as follows:

Edge	Set of Chains	P
—	$(B_0)_E$, $(B_1)_E$, $(B_2)_E$, $(B_3)_E$, $(B_4)_E$	0
(B_3, B_4)	$(B_0)_E$, $(B_1)_E$, $(B_2)_E$, $(B_3, B_4)_0$	1
(B_0, B_3)	$(B_0, B_3, B_4)_0$, $(B_1)_E$, $(B_2)_E$	2
(B_2, B_4)	$(B_0, B_3, B_4)_0$, $(B_1)_E$, $(B_2)_E$	2
(B_0, B_2)	$(B_0, B_3, B_4)_0$, $(B_1)_E$, $(B_2)_E$	2
(B_1, B_3)	$(B_0, B_3, B_4)_0$, $(B_1)_E$, $(B_2)_E$	2
(B_0, B_1)	$(B_0, B_3, B_4)_0$, $(B_1)_E$, $(B_2)_E$	2

On this graph, the algorithm halts with one multinode chain and two degenerate chains, both of which have their initial high priority.

The layout algorithm first places (B_0, B_3, B_4). When it examines the outbound edges from the placed nodes, it adds both of the degenerate blocks to the worklist. The next two iterations remove the degenerate blocks, in arbitrary order, and place them. There is no reason to prefer one order over the other.

SECTION REVIEW

Optimizations that examine an entire procedure have opportunities for improvement that are not available at smaller scopes. Because the global, or procedure-wide, scope includes cyclic paths and backward branches, global optimizations usually need global analysis. As a consequence, these algorithms have an offline flavor; they consist of an analysis phase followed by a transformation phase.

This section highlighted two distinct kinds of analysis: global data-flow analysis and runtime collection of profile data. Data-flow analysis is a

compile-time technique that accounts, mathematically, for the effects along all possible paths through the code. In contrast, profile data records what actually happened on a single run of the code, with a single set of input data. Data-flow analysis is conservative, in that it accounts for all possibilities. Runtime profiling is aggressive, in that it assumes that future runs will share runtime characteristics with the profiling run. Both can play an important role in optimization.

Review Questions

1. In some situations, the compiler needs to know that a variable is live along *all* paths that leave a block, rather than live along *some* path. Reformulate the equations for LiveOut so that they compute the set of names that are used before definition along every path from the end of the block to the CFG's exit node, n_f.
2. To collect accurate edge-count profiles, the compiler can instrument each edge in the profiled procedure's CFG. A clever implementation can instrument a subset of those edges and deduce the counts for the rest. Devise a scheme that derives accurate edge-count data without instrumenting each branch. On what principles does your scheme rely?

8.7 INTERPROCEDURAL OPTIMIZATION

As discussed in Chapter 6, procedure calls form boundaries in software systems. The division of a program into multiple procedures has both positive and negative impacts on the compiler's ability to generate efficient code. On the positive side, it limits the amount of code that the compiler considers at any one time. This effect keeps compile-time data structures small and limits the cost of various compile-time algorithms by limiting the problem sizes.

On the negative side, the division of the program into procedures limits the compiler's ability to understand what happens inside a call. For example, consider a call from `fee` to `fie` that passes a variable x as a call-by-reference parameter. If the compiler knows that x has the value 15 before the call, it cannot use that fact after the call, unless it knows that the call cannot change x. To use the value of x after the call, the compiler must prove that the formal parameter corresponding to x is not modified by `fie` or any procedure that it calls, directly or indirectly.

A second major source of inefficiency introduced by procedure calls arises from the fact that each call entails executing a precall and a postreturn sequence in the caller and a prolog and an epilog sequence in the callee. The operations implemented in these sequences take time. The transitions

between these sequences require (potentially disruptive) jumps. These operations are all overhead needed in the general case to implement the abstractions of the source language. At any specific call, however, the compiler may be able to tailor the sequences or the callee to the local runtime environment and achieve better performance.

These effects, on compile-time knowledge and on runtime actions, can introduce inefficiencies that intraprocedural optimization cannot address. To reduce the inefficiencies introduced by separate procedures, the compiler may analyze and transform multiple procedures together, using interprocedural analysis and optimization. These techniques are equally important in Algol-like languages and in object-oriented languages.

The term "whole program" clearly implies analyzing all the code. We prefer the term "interprocedural" when we talk about analyzing some, but not all, of the procedures.

In this section, we will examine two different interprocedural optimizations: inline substitution of procedure calls and procedure placement for improved code locality. Because whole-program optimization requires that the compiler have access to the code being analyzed and transformed, the decision to perform whole-program optimization has implications for compiler structure. Thus, the final subsection discusses the structural issues that arise in a system that includes interprocedural analysis and optimization.

