18

Loop Optimizations

loop: a series of instructions that is repeated until a terminating condition is reached

Webster's Dictionary

Loops are pervasive in computer programs, and a great proportion of the execution time of a typical program is spent in one loop or another. Hence it is worthwhile devising optimizations to make loops go faster. Intuitively, a loop is a sequence of instructions that ends by jumping back to the beginning. But to be able to optimize loops effectively we will use a more precise definition.

A *loop* in a control-flow graph is a set of nodes S including a *header* node h with the following properties:

- From any node in S there is a path of directed edges leading to h.
- There is a path of directed edges from h to any node in S.
- There is no edge from any node outside S to any node in S other than h.

Thus, the dictionary definition (from *Webster's*) is not the same as the technical definition.

Figure 18.1 shows some loops. A *loop entry* node is one with some predecessor outside the loop; a *loop exit* node is one with a successor outside the loop. Figures 18.1c, 18.1d, and 18.1e illustrate that a loop may have multiple exits, but may have only one entry. Figures 18.1e and 18.1f contain nested loops.

REDUCIBLE FLOW GRAPHS

A *reducible flow graph* is one in which the dictionary definition of *loop* corresponds more closely to the technical definition; but let us develop a more precise definition.

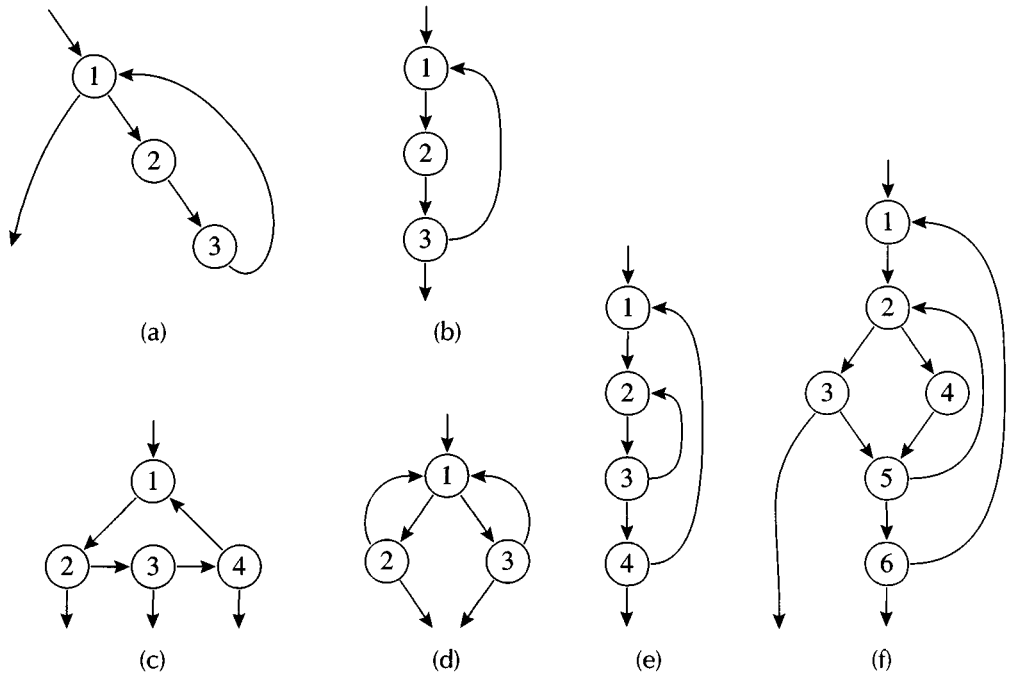


FIGURE 18.1. Some loops; in each case, 1 is the header node.

Figure 18.2a does not contain a loop; either node in the strongly connected component (2, 3) can be reached without going through the other.

Figure 18.2c contains the same pattern of nodes 1, 2, 3; this becomes more clear if we repeatedly delete edges and collapse together pairs of nodes (x, y), where x is the only predecessor of y. That is: delete $6 \to 9$, $5 \to 4$, collapse $(7, 9)$, $(3, 7)$, $(7, 8)$, $(5, 6)$, $(1, 5)$, $(1, 4)$; and we obtain Figure 18.2a.

An *irreducible flow graph* is one in which – after collapsing nodes and deleting edges – we can find a subgraph like Figure 18.2a. A *reducible flow graph* is one that cannot be collapsed to contain such a subgraph. Without such subgraphs, then any cycle of nodes does have a unique header node.

Common control-flow constructs such as **if-then**, **if-then-else**, **while-do**, **repeat-until**, **for**, and **break** (even multilevel **break**) can only generate reducible flow graphs. Thus, the control-flow graph for a Tiger or Java function, or a C function without **goto**, will always be reducible.

The following program corresponds to flow graph 18.1e, assuming Tiger were augmented with **repeat-until** loops:

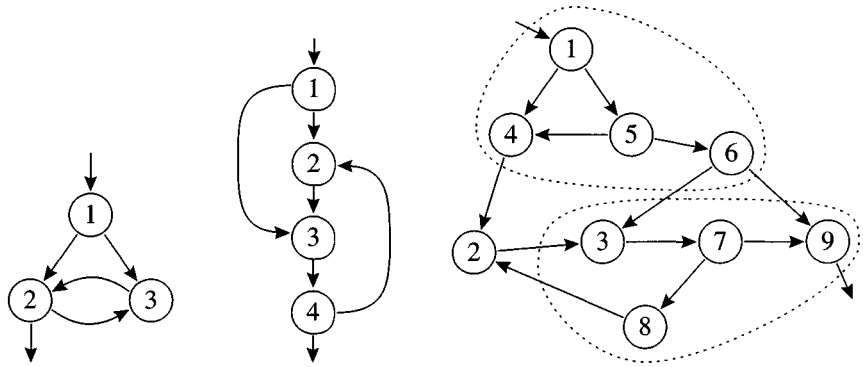


FIGURE 18.2. None of these contains a loop. Dotted lines indicate reduction of graph (c) by deleting edges and collapsing nodes.

```
function isPrime(n: int) : int =
  (i := 2;
   repeat j := 2;
          repeat if i*j=n
                    then return 0
                    else j := j+1
          until j=n;
          i := i+1
   until i=n;
   return 1)
```

