Engineering a Compiler

Manuscript for the Third Edition (EaC 3e)

Keith D. Cooper
Linda Torczon

Rice University
Houston, Texas

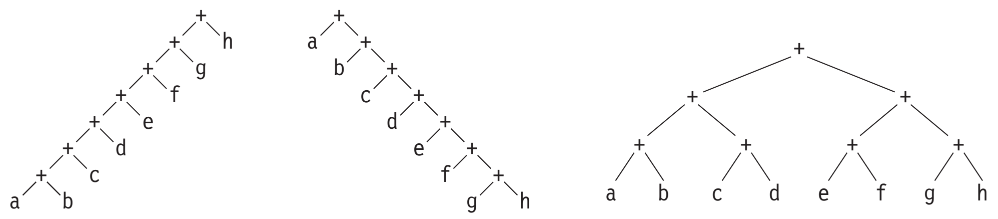


FIGURE 8.5 Potential Tree Shapes for a + b + c + d + e + f + g + h

8.4.2 Tree-Height Balancing

Specific details as to how the compiler encodes a computation can affect the compiler's ability to optimize that computation. For example, almost all modern processors have multiple functional units; they can execute multiple independent operations in each cycle. If the compiler can arrange the instruction stream so that it contains independent operations, encoded in the appropriate, machine-specific way, then the application will run more quickly.

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

Consider the expression a + b + c + d + e + f + g + h. The code in the margin encodes a left-to-right evaluation, which is equivalent to a postorder walk of the left-associative tree in Figure 8.5.a. A right-to-left evaluation of the expression would correspond to the tree in panel (b). Each tree constrains the execution order in ways that are more restrictive than the rules of addition. The left-associative tree forces the code to evaluate a + b before it can add either g or h. The right-associative tree requires that g + h precede additions involving a or b. The balanced tree in panel (c) imposes fewer constraints, but it still restricts evaluation order more than does the underlying arithmetic.

	Adder 0	Adder 1
1	$t_1 \leftarrow$ a + b	—
2	$t_2 \leftarrow t_1$ + c	—
3	$t_3 \leftarrow t_2$ + d	—
4	$t_4 \leftarrow t_3$ + e	—
5	$t_5 \leftarrow t_4$ + f	—
6	$t_6 \leftarrow t_5$ + g	—
7	$t_7 \leftarrow t_6$ + h	—

(a) Left-Associative Tree

	Adder 0	Adder 1
1	$t_1 \leftarrow$ a + b	$t_2 \leftarrow$ c + d
2	$t_3 \leftarrow$ e + f	$t_4 \leftarrow$ g + h
3	$t_5 \leftarrow t_1 + t_2$	$t_6 \leftarrow t_3 + t_4$
4	$t_7 \leftarrow t_5 + t_6$	—
5	—	—
6	—	—
7	—	—

(b) Balanced Tree

FIGURE 8.6 Schedules from Trees in Figure 8.5

If the processor can perform multiple concurrent additions, then the balanced tree should lead to a shorter schedule for the computation. Figure 8.6 shows the shortest possible schedules for the balanced tree and the left-associative tree on a computer with two single-cycle adders. The balanced tree can execute in four cycles, with one unit idle in the fourth cycle. By contrast, the left-associative tree takes seven cycles to execute because it serializes the additions. It leaves one adder idle. The right-associative tree produces similar results.

This small example suggests an important optimization: using the commutativity and associativity to expose additional parallelism in expression evaluation. The rest of this section presents an algorithm that rewrites a single block to improve balance across its forest of expression trees. This transformation exposes more *instruction-level parallelism* (ILP) to the instruction scheduler.

Instruction-level parallelism: Operations that are independent can execute in the same cycle. This kind of concurrency is called instruction-level parallelism.