8.7.1 Inline Substitution

As we saw in Chapters 6 and 7, the code that the compiler must generate to implement a procedure call involves a significant number of operations. The code must allocate an activation record, evaluate each actual parameter, preserve the caller's state, create the callee's environment, transfer control from caller to callee and back, and, if necessary, return values from callee to caller. In a sense, these runtime actions are part of the overhead of using a programming language; they maintain programming-language abstractions but are not strictly necessary to compute the results. Optimizing compilers try to reduce the cost of such overheads.

Inline substitution
a transformation that replaces a call site with a copy of the callee's body, rewritten to reflect parameter bindings

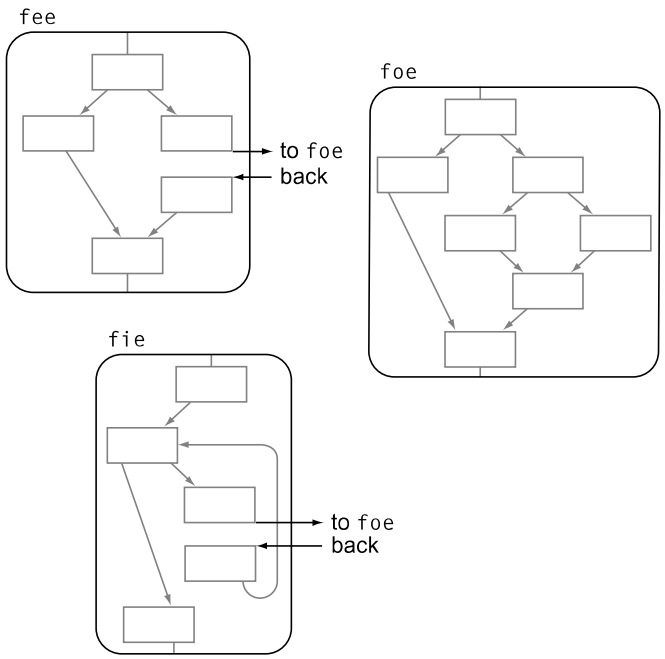
In some cases, the compiler can improve the efficiency of the final code by replacing the call site with a copy of the callee's body, appropriately tailored to the specific call site. This transformation, called *inline substitution*, allows the compiler to avoid most of the procedure linkage code and to tailor the new copy of the callee's body to the caller's context. Because the transformation moves code from one procedure to another and alters the program's call graph, inline substitution is considered an interprocedural transformation.

As with many optimizations, inline substitution has a natural partition into two subproblems: the actual transformation and a decision procedure that chooses call sites to inline. The transformation itself is relatively simple. The decision procedure is more complex and has a direct impact on performance.

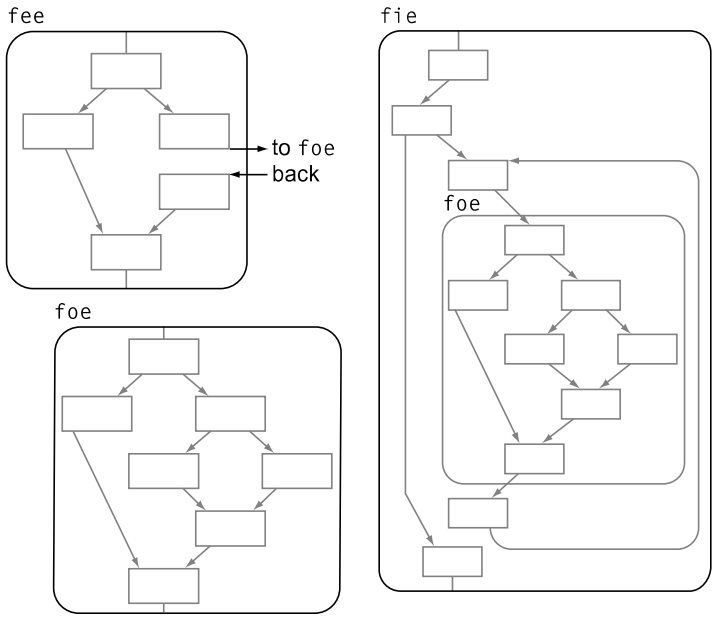
The Transformation

To perform inline substitution, the compiler rewrites a call site with the body of the callee, while making appropriate modifications to model the effects of parameter binding. Figure 8.18 shows two procedures, `fee` and `fie`, both of which call a third procedure, `foe`. Figure 8.19 depicts the control flow after inlining the call from `fie` to `foe`. The compiler has created a copy of `foe` and moved it inside `fie`, connected `fie`'s precall sequence directly to the prolog of its internal copy of `foe` and connected the epilog to the postcall sequence in a similar fashion. Some of the resulting blocks can be merged, enabling improvement with subsequent optimization.

Of course, the compiler must use an IR that can represent the inlined procedure. Some source-language constructs can create arbitrary and unusual control-flow constructs in the resulting code. For example, a callee with multiple premature returns may generate a complex control-flow graph. Similarly, FORTRAN's *alternate return* construct allows the caller to pass labels into the callee; the callee can then cause control to return to any of those labels. In either case, the resulting control-flow graph may be hard to represent in a near-source AST.



