17 Code Improvement

In Chapter 15 we discussed the generation, assembly, and linking of target code in the middle and back end of a compiler. The techniques we presented led to correct but highly suboptimal code: there were many redundant computations, and inefficient use of the registers, multiple functional units, and cache of a modern microprocessor. This chapter takes a look at *code improvement*: the phases of compilation devoted to generating *good* code. For the most part we will interpret "good" to mean *fast*. In a few cases we will also consider program transformations that decrease memory requirements. On occasion a real compiler may try to minimize power consumption, dollar cost of execution on a commercial cloud server, or demand for some other resource; we will not consider these issues here.

There are several possible levels of "aggressiveness" in code improvement. In a very simple compiler, or in a "nonoptimizing" run of a more sophisticated compiler, we can use a *peephole optimizer* to peruse already-generated target code for obviously suboptimal sequences of adjacent instructions. At a slightly higher level, typical of the baseline behavior of production-quality compilers, we can generate near-optimal code for *basic blocks*. As described in Chapter 15, a basic block is a maximal-length sequence of instructions that will always execute in its entirety (assuming it executes at all). In the absence of delayed branches, each basic block in assembly language or machine code begins with the target of a branch or with the instruction after a conditional branch, and ends with a branch or with the instruction before the target of a branch. As a result, in the absence of hardware exceptions, control never enters a basic block except at the beginning, and never exits except at the end. Code improvement at the level of basic blocks is known as *local* optimization. It focuses on the elimination of redundant operations (e.g., unnecessary loads or common subexpression calculations), and on effective instruction scheduling and register allocation.

At higher levels of aggressiveness, production-quality compilers employ techniques that analyze entire subroutines for further speed improvements. These techniques are known as *global* optimization.[1] They include multi-basic-block versions of redundancy elimination, instruction scheduling, and register allocation,

plus code modifications designed to improve the performance of loops. Both global redundancy elimination and loop improvement typically employ a *control flow graph* representation of the program, as described in Section 15.1.1. Both employ a family of algorithms known as *data flow analysis* to trace the flow of information across the boundaries between basic blocks.

At the highest levels of aggressiveness, compilers may perform various forms of *interprocedural* code improvement. Interprocedural improvement is difficult for two main reasons. First, because a subroutine may be called from many different places in a program, it is difficult to identify (or fabricate) conditions (available registers, common subexpressions, etc.) that are guaranteed to hold at all call sites. Second, because many subroutines are separately compiled, an interprocedural code improver must generally subsume some of the work of the linker.

In the sections below we consider peephole, local, and global code improvement. We will not cover interprocedural improvement; interested readers are referred to other texts (see the Bibliographic Notes at the end of the chapter). Moreover, even for the subjects we cover, our intent will be more to "demystify" code improvement than to describe the process in detail. Much of the discussion (beginning in Section C-17.3) will revolve around the successive refinement of code for a single subroutine. This extended example will allow us to illustrate the effect of several key forms of code improvement without dwelling on the details of how they are achieved. Entire books continue to be written on code improvement; it remains a very active research topic.

As in most texts, we will sometimes refer to code improvement as "optimization," though this term is really a misnomer: we will seldom have any guarantee that our techniques will lead to optimal code. As it turns out, even some of the relatively simple aspects of code improvement (e.g., minimizing the number of registers needed in a basic block) can be shown to be NP-hard. True optimization is a realistic option only for small, special-purpose program fragments [Mas87]. Our discussion will focus on the improvement of code for imperative programs. Optimizations specific to functional or logic languages are beyond the scope of this book.

We begin in Section C-17.1 with a more detailed consideration of the phases of code improvement. We then turn to peephole optimization in Section C-17.2. It can be performed in the absence of other optimizations if desired, and the discussion introduces some useful terminology. In Sections C-17.3 and C-17.4 we consider local and global redundancy elimination. Sections C-17.5 and C-17.7 cover code improvement for loops. Section C-17.6 covers instruction scheduling. Section C-17.8 covers register allocation.

---

1 The adjective "global" is standard but somewhat misleading in this context, since the improvements do not consider the program as a whole; "subroutine-level" might be more accurate.

17.1 Phases of Code Improvement

As we noted in Chapter 15, the structure of the middle and back end varies considerably from compiler to compiler. For simplicity of presentation we will continue to focus on the structure introduced in Section 15.1. In that section (as in Section 1.6) we characterized machine-independent and machine-specific code improvement as individual phases of compilation, separated by target code generation. We must now acknowledge that this was an oversimplification. In reality, code improvement is a substantially more complicated process, often comprising a very large number of phases. As noted in Section C-15.2.1, `gcc` has more than 140 phases in its middle end, and 70 in the back end—far more than we can cover in this chapter. In some cases optimizations depend on one another, and must be performed in a particular order. In other cases they are independent, and can be performed in any order. In still other cases it can be important to *repeat* an optimization, in order to recognize new opportunities for improvement that were not visible until some other optimization was applied.

EXAMPLE 17.1
Code improvement phases

We will concentrate in our discussion on the forms of code improvement that tend to achieve the largest increases in execution speed, and are most widely used. Compiler phases to implement these improvements are shown in Figure C-17.1. Within this structure, the middle end begins with intermediate code generation. This phase identifies fragments of the syntax tree that correspond to basic blocks. It then creates a control flow graph in which each node contains a linear sequence of three-address instructions for an idealized machine, typically one with an unlimited supply of *virtual registers*. The (machine-specific) back end begins with target code generation. This phase strings the basic blocks together into a linear program, translating each block into the instruction set of the target machine and generating branch instructions that correspond to the arcs of the control flow graph.

Machine-independent code improvement in Figure C-17.1 is shown as three key phases. The first of these identifies and eliminates redundant loads, stores, and computations within each basic block. The second deals with similar redundancies across the boundaries between basic blocks (but within the bounds of a single subroutine). The third effects several improvements specific to loops; these are particularly important, since most programs spend most of their time in loops. In Sections C-17.4, C-17.5, and C-17.7, we shall see that global redundancy elimination and loop improvement may actually be subdivided into several separate phases.

We have shown machine-specific code improvement as four separate phases. The first and third of these are essentially identical. As we noted in Section C-5.5.2, register allocation and instruction scheduling tend to interfere with one another: the instruction schedules that do the best job of minimizing pipeline stalls tend to increase the demand for architectural registers (this demand is commonly known as *register pressure*). A common strategy, assumed in our discussion, is to schedule instructions first, then allocate architectural registers, then schedule instructions again. If it turns out that there aren't enough architectural registers to go around, the register allocator will generate additional load and store instructions to *spill*

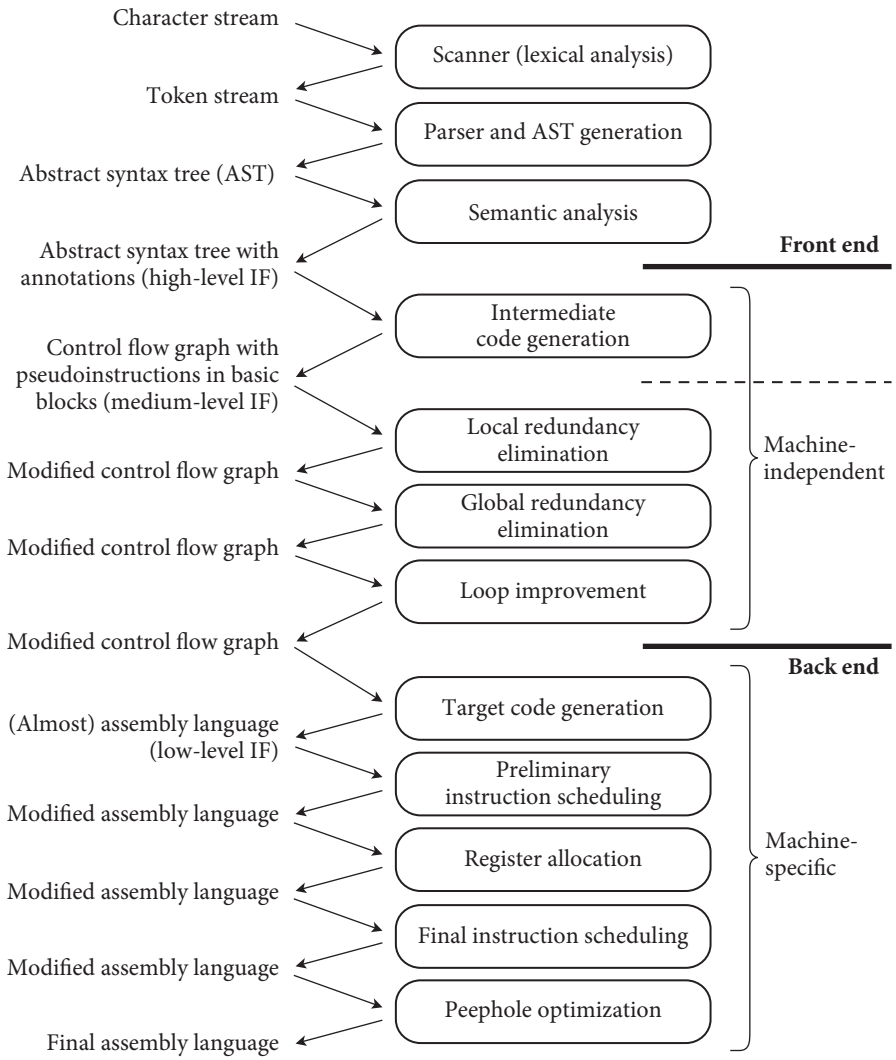


Figure 17.1 A more detailed view of the compiler structure originally presented in Figure 15.1. Both machine-independent and machine-specific code improvement have been divided into multiple phases. As before, the dashed line shows a common "break point" for a two-pass compiler. Machine-independent code improvement may sometimes be located in a separate "middle end" pass.

registers temporarily to memory. The second round of instruction scheduling attempts to fill any delays induced by the extra loads.

17.2 Peephole Optimization

In a simple compiler with no machine-independent code improvement, a code generator can simply walk the abstract syntax tree, producing naive code, either as output to a file or global list, or as annotations in the tree. As we saw in Chapters 1 and 15, however, the result is generally of very poor quality (contrast the code of Example 1.2 with that of Figure 1.7). Among other things, every use of a variable as an r-value results in a load, and every assignment results in a store.

A relatively simple way to significantly improve the quality of naive code is to run a *peephole optimizer* over the target code. A peephole optimizer works by sliding a several-instruction window (a peephole) over the target code, looking for suboptimal patterns of instructions. The set of patterns to look for is heuristic; generally one creates patterns to match common suboptimal idioms produced by a particular code generator, or to exploit special instructions available on a given machine. Here are a few examples:

EXAMPLE 17.2

Elimination of redundant loads and stores: The peephole optimizer can often recognize that the value produced by a load instruction is already available in a register. For example:

```
r2 := r1 + 5                    r2 := r1 + 5
i := r2          becomes        i := r2
r3 := i                         r3 := r2 × 3
r3 := r3 × 3
```

In a similar but less common vein, if there are two stores to the same location within the optimizer's peephole (with no possible intervening load from that location), then we can generally eliminate the first.

EXAMPLE 17.3

Constant folding: A naive code generator may produce code that performs calculations at run time that could actually be performed at compile time. A peephole optimizer can often recognize such code. For example:

```
r2 := 3 × 2        becomes        r2 := 6
```

EXAMPLE 17.4

Constant propagation: Sometimes we can tell that a variable will have a constant value at a particular point in a program. We can then replace occurrences of the variable with occurrences of the constant:

```
r2 := 4                         r2 := 4                          r3 := r1 + 4
r3 := r1 + r2      becomes      r3 := r1 + 4      and then       r2 := . . .
r2 := . . .                     r2 := . . .
```

The final assignment to r2 tells us that the previous value (the 4) in r2 was *dead*—it was never going to be needed. (By analogy, a value that may be needed

in some future computation is said to be *live*.) Loads of dead values can be eliminated. Similarly,

```
r2 := 4                    r3 := r1 + 4                    r3 := *(r1 +4)
r3 := r1 + r2    becomes   r3 := *r3        and then       r2 := . . .
r3 := *r3                  r2 := . . .
r2 := . . .
```

(This again leverages that fact that the 4 in r2 is dead at the final assignment.)

Often constant folding will reveal an opportunity for constant propagation. Sometimes the reverse occurs:

```
r1 := 3                    r1 := 3                    r1 := 3
r2 := r1 × 2     becomes   r2 := 3 × 2      and then   r2 := 6
```

If the 3 in r1 is dead, then the initial load can also be eliminated. ■

EXAMPLE 17.5

Common subexpression elimination: When the same calculation occurs twice within the peephole of the optimizer, we can often eliminate the second calculation:

```
r2 := r1 × 5               r4 := r1 × 5
r2 := r2 + r3    becomes   r2 := r4 + r3
r3 := r1 × 5               r3 := r4
```

Often, as shown here, an extra register will be needed to hold the common value. ■

EXAMPLE 17.6

Copy propagation: Even when we cannot tell that the contents of register *b* will be constant, we may sometimes be able to tell that register *b* will contain the same value as register *a*. We can then replace uses of *b* with uses of *a*, so long as neither *a* nor *b* is modified:

```
r2 := r1                   r2 := r1                   r3 := r1 + r1
r3 := r1 + r2    becomes   r3 := r1 + r1    and then   r2 := 5
r2 := 5                    r2 := 5
```

Performed early in code improvement, copy propagation can serve to decrease register pressure. In a peephole optimizer it may allow us (as in this case, in which the copy of r1 in r2 is dead) to eliminate one or more instructions. ■

EXAMPLE 17.7

Strength reduction: Numeric identities can sometimes be used to replace a comparatively expensive instruction with a cheaper one. In particular, multiplication or division by powers of two can be replaced with adds or shifts:

```
r1 := r2 × 2     becomes   r1 := r2 + r2      or   r1 := r2 << 1
r1 := r2 / 2     becomes   r1 := r2 >> 1
```

(This last replacement may not be correct when r2 is negative; see Exercise C-17.1.) In a similar vein, algebraic identities allow us to perform simplifications like the following:

```
r1 := r2 × 0     becomes   r1 := 0
```

■

EXAMPLE 17.8

Elimination of useless instructions: Instructions like the following can be dropped entirely:

```
r1 := r1 + 0
r1 := r1 × 1
```

Filling of load and branch delays: Several examples of delay-filling transformations were presented in Section C-5.5.1.

EXAMPLE 17.9

Exploitation of the instruction set: Particularly on CISC machines, sequences of simple instructions can often be replaced by a smaller number of more complex instructions. For example,

```
r1 := r1 & 0x0000FF00
r1 := r1 >> 8
```

can be replaced by an "extract byte" instruction. The sequence

```
r1 := r2 + 8
r3 := *r1
```

where r1 is dead at the end can be replaced by a single load of r3 using a base plus displacement addressing mode. Similarly,

```
r1 := *r2
r2 := r2 + 4
```

where *r2 is a 4-byte quantity can be replaced by a single load with an autoincrement addressing mode. On many machines, a series of loads from consecutive locations can be replaced by a single, multiple-register load.

Because they use a small, fixed-size window, peephole optimizers tend to be very fast: they impose a small, constant amount of overhead per instruction. They are also relatively easy to write and, when used on naive code, can yield dramatic performance improvements.

It should be emphasized, however, that most of the forms of code improvement in Examples C-17.2 through C-17.9 are not specific to peephole optimization. In fact, all but the last (exploitation of the instruction set) will appear in our discussion of more general forms of code improvement. The more general forms will do a better job, because they won't be limited to looking at a narrow window of instructions. In a compiler with good machine-specific and machine-independent code improvers,

DESIGN & IMPLEMENTATION

17.1 Peephole optimization

In many cases, it is easier to count on the code improver to catch and fix suboptimal idioms than it is to generate better code in the first place. Even a peephole optimizer will catch such common examples as multiplication by one or addition of zero; there is no point adding complexity to the code generator to treat these cases specially.

there may be no need for the peephole optimizer to eliminate redundancies or useless instructions, fold constants, perform strength reduction, or fill load and branch delays. In such a compiler the peephole optimizer serves mainly to exploit idiosyncrasies of the target machine, and perhaps to clean up certain suboptimal code idioms that leak through the rest of the back end.

17.3 Redundancy Elimination in Basic Blocks

To implement local optimizations, the compiler must first identify the fragments of the syntax tree that correspond to basic blocks, as described in Section 15.1.1. Roughly speaking, these fragments consist of tree nodes that are adjacent according to in-order traversal, and contain no selection or iteration constructs. In Figure 15.6, we presented inference rules to generate linear (goto-containing) code for simple syntax trees. A similar set of inference rules can be used to create a control flow graph (Exercise 15.6).

A call to a user subroutine within a control flow graph could be treated as a pair of branches, defining a boundary between basic blocks, but as long as we know that the call will return we can simply treat it as an instruction with potentially wide-ranging side effects (i.e., as an instruction that may overwrite many registers and memory locations). As we noted in Section 9.2.4, the compiler may also choose to expand small subroutines in-line. In this case the behavior of the "call" is completely visible. If the called routine consists of a single basic block, it becomes a part of the calling block. If it consists of multiple blocks, its prologue and epilogue become part of the blocks before and after the call.