Candidate Trees

As a first step, the algorithm must identify trees that are promising candidates for re-balancing. A candidate tree must contain only one kind of operator, such as + or ×. That operator must be both associative and commutative, to allow rearrangement of the operands. The tree should be as large as possible, to maximize ILP. If the tree is too small, the optimization will make no difference.

To ensure that the rewritten code works in its surrounding context, that code must preserve any values that are used outside of the expression. A value is *exposed* if it is used after the block; if it is used more than once within the block; or if it is appears as an argument to an operator of another type. Exposed values must be preserved—that is, the rewritten computation must produce the same value for the exposed value as the original computation. This restriction can constrain the algorithm's ability to rearrange code.

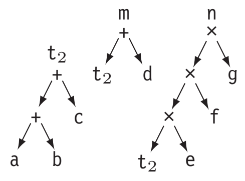
The code fragment shown in the margin computes two large expressions: $m \leftarrow (a+b+c+d)$, and $n \leftarrow (a+b+c) \times e \times f \times g$. Assume that m and n are used later, so they are exposed values. t_2 is also exposed; it must be preserved for its use in the computation of t_3.

```
t1 ← a + b
t2 ← t1 + c
m ← t2 + d
t3 ← t2 × e
t4 ← t4 × f
n ← t4 × g
```

Example Basic Block

The fact that t_2 is exposed affects the code in two distinct ways. First, the compiler cannot rewrite $(a+b+c+d)$ as $(a+b)+(c+d)$ because the latter expression never computes the value of t_2. Second, the computation of t_2 must precede the computation of t_3, which, again, restricts the order of operations.

The algorithm to find candidate trees must treat each exposed value as the root of a tree. In the example, this produces the forest shown in the margin. Notice that t_2 is defined once and used twice. The trees for t_2 and y are too small to benefit from tree-height balancing, while the tree for n, with four operands, can be balanced.



Trees in Example

```
// All operations have the form "$T_i \leftarrow L_i\ Op_i\ R_i$"
// Phase 1: build a queue, Roots, of the candidate trees
Roots ← new queue of names
for i ← 0 to n - 1
    Rank($T_i$) ← -1
    if $Op_i$ is commutative and associative and $T_i$ is exposed then
        mark $T_i$ as a root
        Enqueue(Roots, $T_i$, precedence of $Op_i$)
// Phase 2: remove a tree from Roots and rebalance it
while (Roots is not empty)
    var ← Dequeue(Roots)
    Balance(var)
```

FIGURE 8.7 Tree-Height Balancing Algorithm

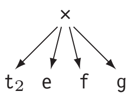
Upward exposed: A name *x* is *upward exposed* in block *b* if the first use of *x* in *b* refers to a value computed before entering *b*.

If the compiler's definitive IR is not a collection of expression trees, it can either build the expression trees for the block as a derivative IR, or it can interpret the definitive IR as expression trees to drive the algorithm. The latter approach requires some simple data structures to link each use in the block with either the definition in the block that reaches it or a notation that indicates that the use is *upward exposed*.

High-Level Sketch of Algorithm

The tree-height balancing algorithm, shown in Figure 8.7, consists of two phases. The first phase finds the roots of the candidate expression trees in the block. It ignores operators unless they are both commutative and associative. If a value *T* is both an exposed value and the result of a commutative and associative operator, then it adds *T* to a queue of roots, ordered by the operator's arithmetic precedence.

The second phase takes the roots, in precedence order from lowest to highest, and creates a balanced tree by flattening the tree into a single *n*-ary operator and then rebuilding it in balanced form. The *n*-ary tree for n in our continuing example appears in the margin.



n-ary Tree for n

Phase I: Finding Candidate Trees

Phase one of the algorithm is straightforward. The algorithm walks over the operations in the block, in order, and tests each operation to determine if it should be the root of a candidate tree. The test relies on observations made earlier. A candidate tree root must be the result of applying an operator that is both commutative and associative, and the root's value must be exposed.