■ **FIGURE 8.18** Before Inline Substitution.



■ FIGURE 8.19 After Inline Substitution.

In the implementation, the compiler writer should pay attention to the proliferation of local variables. A simple implementation would create one new local variable in the caller for each local variable in the callee. If the compiler inlines several procedures, or several call sites to the same callee, the local name space can grow quite large. While growth in the name space is not a correctness issue, it can increase the cost of compiling the transformed code and, in some cases, it can hurt performance in the final code. Attention to this detail can easily avoid the problem by reusing names across multiple inlined callees.

The Decision Procedure

Choosing which call sites to inline is a complex task. Inlining a given call site can improve performance; unfortunately, it can also degrade performance. To make intelligent choices, the compiler must consider a broad range of characteristics of the caller, the callee, and the call site. The compiler must also understand its own strengths and weaknesses.

The primary sources of improvement from inlining are direct elimination of operations and improved effectiveness of other optimizations. The former effect can occur when parts of the linkage sequence can be eliminated;

for example, register save and restore code might be eliminated in favor of allowing the register allocator make those decisions. Knowledge from the caller may prove other code inside the callee dead or useless as well. The latter effect arises from having more contextual information in global optimization.

The primary source of degradation from inline substitution is decreased effectiveness of code optimization on the resulting code. Inlining the callee can increase code size and the name space size. It can increase demand for registers in the neighborhood of the original call site. Eliminating the register save and restore code changes the problem seen by the register allocator. In practice, any of these can lead to a decrease in optimization effectiveness.

Changes in architecture, such as larger register sets, can increase the cost of a procedure call. That change can, in turn, make inlining more attractive.

At each call site, the compiler must decide whether or not to inline the call. To complicate matters, a decision made at one call site affects the decision at other call sites. For example, if *a* calls *b* which calls *c*, choosing to inline *c* into *b* changes both the characteristics of the procedure that might be inlined into *a* and the call graph of the underlying program. Furthermore, inlining has effects, such as code size growth, that must be viewed across the whole program; the compiler writer may want to limit the overall growth in code size.

Decision procedures for inline substitution examine a variety of criteria at each call site. These include:

- *Callee size* If the callee is smaller than the procedure linkage code (pre-call, post-return, prolog, and epilog), then inlining the callee should reduce code size and execute fewer operations. This situation arises surprisingly often.
- *Caller size* The compiler may limit the overall size of any procedure to mitigate increases in compile time and decreases in optimization effectiveness.
- *Dynamic call count* An improvement at a frequently executed call site provides greater benefit than the same improvement at an infrequently executed call site. In practice, compilers use either profile data or simple estimates, such as 10 times the loop nesting depth.
- *Constant-valued actual parameters* The use of actual parameters that have known-constant values at a call site creates the potential for improvement as those constants are folded into the body of the callee.
- *Static call count* Compilers often track the number of distinct sites that call a procedure. Any procedure called from just one call site can be

```
Inline any call site that matches one of the following:
(1)  The callee uses more than t0 percent of execution time, and
     (a) the callee contains no calls, or
     (b) the static call count is one, or
     (c) the call site has more than t1 constant-valued parameters.
(2) The call site represents more than t2 percent of all calls, and
     (a) the callee is smaller than t3 , or
     (b)  inlining the call will produce a procedure smaller than t4
```

■ **FIGURE 8.20** A Typical Decision Heuristic for Inline Substitution.

inlined without any code space growth. The compiler should update this metric as it inlines, to detect procedures that it reduces to one call site.

- *Parameter count* The number of parameters can serve as a proxy for the cost of the procedure linkage, as the compiler must generate code to evaluate and store each actual parameter.
- *Calls in the procedure* Tracking the number of calls in a procedure provides an easy way to detect leaves in the call graph—they contain no calls. Leaf procedures are often good candidates for inlining.
- *Loop nesting depth* Call sites in loops execute more frequently than call sites outside loops. They also disrupt the compiler's ability to schedule the loop as a single unit (see Section 12.4).
- *Fraction of execution time* Computing the fraction of execution time spent in each procedure from profile data can prevent the compiler from inlining routines that cannot have a significant impact on performance.

In practice, compilers precompute some or all of these metrics and then apply a heuristic or set of heuristics to determine which call sites to inline. Figure 8.20 shows a typical heuristic. It relies on a series of threshold parameters, named t_0 through t_4. The specific values chosen for the parameters will govern much of the heuristic's behavior; for example, t_3 should undoubtedly have a value greater than the size of the standard precall and postreturn sequences. The best settings for these parameters is undoubtedly program specific.

8.7.2 Procedure Placement

The global code placement technique from Section 8.6.2 rearranged blocks within a single procedure. An analogous problem exists on the interprocedural scale: rearranging procedures within an executable image.