In a functional language, loops are generally expressed using tail-recursive function calls. The `isPrime` program might be written as:

```
  function isPrime(n: int) : int =
0      tryI(n,2)

  function tryI(n: int, i: int) : int =
1      tryJ(n,i,2)

  function tryJ(n: int, i: int, j: int) : int =
2      if i*j=n
3         then 0
4         else nextJ(n,i,j+1)

  function nextJ(n: int, i: int, j: int) : int =
5      if j=n
         then nextI(n,i+1)
         else tryJ(n,i,j)

  function nextI(n: int, i: int) : int =
6      if i=n
         then 1
         else tryI(n,i)
```

where the numbers 1–6 show the correspondence with the flow-graph nodes of Figure 18.1f.

Because the programmer can arrange these functions in arbitrary ways, flow graphs produced by the tail-call structure of functional programs are sometimes irreducible.

Advantages of reducible flow graphs. Many dataflow analyses (presented in Chapter 17) can be done very efficiently on reducible flow graphs. Instead of using fixed-point iteration ("keep executing assignments until there are no changes"), we can determine an order for computing the assignments, and calculate in advance how many assignments will be necessary – that is, there will never be a need to check to see if anything changed.

However, for the remainder of this chapter we will assume that our control-flow graphs may be reducible or irreducible.

18.1 DOMINATORS

Before we optimize the loops, we must find them in the flow graph. The notion of *dominators* is useful for that purpose.

Each control-flow graph must have a start node s_0 with no predecessors, where program (or procedure) execution is assumed to begin.

A node d *dominates* a node n if every path of directed edges from s_0 to n must go through d. Every node dominates itself.

ALGORITHM FOR FINDING DOMINATORS

Consider a node n with predecessors $p_1, \ldots, p_k$, and a node d (with $d \neq n$). If d dominates each one of the p_i, then it must dominate n, because every path from s_0 to n must go through one of the p_i, but every path from s_0 to a p_i must go through d. Conversely, if d dominates n, it must dominate all the p_i; otherwise there would be a path from s_0 to n going through the predecessor not dominated by d.

Let $D[n]$ be the set of nodes that dominate n. Then

$$D[s_0] = \{s_0\} \qquad D[n] = \{n\} \cup \left(\bigcap_{p \in \text{pred}[n]} D[p] \right) \qquad \text{for } n \neq s_0$$

The simultaneous equations can be solved, as usual, by iteration, treating each equation as an assignment statement. However, in this case each set $D[n]$ (for

$n \neq s_0$) must be initialized to hold all the nodes in the graph, because each assignment $D[n] \leftarrow \{n\} \cup \ldots$ makes $D[n]$ smaller (or unchanged), not larger.

This algorithm can be made more efficient by ordering the set assignments in quasi-topological order, that is, according to a depth-first search of the graph (Algorithm 17.5). Section 19.2 describes a faster algorithm for computing dominators.

Technically, an unreachable node is dominated by every node in the graph; we will avoid the pathologies this can cause by deleting unreachable nodes from the graph before calculating dominators and doing loop optimizations. See also Exercise 18.4.

IMMEDIATE DOMINATORS

Theorem: In a connected graph, suppose d dominates n, and e dominates n. Then it must be that either d dominates e, or e dominates d.

Proof: (By contradiction.) Suppose neither d nor e dominates the other. Then there is some path from s_0 to e that does not go through d. Therefore any path from e to n must go through d; otherwise d would not dominate n.

Conversely, any path from d to n must go through e. But this means that to get from e to n the path must infinitely loop from d to e to d ... and never get to n.

This theorem tells us that every node n has no more than one *immediate dominator*, $idom(n)$, such that

1. $idom(n)$ is not the same node as n,
2. $idom(n)$ dominates n, and
3. $idom(n)$ does not dominate any other dominator of n.

Every node except s_0 is dominated by at least one node other than itself (since s_0 dominates every node), so every node except s_0 has exactly one immediate dominator.

Dominator tree. Let us draw a graph containing every node of the flow graph, and for every node n an edge from $idom(n)$ to n. The resulting graph will be a tree, because each node has exactly one immediate dominator. This is called the *dominator tree*.

Figure 18.3 shows a flow graph and its dominator tree. Some edges in the dominator tree correspond single flow-graph edges (such as $4 \rightarrow 6$), but others do not (such as $4 \rightarrow 7$). That is, the immediate dominator of a node is not necessarily its predecessor in the flow graph.

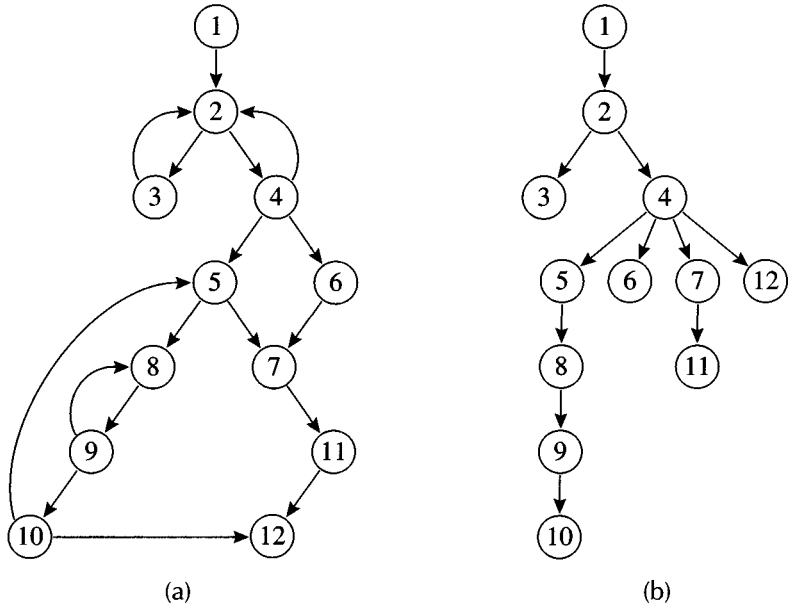


FIGURE 18.3. (a) A flow graph; (b) its dominator tree.