To determine if the value T_i defined by operation *i* is exposed, the

```
Balance(root)        // Create balanced tree from its root, $T_i$
    if Rank(root) $\geq$ 0 then
        return                          // already processed this tree

    q $\leftarrow$ new queue of names          // Flatten the tree, then rebuild it
    Rank(root) $\leftarrow$ Flatten($L_i$,q) + Flatten($R_i$,q)
    Rebuild(root, q, $Op_i$)


Flatten(var,q)       // Flatten computes a rank for var & builds the queue
    if var is a constant then           // Constant is a leaf in the tree
        Rank(var) $\leftarrow$ 0
        Enqueue(q,var,Rank(var))
    else if var is upward exposed then  // UE is a leaf in the tree
        Rank(var) $\leftarrow$ 1
        Enqueue(q,var,Rank(var))
    else if var is a root               // Root of another tree is a leaf
        Balance(var)                    // Balance sets Rank(var)
        Enqueue(q,var,Rank(var))
    else                                // interior node for $T_i \leftarrow L_i\ Op_i\ R_i$
        Rank(var) $\leftarrow$ Flatten($L_j$,q) + Flatten($R_j$,q)
    return Rank(var)
```

FIGURE 8.8 Tree-Height Balancing Algorithm: Balance and Flatten

compiler must understand where T_i is used. The compiler can compute LIVEOUT sets for each block and use that information to determine if T_i is used after the current block (see Section 8.6.1). As part of computing the initial information for that computation, the compiler can also determine how many times T_i is used within the block.

The test for a candidate-tree root, then, can be expressed concisely. For an operation $T \leftarrow L\ Op\ R$ in block B, T is a root if and only if:

Op is both commutative and associative and
(T is used more than once in B or $T \in$ LIVEOUT(B))

Phase one must initialize the ranks on each T_i defined in the block, mark the roots so that later processing need not retest them, and add each root to a priority queue of roots. The queue of roots for the continuing example appears in the margin, assuming that + has precedence one and × has precedence two.

$(\langle m,1\rangle, \langle t_2,1\rangle, \langle n,2\rangle)$
Queue of roots for example

Phase II: Rebuilding the Block in Balanced Form

Phase 2 takes the queue of candidate-tree roots and builds, from each root, an approximately balanced tree. Phase 2 starts with a while loop

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

        if NewL and NewR are both constants then
            // evaluate and fold the result
            NewT ← Fold(op, NewL, NewR)
            if q is empty then
                create the op "root ← NewT"
                Rank(root) = 0
            else
                Enqueue(q, NewT, 0)
                Rank(NewT) = 0

        else if q is empty          // not a constant; name the result
                NewT ← root
             else
                NewT ← new name