17.3.1 A Running Example

EXAMPLE 17.10
The `combinations` subroutine

Throughout much of the remainder of this chapter we will trace the improvement of code for a single subroutine: specifically, one that calculates into an array the

DESIGN & IMPLEMENTATION

17.2 Basic blocks

Many of a program's basic blocks are obvious in the source. Some, however, are created by the compiler during the translation process. Loops may be created, for example, to copy or initialize large records or subroutine parameters. Run-time semantic checks, likewise, induce large numbers of implicit selection statements. Moreover, as we shall see in Sections C-17.4.2, C-17.5, and C-17.7, many optimizations move code from one basic block to another, create or destroy basic blocks, or completely restructure loop nests. As a result of these optimizations, the final control flow graph may be very different from what the programmer might naively expect.

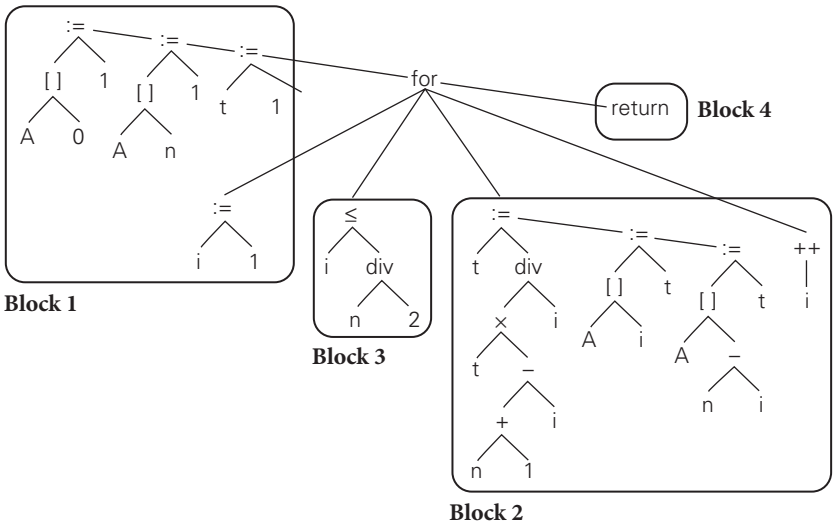


Figure 17.2 Syntax tree for the `combinations` subroutine. Portions of the tree corresponding to basic blocks have been circled.

binomial coefficients $\binom{n}{m}$ for all $0 \leq m \leq n$. These are the elements of the nth row of Pascal's triangle. The mth element of the row indicates the number of distinct combinations of m items that may be chosen from among a collection of n items. In C, the code looks like this:

```
void combinations(int n, int *A) {
    int i, t;
    A[0] = 1;
    A[n] = 1;
    t = 1;
    for (i = 1; i <= n/2; i++) {
        t = (t * (n+1-i)) / i;
        A[i] = t;
        A[n-i] = t;
    }
}
```

This code capitalizes on the fact that $\binom{n}{m} = \binom{n}{n-m}$ for all $0 \leq m \leq n$. One can prove (Exercise C-17.2) that the use of integer arithmetic will not lead to round-off errors.

EXAMPLE 17.11
Syntax tree and naive control flow graph

A syntax tree for our subroutine appears in Figure C-17.2, with basic blocks identified. The corresponding control flow graph appears in Figure C-17.3. To avoid artificial interference between instructions at this early stage of code improvement, we employ a medium-level intermediate form (IF) in which every calculated value is placed in a separate register. To emphasize that these are virtual registers (of

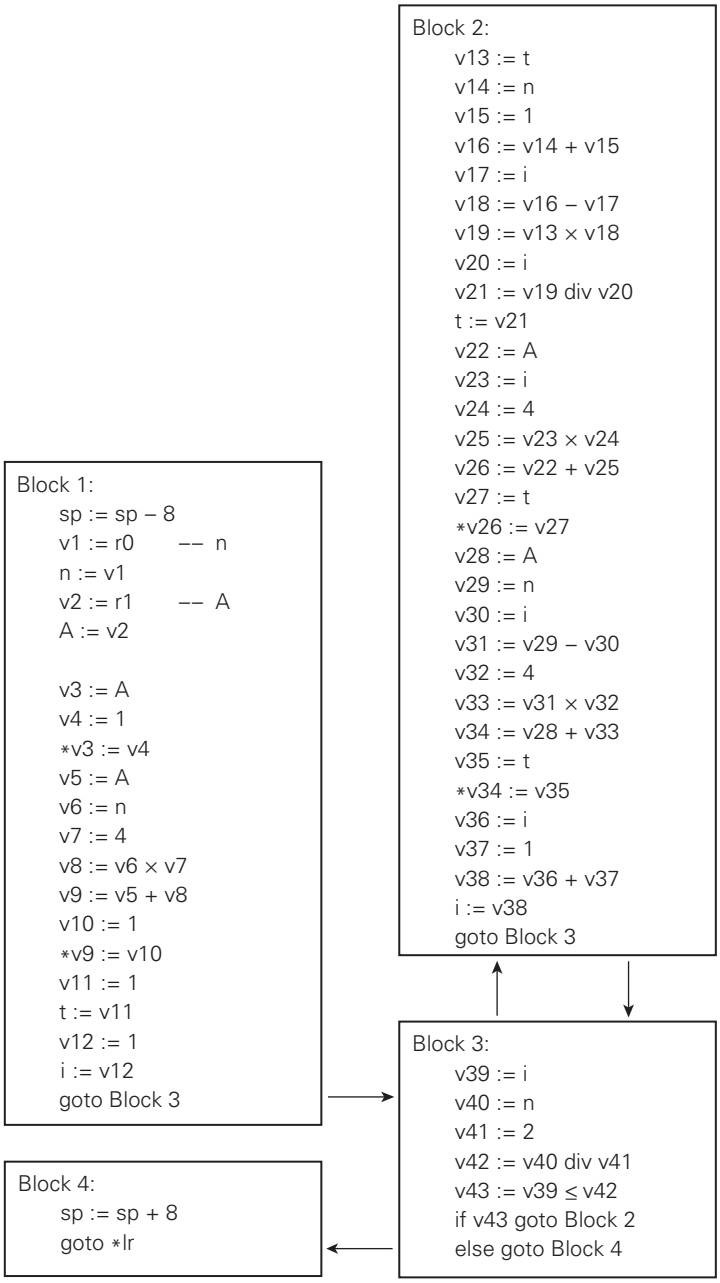


Figure 17.3 **Naive control flow graph for the `combinations` subroutine.** Note that reference parameter A contains the *address* of the array into which to write results; hence we write v3 := A instead of v3 := &A.

which there is an unlimited supply), we name them v1, v2, ... We will use r1, r2, ... to represent architectural registers in Section C-17.8.

The fact that no virtual register is assigned a value by more than one instruction in the original control flow graph is crucial to the success of our code improvement techniques. Informally, it says that every value that could eventually end up in a separate architectural register will, at least at first, be placed in a separate virtual register. Of course if an assignment to a virtual register appears within a loop, then the register may take on a different value in every iteration. In addition, as we move through the various phases of code improvement we will relax our rules to allow a virtual register to be assigned a value in more than one place. The key point is that by employing a new virtual register whenever possible at the outset we maximize the degrees of freedom available to later phases of code improvement.

In the initial (entry) and final (exit) blocks, we have included code for the subroutine prologue and epilogue. We have assumed naive Arm calling conventions, as described in Section C-9.2.2. We have also assumed that the compiler has recognized that our subroutine is a leaf, and that it therefore has no need to save the return address (link register—lr) or frame pointer (r7) registers. In all cases, accesses to n, A, i, and t in memory should be interpreted as performing the appropriate displacement addressing with respect to the stack pointer (sp) register. Though we assume that parameter values were passed in registers (architectural registers r0 and r1 on Arm), our original (naive) code immediately saves these values to memory, so that subsequent accesses can be handled in the same way as they are for local variables. We make the saves by way of virtual registers so that they will be visible to the global value numbering algorithm described in Section C-17.4.1. Eventually, after several stages of improvement, we will find that both the parameters and the local variables can be kept permanently in registers, eliminating the need for the various loads, stores, and copy operations. ■

17.3.2 Value Numbering

To improve the code within basic blocks, we need to minimize loads and stores, and to identify redundant calculations. One common way to accomplish these tasks is to translate the syntax tree for a basic block into an *expression DAG* (directed acyclic graph) in which redundant loads and computations are merged into individual nodes with multiple parents [ALSU07, Secs. 6.1.1 and 8.5.1]. Similar functionality can also be obtained without an explicitly graphical program representation, through a technique known as local *value numbering* [Muc97, Sec. 12.4]. We describe this technique below.

Value numbering assigns the same name (a "number"—historically, a table index) to any two or more symbolically equivalent computations ("values"), so that redundant instances will be recognizable by their common name. In the formulation here, our names are virtual registers, which we merge whenever they are guaranteed to hold a common value. While performing local value numbering, we will also implement local constant folding, constant propagation, copy propagation,

common subexpression elimination, strength reduction, and useless instruction elimination. (The distinctions among these optimizations will be clearer in the global case.)

We scan the instructions of a basic block in order, maintaining a dictionary to keep track of values that have already been loaded or computed, and writing instructions to a new, improved basic block that will replace the original one. For a load instruction, v_i := x, we consult the dictionary to see whether x is already in some register v_j. If so, we simply add an entry to the dictionary indicating that uses of v_i should be replaced by uses of v_j. If x is not in the dictionary, we generate a load in the new version of the basic block, and add an entry to the dictionary indicating that x is available in v_i. For a load of a constant, v_i := c, we check to see whether c is small enough to fit in the immediate operand of a compute instruction. If so, we add an entry to the dictionary indicating that uses of v_i should be replaced by uses of the constant, but we generate no code: we'll embed the constant directly in the appropriate instructions when we come to them. If the constant is large, we consult the dictionary to see whether it has already been loaded (or computed) into some other register v_j; if so, we note that uses of v_i should be replaced by uses of v_j. If the constant is large and not already available, then we generate instructions to load it into v_i and then note its availability with an appropriate dictionary entry. In all cases, we create a dictionary entry for the target register of a load, indicating whether that register (1) should be used under its own name in subsequent instructions, (2) should be replaced by uses of some other register, or (3) should be replaced by some small immediate constant.

For a compute instruction, v_i := v_j *op* v_k, we first consult the dictionary to see whether uses of v_j or v_k should be replaced by uses of some other registers or small constants v_l and v_m. If both operands are constants, then we can perform the operation at compile time, effecting constant folding. We then treat the constant as we did for loads above: keeping a note of its value if small, or of the register in which it resides if large. We also note opportunities to perform strength reduction or to eliminate useless instructions. If at least one of the operands is nonconstant (and the instruction is not useless), we consult the dictionary again to see whether the result of the (potentially modified) computation is already available in some register v_n. This final lookup operation is keyed by a combination of the operator *op* and the operand registers or constants v_j (or v_l) and v_k (or v_m). If the lookup is successful, we add an entry to the dictionary indicating that uses of v_i should be replaced by uses of v_n. If the lookup is unsuccessful, we generate an appropriate instruction (e.g., v_i := v_j *op* v_k or v_i := v_l *op* v_m) in the new version of the basic block, and add a corresponding entry to the dictionary.

As we work our way through the basic block, the dictionary provides us with four kinds of information:

1. For each already-computed virtual register: whether it should be used under its own name, replaced by some other register, or replaced by an immediate constant
2. For certain variables: what register holds the (current) value

3. For certain large constants: what register holds the value
4. For some (op, arg1, arg2) triples, where arg*i* can be a register name or a constant: what register already holds the result

For a store instruction, x := v*i*, we remove any existing entry for x in the dictionary, and add an entry indicating that x is available in v*i*. We also note (in that entry) that the value of x in memory is stale. If x may be an alias for some other variable y, we must also remove any existing entry for y from the dictionary. (If we are *certain* that y is an alias for x, then we can add an entry indicating that the value of y is available in v*i*.) A similar precaution, ignored in the discussion above, applies to loads: if x may be an alias for y, and if there is an entry for y in the dictionary indicating that the value in memory is stale, then a load instruction v*i* := x must be preceded by a store to y. When we reach the end of the block, we traverse the dictionary, generating store instructions for all variables whose values in memory are stale. If any variables may be aliases for each other, we must take care to generate the stores in the order in which the values were produced. After generating the stores, we generate the branch (if any) that ends the block.

Local Code Improvement

In the process of local value numbering we automatically perform several important operations. We identify common subexpressions (none of which occur in

DESIGN & IMPLEMENTATION

17.3 Common subexpressions

It is natural to think of common subexpressions as something that could be eliminated at the source code level, and programmers are sometimes tempted to do so. The following, for example,

```
x = a + b + c;
y = a + b + d;
```

could be replaced with

```
t = a + b;
x = t + c;
y = t + d;
```

Such changes do not always make the code easier to read, however, and if the compiler is doing its job they don't make it any faster either. Moreover numerous examples of common subexpressions are entirely invisible in the source code. Examples include array subscript calculations (Section 8.2.3), references to variables in lexically enclosing scopes (Section 9.2), and references to nearby fields in complex records (Section 8.1.3). Like the pointer arithmetic discussed in Sidebar 8.8, hand elimination of common subexpressions, unless it makes the code easier to read, is usually not a good idea.

our `combinations` example), allowing us to compute them only once. We also implement constant folding and certain strength reductions. Finally, we perform local constant and copy propagation, and eliminate redundant loads and stores: our use of the dictionary to delay store instructions ensures that (in the absence of potential aliases) we never write a variable twice, or write and then read it again within the same basic block.

To increase the number of common subexpressions we can find, we may want to traverse the syntax tree prior to linearizing it, rearranging expressions into some sort of normal form. For commutative operations, for example, we can swap subtrees if necessary to put operands in lexicographic order. We can then recognize that `a + b` and `b + a` are common subexpressions. In some cases (e.g., in the context of array address calculations, or with explicit permission from the programmer), we may use associative or distributive rules to normalize expressions as well, though as we noted in Section 6.1.4 such changes can in general lead to arithmetic overflow or numerical instability. Unfortunately, straightforward normalization techniques will fail to recognize the redundancy in `a + b + c` and `a + c`; lexicographic ordering is simply a heuristic.

A naive approach to aliases is to assume that assignment to element *i* of an array may alter element *j*, for any *j*; that assignment through a pointer to an object of type *t* (in a type-safe language) may alter any variable of that type; and that a call to a subroutine may alter any variable visible in the subroutine's scope (including at a minimum all globals). These assumptions are overly conservative and can greatly limit the ability of a compiler to generate good code. More aggressive compilers perform extensive symbolic analysis of array subscripts in order to narrow the set of potential aliases for an array assignment. Similar analysis may be able to determine that particular array or record elements can be treated as unaliased scalars, making them candidates for allocation to registers. Recent years have also seen the development of very good alias analysis techniques for pointers (see Sidebar C-17.4).

EXAMPLE 17.12
Result of local redundancy elimination

Figure C-17.4 shows the control flow graph for our `combinations` subroutine after local redundancy elimination. We have eliminated 21 of the instructions in Figure C-17.3, all of them loads of variables or constants. Thirteen of the eliminated

DESIGN & IMPLEMENTATION

17.4 Pointer analysis

The tendency of pointers to introduce aliases is one of the reasons why Fortran compilers have traditionally produced faster code than C compilers. Prior to Fortran 90, the language had no pointers, and many Fortran programs are still written without them. C programs, by contrast, tend to be pointer-rich. Some time ago, alias analysis for pointers reached the point at which good C compilers could rival their Fortran counterparts; it remains an active research topic. For a survey of the field as of 2015, see the tutorial by Smaragdakis and Balatsouras [SB15].

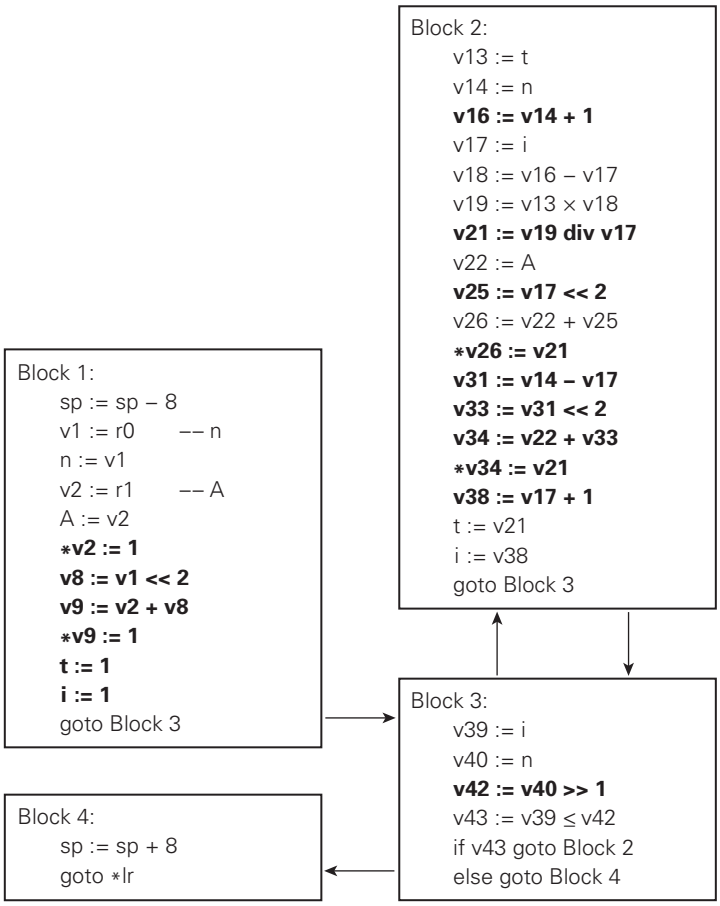


Figure 17.4 Control flow graph for the `combinations` subroutine after local redundancy elimination and strength reduction. Changes from Figure C-17.3 are shown in boldface type.

instructions are in the body of the loop (Blocks 2 and 3) where improvements are particularly important. We have also performed strength reduction on the two instructions that multiply a register by the constant 4 and the one that divides a register by 2, replacing them by equivalent shifts. ■

CHECK YOUR UNDERSTANDING

1. Describe several increasing levels of "aggressiveness" in code improvement.
2. Give three examples of code improvements that must be performed in a particular order. Give two examples of code improvements that should probably be performed more than once (with other improvements in between).