A flow-graph edge from a node n to a node h that dominates n is called a *back edge*. For every back edge there is a corresponding subgraph of the flow graph that is a loop. The back edges in Figure 18.3a are $3 \rightarrow 2$, $4 \rightarrow 2$, $10 \rightarrow 5$, $9 \rightarrow 8$.

LOOPS

The *natural loop* of a back edge $n \rightarrow h$, where h dominates n, is the set of nodes x such that h dominates x and there is a path from x to n not containing h. The *header* of this loop will be h.

The natural loop of the back edge $10 \rightarrow 5$ from Figure 18.3a includes nodes 5, 8, 9, 10 and has the loop 8, 9 nested within it.

A node h can be the header of more than one natural loop, if there is more than one back edge into h. In Figure 18.3a, the natural loop of $3 \rightarrow 2$ consists of the nodes 3, 2 and the natural loop of $4 \rightarrow 2$ consists of 4, 2.

The loop optimizations described in this chapter can cope with any loop, whether it is a natural loop or not, and whether or not that loop shares its header with some other loop. However, we usually want to optimize an *inner* loop first, because most of the program's execution time is expected to be in

the inner loop. If two loops share a header, then it is hard to determine which should be considered the inner loop. A common way of solving this problem is to merge all the natural loops with the same header. The result will not necessarily be a natural loop.

If we merge all the loops with header 2 in Figure 18.3a, we obtain the loop 2, 3, 4 – which is not a natural loop.

Nested loops If A and B are loops with headers a and b respectively, such that $a \neq b$ and b is in A, then the nodes of B are a proper subset of the nodes of A. We say that loop B is nested within A, or that B is the *inner loop*.

We can construct a *loop-nest tree* of loops in a program. The procedure is, for a flow graph G:

1. Compute dominators of G.
2. Construct the dominator tree.
3. Find all the natural loops, and thus all the loop-header nodes.
4. For each loop header h, merge all the natural loops of h into a single loop, $loop[h]$.
5. Construct the tree of loop headers (and implicitly loops), such that h_1 is above h_2 in the tree if h_2 is in $loop[h_1]$.

The leaves of the loop-nest tree are the *innermost loops*.

Just to have a place to put nodes not in any loop, we could say that the entire procedure body is a pseudo-loop that sits at the root of the loop-nest tree. The loop-nest tree of Figure 18.3 is shown in Figure 18.4.

LOOP PREHEADER

Many loop optimizations will insert statements immediately before the loop executes. For example, *loop-invariant hoisting* moves a statement from inside the loop to immediately before the loop. Where should such statements be put? Figure 18.5a illustrates a problem: if we want to insert statement s into a basic block immediately before the loop, we need to put s at the end of blocks 2 and 3. In order to have one place to put such statements, we insert a new, initially empty, *preheader* node p outside the loop, with an edge $p \rightarrow h$. All edges $x \rightarrow h$ from nodes x inside the loop are left unchanged, but all existing edges $y \rightarrow h$ from nodes y outside the loop are redirected to point to p.

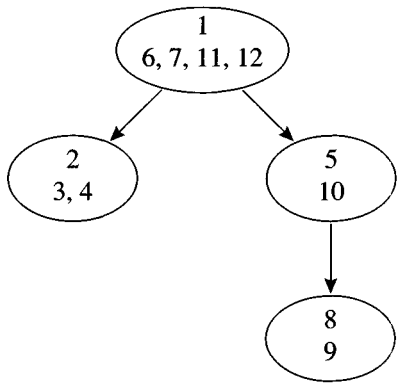


FIGURE 18.4. The loop-nest tree for Figure 18.3a. Each loop header is shown in the top half of each oval (nodes 1, 2, 5, 8); a loop comprises a header node (e.g. node 5), all the other nodes shown in the same oval (e.g. node 10), and all the nodes shown in subtrees of the loop-nest-tree node (e.g. 8, 9).

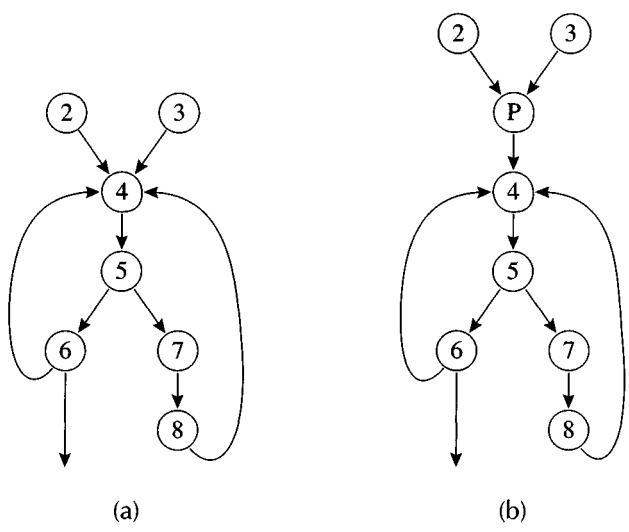


FIGURE 18.5. (a) A loop; (b) the same loop with a preheader.

18.2 LOOP-INVARIANT COMPUTATIONS

If a loop contains a statement $t \leftarrow a \oplus b$ such that a has the same value each time around the loop, and b has the same value each time, then t will also have the same value each time. We would like to *hoist* the computation out of the loop, so it is computed just once instead of every time.