Given the call graph for a program, annotated with either measured or estimated execution frequencies for each call site, rearrange the procedures to reduce virtual-memory working-set sizes and to limit the potential for call-induced conflicts in the instruction cache.

The principle is simple. If procedure p calls q, we would like p and q to occupy adjacent locations in memory.

To solve this problem, we can treat the call graph as a set of constraints on the relative placement of procedures in the executable code. Each call-graph edge, (p,q), specifies an adjacency that should occur in the executable code. Unfortunately, the compiler cannot satisfy all of those adjacencies. For example, if p calls q, r, and s, the compiler cannot place all three of them next to p. Thus, compilers that perform procedure placement tend to use a greedy approximate technique to find a good placement, rather than trying to compute an optimal placement.

Recall that a program's call graph has a node for each procedure and an edge (x,y) for each call from x to y.

Procedure placement differs subtly from the global code placement problem discussed in Section 8.6.2. That algorithm improves the code by ensuring that hot paths can be implemented with fall-through branches. Thus, the chain-construction algorithm in Figure 8.16 ignores any CFG edge unless it runs from the tail of one chain to the head of another. In contrast, as the procedure placement algorithm builds chains of procedures, it can use edges that run between procedures that lie in the middles of their chains because its goal is simply to place procedures near each other—to reduce working set sizes and to reduce interference in the instruction cache. If p calls q and the distance from p to q is less than the size of the instruction cache, placement succeeds. Thus, in some sense, the procedure placement algorithm has more freedom than the block-layout algorithm.

Procedure placement consists of two phases: analysis and transformation. The analysis operates on the program's call graph. It repeatedly selects two nodes in the call graph and combines them. The order of combination is driven by execution frequency data, either measured or estimated. The order of combination determines the final layout. The layout phase is straightforward; it simply rearranges the code for the procedures into the order chosen by the analysis phase.

Figure 8.21 shows a greedy algorithm for the analysis phase of procedure placement. It operates over the program's call graph and iteratively constructs a placement by considering edges in order of their estimated execution frequency. As a first step, it builds the call graph, assigns each edge a weight that corresponds to its estimated execution frequency, and combines all the edges between two nodes into a single edge. As the final part of its

```
// Initialization work
build the call multi-graph G
initialize Q as a priority queue          // Order Q highest to lowest

for each edge (x,y) ∈ G                   // Add weights to the edges
   if (x = y)                             // Self loop is irrelevant
      then delete (x,y) from G
      else weight((x,y)) ← estimated execution frequency for (x,y)

for each node x ∈ G
   list(x) ← {x}                          // Initialize placement lists
   if multiple edges exist from x to y
      then combine them and their weights
   for each edge (x,z) ∈ G                // Put each edge into Q
      Enqueue(Q,(x,z),weight((x,z))

// Iterative reduction of the graph
while Q is not empty
   (x,y) ← Dequeue(Q)                     // Take highest priority edge
   for each edge (y,z) ∈ G                // Move source from y to x
      ReSource((y,z),x)
   for each edge (z,y) ∈ G                // Move target from y to x
      ReTarget((z,y),x)
   append list(y) to list(x)              // Update the placement list
   delete y and its edges from G          // Clean up G
```

■ **FIGURE 8.21** Procedure Placement Algorithm.

initialization work, it builds a priority queue of the call-graph edges, ordered by their weights.

The second half of the algorithm iteratively builds up an order for procedure placement. The algorithm associates with each node in the graph an ordered list of procedures. These lists specify a linear order among the named procedures. When the algorithm halts, the lists will specify a total order on the procedures that can be used to place them in the executable code.

The algorithm uses the call-graph edge weights to guide the process. It repeatedly selects the highest-weight edge, say (x,y), from the priority queue and combines its source x and its sink y. Next, it must update the call graph to reflect the change.

1. For each edge (y, z), it calls `ReSource` to replace (y, z) with (x, z) and to update the priority queue. If (x, z) already exists, `ReSource` combines them.

2. For each edge (z, y), it calls `ReTarget` to replace (z, y) with (z, x) and to update the priority queue. If (z, x) already exists, `ReTarget` combines them.

To affect the placement of y after x, the algorithm appends `list`(y) to `list`(x). Finally, it deletes y and its edges from the call graph.

The algorithm halts when the priority queue is empty. The final graph will have one node for each of the connected components of the original call graph. If all nodes were reachable from the node that represents the program's entry, the final graph will consist of a single node. If some procedures were not reachable, either because no path exists in the program that calls them or because those paths are obscured by ambiguous calls, then the final graph will consist of multiple nodes. Either way, the compiler and linker can use the lists associated with nodes in the final graph to specify the relative placement of procedures.

Example

To see how the procedure placement algorithm works, consider the example call graph shown in panel 0 of Figure 8.22. The edge from P_5 to itself is shown in gray because it only affects the algorithm by changing the execution frequencies. A self loop cannot affect placement since its source and sink are identical.