3. What is *peephole optimization*? Describe at least four different ways in which a peephole optimizer might transform a program.
4. What is *constant folding*? *Constant propagation*? *Copy propagation*? *Strength reduction*?
5. What does it mean for a value in a register to be *live*?
6. What is a *control flow graph*? Why is it central to so many forms of global code improvement? How does it accommodate subroutine calls?
7. What is *value numbering*? What purpose does it serve?
8. Explain the connection between common subexpressions and expression rearrangement.
9. Why is it not practical in general for the programmer to eliminate common subexpressions at the source level?

17.4 Global Redundancy and Data Flow Analysis

In this section we will concentrate on the elimination of redundant loads and computations across the boundaries between basic blocks. We will translate the code of our basic blocks into *static single assignment* (SSA) form, which will allow us to perform global value numbering. Once value numbers have been assigned, we shall be able to perform global common subexpression elimination, constant propagation, and copy propagation. In a compiler both the translation to SSA form and the various global optimizations would be driven by data flow analysis. We will go into some of the details for global optimization (specifically, for the problems of identifying common subexpressions and useless store instructions) after a much more informal presentation of the translation to SSA form. We will also give data flow equations in Section C-17.5 for the calculation of *reaching definitions*, used (among other things) to move invariant computations out of loops.

Global redundancy elimination can be structured in such a way that it catches local redundancies as well, eliminating the need for a separate local pass. The global algorithms are easier to implement and to explain, however, if we assume that a local pass has already occurred. In particular, local redundancy elimination allows us to assume (in the absence of aliases, which we will ignore in our discussion) that no variable is read or written more than once in a basic block.

17.4.1 SSA Form and Global Value Numbering

Value numbering, as introduced in Section C-17.3, assigns a distinct virtual register name to every symbolically distinct value that is loaded or computed in a given body of code, allowing us to recognize when certain loads or computations are

redundant. The first step in *global* value numbering is to distinguish among the values that may be written to a variable in different basic blocks. We accomplish this step using static single assignment (SSA) form.

Our initial translation to medium-level IF ensured that each virtual register was assigned a value by a unique instruction. This uniqueness was preserved by local value numbering. Variables, however, may be assigned in more than one basic block. Our translation to SSA form therefore begins by adding subscripts to variable names: a different one for each distinct store instruction. This convention makes it easier to identify global redundancies. It also explains the terminology: each subscripted variable in an SSA program has a single static (compile time) assignment—a single store instruction.

Following the flow of the program, we assign subscripts to variables in load instructions, to match the corresponding stores. If the instruction v2 := x is guaranteed to read the value of x written by the instruction x_3 := v1, then we replace v2 := x with v2 := x_3. If we cannot tell which version of x will be read, we use a hypothetical *merge function* (also known as a *selection function*, and traditionally represented by the Greek letter ϕ) to choose among the possible alternatives. Fortunately, we won't actually have to compute merge functions at run time. Their only purpose is to help us identify possible code improvements; we will drop them (and the subscripts) prior to target code generation.

In general, the translation to SSA form (and the identification of merge functions in particular) requires the use of data flow analysis. We will describe the concept of data flow in the context of global common subexpression elimination in Section C-17.4.2. In the current subsection we will generate SSA code informally; data flow formulations can be found in more advanced compiler texts [CT11, Sec. 9.3; AK02, Sec. 4.4.4; App97, Sec. 19.1; Muc97, Sec. 8.11].

EXAMPLE 17.13
Conversion to SSA form

In the `combinations` subroutine (Figure C-17.4) we assign the subscript 1 to the stores of t and i at the end of Block 1. We assign the subscript 2 to the stores of t and i at the end of Block 2. Thus at the end of Block 1 t_1 and i_1 are live; at the end of Block 2 t_2 and i_2 are live. What about Block 3? If control enters Block 3 from Block 1, then t_1 and i_1 will be live, but if control enters Block 3 from Block 2, then t_2 and i_2 will be live. We invent a merge function ϕ that returns its first argument if control enters Block 3 from Block 1, and its second argument if control enters Block 3 from Block 2. We then use this function to write values to new names t_3 and i_3. Since Block 3 does not modify either t or i, we know that t_3 and i_3 will be live at the end of the block. Moreover, since control always enters Block 2 from Block 3, t_3 and i_3 will be live at the beginning of Block 2. The load of v13 in Block 2 is guaranteed to return t_3; the loads of v17 in Block 2 and of v39 in Block 3 are guaranteed to return i_3.

SSA form annotates the right-hand sides of loads with subscripts and merge functions in such a way that at any given point in the program, if v*i* and v*j* were given values by load instructions with symbolically identical right-hand sides, then the loaded values are guaranteed to have been produced by (the same execution of) the same prior store instruction. Because ours is a simple subroutine, only one merge function is needed: it indicates whether control entered Block 3 from Block 1

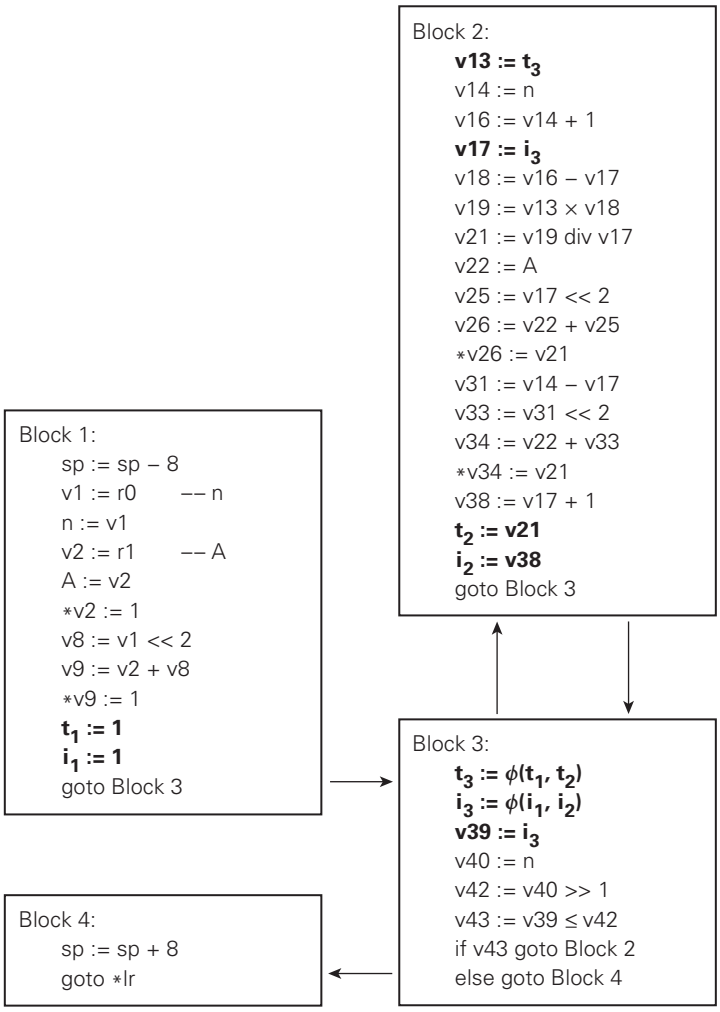


Figure 17.5 Control flow graph for the `combinations` subroutine, in static single assignment (SSA) form. Changes from Figure C-17.4 are shown in boldface type.

or from Block 2. In a more complicated subroutine there could be additional merge functions, for other blocks with more than one predecessor. SSA form for the `combinations` subroutine appears in Figure C-17.5.

EXAMPLE 17.14
Global value numbering

With flow-dependent values determined by merge functions, we are now in a position to perform global value numbering. As in local value numbering, the goal is to merge any virtual registers that are guaranteed to hold symbolically equivalent expressions.

In the local case we were able to perform a linear pass over the code, keeping a dictionary that mapped loaded and computed expressions to the names of virtual

registers that contained them. This approach does not suffice in the global case, because the code may have cycles. The general solution can be formulated using data flow, or obtained with a simpler algorithm [Muc97, Sec. 12.4.2] that begins by unifying all expressions with the same top-level operator, and then repeatedly separates expressions whose operands are distinct, in a manner reminiscent of the DFA minimization algorithm of Section 2.2.1. In contrast to our presentation of local value numbering, where we performed code improvements such as eliminating redundant loads and stores, we perform only global value numbering here, leaving further code improvements to separate dataflow analyses that build on our results. Again, we perform the analysis for our running example informally.

We can begin by adopting the results of local value numbering for Block 1; since this is the first basic block and local redundancies have been removed, its virtual register names have already been merged as much as possible. In Block 2, the second instruction loads n into v14. Since we already used v1 for n in Block 1, we can substitute the same name here. This substitution violates, for the first time, our assumption that every virtual register is given a value by a single static instruction. The "violation" is safe, however: both occurrences of n have the same subscript (none at all, in this case), so we know that at any given point in the code, if v1 and v14 have both been given values, then those values are the same. We can't (yet) eliminate the load in Block 2, because we don't (yet) know that Block 1 will have executed first. For consistency we replace v14 with v1 in the third instruction of Block 2. Then, by similar reasoning, we replace v22 with v2 in the 8th, 10th, and 14th instructions.

In Block 3 we have more replacements. In the first real instruction ($v39 := i_3$), we recall that the same right-hand side is loaded into v17 in Block 2. We therefore replace v39 with v17, in both the first and fourth instructions. Similarly, we replace v40 with v1, in both the second and third instructions. There are no changes in Block 4.

The result of global value numbering on our `combinations` subroutine appears in Figure C-17.6. In this case the only common values identified were variables loaded from memory. In a more complicated subroutine, we would also identify known-to-be-identical computations performed in more than one block (though we would not yet know which, if any, were redundant). As we did with loads, we would rename left-hand sides so that all symbolically equivalent computations place their results in the same virtual register.

Static single assignment form is useful for a variety of code improvements. In our discussion here we use it only for global value numbering. We will drop it in later figures.

17.4.2 Global Common Subexpression Elimination

We have seen an informal example of data flow analysis in the construction of static single assignment form. We will now employ a more formal example for global common subexpression elimination. As a result of global value numbering, we

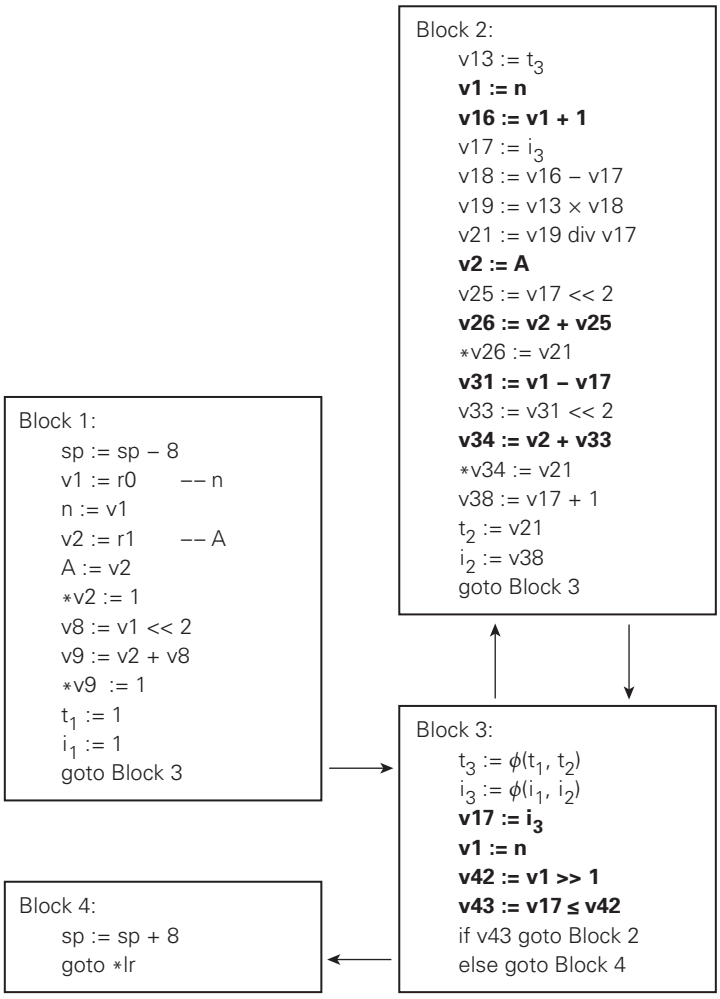


Figure 17.6 Control flow graph for the `combinations` subroutine after global value numbering. Changes from Figure C-17.5 are shown in boldface type.

know that any common subexpression will have been placed into the same virtual register wherever it is computed. We will therefore use virtual register names to represent expressions in the discussion below.[2] The goal of global common subexpression elimination is to identify places in which an instruction that computes

2 As presented here, there is a one–one correspondence among SSA names, global value numbers, and (after global value numbering has been completed) virtual register names. Other texts and papers sometimes distinguish among these concepts more carefully, and use them for different purposes.

a value for a given virtual register can be eliminated, because the computation is certain to already have occurred on every control path leading to the instruction.

Many instances of data flow analysis can be cast in the following framework: (1) four sets for each basic block B, called In_B, Out_B, Gen_B, and $Kill_B$; (2) values for the *Gen* and *Kill* sets; (3) an equation relating the sets for any given block B; (4) an equation relating the *Out* set of a given block to the *In* sets of its successors, or relating the *In* set of the block to the *Out* sets of its predecessors; and (often) (5) certain initial conditions. The goal of the analysis is to find a *fixed point* of the equations: a consistent set of *In* and *Out* sets that satisfy both the equations and the initial conditions. Some problems have a single fixed point. Others may have more than one, in which case we usually want either the least or the greatest fixed point (smallest or largest sets).

EXAMPLE 17.15

Data flow equations for available expressions

In the case of global common subexpression elimination, In_B is the set of expressions (virtual registers) guaranteed to be available at the beginning of block B. These *available expressions* will all have been set by predecessor blocks. Out_B is the set of expressions guaranteed to be available at the end of B. $Kill_B$ is the set of expressions *killed* in B: invalidated by assignment to one of the variables used to calculate the expression, and not subsequently recalculated in B. Gen_B is the set of expressions calculated in B and not subsequently killed in B. The data flow equations for available expression analysis are[3]

$$\begin{aligned} Out_B &= Gen_B \cup (In_B \setminus Kill_B) \\ In_B &= \bigcap_{\text{predecessors } A \text{ of } B} Out_A \end{aligned}$$

Our initial condition is $In_1 = \varnothing$: no expressions are available at the beginning of execution.

Available expression analysis is known as a *forward* data flow problem, because information flows forward across branches: the *In* set of a block depends on the *Out* sets of its predecessors. We shall see an example of a *backward* data flow problem later in this section.

EXAMPLE 17.16

Fixed point for available expressions

We calculate the desired fixed point of our equations in an inductive (iterative) fashion, much as we computed FIRST and FOLLOW sets in Section 2.3.3. Our equation for In_B uses intersection to insist that an expression be available on all paths into B. In our iterative algorithm, this means that In_B can only shrink with subsequent iterations. Because we want to find as many available expressions as possible, we therefore optimistically assume that all expressions are initially available as inputs to all blocks other than the first; that is, $In_{B,B\neq 1} = \{\mathsf{n}, \mathsf{A}, \mathsf{t}, \mathsf{i}, \mathsf{v1}, \mathsf{v2}, \mathsf{v8}, \mathsf{v9}, \mathsf{v13}, \mathsf{v16}, \mathsf{v17}, \mathsf{v18}, \mathsf{v19}, \mathsf{v21}, \mathsf{v25}, \mathsf{v26}, \mathsf{v31}, \mathsf{v33}, \mathsf{v34}, \mathsf{v38}, \mathsf{v42}, \mathsf{v43}\}$.

Our *Gen* and *Kill* sets can be found in a single backward pass over each of the basic blocks. In Block 3, for example, the last assignment defines a value for v43.

---

3 Set notation here is standard: $\bigcup_i S_i$ indicates the union of all sets S_i; $\bigcap_i S_i$ indicates the intersection of all sets S_i; $A \setminus B$, pronounced "A minus B" indicates the set of all elements found in A but not in B.

We therefore know that v43 is in Gen_3. Working backward, so are v42, v1, and v17. As we notice each of these, we also consider their impact on $Kill_3$. Virtual register v43 does not appear on the right-hand side of any assignment in the program (it is not part of the expression named by any virtual register), so giving it a value kills nothing. Virtual register v42 is part of the expression named by v43, but since v43 is given a value later in the block (is already in Gen_3), the assignment to v42 does not force v43 into $Kill_3$. Virtual register v1 is a different story. It is part of the expressions named by v8, v16, v31, and v42. Since v42 is already in Gen_3, we do not add it to $Kill_3$. We do, however, put v8, v16, and v31 in $Kill_3$. In a similar manner, the assignment to v17 forces v18, v21, v25, and v38 into $Kill_3$. Note that we do not have to worry about virtual registers that depend in turn on v8, v16, v18, v21, v25, v31, or v38: our iterative data flow algorithm will take care of that; all we need now is one level of dependence. Stores to program variables (e.g., at the ends of Blocks 1 and 2) kill the corresponding virtual registers.