We cannot always tell if a will have the same value every time, so as usual we will conservatively approximate. The definition $d : t \leftarrow a_1 \oplus a_2$ is loop-invariant within loop L if, for each operand a_i

1. a_i is a constant,
2. *or* all the definitions of a_i that reach d are outside the loop,
3. *or* only one definition of a_i reaches d, and that definition is loop-invariant.

This leads naturally to an iterative algorithm for finding loop-invariant definitions: first find all the definitions whose operands are constant or from outside the loop, then repeatedly find definitions whose operands are loop-invariant.

HOISTING

Suppose $t \leftarrow a \oplus b$ is loop-invariant. Can we hoist it out of the loop? In Figure 18.6a, hoisting makes the program compute the same result faster. But in Figure 18.6b, hoisting makes the program faster but incorrect – the original program does not *always* execute $t \leftarrow a \oplus b$, but the transformed program does, producing an incorrect value for x if $i \geq N$ initially. Hoisting in Figure 18.6c is also incorrect, because the original loop had more than one definition of t, and the transformed program interleaves the assignments to t in a different way. And hoisting in Figure 18.6d is wrong because there is a use of t before the loop-invariant definition, so after hoisting, this use will have the wrong value on the first iteration of the loop.

With these pitfalls in mind, we can set the criteria for hoisting $d : t \leftarrow a \oplus b$ to the end of the loop preheader:

1. d dominates all loop exits at which t is *live-out*;
2. *and* there is only one definition of t in the loop,
3. *and* t is not *live-out* of the loop preheader.

Implicit side effects. These rules need modification if $t \leftarrow a \oplus b$ could raise some sort of arithmetic exception or have other side effects; see Exercise 18.7.

Turning while loops into repeat-until loops. Condition (1) tends to prevent many computations from being hoisted from **while** loops; from Figure 18.7a

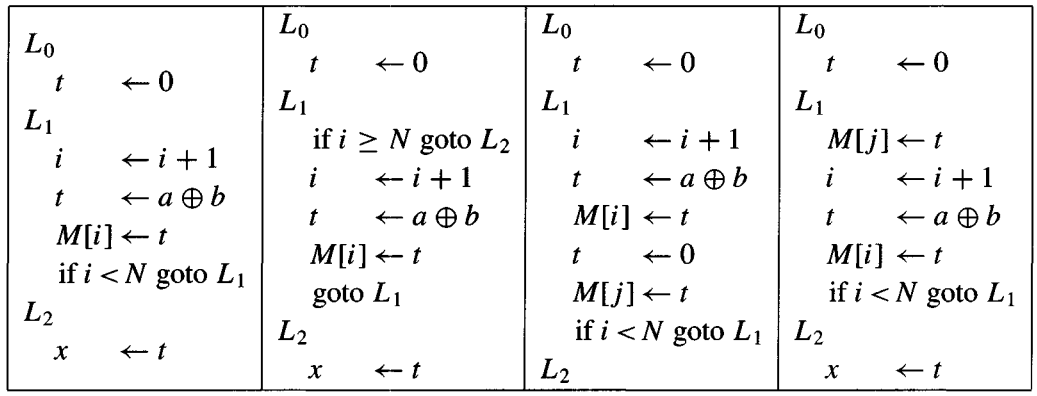


FIGURE 18.6. Some good and bad candidates for hoisting $t \leftarrow a \oplus b$.

it is clear that none of the statements in the loop body dominates the loop exit node (which is the same as the header node). To solve this problem, we can transform the **while** loop into a **repeat** loop preceded by an **if** statement. This requires duplication of the statements in the header node, as shown in Figure 18.7b. Of course, all the statements in the body of a **repeat** loop dominate the loop exit (if there are no **break** or explicit loop-exit statements), so condition (1) will be satisfied.

18.3 INDUCTION VARIABLES

Some loops have a variable i that is incremented or decremented, and a variable j that is set (in the loop) to $i \cdot c + d$ where c and d are loop-invariant. Then we can calculate j's value without reference to i; whenever i is incremented by a we can increment j by $c \cdot a$.

Consider, for example, Program 18.8a, which sums the elements of an array. Using *induction-variable analysis* to find that i and j are related induction variables, *strength reduction* to replace a multiplication by 4 with an addition, then *induction-variable elimination* to replace $i \geq n$ by $k \geq 4n + a$, followed by miscellaneous copy propagation, we get Program 18.8b. The transformed loop has fewer quadruples; it might even run faster. Let us now take the series of transformations one step at a time.

We say that a variable such as i is a *basic induction variable*, and j and k

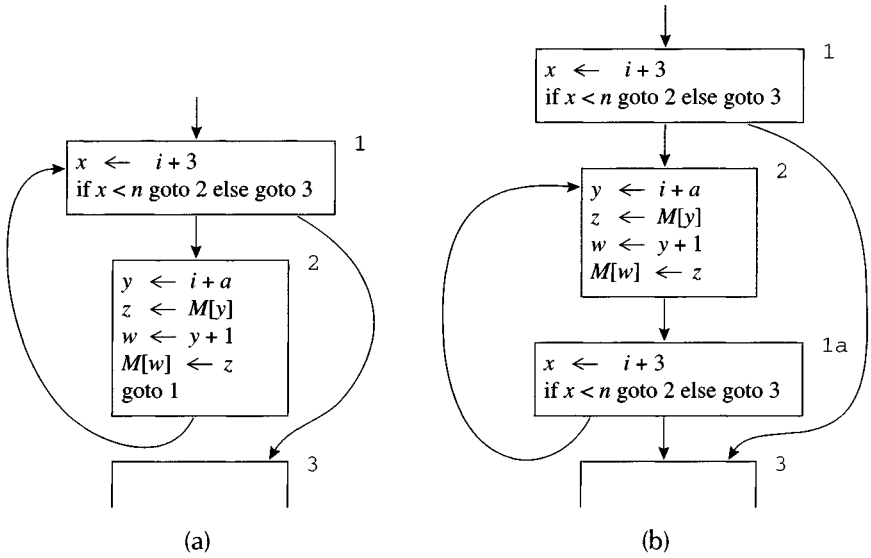


FIGURE 18.7. A **while** loop (a), transformed into a **repeat** loop (b).

```
    s ← 0
    i ← 0
L1 : if i ≥ n goto L2
    j ← i · 4
    k ← j + a
    x ← M[k]
    s ← s + x
    i ← i + 1
    goto L1
L2
```