Panel 0 shows the state of the algorithm immediately before the iterative reduction begins. Each node has the trivial list that contains its own name. The priority queue has every edge, except the self loop, ranked by execution frequency.

Panel 1 shows the state of the algorithm after the first iteration of the while loop. The algorithm collapsed P_6 into P_5, and updated both the list for P_5 and the priority queue.

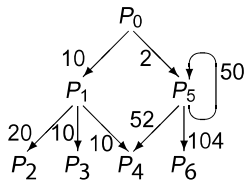
In panel 2, the algorithm has collapsed P_4 into P_5. It retargeted (P_1, P_4) onto P_5 and changed the corresponding edge name in the priority queue. In addition, it removed P_4 from the graph and updated the list for P_5.

The other iterations proceed in a similar fashion. Panel 4 shows a situation where it combined edges. When it collapsed P_5 into P_1, it retargeted (P_0, P_5) onto P_1. Since (P_0, P_1) already existed, it simply combined their weights and updated the priority queue by deleting (P_0, P_5) and changing the weight on (P_0, P_1).

At the end of the iterations, the graph has been collapsed to a single node, P_0. While this example constructed a layout that begins with the entry node, that happened because of the edge weights rather than by algorithmic design.

Initial State

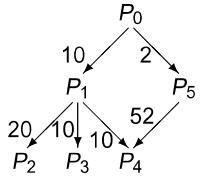
0.



	P_0	P_1	P_2	P_3	P_4	P_5	P_6
List(x)	{ P_0 }	{ P_1 }	{ P_2 }	{ P_3 }	{ P_4 }	{ P_5 }	{ P_6 }

Q: $\{(P_5, P_6)_{104},\ (P_5, P_4)_{52},\ (P_1, P_2)_{20},\ (P_1, P_4)_{10},\ (P_1, P_3)_{10},\ (P_0, P_1)_{10},\ (P_0, P_5)_2\ \}$

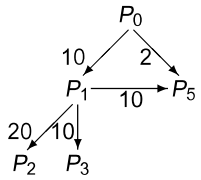
1.



	P_0	P_1	P_2	P_3	P_4	P_5	
List(x)	{ P_0 }	{ P_1 }	{ P_2 }	{ P_3 }	{ P_4 }	{ P_5, P_6 }	

Q: $\{(P_5, P_4)_{52},\ (P_1, P_2)_{20},\ (P_1, P_4)_{10},\ (P_1, P_3)_{10},\ (P_0, P_1)_{10},\ (P_0, P_5)_2\ \}$

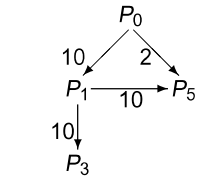
2.



	P_0	P_1	P_2	P_3	P_5	
List(x)	{ P_0 }	{ P_1 }	{ P_2 }	{ P_3 }	{ P_5, P_6, P_4 }	

Q: $\{(P_1, P_2)_{20},\ (P_1, P_5)_{10},\ (P_1, P_3)_{10},\ (P_0, P_1)_{10},\ (P_0, P_5)_2\ \}$

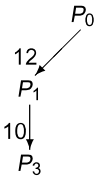
3.



	P_0	P_1	P_3	P_5	
List(x)	{ P_0 }	{ P_1, P_2 }	{ P_3 }	{ P_5, P_6, P_4 }	

Q: $\{(P_1, P_5)_{10},\ (P_1, P_3)_{10},\ (P_0, P_1)_{10},\ (P_0, P_5)_2\ \}$

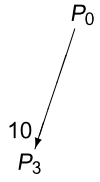
4.



	P_0	P_1	P_3	
List(x)	{ P_0 }	{ P_1, P_2, P_5, P_6, P_4 }	{ P_3 }	

Q: $\{(P_0, P_1)_{12},\ (P_1, P_3)_{10}\ \}$

5.



	P_0	P_3
List(x)	{ $P_0, P_1, P_2, P_5, P_6, P_4$ }	{ P_3 }

Q: $\{(P_1, P_3)_{10}\ \}$

6.

	P_0	
List(x)	{ $P_0, P_1, P_2, P_5, P_6, P_4, P_3$ }	

Q: $\emptyset$

■ **FIGURE 8.22** Steps of the Procedure Placement Algorithm.

8.7.3 Compiler Organization for Interprocedural Optimization

Building a compiler that performs analysis and optimization across two or more procedures fundamentally changes the relationship between the compiler and the code that it produces. Traditional compilers have compilation units of a single procedure, a single class, or a single file of code; the resulting code depends solely on the contents of that compilation unit. Once the compiler uses knowledge about one procedure to optimize another, the correctness of the resulting code depends on the state of both procedures.

Compilation unit
The portion of a program presented to the compiler is often called a *compilation unit*.