After completing a backward scan of all four blocks, we have the following *Gen* and *Kill* sets:

$Gen_1 = \{v1, v2, v8, v9\}$
$Gen_2 = \{v1, v2, v13, v16, v17, v18, v19, v21, v25, v26, v31, v33, v34, v38\}$
$Gen_3 = \{v1, v17, v42, v43\}$
$Gen_4 = \varnothing$

$Kill_1 = \{v13, v16, v17, v26, v31, v34, v42\}$
$Kill_2 = \{v8, v9, v13, v17, v42, v43\}$
$Kill_3 = \{v8, v16, v18, v21, v25, v31, v38\}$
$Kill_4 = \varnothing$

Applying the first of our data flow equations ($Out_B = Gen_B \cup (In_B \setminus Kill_B)$) to all blocks, we obtain

$Out_1 = \{v1, v2, v8, v9\}$
$Out_2 = \{v1, v2, v13, v16, v17, v18, v19, v21, v25, v26, v31, v33, v34, v38\}$
$Out_3 = \{v1, v2, v9, v13, v17, v19, v26, v33, v34, v42, v43\}$
$Out_4 = \{v1, v2, v8, v9, v13, v16, v17, v18, v19, v21, v25, v26, v31, v33, v34, v38, v42, v43\}$

If we now apply our second equation ($In_B = \bigcap_A Out_A$) to all blocks, followed by a second iteration of the first equation, we obtain

$In_1 = \varnothing$
$In_2 = \{v1, v2, v9, v13, v17, v19, v26, v33, v34, v42, v43\}$
$In_3 = \{v1, v2\}$
$In_4 = \{v1, v2, v9, v13, v17, v19, v26, v33, v34, v42, v43\}$

$Out_1 = \{v1, v2, v8, v9\}$
$Out_2 = \{v1, v2, v13, v16, v17, v18, v19, v21, v25, v26, v31, v33, v34, v38\}$
$Out_3 = \{v1, v2, v17, v42, v43\}$
$Out_4 = \{v1, v2, v9, v13, v17, v19, v26, v33, v34, v42, v43\}$

One more iteration of each equation yields the fixed point:

$In_1 = \varnothing$ $Out_1 = \{v1, v2, v8, v9\}$
$In_2 = \{v1, v2, v17, v42, v43\}$ $Out_2 = \{v1, v2, v13, v16, v17, v18, v19, v21, v25, v26, v31, v33, v34, v38\}$
$In_3 = \{v1, v2\}$ $Out_3 = \{v1, v2, v17, v42, v43\}$
$In_4 = \{v1, v2, v17, v42, v43\}$ $Out_4 = \{v1, v2, v17, v42, v43\}$

EXAMPLE 17.17
Result of global common subexpression elimination

We can now exploit what we have learned. Whenever a virtual register is in the *In* set of a block, we can drop any assignment of that register in the block. In our example subroutine, we can drop the loads of v1, v2, and v17 in Block 2, and the load of v1 in Block 3. In addition, whenever a virtual register corresponding to a variable is in the *In* set of a block, we can replace a load of that variable with a register–register move on each of the potential paths into the block. In our example, we can replace the load of t in Block 2 and the load of i in Block 3 (the load of i in Block 2 has already been eliminated). To compensate, we must load v13 and v17 with the constant 1 at the end of Block 1, and move v21 into v13 and v38 into v17 at the end of Block 2. The final result appears in Figure C-17.7.

(The careful reader may note that v21 and v38 are not strictly necessary: if we computed new values directly into v13 and v17, we could eliminate the two register–register moves. This observation, while correct, need not be made at this time; it can wait until we perform induction variable optimizations and register allocation, to be described in Sections C-17.5.2 and C-17.8, respectively.)

Splitting Control Flow Edges

EXAMPLE 17.18
Edge splitting transformations

If the block (call it A) in which a variable is written has more than one successor, only one of which (call it B) contains a redundant load, and if B has more than one predecessor, then we need to create a new block on the arc between A and B to hold the register–register move. This way the move will not be executed on code paths that don't need it. In a similar vein, if an expression is available from A but not from B's other predecessor, then we can move the load or computation of the expression back into the predecessor that lacks it or, if that predecessor has more than one successor, into a new block on the connecting arc. This move will eliminate a redundancy on the path through A. These "edge splitting" transformations are illustrated in Figure C-17.8. In general, a load or computation is said to be *partially redundant* if it is a repetition of an earlier load or store on some paths through the flow graph, but not on others. No edge splits are required in the `combinations` example.

Common subexpression elimination can have a complicated effect on register pressure. If we realize that the expression v10 + v20 has been calculated into, say, register v30 earlier in the program, and we exploit this knowledge to replace a later recalculation of the expression with a direct use of v30, then we may expand v30's *live range*—the span of instructions over which its value is needed. At the same time, if v10 and v20 are not used for other purposes in the intervening region of the program, we may *shrink* the range over which *they* are live. In a subroutine with a high level of register pressure, a good compiler may sometimes perform the

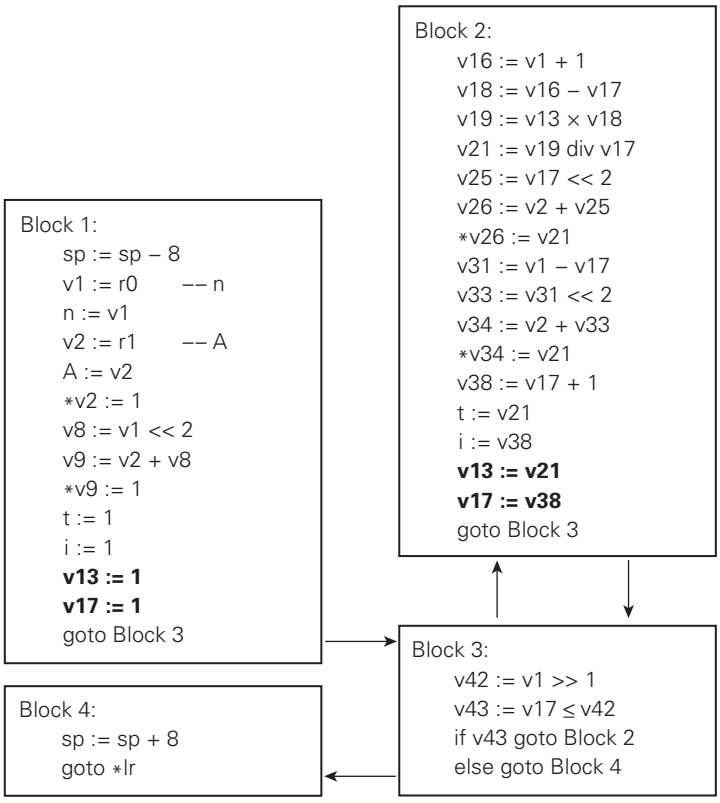


Figure 17.7 Control flow graph for the `combinations` subroutine after performing global common subexpression elimination. Note the absence of the many load instructions of Figure C-17.6. Compensating register–register moves are shown in boldface type.

inverse of common subexpression elimination (known as *forward substitution*) in order to shrink live ranges.

Live Variable Analysis

Constant propagation and copy propagation, like common subexpression elimination, can be formulated as instances of data flow analysis. We skip these analyses here; none of them yields improvements in our example. Instead, we turn our attention to *live variable analysis*, which is very important in our example, and in general in any subroutine in which global common subexpression analysis has eliminated load instructions.

Live variable analysis is the *backward* flow problem mentioned above. It determines which instructions produce values that will be needed in the future, allowing us to eliminate *dead* (useless) instructions. In our example we will concern ourselves only with values written to memory and with the elimination of dead stores. When applied to values in virtual registers as well, live variable analysis

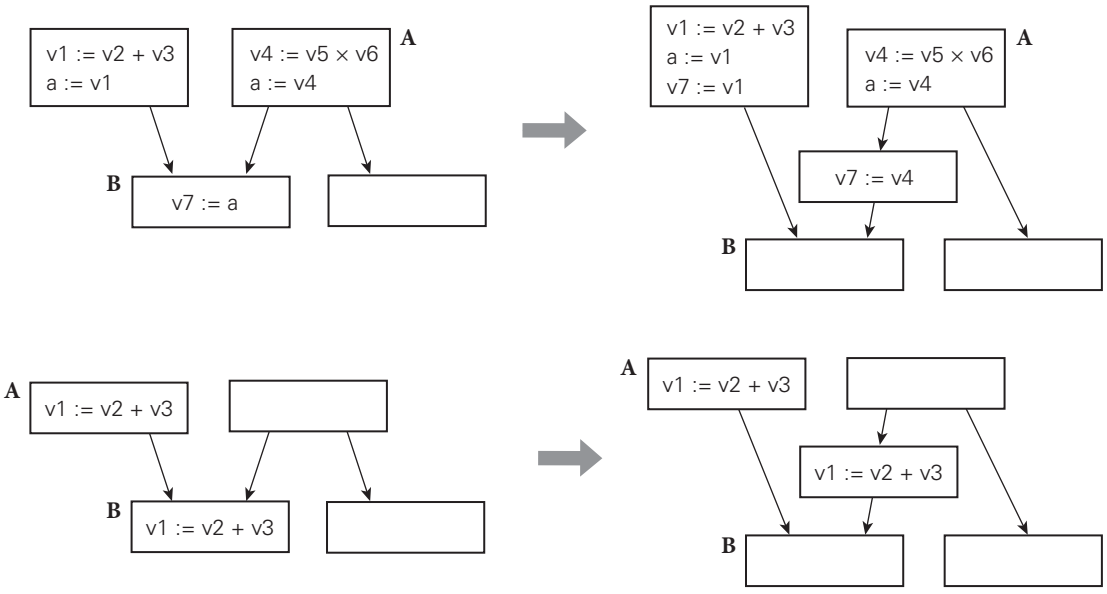


Figure 17.8 Splitting an edge of a control flow graph to eliminate a redundant load (*top*) or a partially redundant computation (*bottom*).

can help to identify other dead instructions. (None of these arise this early in the `combinations` example.)

EXAMPLE 17.19
Data flow equations for live variables

For this instance of data flow analysis, In_B is the set of variables that are live at the beginning of block B. Out_B is the set of variables that are live at the end of the block. Gen_B is the set of variables read in B without first being written in B. $Kill_B$ is the set of variables written in B without having been read first. The data flow equations are

$$In_B = Gen_B \cup (Out_B \setminus Kill_B)$$
$$Out_B = \bigcup_{\text{successors } C \text{ of } B} In_C$$

Our initial condition is $Out_4 = \varnothing$: no variables are live at the end of execution. (If our subroutine wrote any nonlocal [e.g., global] variables, these would be initial members of Out_4.)

In comparison to the equations for available expression analysis, the roles of *In* and *Out* have been reversed (that's why it's a backward problem), and the intersection operator in the second equation has been replaced by a union. Intersection ("all paths") problems require that information flow over *all* paths between blocks; union ("any path") problems require that it flow along *some* path. Further data flow examples appear in Exercises C-17.7 and C-17.9.

EXAMPLE 17.20
Fixed point for live variables

In our example program, we have

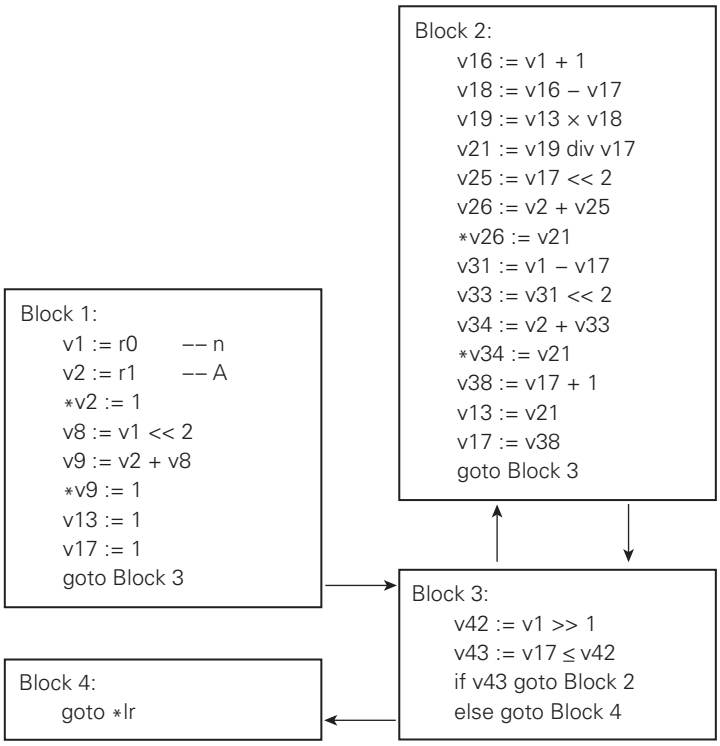


Figure 17.9 Control flow graph for the `combinations` subroutine after performing live variable analysis. Starting with Figure C-17.7, the compiler has eliminated all stores to n, A, t, and i. It has also dropped the changes to the stack pointer that used to appear in the subroutine prologue and epilogue: we don't need space for local variables anymore.

$$
\begin{aligned}
Gen_1 &= \varnothing & Kill_1 &= \{\mathsf{n}, \mathsf{A}, \mathsf{t}, \mathsf{i}\} \\
Gen_2 &= \varnothing & Kill_2 &= \{\mathsf{t}, \mathsf{i}\} \\
Gen_3 &= \varnothing & Kill_3 &= \varnothing \\
Gen_4 &= \varnothing & Kill_4 &= \varnothing
\end{aligned}
$$

Our use of union means that *Out* sets can only grow with each iteration, so we begin with $Out_B = \varnothing$ for all blocks B (not just B_4). One iteration of our data flow equations gives us $In_B = Gen_B$ and $Out_B = \varnothing$ for all blocks B. But since $Gen_B = \varnothing$ for all B, this is our fixed point! Common subexpression elimination has left us with a situation in which none of our parameters or local variables is live; all of the stores of A, n, t, and i can be eliminated. Moreover, now that computation works entirely in registers, we don't even need a stack frame: we can eliminate the updates of the stack pointer in the subroutine prologue and epilogue, leaving us with the code in Figure C-17.9.

Aliases must be treated in a conservative fashion in both common subexpression elimination and live variable analysis. If a store instruction might modify variable

x, then for purposes of common subexpression elimination we must consider the store as killing any expression that depends on x. If a load instruction might access x, and x is not written earlier in the block containing the load, then x must be considered live at the beginning of the block. In our example we have assumed that the compiler is able to verify that, as a reference parameter, array A cannot alias either value parameter n or local variables t and i.

CHECK YOUR UNDERSTANDING

10. What is *static single assignment (SSA) form*? Why is SSA form needed for global value numbering, but not for local value numbering?
11. What are *merge functions* in the context of SSA form?
12. Give three distinct examples of *data flow analysis*. Explain the difference between *forward* and *backward* flow. Explain the difference between *all-paths* and *any-path* flow.
13. Explain the role of the *In*, *Out*, *Gen*, and *Kill* sets common to many examples of data flow analysis.
14. What is a *partially redundant* computation? Why might an algorithm to eliminate partial redundancies need to *split* an edge in a control flow graph?
15. What is an *available expression*?
16. What is *forward substitution*?
17. What is *live variable analysis*? What purpose does it serve?
18. Describe at least three instances in which code improvement algorithms must consider the possibility of aliases.

17.5 Loop Improvement I

Because programs tend to spend most of their time in loops, code improvements that improve the speed of loops are particularly important. In this section we consider two classes of loop improvements: those that move *invariant* computations out of the body of a loop and into its header, and those that reduce the amount of time spent maintaining *induction variables*. In Section C-17.7 we will consider transformations that improve instruction scheduling by restructuring a loop body to include portions of more than one iteration of the original loop, and that manipulate multiply nested loops to improve cache performance or increase opportunities for parallelization.

17.5.1 Loop Invariants

A *loop invariant* is an instruction (i.e., a load or calculation) in a loop whose result is guaranteed to be the same in every iteration.[4] If a loop is executed n times and we are able to move an invariant instruction out of the body and into the header (saving its result in a register for use within the body), then we will eliminate $n - 1$ calculations from the program, a potentially significant savings.

In order to tell whether an instruction is invariant, we need to identify the bodies of loops, and we need to track the locations at which operand values are defined. The first task—identifying loops—is easy in a language that relies exclusively on structured control flow: we simply save appropriate markers when linearizing the syntax tree. In a language with `goto` statements we may need to construct (recover) the loops from a less structured control flow graph.

Tracking the locations at which an operand may have been defined amounts to the problem of *reaching definitions*. Formally, we say an instruction that assigns a value v into a location (variable or register) l *reaches* a point p in the code if v may still be in l at p. Like the conversion to static single assignment form, considered informally in Section C-17.4.1, the problem of reaching definitions can be structured as a set of forward, any-path data flow equations. We let Gen_B be the set of final assignments in block B (those that are not overwritten later in B). For each assignment in B we also place in $Kill_B$ all *other* assignments (in any block) to the same location. Then we have

EXAMPLE 17.21
Data flow equations for reaching definitions

$$\begin{aligned} Out_B &= Gen_B \cup (In_B \setminus Kill_B) \\ In_B &= \bigcup_{\text{predecessors } C \text{ of } B} Out_C \end{aligned}$$

Our initial condition is that $In_1 = \varnothing$: no definitions in the function reach its entry point. Given In_B (the set of reaching definitions at the beginning of the block), we can determine the reaching definitions of all values used *within* B by a simple linear perusal of the code. Because our union operator will iteratively grow the sets of reaching definitions, we begin our computation with $In_B = \varnothing$ for all blocks B (not just B_1). ■

DESIGN & IMPLEMENTATION

17.5 Loop invariants

Many loop invariants arise from address calculations, especially for arrays. Like the common subexpressions discussed in Sidebar C-17.3, they are often not explicit in the program source, and thus cannot be hoisted out of loops by handwritten optimization.