(a) Before

```
    s ← 0
    k' ← a
    b ← n · 4
    c ← a + b
L1 : if k' ≥ c goto L2
    x ← M[k']
    s ← s + x
    k' ← k' + 4
    goto L1
L2
```

(b) After

PROGRAM 18.8. A loop before and after induction-variable optimizations.

```
        s ← 0                          s  ← 0
                                       j' ← i · 4
                                       b' ← b · 4
                                       n' ← n · 4
L1 : if s > 0 goto L2          L1 : if s > 0 goto L2
        i ← i + b                      j' ← j' + b'
        j ← i · 4                      j  ← j'
        x ← M[j]                       x  ← M[j]
        s ← s − x                      s  ← s − x
        goto L1                        goto L1
L2 : i ← i + 1                 L2 : j' ← j' + 4
        s ← s + j                      s  ← s + j
        if i < n goto L1               if j' < n' goto L1
```

(a) Before (b) After

FIGURE 18.9. The basic induction variable i is incremented by different amounts in different iterations; the derived induction variable j is not changed in every iteration.

are *derived induction variables in the family of* i. Right after j is defined (in the original loop), we have $j = a_j + i \cdot b_j$, where $a_j = 0$ and $b_j = 4$. We can completely characterize the value of j at its definition by (i, a, b), where i is a basic induction variable and a and b are loop-invariant expressions.

If there is another derived induction variable with definition $k \leftarrow j + c_k$ (where c_k is loop-invariant), then k is also in the family of i. We can characterize k by the triple (i, c_k, b_j), that is, $k = c_k + i \cdot b_j$.

We can characterize the basic induction variable i by a triple in the same way, that is $(i, 0, 1)$, meaning that $i = 0 + i \cdot 1$. Thus every induction variable can be characterized by such a triple.

If an induction variable changes by the same (constant or loop-invariant) amount in every iteration of the loop, we say it is a *linear induction variable*. In Figure 18.9a, the induction variable i is not linear: incremented by b in some iterations and by 1 in other iterations. Furthermore, in some iterations $j = i \cdot 4$ and in other iterations the derived induction variable j gets (temporarily) left behind as i is incremented.

DETECTION OF INDUCTION VARIABLES

Basic induction variables. The variable i is a basic induction variable in a loop L with header node h if the only definitions of i within L are of the form $i \leftarrow i + c$ or $i \leftarrow i - c$ where c is loop-invariant.

Derived induction variables. The variable k is a *derived induction variable* in loop L if:

1. There is only one definition of k within L, of the form $k \leftarrow j \cdot c$ or $k \leftarrow j + d$, where j is an induction variable and c, d are loop-invariant;
2. *and* if j is a derived induction variable in the family of i, then:
   (a) the only definition of j that reaches k is the one in the loop,
   (b) *and* there is no definition of i on any path between the definition of j and the definition of k.

Assuming j is characterized by (i, a, b), then k is described by $(i, a \cdot c, b \cdot c)$ or $(i, a + d, b)$, depending on whether k's definition was $j \cdot c$ or $j + d$.

Statements of the form $k \leftarrow j - c$ can be treated as $k \leftarrow j + (-c)$ for purposes of induction-variable analysis (unless $-c$ is not representable, which can sometimes happen with two's complement arithmetic).

Division. Statements of the form $k \leftarrow j/c$ can be rewritten as $k \leftarrow j \cdot (\frac{1}{c})$, so that k could be considered an induction variable. This fact is useful for floating-point calculations – though we must beware of introducing subtle numerical errors if $1/c$ cannot be represented exactly. If this is an integer division, we cannot represent $1/c$ at all.

STRENGTH REDUCTION

On many machines, multiplication is more expensive than addition. So we would like to take a derived induction variable whose definition is of the form $j \leftarrow i \cdot c$ and replace it with an addition.

For each derived induction variable j whose triple is (i, a, b), make a new variable j' (although different derived induction variables with the same triple can share the same j' variable). After each assignment $i \leftarrow i + c$, make an assignment $j' \leftarrow j' + c \cdot b$ where $c \cdot b$ is a loop-invariant expression that may be computed in the loop preheader. If c and b are both constant, then the multiplication may be done at compile-time. Replace the (unique) assigment to j with $j \leftarrow j'$. Finally, it is necessary to initialize j' at the end of the loop preheader, with $j' \leftarrow a + i \cdot b$.

We say two induction variables x, y in the family of i are *coordinated* if $(x - a_x)/b_x = (y - a_y)/b_y$ at every time during the execution of the loop, except during a sequence of statements $z_i \leftarrow z_i + c_i$, where c_i is loop-invariant. Clearly, all the new variables in the family of i introduced by strength reduction are coordinated with each other, and with i.

When the definition of an induction variable $j \leftarrow \cdots$ is replaced by $j \leftarrow j'$, we know that j' is coordinated but j might not be. However, the standard *copy propagation* algorithm can help here, replacing uses of j by uses of j' where there is no intervening definition of j'.

Thus, instead of using some sort of flow analysis to learn whether j is coordinated, we just use j' instead, where copy propagation says it is legal to do so.

After strength reduction there is still a multiplication, but it is outside the loop. If the loop executes more than one iteration, then the program should run faster with additions instead of multiplication, on many machines. The results of strength reduction may be disappointing on processors that can schedule multiplications to hide their latency.