Consider the impact of inline substitution on the validity of the optimized code. Assume that the compiler inlines `fie` into `fee`. Any subsequent editing change to `fie` will necessitate recompilation of `fee`—a dependence that results from an optimization decision rather than from any relationship exposed in the source code.

If the compiler collects and uses interprocedural information, similar problems can arise. For example, `fee` may call `fie`, which calls `foe`; assume that the compiler relies on the fact that the call to `fie` does not change the known constant value of the global variable x. If the programmer subsequently edits `foe` so that it modifies x, that change can invalidate the prior compilation of both `fee` and `fie`, by changing the facts upon which optimization relies. Thus, a change to `foe` can necessitate a recompilation of other procedures in the program.

To address this fundamental issue, and to provide the compiler with access to all the source code that it needs, several different structures have been proposed for compilers that perform whole-program or interprocedural optimization: enlarging the compilation units, embedding the compiler in an integrated development environment, and performing the optimization at link time.

- *Enlarging Compilation Units* The simplest solution to the practical problems introduced by interprocedural optimization is to enlarge the compilation units. If the compiler only considers optimization and analysis within a compilation unit, and those units are consistently applied, then it can sidestep the problems. It can only analyze and optimize code that is compiled together; thus, it cannot introduce dependences between compilation units and it should not require access to either source code or facts about other units. The IBM PL/I optimizing compiler took this approach; code quality improved as related procedures were grouped together in the same file.

  Of course, this approach limits the opportunities for interprocedural optimization. It also encourages the programmer to create larger

compilation units and to group together procedures that call one another. Both of these may introduce practical problems in a system with multiple programmers. Still, as a practical matter, this organization is attractive because it least disturbs our model of the compiler's behavior.

- *Integrated Development Environments* If the design embeds the compiler inside an integrated development environment (IDE), the compiler can access code as needed through the IDE. The IDE can notify the compiler when source code changes, so that the compiler can determine if recompilation is needed. This model shifts ownership of both the source code and the compiled code from the developer to the IDE. Collaboration between the IDE and the compiler then ensures that appropriate actions are taken to guarantee consistent and correct optimization.
- *Link-time Optimization* The compiler writer can shift interprocedural optimization into the linker, where it will have access to all of the *statically* linked code. To obtain the benefits of interprocedural optimization, the linker may also need to perform subsequent global optimization. Since the results of link-time optimization are only recorded in the executable, and that executable is discarded on the next compilation, this strategy sidesteps the recompilation problem. It almost certainly performs more analysis and optimization that the other approaches, but it offers both simplicity and obvious correctness.

Many modern systems make use of runtime, or *dynamic*, linking for shared libraries. Runtime linking limits the opportunities for link-time optimization.

SECTION REVIEW

Analysis and optimization across procedure boundaries can reveal new opportunities for code improvement. Examples include tailoring the procedure linkage (precall, prolog, epilog, and postcall sequences) to a specific call site through exposing constant values or redundant values across a call. Many techniques have been proposed to recognize and exploit these opportunities; inline substitution is one of the best known and broadly effective of these techniques.

A compiler that applies interprocedural analysis and optimization must take care to ensure that the executables it builds are based on a consistent view of the entire program. Using facts from one procedure to modify the code in another can introduce subtle dependences between the code in distant procedures, dependences that the compiler must recognize and respect. Several strategies have been proposed to mitigate these effects; perhaps the simplest is to perform interprocedural transformations at link time.

Review Questions

1. Suppose procedure *a* invokes *b* and *c*. If the compiler inlines the call to *b*, what code space and data space savings might arise? If it inlines *c* as well, are further data-space savings possible?
2. In procedure placement, what happens to a procedure whose incoming edges all have estimated execution frequencies of zero? Where should the algorithm place such a procedure? Does the treatment of such a procedure affect execution time performance? Can the compiler eliminate them as useless?

8.8 SUMMARY AND PERSPECTIVE

The optimizer in a modern compiler contains a collection of techniques that try to improve the performance of the compiled code. While most optimizations try to improve runtime speed, optimizations can also target other measures, such as code size or energy consumption. This chapter has shown a variety of techniques that operate over scopes that range from single basic blocks through entire programs.

Optimizations improve performance by tailoring general translation schemes to the specific details of the code at hand. The transformations in an optimizer try to remove the overhead introduced in support of source-language abstractions, including data structures, control structures, and error checking. They try to recognize special cases that have efficient implementations and rewrite the code to realize those savings. They try to match the resource needs of the program against the actual resources available on the target processor, including functional units, the capacity and bandwidth of each level in the memory hierarchy (registers, cache, translation lookaside buffers, and memory), and instruction-level parallelism.

Before the optimizer can apply a transformation, it must determine that the proposed rewrite of the code is safe—that it preserves the code's original meaning. Typically, this requires that the optimizer analyze the code. In this chapter, we saw a number of approaches to proving safety, ranging from the bottom-up construction of the value table in local value numbering through computing LIVEOUT sets to detect uninitialized variables.