4 Note that this use of the term is unrelated to the notion of loop invariants in axiomatic semantics (discussed under "Assertions" in Section 4.4).

Given reaching definitions, we define an instruction to be a loop invariant if each of its operands (1) is a constant, (2) has reaching definitions that all lie outside the loop, or (3) has a single reaching definition, even if that definition is an instruction *d* located inside the loop, so long as *d* is itself a loop invariant. (If there is more than one reaching definition for a particular variable, then we cannot be sure of invariance unless we know that all definitions will assign the same value, something that most compilers do not attempt to infer.) As in previous analyses, we begin with the obvious cases and proceed inductively until we reach a fixed point.

EXAMPLE 17.22
Result of hoisting loop invariants

In our `combinations` example, visual inspection of the code reveals two loop invariants: the assignment to v16 in Block 2 and the assignment to v42 in Block 3. Moving these invariants out of the loop (and dropping the dead stores and stack pointer updates of Figure C-17.7) yields the code of Figure C-17.10. ■

In the new version of the code, v16 and v42 will be calculated even if the loop is executed zero times. In general this precalculation may not be a good idea. If an invariant calculation is expensive and the loop is not in fact executed, then we may have made the program slower. Worse, if an invariant calculation may produce a run-time error (e.g., divide by zero), we may have made the program incorrect. A safe and efficient general solution is to insert an initial test for zero iterations *before* any invariant calculations; we consider this option in Exercise C-17.4. In the specific case of the `combinations` subroutine, our more naive transformation is both safe and (in the common case) efficient.

17.5.2 Induction Variables

An *induction variable* (or register) is one that takes on a simple progression of values in successive iterations of a loop. We will confine our attention here to arithmetic progressions; more elaborate examples appear in Exercises C-17.11 and C-17.12. Induction variables commonly appear as loop indices, subscript computations, or variables incremented or decremented explicitly within the body of the loop. Induction variables are important for two main reasons:

- They commonly provide opportunities for strength reduction, most notably by replacing multiplication with addition. For example, if i is a loop index variable,

EXAMPLE 17.23
Induction variable strength reduction

DESIGN & IMPLEMENTATION

17.6 Control flow analysis

Most of the loops in a modern language, with structured control flow, correspond directly to explicit constructs in the syntax tree. A few may be implicit; examples include the loops required to initialize or copy large records or subroutine parameters, or to capture tail recursion. For older languages, the recovery of structure depends on a technique known as *control flow analysis*. A detailed treatment can be found in standard compiler texts [AK02, Sec. 4.5; App97, Sec. 18.1; Muc97, Chap. 7]; we do not discuss it further here.

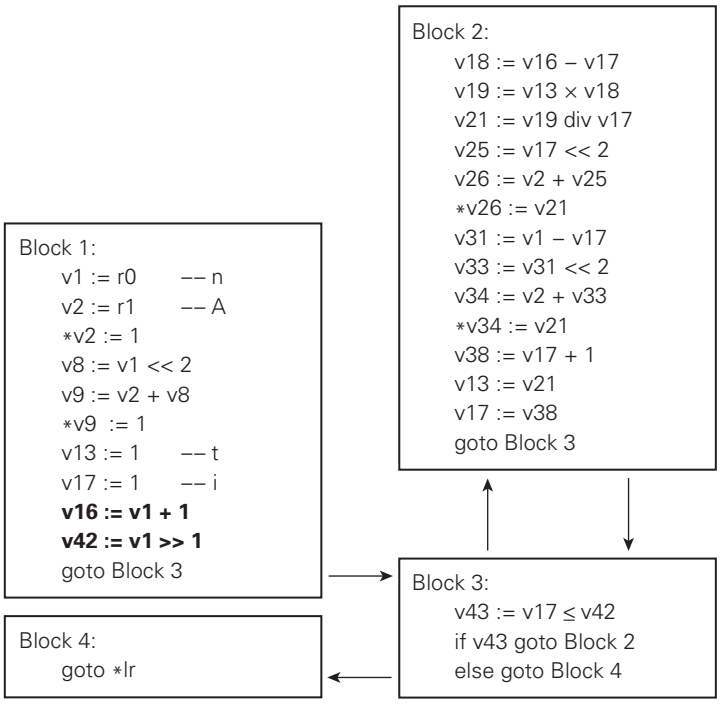


Figure 17.10 **Control flow graph for the `combinations` subroutine after moving the invariant calculations of** v16 **and** v42 **(shown in boldface type) out of the loop.** We have also dropped the dead stores of Figure C-17.7, and have eliminated the stack space for t and i, which now reside entirely in registers.

then expressions of the form $t := k \times i + c$ for $i > a$ can be replaced by $t_i := t_{i-1} + k$, where $t_a = k \times a + c$.

- They are commonly redundant: instead of keeping several induction variables in registers across all iterations of the loop, we can often keep a smaller number and calculate the remainder from those when needed (assuming the calculations are sufficiently inexpensive). The result is often a reduction in register pressure with no increase—and sometimes a decrease—in computation cost. In particular, after strength-reducing other induction variables, we can often eliminate the loop index variable itself, with an appropriate change to the end test (see Figure C-17.11 for an example).

EXAMPLE 17.24
Induction variable elimination

The algorithms required to identify, strength-reduce, and possibly eliminate induction variables are more or less straightforward, but fairly tedious [AK02, Sec. 4.5; App97, Sec. 18.3; Muc97, Chap. 14]; we do not present the details here. Similar algorithms can be used to eliminate array and subrange bounds checks in many applications.

```
A : array [1..n] of record
        key : integer
        // other stuff
for i in 1..n
    A[i].key := 0
```

(a)

```
    v1 := 1
    v2 := n
    v3 := sizeof(record)
    v4 := &A − v3
L:  v5 := v1 × v3
    v6 := v4 + v5
    *v6 := 0
    v1 := v1 + 1
    v7 := v1 ≤ v2
    if v7 goto L
```

(b)

```
    v1 := 1
    v2 := n
    v3 := sizeof(record)
    v5 := &A
L:  *v5 := 0
    v5 := v5 + v3
    v1 := v1 + 1
    v7 := v1 ≤ v2
    if v7 goto L
```

(c)

```
    v2 := &A + (n−1) × sizeof(record)
        -- may take >1 instructions
    v3 := sizeof(record)
    v5 := &A
L:  *v5 := 0
    v5 := v5 + v3
    v7 := v5 ≤ v2
    if v7 goto L
```

(d)

Figure 17.11 **Code improvement of induction variables.** High-level pseudocode source is shown in (a). Target code prior to induction variable optimizations is shown in (b). In (c) we have performed strength reduction on v5, the array index, and eliminated v4, at which point v5 no longer depends on v1 (i). In (d) we have modified the end test to use v5 instead of v1, and have eliminated v1.

EXAMPLE 17.25

Result of induction variable optimization

For our `combinations` example, the code resulting from induction variable optimizations appears in Figure C-17.12. Two induction variables—the array pointers v26 and v34—have undergone strength reduction, eliminating the need for v25, v31, and v33. Similarly v18 has been made independent of v17, eliminating the need for v16. A fifth induction variable—v38—has been eliminated by replacing its single use (the right-hand side of a register–register move) with the addition that computed it. We assume that a repeat of local redundancy elimination in Block 1 has allowed the initialization of v34 to capitalize on the value known to reside in v9.

For presentation purposes, we have also calculated the division operation directly into v13, allowing us to eliminate v21 and its later assignment into v13. A real compiler would probably not make this change until the register allocation phase of compilation, when it would verify that the previous value in v13 is dead at the time of the division (v21 is not an induction variable; its progression of values is not sufficiently simple). Making the change now eliminates the last redun-

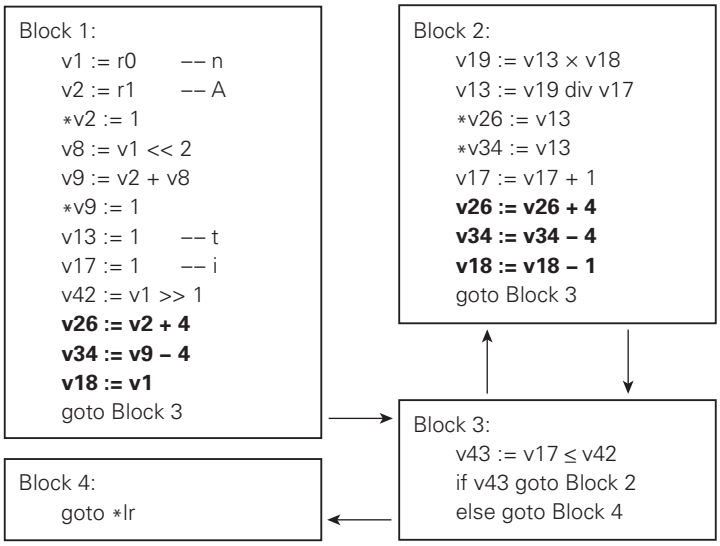


Figure 17.12 Control flow graph for the `combinations` subroutine after optimizing induction variables. Registers v26 and v34 have undergone strength reduction, allowing v25, v31, and v33 to be eliminated. Registers v38 and v21 have been merged into v17 and v13. The update to v18 has also been simplified, allowing v16 to be eliminated.

dant instruction in the block, and allows us to discuss instruction scheduling in comparative isolation from other issues. ■

17.6 Instruction Scheduling

In the example compiler structure of Figure C-17.1, the next phase after loop optimization is target code generation. As noted in Chapter 15, this phase linearizes the control flow graph and replaces the instructions of the medium-level intermediate form with target machine instructions. The replacements are often driven by an automatically generated pattern-matching algorithm. We will continue to employ our pseudo-assembly "instruction set," so linearization will be the only change we see. Specifically, we will assume that the blocks of the program are concatenated in the order suggested by their names. Control will "fall through" from Block 2 to Block 3, and from Block 3 to Block 4 in the last iteration of the loop.

We will perform two rounds of instruction scheduling separated by register allocation. Given our use of pseudo-assembly, we won't consider peephole optimization in any further detail. In Section C-17.7, however, we will consider additional forms of code improvement for loops that could be applied *prior* to target code generation. We delay discussion of these because the need for them will be clearer after considering instruction scheduling.

On a pipelined machine—particularly one that always executes instructions in program order—performance depends critically on the extent to which the compiler is able to keep the pipeline full. As explained in Section C-5.5.1, delays may result when an instruction (1) needs a functional unit still in use by an earlier instruction, (2) needs data still being computed by an earlier instruction, or (3) cannot even be selected for execution until the outcome or target of a branch has been determined. In this section we consider cases (1) and (2), which can be addressed by reordering instructions within a basic block. A good solution to (3) requires branch prediction, generally with hardware assist. A compiler can solve the subproblem of filling branch delays in a more or less straightforward fashion [Muc97, Sec. 17.1.1].

EXAMPLE 17.26
Remaining pipeline delays

If we examine the body of the loop in our `combinations` example, we find that the optimizations described thus far have transformed Block 2 from the 30 instruction sequence of Figure C-17.3 into the eight-instruction sequence of Figure C-17.12 (not counting the final `goto`s). Unfortunately, on a pipelined machine without instruction reordering, this code is still distinctly suboptimal. In particular, the results of the second and third instructions are used immediately, but the results of multiplies and divides are commonly not available for several cycles. If we assume four-cycle delays, then our block will take 16 cycles to execute. ■

Dependence Analysis

To schedule instructions to make better use of the pipeline, we first arrange them into a directed acyclic graph (DAG), in which each node represents an instruction, and each arc represents a *dependence*,[5] as described in Section C-5.5.1. Most arcs will represent *flow* dependences, in which one instruction uses a value produced by a previous instruction. A few will represent *anti*-dependences, in which a later instruction overwrites a value read by a previous instruction. In our example, these will correspond to updates of induction variables. If we were performing instruction scheduling after architectural register allocation, then uses of the same register for independent values could increase the number of anti-dependences, and could also induce so-called *output* dependences, in which a later instruction overwrites a value written by a previous instruction. Anti- and output dependences can be hidden on many machines by hardware register renaming (Section C-5.4.3).

EXAMPLE 17.27
Value dependence DAG

Because common subexpression analysis has eliminated all of the loads and stores of `i`, `n`, and `t` in the `combinations` subroutine, and because there are no loads of elements of `A` (only stores), dependence analysis in our example will be dealing solely with values in registers. In general we should need to deal with values in memory as well, and to rely on alias analysis to determine when two instructions might access the same location, and therefore share a dependence. On a target

---

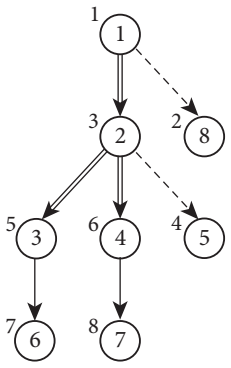
5 What we are discussing here is a *dependence DAG*. It is related to, but distinct from, the expression DAG mentioned in Section C-17.3. In particular, the dependence DAG is constructed *after* the assignment of virtual registers to expressions, and its nodes represent instructions, rather than variables and operators.

Block 2:

```
1.  v19 := v13 × v18
    —
    —
    —
    —
2.  v13 := v19 div v17
    —
    —
    —
    —
3.  *v26 := v13
4.  *v34 := v13
5.  v17 := v17 + 1
6.  v26 := v26 + 4
7.  v34 := v34 – 4
8.  v18 := v18 – 1
    -- fall through to Block 3
```

Scheduled:

```
v19 := v13 × v18
v18 := v18 – 1
—
—
—
v13 := v19 div v17
v17 := v17 + 1
—
—
—
*v26 := v13
*v34 := v13
v26 := v26 + 4
v34 := v34 – 4
```



Block 3:

```
    v43 := v17 ≤ v42
    if v43 goto Block 2
    -- else fall through to Block 4
```

(same)

Figure 17.13 **Dependence DAG for Block 2 of Figure C-17.12, together with pseudocode for the entire loop, both before (left) and after (right) instruction scheduling.** Circled numbers in the DAG correspond to instructions in the original version of the loop. Smaller adjacent numbers give the schedule order in the new loop. Solid arcs indicate flow dependences; dashed arcs indicate anti-dependences. Double arcs indicate pairs of instructions that must be separated by four additional instructions in order to avoid pipeline delays on our hypothetical machine. Delays are shown explicitly in Block 2. Unless we modify the array indexing code (Exercise C-17.20), only two instructions can be moved.

machine with condition codes (i.e., most machines today—see Section C-5.3), we should need to model these explicitly, tracking flow, anti-, and output dependences.

The dependence DAG for Block 2 of our `combinations` example appears in Figure C-17.13. In this case the DAG turns out to be a tree. It was generated by examining the code from top to bottom, linking each instruction i to each subsequent instruction j such that j reads a register written by i (solid arcs) or writes a register read by i (dashed arcs).

Any topological sort of a dependence DAG (i.e., any enumeration of the nodes in which each node appears before its children) will represent a correct schedule. Ideally we should like to choose a sort that minimizes overall delay. As with many aspects of code improvement, this task is NP-hard, so practical techniques rely upon heuristics.

To capture timing information, we define a function $latency(i, j)$ that returns the number of cycles that must elapse between the scheduling of instructions i

and j if j is to run after i in the same pipeline without stalling. (To maintain machine independence, this portion of the code improver must be driven by tables of machine characteristics; those characteristics must not be "hard-coded.") Nontrivial latencies can result from data dependences or from conflicts for use of some physical resource, such as an incompletely pipelined functional unit. We will assume in our example that all units are fully pipelined, so all latencies are due to data dependences.

We now traverse the DAG from the roots down to the leaves. At each step we first determine the set of *candidate* nodes: those for which all parents have been scheduled. For each candidate i we then use the *latency* function with respect to already-scheduled nodes to determine the earliest time at which i could execute without stalling. We also precalculate the maximum over all paths from i to a leaf of the sums of the latencies on arcs; this gives us a lower bound on the time that will be required to finish the basic block after i has been scheduled. In our examples we will use the following three heuristics to choose among candidate nodes:

1. Favor nodes that can be started without stalling.
2. If there is a tie, favor nodes with the maximum delay to the end of the block.
3. If there is still a tie, favor the node that came first in the original source code (this strategy leads to more intuitive assembly language, which can be helpful in debugging).

Other possible scheduling heuristics include:

- Favor nodes that have a large number of children in the DAG (this increases flexibility for future iterations of the scheduling algorithm).
- Favor nodes that are the final use of a register (this reduces register pressure).
- If there are multiple pipelines, favor nodes that can use a pipeline that has not received an instruction recently.

If our target machine has multiple pipelines, then we must keep track for each instruction of the pipeline we think it will use, so we can distinguish between candidates that can start in the current cycle and those that cannot start until the next. (Imprecise machine models, cache misses, or other unpredictable delays may cause our guess to be wrong some of the time.)