Example. Let us perform strength reduction on Program 18.8a. We find that j is a derived induction variable with triple $(i, 0, 4)$, and k has triple $(i, a, 4)$. After strength reduction on both j and k, we have

$$
\begin{array}{ll}
 & s \leftarrow 0 \\
 & i \leftarrow 0 \\
 & j' \leftarrow 0 \\
 & k' \leftarrow a \\
L_1 : & \text{if } i \geq n \text{ goto } L_2 \\
 & j \leftarrow j' \\
 & k \leftarrow k' \\
 & x \leftarrow M[k] \\
 & s \leftarrow s + x \\
 & i \leftarrow i + 1 \\
 & j' \leftarrow j' + 4 \\
 & k' \leftarrow k' + 4 \\
 & \text{goto } L_1 \\
L_2 &
\end{array}
$$

We can perform *dead-code elimination* to remove the statement $j \leftarrow j'$. We would also like to remove all the definitions of the *useless variable* j', but

technically it is not dead, since it is used in every iteration of the loop.

ELIMINATION

After strength reduction, some of the induction variables are not used at all in the loop, and others are used only in comparisons with loop-invariant variables. These induction variables can be deleted.

A variable is *useless* in a loop L if it is dead at all exits from L, and its only use is in a definition of itself. All definitions of a useless variable may be deleted.

In our example, after the removal of j, the variable j' is useless. We can delete $j' \leftarrow j' + 4$. This leaves a definition of j' in the preheader that can now be removed by dead-code elimination.

REWRITING COMPARISONS

A variable k is *almost useless* if it is used only in comparisons against loop-invariant values and in definitions of itself, and there is some other induction variable in the same family that is not useless. An almost-useless variable may be made useless by modifying the comparison to use the related induction variable.

If we have $k < n$, where j and k are coordinated induction variables in the family of i, and n is loop-invariant; then we know that $(j - a_j)/b_j = (k - a_k)/b_k$, so therefore the comparison $k < n$ can be written as

$$a_k + \frac{b_k}{b_j}(j - a_j) < n$$

Now, we can subtract a_k from both sides and multiply both sides by b_j/b_k. If b_j/b_k is positive, the resulting comparison is:

$$j - a_j < \frac{b_j}{b_k}(n - a_k)$$

but if b_j/b_k is negative, then the comparison becomes

$$j - a_j > \frac{b_j}{b_k}(n - a_k)$$

instead. Finally, we add a_j to both sides (here we show the positive case):

$$j < \frac{b_j}{b_k}(n - a_k) + a_j$$

The entire right-hand side of this comparison is loop-invariant, so it can be computed just once in the loop preheader.

Restrictions:

1. If $b_j(n - a_k)$ is not evenly divisible by b_k, then this transformation cannot be used, because we cannot hold a fractional value in an integer variable.
2. If b_j or b_k is not constant, but is a loop-invariant value whose sign is not known, then the transformation cannot be used since we won't know which comparison (less-than or greater-than) to use.

Example. In our example, the comparison $i < n$ can be replaced by $k' < a + 4 \cdot n$. Of course, $a + 4 \cdot n$ is loop-invariant and should be hoisted. Then i will be useless and may be deleted. The transformed program is:

$$
\begin{array}{ll}
 & s \leftarrow 0 \\
 & k' \leftarrow a \\
 & b \leftarrow n \cdot 4 \\
 & c \leftarrow a + b \\
L_1: & \text{if } k' < c \text{ goto } L_2 \\
 & k \leftarrow k' \\
 & x \leftarrow M[k] \\
 & s \leftarrow s + x \\
 & k' \leftarrow k' + 4 \\
 & \text{goto } L_1 \\
L_2 &
\end{array}
$$

Finally, copy propagation can eliminate $k \leftarrow k'$, and we obtain Program 18.8b.

18.4 ARRAY-BOUNDS CHECKS

Safe programming languages automatically insert array-bounds checks on every subscript operation (see the sermon on page 160). Of course, in well written programs all of these checks are redundant, since well written programs don't access arrays out of bounds. We would like safe languages to achieve the fast performance of unsafe languages. Instead of turning off the bounds checks (which would not be safe) we ask the compiler to remove any checks that it can prove are redundant.

We cannot hope to remove all the redundant bounds checks, because this problem is not computable (it is as hard as the halting problem). But many array subscripts are of the form $a[i]$, where i is an induction variable. These the compiler can often understand well enough to optimize.

The bounds for an array are generally of the form $0 \leq i \wedge i < N$. When N is nonnegative, as it always is for array sizes, this can be implemented as $i <_u N$, where $<_u$ is the unsigned comparison operator.

Conditions for eliminating array-bounds checking. Although it seems natural and intuitive that an induction variable must stay within a certain range, and we should be able to tell whether that range does not exceed the bounds of the array, the criteria for eliminating a bounds check from a loop L are actually quite complicated:

1. There is an induction variable j and a loop-invariant u used in a statement s_1, taking one of the following forms:

   if $j < u$ goto L_1 else goto L_2
   if $j \geq u$ goto L_2 else goto L_1
   if $u > j$ goto L_1 else goto L_2
   if $u \leq j$ goto L_2 else goto L_1

   where L_2 is out of the loop.
2. There is a statement s_2 of the form

   if $k <_u n$ goto L_3 else goto L_4

   where k is an induction variable coordinated with j, n is loop-invariant, and s_1 dominates s_2.
3. There is no loop nested within L containing a definition of k.
4. k increases when j does, that is, $b_j/b_k > 0$.

Often, n will be an array length. In a language with static arrays an array length n is a constant. In many languages with dynamic arrays, array lengths are loop-invariant. In Tiger, Java, and ML the length of an array cannot be dynamically modified once the array has been allocated. The array-length n will typically be calculated by fetching the *length* field of some array-pointer v. For the sake of illustration, assume the length field is at offset 0 in the array object. To avoid the need for complicated alias analysis, the semantic analysis phase of the compiler should mark the expression $M[v]$ as *immutable*, meaning that no other store instruction can possibly update the contents of the *length* field of the array v. If v is loop-invariant, then n will also be loop-invariant. Even if n is not an array length but is some other loop invariant, we can still optimize the comparison $k <_u n$.