Once the optimizer has determined that it can safely apply a transformation, it must decide whether or not the rewrite will improve the code. Some techniques, such a local value numbering, simply assume that the rewrites

they use are profitable. Other techniques, such as inline substitution, require complicated decision procedures to determine when a transformation might improve the code.

This chapter provided a basic introduction to the field of compiler-based code optimization. It introduced many of the terms and issues that arise in optimization. It does not include an "Advanced Topics" section; instead, the interested reader will find additional material on static analysis in support of optimization in Chapter 9 and on optimizing transformations in Chapter 10.

CHAPTER NOTES

The field of code optimization has a long and detailed literature. For a deeper treatment, the reader should consider some of the specialized books on the subject [20, 268, 270]. It would be intellectually pleasing if code optimization had developed in a logical and disciplined way, beginning with local techniques, extending them first to regions, then entire procedures, and finally entire programs. As it happened, however, development has occurred in a more haphazard fashion. For example, the original Fortran compiler [27] performed both local and global optimization—the former on expression trees and the latter for register allocation. Interest in both regional techniques, such as loop optimization [252], and whole-program techniques, such as inline substitution, crops up early in the literature, as well [16].

Local value numbering, with its extensions for algebraic simplifications and constant folding, is usually credited to Balke in the late 1960s [16, 87], although it is clear that Ershov achieved similar effects in a much earlier system [139]. Similarly, Floyd mentioned the potential for both local redundancy elimination and application of commutativity [150]. The extension to EBBs in superlocal value numbering is natural and has, undoubtedly, been invented and reinvented in many compilers. Our treatment derives from Simpson [53].

The tree-height balancing algorithm is due to Hunt [200]; it uses a rank function inspired by Huffman codes, but is easily adapted to other metrics. The classic algorithm for balancing instruction trees is due to Baer and Bovet [29]. The entire issue of finding and exploiting instruction-level parallelism is intimately related to instruction scheduling (see Chapter 12).

Loop unrolling is the simplest of loop nest optimizations. It has a long history in the literature [16]. The use of unrolling to eliminate register-to-register copy operations as in review question 2 for Section 8.5 on

page 444 is due to Kennedy [214]. Unrolling can have subtle and surprising effects [108]. Selection of unroll factors has also been studied [114, 325].

The ideas that underlie live analysis have been around as long as compilers have been automatically allocating storage locations for values [242]. Beatty first defined live analysis in an internal IBM technical report [15]. Lowry and Medlock discuss "busy" variables [p. 16, 252] and the use of this information in both the elimination of dead code and in reasoning about interference (see Chapter 13). The analysis was formulated as a global data-flow analysis problem by 1971 [13, 213]. Live analysis will appear again in the construction of SSA form in Chapter 9 and in the discussion of register allocation in Chapter 13.

The code-placement algorithms, at both the global and whole-program scopes, are taken from Pettis and Hansen [284]. Subsequent work on this problem has focused on collecting better profile data and improving the placements [161, 183]. Later work includes work on branch alignment [66, 357] and code layout [78, 93, 161].

Inline substitution has been discussed in the literature for decades [16]. While the transformation is straightforward, its profitability has been the subject of many studies [31, 99, 119, 301].

Interprocedural analysis and optimization has been discussed in the literature for decades [18, 34, 322]. Inline substitution has a long history in the literature [16]. All of the scenarios mentioned in Section 8.7.3 have been explored in real systems [104, 322, 341]. Recompilation analysis is treated in depth by Burke and Torczon [64, 335]. See the notes for Chapter 9 for more references on interprocedural analysis.

EXERCISES

1. Apply the algorithm from Figure 8.4 to each of the following blocks: Section 8.4

Block b_0	Block b_1
$t_1 \leftarrow a + b$	$t_1 \leftarrow a \times b$
$t_2 \leftarrow t_1 + c$	$t_2 \leftarrow t_1 \times 2$
$t_3 \leftarrow t_2 + d$	$t_3 \leftarrow t_2 \times c$
$t_4 \leftarrow b + a$	$t_4 \leftarrow 7 + t_3$
$t_5 \leftarrow t_3 + e$	$t_5 \leftarrow t_4 + d$
$t_6 \leftarrow t_4 + f$	$t_6 \leftarrow t_5 + 3$
$t_7 \leftarrow a + b$	$t_7 \leftarrow t_4 + e$
$t_8 \leftarrow t_4 - t_7$	$t_8 \leftarrow t_6 + f$
$t_9 \leftarrow t_8 * t_6$	$t_9 \leftarrow t_1 + 6$

2. Consider a basic block, such as b_0 or b_1 in question Section 8.8 above. It has n operations, numbered 0 to $n-1$.

a. For a name x, USES(x) contains the index in b of each operation that uses x as an operand. Write an algorithm to compute the USES set for every name mentioned in block b. If $x \in$ LIVEOUT(b), then add two dummy entries ($> n$) to USES(x).

b. Apply your algorithm to blocks b_0 and b_1 above.