EXAMPLE 17.28
Result of instruction scheduling

Unfortunately, our example DAG leaves very little room for choice. The only possible improvements are to move Instruction 8 into one of the multiply or divide delay slots and Instruction 5 into one of the divide delay slots, reducing the total cycle count of Block 2 from 16 to 14. If we assume (1) that our target machine correctly predicts a backward branch at the bottom of the loop, and (2) that we can replicate the first instruction of Block 2 into a nullifying delay slot of the branch, then we incur no additional delays in Block 3 (except in the last iteration). The overall duration of the loop is therefore 18 cycles per iteration before scheduling, 16 cycles per iteration after scheduling—an improvement of 11%. In Section C-17.7 we will consider other versions of the block, in which rescheduling yields significantly faster code. ■

As noted near the end of Section C-17.1, we shall probably want to repeat instruction scheduling after global code improvement and register allocation. If there are times when the number of virtual registers with useful values exceeds the number of architectural registers on the target machine, then we shall need to generate code to *spill* some values to memory and load them back in again later. Rescheduling will be needed to handle any delays induced by the loads.

CHECK YOUR UNDERSTANDING

19. What is a *loop invariant*? A *reaching definition*?
20. Why might it sometimes be unsafe to hoist an invariant out of a loop?
21. What are *induction variables*? What is *strength reduction*?
22. What is *control flow analysis*? Why is it less important than it used to be?
23. What is *register pressure*? *Register spilling*?
24. Is instruction scheduling a machine-independent code improvement technique? Explain.
25. Describe the creation and use of a *dependence DAG*. Explain the distinctions among *flow*, *anti-*, and *output* dependences.
26. Explain the tension between instruction scheduling and register allocation.
27. List several heuristics that might be used to prioritize instructions to be scheduled.

17.7 Loop Improvement II

As noted in Section C-17.5, code improvements that improve the speed of loops are particularly important, because loops are where most programs spend most of their time. In this section we consider transformations that improve instruction scheduling by restructuring a loop body to include portions of more than one iteration of the original loop, and that manipulate multiply nested loops to improve cache performance or increase opportunities for parallelization. Extensive coverage of loop transformations and dependence theory can be found in Allen and Kennedy's text [AK02].

17.7.1 Loop Unrolling and Software Pipelining

Loop *unrolling* is a transformation that embeds two or more iterations of a source-level loop in a single iteration of a new, longer loop, allowing the scheduler to intermingle the instructions of the original iterations. If we unroll two iterations of

EXAMPLE 17.29
Result of loop unrolling

our `combinations` example we obtain the code of Figure C-17.14. We have used separate names (here starting with the letter 't') for registers written in the initial half of the loop. This convention minimizes anti- and output dependences, giving us more latitude in scheduling. In an attempt to minimize loop overhead, we have also recognized that the array pointer induction variables (v26 and v34) need only be updated once in each iteration of the loop, provided that we use displacement addressing in the second set of store instructions. The new instructions added to the end of Block 1 cover the case in which n div 2, the number of iterations of the original loop, is not an even number.

Again assuming that the branch in Block 3 can be scheduled without delays, the total time for our unrolled loop (prior to scheduling) is 32 cycles, or 16 cycles per iteration of the original loop. After scheduling, this number is reduced to 12 cycles per iteration of the original loop. Unfortunately, eight cycles (four per original iteration) are still being lost to stalls.

EXAMPLE 17.30
Result of software pipelining

If we unroll the loop three times instead of two (see Exercise C-17.21), we can bring the cost (with rescheduling) down to 11.3 cycles per original iteration, but this is not much of an improvement. The basic problem is illustrated in the top half of Figure C-17.15. In the original version of the loop, the two store instructions cannot begin until after the divide delay. If we unroll the loop, then instructions of the internal iterations can be intermingled, but six cycles of "shut-down" cost (four delay slots and two stores) are still needed after the final divide.

A *software-pipelined* version of our `combinations` subroutine appears schematically in the bottom half of Figure C-17.15, and as a control flow graph in Figure C-17.16. The idea is to build a loop whose body comprises portions of several consecutive iterations of the original loop, with no internal start-up or shut-down cost. In our example, each iteration of the software-pipelined loop contributes to three separate iterations of the original loop. Within each new iteration (shown between vertical bars) nothing needs to wait for the divide to complete. To avoid delays, we have altered the code in several ways. First, because each iteration of the new loop contributes to several iterations of the original loop, we must ensure that there are enough iterations to run the new loop at least once (this is the purpose of the test in the new Block 1). Second, we have preceded and followed the loop with code to "prime" and "flush" the "pipeline": to execute the early portions of the first iteration and the final portions of the last few. As we did when unrolling the loop, we use a separate name (t13 in this case) for any register written in the new "pipeline flushing" code. Third, to minimize the amount of priming required we have initialized v26 and v34 one slot before their original positions, so that the first iteration of the pipelined loop can "update" them as part of a "zero-th" original iteration. Finally, we have dropped the initialization of v13 in Block 1: our priming code has left that register dead at the end of the block. (Live variable analysis on virtual registers could have been used to discover this fact.)

Both the original and pipelined versions of the loop carry five nonconstant values across the boundary between iterations, but one of these has changed identity: whereas the original loop carried the result of the divide around to the next multiply

Block 1:
… –– code from Block 1, figure 16.11
v44 := v42 & 01
if !v44 goto Block 3
–– else fall through to Block 1a

Block 1a:
*v26 := 1
*v34 := 1
v17 := 2
v26 := v26 + 4
v34 := v34 – 4
v18 := v18 – 1
goto Block 3

Block 2:

		Scheduled:
1.	t19 := v13 × v18	t19 := v13 × v18
	—	t18 := v18 – 1
	—	t17 := v17 + 1
	—	v18 := t18 – 1
	—	—
2.	t13 := t19 div v17	t13 := t19 div v17
	—	v17 := t17 + 1
	—	—
	—	—
	—	—
3.	*v26 := t13	v19 := t13 × t18
4.	*v34 := t13	*v26 := t13
5.	t17 := v17 + 1	*v34 := t13
6.	v26 := v26 + 8	v26 := v26 + 8
7.	v34 := v34 – 8	v34 := v34 – 8
8.	t18 := v18 – 1	v13 := v19 div t17
9.	v19 := t13 × t18	—
	—	—
	—	—
	—	—
	—	*(v26–4) := v13
10.	v13 := v19 div t17	*(v34+4) := v13
	—	
	—	
	—	
	—	
11.	*(v26–4) := v13	
12.	*(v34+4) := v13	
13.	v17 := t17 + 1	
14.	v18 := t18 – 1	

–– fall through to Block 3

Block 3:
v43 := v17 ≤ v42 (same)
if v43 goto Block 2
–– else fall through to Block 4

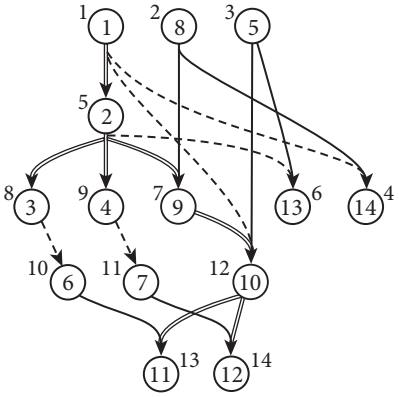


Figure 17.14 **Dependence DAG for Block 2 of the `combinations` subroutine after unrolling two iterations of the body of the loop.** Also shown is linearized pseudocode for the entire loop, both before (left) and after (right) instruction scheduling. New instructions added to the end of Block 1 cover the case in which the number of iterations of the original loop is not a multiple of two.

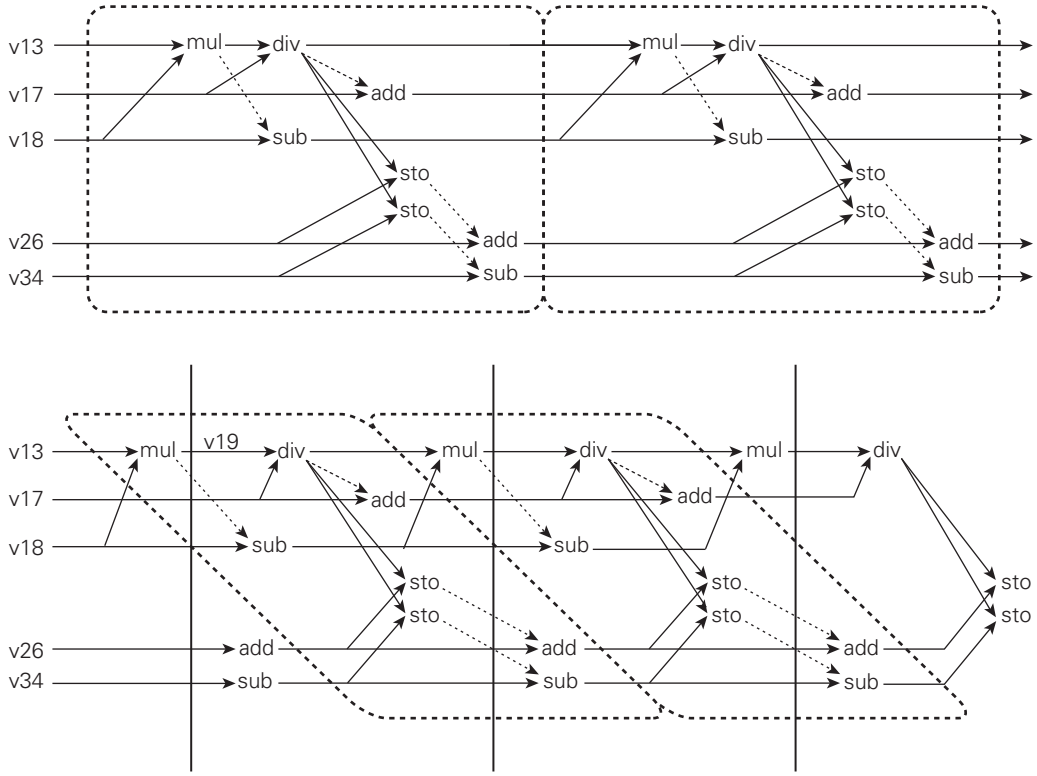


Figure 17.15 **Software pipelining.** The top diagram illustrates the execution of the original (nonpipelined) loop. In the bottom diagram, each iteration of the original loop has been spread across three iterations of the pipelined loop. Iterations of the original loop are enclosed in a dashed-line box; iterations of the pipelined loop are separated by solid vertical lines. In the bottom diagram we have also shown the code to prime the pipeline prior to the first iteration, and to flush it after the last.

in register v13, the pipelined loop carries the result of the multiply forward to the divide in register v19. In more complicated loops it may be necessary to carry two or even three versions of a single register (corresponding to two or more iterations of the original loop) across the boundary between iterations of the pipelined loop. We must invent new virtual registers (similar to the new t13 and to the t registers in the unrolled version of the `combinations` example) to hold the extra values. In such a case software pipelining has the side effect of increasing register pressure.

Each of the instructions in the loop of the pipelined version of the `combinations` subroutine can proceed without delay. The total number of cycles per iteration has been reduced to 10. We can do even better if we combine loop unrolling and software pipelining. For example, by embedding two multiply–divide pairs in each iteration (drawn, with their accompanying instructions, from four iterations of the original loop, rather than just three), we can update the array

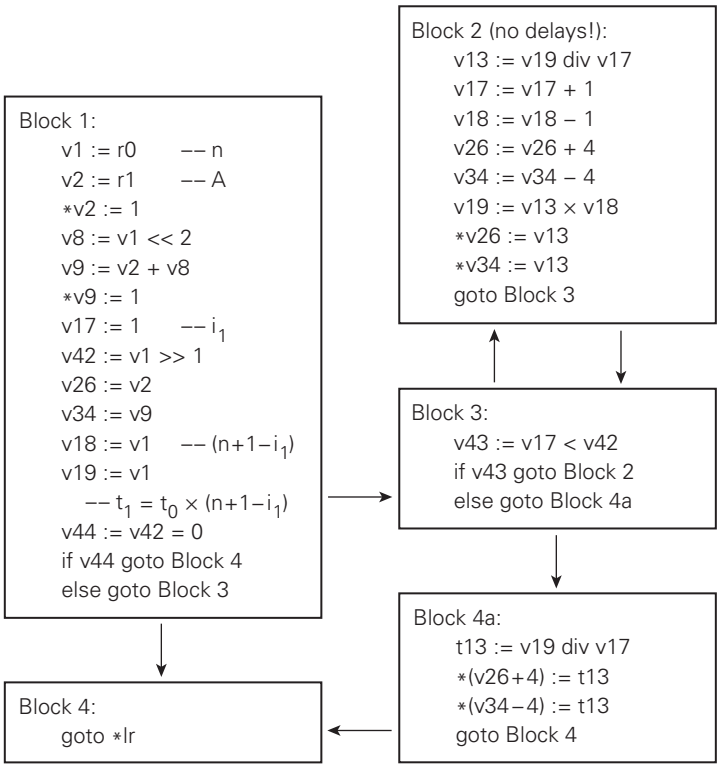


Figure 17.16 Control flow graph for the `combinations` subroutine after software pipelining. The additional code and test at the end of Block 1, the change to the test in Block 3 (< instead of ≤), and the new block (4a) make sure that there are enough iterations to accommodate the pipeline, prime it with the beginnings of the initial iteration, and flush the end of the final iteration. Suffixes on variable names in the comments in Block 1 refer to loop iterations: t_1 is the value of t in the first iteration of the loop; t_0 is a "zero-th" value used to prime the pipeline.

pointers and check the termination condition half as often, for a net of only eight cycles per iteration of the original loop (see Exercise C-17.22).

To summarize, loop unrolling serves to reduce loop overhead, and can also increase opportunities for instruction scheduling. Software pipelining does a better job of facilitating scheduling, but does not address loop overhead. A reasonable code improvement strategy is to unroll loops until the per-iteration overhead falls below some acceptable threshold of the total work, then employ software pipelining if necessary to eliminate scheduling delays.

17.7.2 Loop Reordering

The code improvement techniques that we have considered thus far have served two principal purposes: to eliminate redundant or unnecessary instructions, and to

minimize stalls on a pipelined machine. Two other goals have become increasingly important over the years. First, as improvements in processor speed have outstripped improvements in memory latency, it has become increasingly important to minimize cache misses. Second, for parallel machines, it has become important to identify sections of code that can execute concurrently. As with other optimizations, the largest benefits come from changing the behavior of loops. We touch on some of the issues here; suggestions for further reading can be found at the end of the chapter.

Cache Optimizations

EXAMPLE 17.31
Loop interchange

Probably the simplest example of cache optimization can be seen in code that traverses a multidimensional matrix (array):

```
for i := 1 to n
    for j := 1 to n
        A[i, j] := 0
```

If A is laid out in row-major order, and if each cache line contains m elements of A, then this code will suffer n^2/m cache misses. On the other hand, if A is laid out in column-major order, and if the cache is too small to hold n lines of A, then the code will suffer n^2 misses, fetching the entire array from memory m times. The difference can have an enormous impact on performance. A loop-reordering compiler can improve this code by *interchanging* the nested loops:

```
for j := 1 to n
    for i := 1 to n
        A[i, j] := 0
```

EXAMPLE 17.32
Loop tiling (blocking)

In more complicated examples, interchanging loops may improve locality of reference in one array but worsen it in others. Consider this code to transpose a two-dimensional matrix:

```
for j := 1 to n
    for i := 1 to n
        A[i, j] := B[j, i]
```

If A and B are laid out the same way in memory, one of them will be accessed along cache lines, but the other will be accessed across them. In this case we may improve locality of reference by *tiling* or *blocking* the loops:

```
for it := 1 to n by b
    for jt := 1 to n by b
        for i := it to min(it + b – 1, n)
            for j := jt to min(jt + b – 1, n)
                A[i, j] := B[j, i]
```

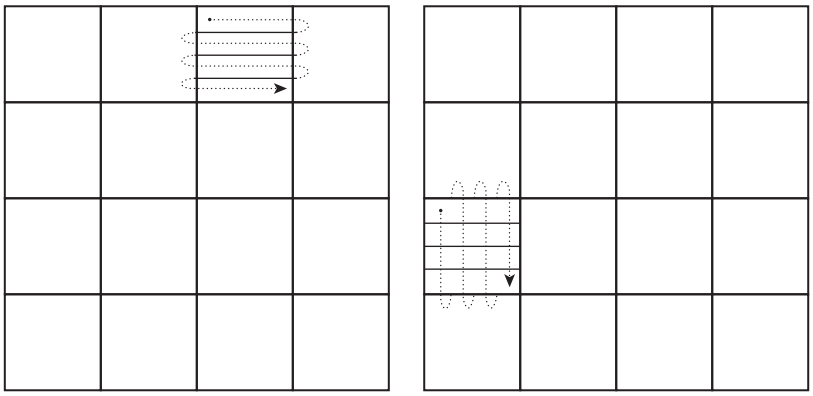


Figure 17.17 Tiling (blocking) of a matrix operation. As long as one tile of A and one tile of B can fit in the cache simultaneously, only one access in m will cause a cache miss (where m is the number of elements per cache line).

Here the min calculations cover the possibility that b does not divide n evenly. They can be dropped if n is known to be a multiple of b. Alternatively, if we are willing to replicate the code inside the innermost loop, then we can generate different code for the final iteration of each loop (Exercise C-17.25).

The new code iterates over $b \times b$ blocks of A and B, one in row-major order, the other in column-major order, as shown in Figure C-17.17. If we choose b to be a multiple of m such that the cache can hold two $b \times b$ blocks of data simultaneously, then both A and B will suffer only one cache miss per m array elements, fetching everything from memory exactly once.[6] Tiling is useful in a wide variety of algorithms on multidimensional arrays. Exercise C-17.23 considers matrix multiplication.

Two other transformations that may sometimes improve cache locality are loop *distribution* (also called *fission* or *splitting*), and its inverse, loop *fusion* (also known as *jamming*). Distribution splits a single loop into multiple loops, each of which contains some fraction of the statements of the original loop. Fusion takes separate loops and combines them.