We want to put a test in the loop preheader that expresses the idea that in every iteration, $k \geq 0 \wedge k < n$. Let k_0 be the value of k at the end of the

preheader, and let $\Delta k_1, \Delta k_2, \ldots, \Delta k_m$ be all the loop-invariant values that are added to k inside the loop. Then we can ensure $k \geq 0$ by testing

$$k \geq 0 \wedge \Delta k_1 \geq 0 \wedge \cdots \wedge \Delta k_m \geq 0$$

at the end of the preheader.

Let $\Delta k_1, \Delta k_2, \ldots, \Delta k_p$ be the set of loop-invariant values that are added to k on any path between s_1 and s_2 that does not go through s_1 (again). Then, to ensure $k < n$ at s_2, it is sufficient to ensure that $k < n - (\Delta k_1 + \cdots + \Delta k_p)$ at s_1. Since we know $(k - a_k)/b_k = (j - a_j)/b_j$, this test becomes

$$j < \frac{b_j}{b_k}(n - (\Delta k_1 + \cdots + \Delta k_p) - a_k) + a_j$$

This will always be true if

$$u < \frac{b_j}{b_k}(n - (\Delta k_1 + \cdots + \Delta k_p) - a_k) + a_j$$

since the test $j < u$ dominates the test $k < n$.

Since everything in this comparison is loop-invariant, we can move it to the preheader as follows. First, ensure that definitions of loop-invariants are hoisted out of the loop. Then, rewrite the loop L as follows: copy all the statements of L to make a new loop L' with header L'_h. Inside L', replace the statement

$$\text{if } k < n \text{ goto } L'_3 \text{ else goto } L'_4$$

by **goto** L'_3. At the end of the preheader of L, put statements equivalent to

$$\begin{array}{l}\text{if } k \geq 0 \ \wedge \ k_1 \geq 0 \ \wedge \ \cdots \ \wedge \ k_m \geq 0 \\ \quad \wedge \ u < \frac{b_j}{b_k}(n - (\Delta k_1 + \cdots + \Delta k_p) - a_k) + a_j \\ \quad\ \text{goto } L'_h \\ \ \text{else goto } L_h\end{array}$$

The conditional **goto** tests whether k will always be between 0 and n.

Sometimes we will have enough information to evaluate this complicated condition at compile time. This will be true in at least two situations:

1. all the loop-invariants mentioned in it are constants, or
2. n and u are the same temporary variable, $a_k = a_j$, $b_k = b_j$, and there are no Δk's added to k between s_1 and s_2. In a language like Tiger or Java or ML, this could happen if the programmer writes,

```
let var u := length(A)
    var i := 0
 in while i<u
     do (sum := sum + A[i];
         i := i+1)
end
```

The quadruples for `length(A)` will include $u \leftarrow M[A]$, assuming that the length of an array is fetched from offset zero from the array pointer; and the quadruples for `A[i]` will include $n \leftarrow M[A]$, to fetch n for doing the bounds check. Now the expressions defining u and n are common subexpressions, assuming the expression $M[A]$ is marked so that we know that no other STORE instruction is modifying the contents of memory location $M[A]$.

If we can evaluate the big comparison at compile time, then we can unconditionally use loop L or loop L', and delete the loop that we are not using.

Cleaning up. After this optimization, the program may have several loose ends. Statements after the label L'_4 may be unreachable; there may be several useless computations of n and k within L'. The former can be cleaned up by *unreachable-code elimination*, and the latter by *dead-code elimination*.

Generalizations. To be practically useful, the algorithm needs to be generalized in several ways:

1. The loop-exit comparison might take one of the forms

   if $j \leq u'$ goto L_1 else goto L_2
   if $j > u'$ goto L_2 else goto L_1
   if $u' \geq j$ goto L_1 else goto L_2
   if $u' < j$ goto L_2 else goto L_1

   which compares $j \leq u'$ instead of $j < u$.
2. The loop-exit test might occur at the bottom of the loop body, instead of before the array-bounds test. We can describe this situation as follows: There is a test

   s_2 : if $j < u$ goto L_1 else goto L_2

   where L_2 is out of the loop and s_2 dominates all the loop back edges. Then the Δk_i of interest are the ones between s_2 and any back edge, and between the loop header and s_1.
3. We should handle the case where $b_j/b_k < 0$.
4. We should handle the case where j counts downward instead of up, and the loop-exit test is something like $j \geq l$, for l a loop-invariant lower bound.
5. The induction-variable increments might be "undisciplined"; for example,

$$
\begin{array}{l}
L_1 : x \leftarrow M[i] \\
\quad s \leftarrow s + x \\
\quad i \leftarrow i + 4 \\
\quad \text{if } i < n \text{ goto } L_1 \text{ else } L_2 \\
L_2
\end{array}
$$

(a) Before

$$
\begin{array}{l}
L_1 : x \leftarrow M[i] \\
\quad s \leftarrow s + x \\
\quad i \leftarrow i + 4 \\
\quad \text{if } i < n \text{ goto } L'_1 \text{ else } L_2 \\
L'_1 : x \leftarrow M[i] \\
\quad s \leftarrow s + x \\
\quad i \leftarrow i + 4 \\
\quad \text{if } i < n \text{ goto } L_1 \text{ else } L_2 \\
L_2
\end{array}
$$

(b) After

PROGRAM 18.10. Useless loop unrolling.

```
while i<n-1
 do (if sum<0
       then (i:=i+1; sum:= sum+i; i:=i+1)
       else i := i+2;
     sum := sum + a[i])
```

Here there are three Δi, (of 1, 1, and 2 respectively). Our analysis will assume that any, all, or none of these increments may be applied; but clearly the effect is $i \leftarrow i + 2$ on either path. In such cases, an analysis that hoists (and merges) the increments above the **if** will be useful.