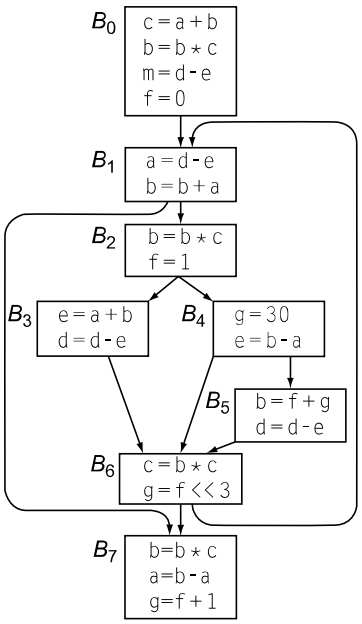
c. For a reference to x in operation i of block b, DEF(x,i) is the index in b where the value of x visible at operation i was defined. Write an algorithm to compute DEF(x,i) for each reference x in b. If x is upward exposed at i, then DEF(x,i) should be -1.

d. Apply your algorithm to blocks b_0 and b_1 above.

3. Apply the tree-height balancing algorithm from Figures 8.7 and 8.8 to the two blocks in problem 1. Use the information computed in problem 2b above. In addition, assume that LIVEOUT(b_0) is {t3, t9}, that LIVEOUT(b_1) is {t7, t8, t9}, and that the names a through f are upward-exposed in the blocks.

Section 8.5

4. Consider the following control-flow graph:



a. Find the extended basic blocks and list their distinct paths.
b. Apply local value numbering to each block.
c. Apply superlocal value numbering to the EBBs and note any improvements that it finds beyond those found by local value numbering.

5. Consider the following simple five-point stencil computation:

```
      do 20 i = 2, n-1, 1
         t1 = A(i,j-1)
         t2 = A(i,j)
         do 10 j = 2, m-1, 1
            t3 = A(i,j+1)
            A(i,j) = 0.2 × (t1 + t2 + t3 + A(i-1,j) + A(i+1,j))
            t1 = t2
            t2 = t3
10       continue
20    continue
```

Each iteration of the loop executes two copy operations.
a. Loop unrolling can eliminate the copy operations. What unroll factor is needed to eliminate all copy operations in this loop?
b. In general, if a loop contains multiple cycles of copy operations, how can you compute the unroll factor needed to eliminate all of the copy operations?

Section 8.6

6. At some point p, LIVE(p) is the set of names that are *live* at p. LIVEOUT(b) is just the LIVE set at the end of block b.
a. Develop an algorithm that takes as input a block b and its LIVEOUT set and produces as output the LIVE set for each operation in the block.
b. Apply your algorithm to blocks b_0 and b_1 in problem 1, using LIVEOUT(b_0) = {t_3, t_9} and LIVEOUT(b_1) = {t_7, t_8, t_9}.

7. Figure 8.16 shows an algorithm for constructing hot paths in the CFG.
a. Devise an alternate hot-path construction that pays attention to ties among equal-weight edges.
b. Construct two examples where your algorithm leads to a code layout that improves on the layout produced by the book's algorithm. Use the code-layout algorithm from Figure 8.17 with the chains constructed by your algorithm and those built by the book's algorithm.

Section 8.7

8. Consider the following code fragment. It shows a procedure `fee` and two call sites that invoke `fee`.

```
static int A[1000,1000], B[1000];
   ...
x = A[i,j] + y;
call fee(i,j,1000);
   ...
call fee(1,1,0);
   ...
fee(int row; int col; int ub) {
   int i, sum;
   sum = A[row,col];
   for (i=0; i<ub; i++) {
      sum = sum + B[i];
   }
}
```

a. What optimization benefits would you expect from inlining `fee` at each of the call sites? Estimate the fraction of `fee`'s code that would remain after inlining and subsequent optimization.

b. Based on your experience in part a, sketch a high-level algorithm for estimating the benefits of inlining a specific call site. Your technique should consider both the call site and the callee.

9. In Problem 8, features of the call site and its context determined the extent to which the optimizer could improve the inlined code. Sketch, at a high level, a procedure for estimating the improvements that might accrue from inlining a specific call site. (With such an estimator, the compiler could inline the call sites with the highest estimated profit, stopping when it reached some threshold on procedure size or total program size.)

10. When the procedure placement algorithm, shown in Figure 8.21, considers an edge $\langle p,q \rangle$ it always places p before q.

a. Formulate a modification of the algorithm that would consider placing the sink of an edge before its source.

b. Construct an example where this approach places two procedures closer together than the original algorithm. Assume that all procedures are of uniform size.