EXAMPLE 17.33
Loop distribution

Consider, for example, the following code to reorganize a pair of arrays:

```
for i := 0 to n−1
    A[i] := B[M[i]];
    C[i] := D[M[i]];
```

6 Although A is being written, not read, the hardware will fetch each line of A from memory on the first write to the line, so that the single modified element can be updated within the cache. The hardware has no way to know that the entire line will be modified before it is written back to memory.

Here M defines a mapping from locations in B or D to locations in A or C. If either B or D, but not both, can fit into the cache at once, then we may get faster code through distribution:

```
for i := 1 to n
    A[i] := B[M[i]];
for i := 1 to n
    C[i] := D[M[i]];
```

EXAMPLE 17.34
Loop fusion

On the other hand, in the following code, separate loops may lead to *poorer* locality:

```
for i := 1 to n
    A[i] := A[i] + c
for i := 1 to n
    if A[i] < 0 then A[i] := 0
```

If A is too large to fit in the cache in its entirety, then these loops will fetch the entire array from memory twice. If we fuse them, however, we need only fetch A once:

```
for i := 1 to n
    A[i] := A[i] + c
    if A[i] < 0 then A[i] := 0
```

If two loops do not have identical bounds, it may still be possible to fuse them if we transform induction variables or *peel* some constant number of iterations off of one of the loops.

EXAMPLE 17.35
Obtaining a perfect loop nest

Loop distribution may serve to facilitate other transformations (e.g., loop interchange) by transforming an "imperfect" loop nest into a "perfect" one:

```
for i := 1 to n
    A[i] := A[i] + c
    for j := 1 to n
        B[i, j] := B[i, j] × A[i]
```

This nest is called imperfect because the outer loop contains more than just the inner loop. Distribution yields two outermost loops:

```
for i := 1 to n
    A[i] := A[i] + c
for i := 1 to n
    for j := 1 to n
        B[i, j] := B[i, j] × A[i]
```

The nested loops are now perfect, and can be interchanged if desired.

In keeping with our earlier discussions of loop optimizations, we note that loop distribution can reduce register pressure, while loop fusion can reduce loop overhead.

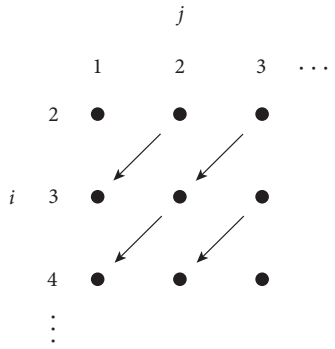
Loop Dependences

When reordering loops, we must be extremely careful to respect all data dependences. Of particular concern are so-called *loop-carried* dependences, which constrain the orders in which iterations can occur. Consider, for example, the following:

EXAMPLE 17.36
Loop-carried dependences

```
for i := 2 to n
    for j := 1 to n−1
        A[i, j] := A[i, j] − A[i−1, j+1]
```

Here the calculation of A[i, j] in iteration (i, j) depends on the value of A[i−1, j+1], which was calculated in iteration $(i-1, j+1)$. This dependence is often represented by a diagram of the *iteration space*:



The *i* and *j* dimensions in this diagram represent loop indices, *not* array subscripts. The arcs represent the loop-carried flow dependence.

If we wish to interchange the *i* and *j* loops of this code (e.g., to improve cache locality), we find that we cannot do it, because of the dependence: we would end up trying to write A[i, j] before we had written A[i−1, j+1]. ■

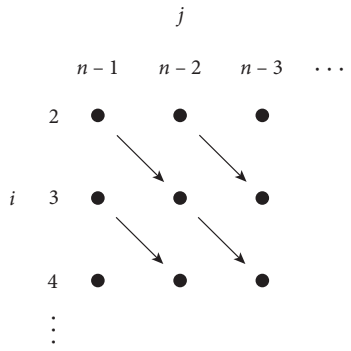
To analyze loop-carried dependences, high-performance optimizing compilers use symbolic mathematics to characterize the sets of index values that may cause the subscript expressions in different array references to evaluate to the same value. Compilers differ somewhat in the sophistication of this analysis. Most can handle linear combinations of loop indices. None, of course, can handle all expressions, since equivalence of general formulae is undecidable. When unable to fully characterize subscripts, a compiler must conservatively assume the worst, and rule out transformations whose safety cannot be proven.

In many cases a loop with a fully characterized dependence that precludes a desired transformation can be modified in a way that eliminates the dependence. In Example C-17.36 above, we can *reverse* the order of the *j* loop without violating the dependence:

EXAMPLE 17.37
Loop reversal and interchange

```
for i := 2 to n
    for j := n−1 to 1 by−1
        A[i, j] := A[i, j] − A[i−1, j+1]
```

This change transforms the iteration space:



And now the loops can safely be interchanged:

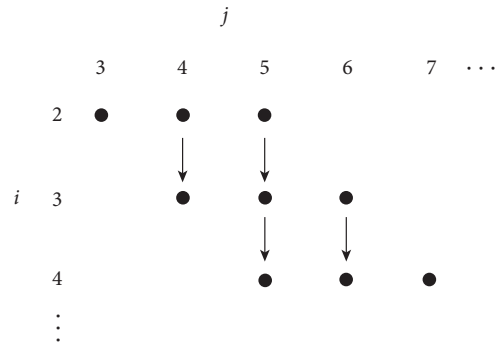
```
for j := n−1 to 1 by−1
    for i := 2 to n
        A[i, j] := A[i, j] − A[i−1, j+1]
```

EXAMPLE 17.38
Loop skewing

Another transformation that sometimes serves to eliminate a dependence is known as loop *skewing*. In essence, it reshapes a rectangular iteration space into a parallelogram, by adding the outer loop index to the inner one, and then subtracting from the appropriate subscripts:

```
for i := 2 to n
    for j := i+1 to i+n−1
        A[i, j−i] := A[i, j−i] − A[i−1, j+1−i]
```

A moment's consideration will reveal that this code accesses the exact same elements as before, in the exact same order. Its iteration space, however, looks like this:



Now the loops can safely be interchanged. The transformation is complicated by the need to accommodate the sloping sides of the iteration space. To avoid using min and max functions, we can divide the space into two triangular sections, each of which has its own loop nest:

```
for j := 3 to n+1
    for i := 2 to j−1
        A[i, j−i] := A[i, j−i] − A[i−1, j+1−i]
for j := n+2 to 2×n−1
    for i := j−n+1 to n
        A[i, j−i] := A[i, j−i] − A[i−1, j+1−i]
```

Skewing has led to more complicated code than did reversal of the *j* loop, but it could be used in the presence of other dependences that would eliminate reversal as an option.

Several other loop transformations, including distribution, can also be used in certain cases to eliminate loop-carried dependences, allowing us to apply techniques that improve cache locality or (as discussed immediately below) enable us to execute code in parallel on a vector or multicore machine. Of course, no set of transformations can eliminate all dependences; some code simply can't be improved.

Parallelization

Loop iterations (at least in nonrecursive programs) constitute the principal source of operations that can execute in parallel. Ideally, one needs to find *independent* loop iterations: ones with no loop-carried dependences. (In some cases, iterations can also profitably be executed in parallel even if they have dependences, so long as they synchronize their operations appropriately.) In Example 13.8 and Section 13.4.6 we considered loop constructs that allow the programmer to specify parallel execution. Even in languages without such special constructs, a compiler can often *parallelize* code by identifying—or creating—loops with as few loop-carried dependences as possible. The transformations described above are valuable tools in this endeavor.

Given a parallelizable loop, the compiler must consider several other issues in order to ensure good performance. One of the most important of these is the *granularity* of parallelism. For a very simple example, consider the problem of "zeroing out" a two-dimensional array, here indexed from 0 to $n-1$ in each dimension, and laid out in row-major order:

EXAMPLE 17.39
Coarse-grain parallelization

```
for i := 0 to n−1
    for j := 0 to n−1
        A[i, j] := 0
```

On a machine comprising several general-purpose processor cores, we would probably parallelize the outer loop:

```
−− on processor core pid:
for i := (n/p × pid) to (n/p × (pid + 1) − 1)
    for j := 1 to n
        A[i, j] := 0
```

Here we have given each core a band of rows to initialize. We have assumed that cores are numbered from 0 to $p-1$, and that p divides n evenly.

The strategy on a vector machine is very different. Such a machine includes a collection of v-element vector registers, and instructions to load, store, and compute on vector data. The vector instructions are deeply pipelined, allowing the machine to exploit a high degree of *fine-grain* parallelism. To satisfy the hardware, the compiler needs to parallelize *inner* loops:

EXAMPLE 17.40
Strip mining

```
for i := 0 to n−1
    for j := 0 to n−1 by v
        A[i, j:j+v−1] := 0        −− vector operation
```

Here the notation A[i, j:j+v−1] represents a v-element *slice* of A. The constant v should be set to the length of a vector register (which we again assume divides n evenly). The code transformation that extracts v-element operations from longer loops is known as *strip mining*. It is essentially a one-dimensional form of tiling. ■

Other issues of importance in parallelizing compilers include *communication* and *load balance*. Just as locality of reference reduces communication between the cache and main memory on a single-core machine, locality in parallel programs reduces communication among cores and between the cores and memory. Optimizations similar to those employed to reduce the number of cache misses on a single-core machine can be used to reduce communication traffic on a multicore machine.

Load balance refers to the division of labor among processor cores. If we divide the work of a program among 16 cores, we shall obtain a speedup of close to 16 only if each core takes the same amount of time to do its work. If we accidentally assign 5% of the work to each of 15 cores and 25% of the work to the 16th, we are likely to see a speedup of no more than 4×. For simple loops it is often possible to predict performance accurately enough to divide the work among cores at compile time. For more complex loops, in which different iterations perform different amounts of work or have different cache behavior, it is often better to generate *self-scheduled* code, which divides the work up at run time. In its simplest form, self scheduling creates a "bag of tasks," as described in Section 13.2. Each task consists of a set of loop iterations. The number of such tasks is chosen to be significantly larger than the number of cores. When finished with a given task, a core goes back to the bag to get another.

17.8 Register Allocation

In a simple compiler with no global optimizations, register allocation can be performed independently in every basic block. To avoid the obvious inefficiency of storing frequently accessed variables to memory at the end of many blocks, and reading them back in again in others, simple compilers usually apply a set of heuristics to identify such variables and allocate them to registers over the life of a subroutine. Obvious candidates for a dedicated register include loop indices and scalar local variables and parameters.

It has been known since the early 1970s that register allocation is equivalent to the NP-hard problem of graph coloring. Following the work of Chaitin et al. [CAC+81], heuristic (nonoptimal) implementations of graph coloring have become a common approach to register allocation in aggressive optimizing compilers. We describe the basic idea here; for more detail see Cooper and Torczon's text [CT11, Chap. 13].

The first step is to identify virtual registers that *cannot* share an architectural register, because they contain values that are live concurrently. To accomplish this step we use reaching definitions data flow analysis (Section C-17.5.1). For the software-pipelined version of our `combinations` subroutine (Figure C-17.16), we can chart the *live ranges* of the virtual registers as shown in Figure C-17.18. Note that the live range of v19 spans the backward branch at the end of Block 2; though typographically disconnected it is contiguous in time.

EXAMPLE 17.41
Live ranges of virtual registers

Given these live ranges, we construct a *register interference graph*. The nodes of this graph represent virtual registers. Registers v*i* and v*j* are connected by an arc if they are simultaneously live. The interference graph corresponding to Figure C-17.18 appears in Figure C-17.19. The problem of mapping virtual registers onto the smallest possible number of architectural registers now amounts to finding a *minimal coloring* of this graph: an assignment of "colors" to nodes such that no arc connects two nodes of the same color.

EXAMPLE 17.42
Register coloring

In our example, we can find one of several optimal solutions by inspection. The six registers in the center of the figure constitute a clique (a completely connected subgraph); each must be mapped to a separate architectural register. Moreover there are three cases—registers v1 and v19, v2 and v26, and v9 and v34—in which one register is copied into the other somewhere in the code, but the two are never simultaneously live. If we use a common architectural register in each of these cases then we can eliminate the copy instructions; this optimization is known as *live range coalescing*. Registers v13, v43, and v44 are connected to every member of the clique, but not to each other; they can share a seventh architectural register. Register v8 is connected to v1, v2, and v9, but not to anything else; we have arbitrarily chosen to have both it and t13 share with the three registers on the right.

Final code for the `combinations` subroutine appears in Figure C-17.20. We have left v1/v19 and v2/v26 in r0 and r1, the registers in which their initial values were passed. Because our subroutine is a leaf, these registers are never needed for other arguments. Following Arm conventions (Section C-5.4.5), we have used registers r2 through r6 as additional temporary registers. Of these, r4 through r6 are callee-saves, so we have pushed their old values in the prologue and popped them in the epilogue.

EXAMPLE 17.43
Optimized `combinations` subroutine

We have glossed over two important issues. First, on almost any real machine, architectural registers are not uniform. Integer registers cannot be used for floating-point operations. Caller-saves registers should not be used for variables whose values are needed across subroutine calls. Registers that are overwritten by special instructions (e.g., byte string search on a CISC machine) should not be used to hold values that are needed across such instructions. To handle constraints of this type, the register interference graph is usually extended to contain nodes for

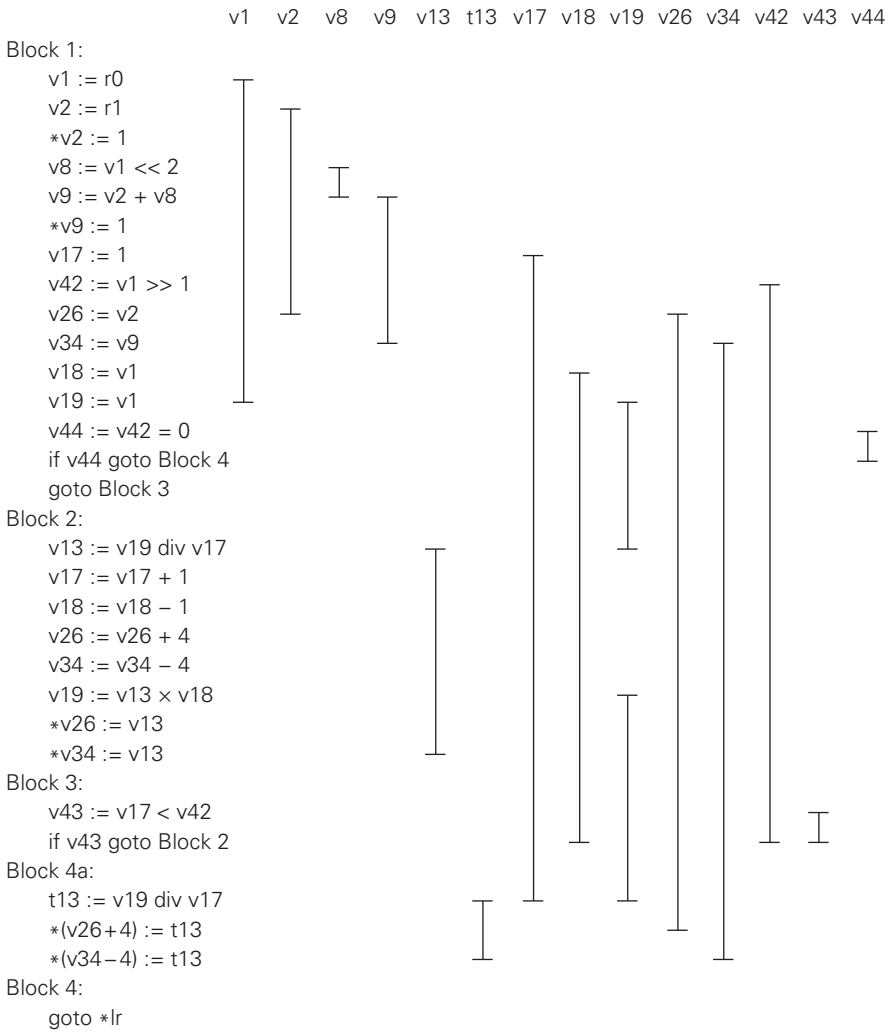


Figure 17.18 Live ranges for virtual registers in the software-pipelined version of the `combinations` subroutine (Figure C-17.16).

both virtual and architectural registers. Arcs are then drawn from each virtual register to the architectural registers to which it should not be mapped. Each architectural register is also connected to every other, to force them all to have separate colors. After coloring the resulting graph, we assign each virtual register to the architectural register of the same color. On Arm (for which we are supposedly generating code), v43 and v44 must actually be mapped to the condition codes in the processor status register (psr). The astute reader may have noticed that we did

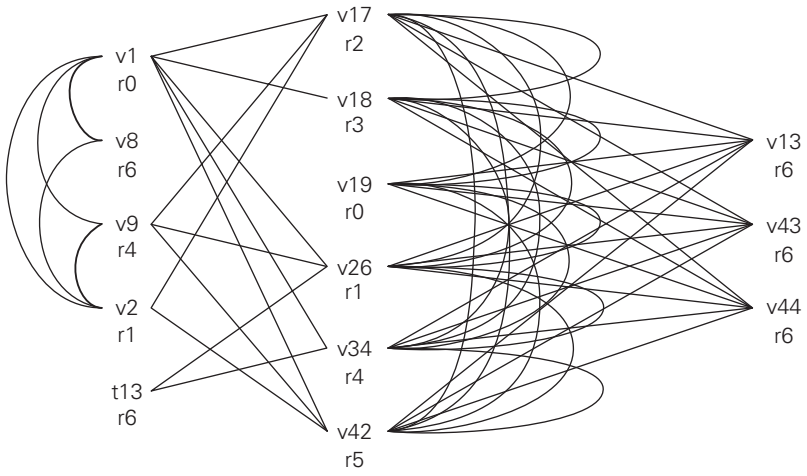


Figure 17.19 **Register interference graph for the software pipelined version of the `combinations` subroutine.** Using architectural register names, we have indicated one of several possible seven-colorings.

```
Block 1:
    push {r4, r5, r6}
    *r1 := 1
    r6 := r0 << 2
    r4 := r1 + r6
    *r4 := 1
    r2 := 1
    r5 := r0 >> 1
    r3 := r0
    cc := r5 = 0
    if cc goto Block 4
    goto Block 3
Block 2:
    r6 := r0 div r2
    r2 := r2 + 1
    r3 := r3 - 1
    r1 := r1 + 4
    r4 := r4 - 4
    r0 := r6 × r3
    *r1 := r6
    *r4 := r6
Block 3:
    cc := r2 < r5
    if cc goto Block 2
Block 4a:
    r6 := r0 div r2
    *(r1+4) := r6
    *(r4-4) := r6
Block 4:
    pop {r4, r5, r6}
    goto *lr
```

Figure 17.20 **Final code for the `combinations` subroutine, after assigning architectural registers and eliminating useless copy instructions.**

so in Figure C-17.20. In our particular example, the change has no impact on the number of colors required for the remaining virtual registers.