18.5 LOOP UNROLLING

Some loops have such a small body that most of the time is spent incrementing the loop-counter variable and testing the loop-exit condition. We can make these loops more efficient by *unrolling* them, putting two or more copies of the loop body in a row.

Given a loop L with header node h and back edges $s_i \rightarrow h$, we can unroll the loop as follows:

1. Copy the nodes to make a loop L' with header h' and back edges $s'_i \rightarrow h'$.
2. Change all the back edges in L from $s_i \rightarrow h$ to $s_i \rightarrow h'$.
3. Change all the back edges in L' from $s'_i \rightarrow h'$ to $s'_i \rightarrow h$.

For example, Program 18.10a unrolls into Program 18.10b. But nothing useful has been accomplished; each "original" iteration still has an increment and a conditional branch.

```
L1 : x ← M[i]
     s ← s + x
     x ← M[i + 4]
     s ← s + x
     i ← i + 8
     if i < n goto L1 else L2
L2
```

(a) Fragile

```
     if i < n − 8 goto L1 else L2
L1 : x ← M[i]
     s ← s + x
     x ← M[i + 4]
     s ← s + x
     i ← i + 8
     if i < n − 8 goto L1 else L2
L2   x ← M[i]
     s ← s + x
     i ← i + 4
     if i < n goto L2 else L3
L3
```

(b) Robust

PROGRAM 18.11. Useful loop unrolling; (a) works correctly only for an even number of iterations of the original loop; (b) works for any number of iterations of the original loop.

By using information about induction variables, we can do better. We need an induction variable i such that every increment $i \leftarrow i + \Delta$ dominates every back edge of the loop. Then we know that each iteration increments i by exactly the sum of all the Δ, so we can agglomerate the increments and loop-exit tests to get Program 18.11a. But this unrolled loop works correctly only if the original loop iterated an even number of times. We execute "odd" iterations in an *epilogue*, as shown in Program 18.11b.

Here we have shown only the case of unrolling by a factor of two. When a loop is unrolled by a factor of K, then the epilogue is a loop (much like the original one) that iterates up to $K - 1$ times.

FURTHER READING

Lowry and Medlock [1969] characterized loops using dominators and performed induction-variable optimizations. Allen [1970] introduced the notion of reducible flow graphs. Aho et al. [1986] describe many optimizations, analyses, and transformations on loops.

Splitting control-flow nodes or edges gives a place into which statements

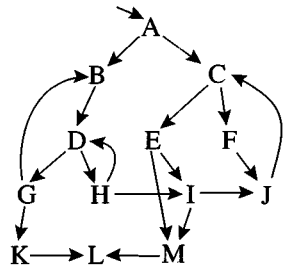
can be moved. The *loop preheader* transformation described on page 410 is an example of such splitting. Other examples are *landing pads* [Cytron et al. 1986] – nodes inserted in each loop-exit edge; *postbody nodes* [Wolfe 1996] – nodes inserted at the end of a loop body (see Exercise 18.6); and edge splitting to ensure a *unique successor or predecessor* property [Rosen et al. 1988] (see Section 19.1).

Chapter 19 describes other loop optimizations and a faster algorithm for computing dominators.

EXERCISES

18.1

a. Calculate the dominators of each node of this flowgraph:



b. Show the immediate dominator tree.

c. Identify the set of nodes in each natural loop.

18.2 Calculate the immediate-dominator tree of each of the following graphs:

a. The graph of Figure 2.8.

b. The graph of Exercise 2.3a.

c. The graph of Exercise 2.5a.

d. The graph of Figure 3.27.

***18.3** Let G be a control-flow graph, h be a node in G, A be the set of nodes in a loop with header h, and B be the set of nodes in a different loop with header h. Prove that the subgraph whose nodes are $A \cup B$ is also a loop.

***18.4** The immediate dominator theorem (page 408) is false for graphs that contain unreachable nodes.

a. Show a graph with nodes d, e, and n such that d dominates n, e dominates n, but neither d dominates e nor e dominates d.

b. Identify which step of the proof is invalid for graphs containing unreachable nodes.

c. In approximately three words, name an algorithm useful in finding unreachable nodes.

***18.5** Show that in a connected flow graph (one without unreachable nodes), a natural loop as defined on page 409 satisfies the definition of loop given on page 404.

18.6 For some purposes it is desirable that each loop header node should have exactly two predecessors, one outside the loop and one inside. We can ensure that there is only one outside predecessor by inserting a *preheader* node, as described in Section 18.1. Explain how to insert a *postbody* node to ensure that the loop header has only one predecessor inside the loop.

***18.7** Suppose any arithmetic overflow or divide-by-zero will raise an exception at run time. If we hoist $t \leftarrow a \oplus b$ out of a loop, and the loop might not have executed the statement at all, then the transformed program may raise the exception where the original program did not. Revise the criteria for *loop-invariant hoisting* to take account of this. Instead of writing something informal like "might not execute the statement," use the terminology of dataflow analysis and dominators.

18.8 On pages 412–413 the transformation of a **while** loop to a **repeat** loop is described. Show how a **while** loop may be characterized in the control-flow graph of basic blocks (using dominators) so that the optimizer can recognize it. The body of the loop may have explicit **break** statements that exit the loop.

***18.9** For bounds-check elimination, we required (on page 420) that the loop-exit test dominate the bounds-check comparison. If it is the other way around, then (effectively) we have one extra array subscript at the end of the loop, so the criterion

$$a_k + i \cdot b_k \geq 0 \ \wedge \ (n - a_k) \cdot b_j < (u - a_j) \cdot b_k$$

is "off by one." Rewrite this criterion for the case where the bounds-check comparison occurs before the loop-exit test.

***18.10** Write down the rules for unrolling a loop, such that the induction-variable increments are agglomerated and the unrolled loop has only one loop-exit test per iteration, as was shown informally for Program 18.10.