The second issue we've ignored is what happens when there aren't enough architectural registers to go around. In this case it will not be possible to color the interference graph. Using a variety of heuristics (which we do not cover here), the compiler chooses virtual registers whose live ranges can be *split* into two or more

subranges. A value that is live at the end of a subrange may be *spilled* (stored) to memory, and reloaded at the beginning of the subsequent subrange. Alternatively, it may be *rematerialized* by repeating the calculation that produced it (assuming the necessary operands are still available). Which is cheaper will depend on the cost of loads and stores and the complexity of the generating calculation.

It is easy to prove that with a sufficient number of range splits it is possible to color any graph, given at least three colors. The trick is to find a set of splits that keeps the cost of spills and rematerialization low. Once register allocation is complete, as noted in Sections C-17.1 and C-17.6, we shall want to repeat instruction scheduling, in order to fill any newly created load delays.

CHECK YOUR UNDERSTANDING

28. What is the difference between *loop unrolling* and *software pipelining*? Explain why the latter may increase register pressure.
29. What is the purpose of *loop interchange*? *Loop tiling* (*blocking*)?
30. What are the potential benefits of *loop distribution*? *Loop fusion*? What is *loop peeling*?
31. What does it mean for loops to be *perfectly nested*? Why are perfect loop nests important?
32. What is a *loop-carried dependence*? Describe three loop transformations that may serve in some cases to eliminate such a dependence.
33. Describe the fundamental difference between the parallelization strategy for multicore machines and the parallelization strategy for vector machines.
34. What is *self scheduling*? When is it desirable?
35. What is the *live range* of a register? Why might it not be a contiguous range of instructions?
36. What is a *register interference graph*? What is its significance? Why do production compilers depend on heuristics (rather than precise solutions) for register allocation?
37. List three reasons why it might not be possible to treat architectural registers uniformly for purposes of register allocation.

17.9 Summary and Concluding Remarks

This chapter has addressed the subject of code improvement ("optimization"). We considered several of the most important optimization techniques, including peephole optimization; local and global (procedure-level) redundancy elimination

(constant folding, constant propagation, copy propagation, common subexpression elimination); loop improvement (invariant hoisting, strength reduction or elimination of induction variables, unrolling and software pipelining, reordering for cache improvement or parallelization); instruction scheduling; and register allocation. Many others techniques, too numerous to mention, can be found in the literature or in production use.

To facilitate code improvement, we introduced several new data structures and program representations, including dependence DAGs (for instruction scheduling), static single-assignment (SSA) form (for many purposes, including global common subexpression elimination via value numbering), and the register interference graph (for architectural register allocation). For many global optimizations we made use of data flow analysis. Specifically, we employed it to identify available expressions (for global common subexpression elimination), to identify live variables (to eliminate useless stores), and to calculate reaching definitions (to identify loop invariants; also useful for finding live ranges of virtual registers). We also noted that it can be used for global constant propagation, copy propagation, conversion to SSA form, and a host of other purposes.

An obvious question for both the writers and users of compilers is: among the many possible code improvement techniques, which produce the most "bang for the buck"? For modern machines, particularly those with in-order pipelines, instruction scheduling and register allocation are definitely on the list. Significant additional benefits accrue from some sort of global register allocation, if only to avoid repeated loads and stores of loop indices and other heavily used local variables and parameters. Beyond these basic techniques, which mainly amount to making good use of the hardware, the most significant benefits in von Neumann programs come from optimizing references to arrays, particularly within loops. Most production-quality compilers (1) perform at least enough common subexpression analysis to identify redundant address calculations for arrays, (2) hoist invariant calculations out of loops, and (3) perform strength reduction on induction variables, eliminating them if possible.

As we noted in the introduction to the chapter, code improvement remains an extremely active area of research. Much of this research addresses language features and computational models for which traditional optimization techniques have not been particularly effective. Examples include alias analysis for pointers in C, static resolution of virtual method calls in object-oriented languages (to permit inlining and interprocedural optimization), streamlined communication in message-passing languages, and a variety of issues for functional and logic languages. In some cases, new programming paradigms can change the goals of code improvement. For just-in-time compilation of Java or C# programs, for example, the speed of the code improver may be as important as the speed of the code it produces. In other cases, new sources of information (e.g., feedback from run-time profiling) create new opportunities for improvement. Finally, advances in processor architecture (multiple pipelines, very wide instruction words, out-of-order execution, architecturally visible caches, speculative instructions) continue to create new challenges; processor design and compiler design are deeply interrelated.

17.10 Exercises

17.1 In Section C-17.2 we suggested replacing the instruction `r1 := r2 / 2` with the instruction `r1 := r2 >> 1`, and noted that the replacement may not be correct for negative numbers. Explain the problem. You will want to learn the difference between *logical* and *arithmetic* shift operations (see almost any assembly language manual). You will also want to consider the issue of rounding.

17.2 Prove that the division operation in the loop of the `combinations` subroutine (Example C-17.10) always produces a remainder of zero. Explain the need for the parentheses around the numerator.

17.3 Certain code improvements can sometimes be performed by the programmer, in the source-language program. Examples include introducing additional variables to hold common subexpressions (so that they need not be recomputed), moving invariant computations out of loops, and applying strength reduction to induction variables or to multiplications by powers of two. Describe several optimizations that cannot reasonably be performed by the programmer, and explain why some that could be performed by the programmer might best be left to the compiler.

17.4 In Section 6.5.1, we suggested that the loop

```
// before
for (i = low; i <= high; i++) {
    // during
}
// after
```

be translated as

```
    -- before
    i := low
    goto test
top:
    -- during
    i +:= 1
test:
    if i ≤ high goto top
    -- after
```

And indeed this is the translation we have used for the `combinations` subroutine. The following is an alternative translation:

```
    -- before
    i := low
    if i > high goto bottom
```

```
top:
    -- during
    i +:= 1
    if i ≤ high goto top
bottom:
    -- after
```

Explain why this translation might be preferable to the one we used. (Hints: Consider the number of branches, the migration of loop invariants, and opportunities to fill delay slots.)

17.5 Beginning with the translation of the previous exercise, reapply the code improvements discussed in this chapter to the `combinations` subroutine.

17.6 Give an example in which the numbered heuristics listed under "Dependence Analysis" in Section C-17.6 do not lead to an optimal code schedule.

17.7 Show that forward data flow analysis can be used to verify that a variable is assigned a value on every possible control path leading to a use of that variable (this is the notion of *definite assignment*, described in Section 6.1.3).

17.8 In Sidebar 16.3, we noted two additional properties (other than definite assignment) that a Java Virtual Machine must verify in order to protect itself from potentially erroneous bytecode. On every possible path to a given statement S (a) every variable read in S must have the same type (which must of course be consistent with operations in S), and (b) the operand stack must have the same current depth, and must not overflow or underflow in S. Describe how data flow analysis can be used to verify these properties.

17.9 Show that *very busy* expressions (those that are guaranteed to be calculated on every future code path) can be detected via backward, all-paths data flow analysis. Suggest a space-saving code improvement for such expressions.

17.10 Explain how to gather information during local value numbering that will allow us to identify the sets of variables and registers that contributed to the value of each virtual register. (If the value of register vi depends on the value of register vj or of variable x, then during available expression analysis we say that $vi \in Kill_B$ if B contains an assignment to vj or x and does not contain a subsequent assignment to vi.)

17.11 Show how to strength-reduce the expression i^2 within a loop, where i is the loop index variable. You may assume that the loop step size is one.

17.12 Division is often much more expensive than addition and subtraction. Show how to replace expressions of the form i div c on the inside of a `for` loop with additions and/or subtractions, where i is the loop index variable and c is an integer constant. You may assume that the loop step size is one.

17.13 Consider the following high-level pseudocode:

```
read(n)
for i in 1 . . 100
    B[i] := n × i
    if n > 0
        A[i] := B[i]
```

The condition n > 0 is loop invariant. Can we move it out of the loop? If so, explain how. If not, explain why.

17.14 Should live variable analysis be performed before or after loop invariant elimination (or should it be done twice, before *and* after)? Justify your answer.

17.15 Starting with the naive `gcd` code of Figure 1.7, show the result of local redundancy elimination (via value numbering) and instruction scheduling.

17.16 Continuing the previous exercise, draw the program's control flow graph and show the result of global value numbering. Next, use data flow analysis to drive any appropriate global optimizations. Then draw and color the register conflict graph in order to perform global register allocation. Finally, perform a final pass of instruction scheduling. How does your code compare to the version in Example 1.2?

17.17 In Section C-17.6, we noted that hardware register renaming can often hide anti- and output dependences. Will it help in Figure C-17.13? Explain.

17.18 Consider the following code:

```
v2 := *v1
v1 := v1 + 20
v3 := *v1
—
v4 := v2 + v3
```

Show how to shorten the time required for this code by moving the update of v1 forward into the delay slot of the second load. (Assume that v1 is still live at the end.) Describe the conditions that must hold for this type of transformation to be applied, and the alterations that must be made to individual instructions to maintain correctness.

17.19 Consider the following code:

```
v5 := v2 × v36
—
—
—
—
v6 := v5 + v1
v1 := v1 + 20
```

Show how to shorten the time required for this code by moving the update of v1 backward into a delay slot of the multiply. Describe the conditions that

must hold for this type of transformation to be applied, and the alterations that must be made to individual instructions to maintain correctness.

17.20 In the spirit of the previous two exercises, show how to shorten the main loop of the `combinations` subroutine (prior to unrolling or pipelining) by moving the updates of v26 and v34 backward into delay slots. What percentage impact does this change make in the performance of the loop?

17.21 Using the code in Figures C-17.12 and C-17.14 as a guide, unroll the loop of the `combinations` subroutine three times. Construct a dependence DAG for the new Block 2. Finally, schedule the block. How many cycles does your code consume per iteration of the original (unrolled) loop? How does it compare to the software pipelined version of the loop (Figure C-17.16)?

17.22 Write a version of the `combinations` subroutine whose loop is both unrolled *and* software pipelined. In other words, build the loop body from the instructions between the left-most and right-most vertical bars of Figure C-17.15, rather than from the instructions between adjacent bars. You should update the array pointers only once per iteration. How many cycles does your code consume per iteration of the original loop? How messy is the code to "prime" and "flush" the pipeline, and to check for sufficient numbers of iterations?

17.23 Consider the following code for matrix multiplication:

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        C[i][j] = 0;
    }
}
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        for (k = 0; k < n; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Describe the access patterns for matrices `A`, `B`, and `C`. If the matrices are large, how many times will each cache line be fetched from memory? Tile the inner two loops. Describe the effect on the number of cache misses.

17.24 Consider the following simple instance of Gaussian elimination:

```
for (i = 0; i < n-1; i++) {
    for (j = i+1; j < n; j++) {
        for (k = n-1; k >= i; k--) {
            A[j][k] -= A[i][k] * A[j][i] / A[i][i];
        }
    }
}
```

(Gaussian elimination serves to triangularize a matrix. It is a key step in the solution of systems of linear equations.) What are the loop invariants in this code? What are the loop-carried dependences? Discuss how to optimize the code. Be sure to consider locality-improving loop transformations.

17.25 Modify the tiled matrix transpose of Example C-17.32 to eliminate the min calculations in the bounds of the inner loops. Perform the same modification on your answer to Exercise C-17.23.

17.11 Explorations

17.26 Investigate the back-end structure of your favorite compiler. What levels of optimization are available? What techniques are employed at each level? What is the default level? Does the compiler generate assembly language or object code?

Experiment with optimization in several program fragments. Instruct the compiler to generate assembly language, or use a disassembler or debugger to examine the generated object code. Evaluate the quality of this code at various levels of optimization.

If your compiler employs a separate assembler, compare the assembler input to its disassembled output. What optimizations, if any, are performed by the assembler?

17.27 As a general rule, a compiler can apply a program transformation only if it preserves the correctness of the code. In some circumstances, however, the correctness of a transformation may depend on information that will not be known until run time. In this case, a compiler may generate two (or more) versions of some body of code, together with a run-time check that chooses which version to use, or customizes a general, parameterized version.

Learn about the "inspector-executor" compilation paradigm pioneered by Saltz et al. [SMC91]. How general is this technique? Under what circumstances can the performance benefits be expected to outweigh the cost of the run-time check and the potential increase in code size?

17.28 Static compiler analysis can be used to check for patterns of information flow that are likely (though not certain) to constitute programming errors. Investigate the work of Guyer et al. [GL05], which performs analysis reminiscent of *taint mode* (Exploration 16.21) at compile time. In a similar vein, investigate the work of Yang et al. [YTEM04] and Chen et al. [CDW04], which use static *model checking* to catch high-level errors. What do you think of such efforts? How do they compare to taint mode or to *proof-carrying code* (Exploration 16.22)? Can static analysis be useful if it has both false negatives (errors it misses) and false positives (correct code it flags as erroneous)?

17.29 In a somewhat gloomy parody of Moore's Law, Todd Proebsting (an eminent compiler researcher formerly at Microsoft Research and now on the faculty

of the University of Arizona) once coined what he called *Proebsting's Law*: "Compiler advances double computing power every 18 years."

Survey the history of compiler technology. What have been the major innovations? Have there been important advances in areas other than speed? Is Proebsting's Law a fair assessment of the field?

17.12 Bibliographic Notes

Mainstream compiler textbooks (e.g., those of Cooper and Torczon [CT11], Grune et al. [GBJ+12], or Aho et al. [ALSU07]) are an accessible source of information on back-end compiler technology. Much of the presentation here was inspired by Muchnick's *Advanced Compiler Design and Implementation*, which contains a wealth of detailed information and citations to related work [Muc97]. Much of the leading-edge compiler research appears in the annual *ACM Conference on Programming Language Design and Implementation* (PLDI). A compendium of "best papers" from the first 20 years of this conference was published in 2004 [McK04].

Throughout our study of code improvement, we concentrated our attention on the von Neumann family of languages. Analogous techniques for functional [App91; Pey87; Pey92; WM95, Chap. 3; App97, Chap. 15; GBJ+12, Chap. 7]; object-oriented [AH95; GDDC97; WM95, Chap. 5; App97, Chap. 14; GBJ+12, Chap. 6]; and logic languages [DRSS96; FSS83; Zho96; WM95, Chap. 4; GBJ+12, Chap. 8] are an active topic of research, but are beyond the scope of this book. A key challenge in functional languages is to identify repetitive patterns of calls (e.g., tail recursion), for which loop-like optimizations can be performed. A key challenge in object-oriented languages is to predict the targets of virtual subroutine calls statically, to permit in-lining and interprocedural code improvement. The dominant challenge in logic languages is to better direct the underlying process of goal-directed search.

Local value numbering is originally due to Cocke and Schwartz [CS69]; the global algorithm described here is based on that of Alpern, Wegman, and Zadeck [AWZ88]. Chaitin et al. [CAC+81] popularized the use of graph coloring for register allocation. Cytron et al. [CFR+91] describe the generation and use of static single-assignment form. Allen and Kennedy [AK02, Sec. 12.2] discuss the general problem of alias analysis in C. Pointers have historically been the most difficult part of this analysis; Smaragdakis and Balatsouras [SB15] provide a tutorial survey. Instruction scheduling from basic-block dependence DAGs is described by Gibbons and Muchnick [GM86]. The general technique is known as *list scheduling*; explanations appear in the texts of Muchnick [Muc97, Sec. 17.1.2] and Cooper and Torczon [CT11, Sec. 12.3]. Massalin provides a delightful discussion of circumstances under which it may be desirable (and possible) to generate a truly *optimal* program [Mas87]. Several projects have expanded on this idea; see for example the work of Schkufza et al. [SSA13].

Sources of information on loop transformations and parallelization include the text of Allen and Kennedy [AK02], the classic text of Wolfe [Wol96], and

the excellent survey of Bacon, Graham, and Sharp [BGS94]. Banerjee provides a detailed discussion of loop dependence analysis [Ban97]. Rau and Fisher discuss fine-grain *instruction-level* parallelism, of the sort exploitable by vector, wide-instruction-word, or superscalar processors [RF93].