        create the op "NewT ← NewL op NewR"
        Rank(NewT) ← Rank(NewL) + Rank(NewR)
        if q is not empty then   // More ops in q ⇒ add NewT to q
            Enqueue(q, NewT, Rank(NewT))
```

FIGURE 8.9 Tree-Height Balancing Algorithm: Rebuild

that calls *Balance* on each candidate tree root. *Balance*, *Flatten*, and *Rebuild* implement phase two, as shown in Figures 8.8 and 8.9.

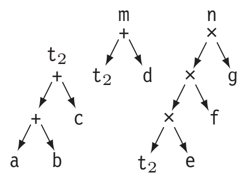
Balance is invoked on each candidate-tree root in increasng priority order. If this tree has not already been balanced, then it uses *Flatten* to create a new priority queue that holds all the operands of the current tree. In essence, the queue represents the single-level, n-ary version of the candidate tree. Once *Flatten* has created the queue, *Balance* invokes *Rebuild* to emit code for the approximately balanced tree.

Recall that phase one ranked the *Roots* queue by arithmetic precedence; that forces *Flatten* to follow the correct evaluation order.

Flatten is straightforward. It recursively walks the candidate tree, assigns each node a rank, and adds them to the queue. *Flatten* assigns a rank to each variable: zero for a constant and one for a leaf. If *Flatten* encounters a root, it calls *Balance* on that root; *Balance* creates a new queue, flattens the root's candidate tree, and rebuilds it. The root keeps the priority assigned to it by that recursive call.

When *Balance* calls *Rebuild*, the queue contains all of the leaves from the candidate tree, in rank order. Intermediate results, represented by interior nodes in the original expression trees are gone.

Rebuild uses a simple algorithm to rewrite the code. It repeatedly removes the two lowest ranked items from the tree. If they are constant, then it evaluates the operation and pushes the result back onto the queue with priority zero. Since all constants have rank zero, *Rebuild* will fold them into one constant. If one or more of the items has non-zero rank, *Rebuild* creates an operation to combine the two items, assigns the result a new rank, and pushes it onto the queue. This process continues until the queue is empty.



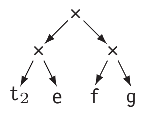
Expression Trees

($\langle$m,1$\rangle$, $\langle t_2$,1$\rangle$, $\langle$n,2$\rangle$)

Roots Queue

The expressions trees and *Roots* queue for the ongoing example are shown in the margin. Phase two will pull the roots from the queue and pass them to *Balance*. If *Dequeue* returns *m* first, then Balance will immediately recur on t_2 because it is a root. Neither the tree for *m* nor t_2 have enough operands to allow the algorithm to improve the code.

After it processes *m* and t_2, phase two pulls *n* from the *Roots* queue. *Flatten* produces a queue with all four operands at rank one. *Rebuild* then creates operations to compute $g+f$ and t_2+e, and enters those sums into the queue with rank two. It then pulls those sums and combines them into the root, *n*. The rebuilt tree for *n* appears in the margin.



Balanced tree for *n*

A Larger Example

As a second example, consider the basic block shown in Figure 8.10.a. This code might result from local value numbering; constants have been folded and redundant computations eliminated. Note that all of s, t, u, v, w, x, y, and z are upward exposed and that t_6, t_{10}, and t_{11} are live on exit from the block.

The block contains several intertwined computations. Panel (b) shows the five distinct expresion trees in the block. The values of t_3 and t_7 are each used multiple times, so we shown them as distinct trees. The connections to t_3 and t_7 are shown in gray.

When we apply phase 1 of the tree-height balancing algorithm to the example, it finds five roots: the three values that are live on exit, plus t_3 andt_7. The roots appear in boxes in panels (b) and (c). At the end of phase 1, *Roots* contains: { $\langle t_{11}$,1$\rangle$, $\langle t_7$,1$\rangle$, $\langle t_3$,1$\rangle$, $\langle t_{10}$,2$\rangle$, $\langle t_6$,2$\rangle$ }.

Phase 2 of the algorithm starts with the *Roots* queue. It removes a node from the *Roots* queue and calls *Balance* to process it. When *Balance* returns, it repeats the process until the *Roots* queue is empty.

Applying the Phase 2 algorithm to our example's *Roots* queue, it first calls *Balance* on t_{11}. Balance calls *Flatten* which builds a queue. When *Flatten* encounters t_3, which is a root, it invokes *Balance(*t_3*)*.

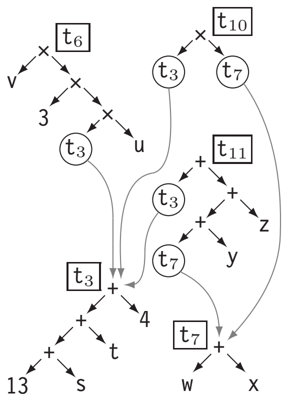
Applied to t_3, *Balance* first calls *Flatten* which produces the queue: {$\langle$4,0$\rangle$,$\langle$13,0$\rangle$,$\langle$t,1$\rangle$,$\langle$s,1$\rangle$}. *Balance* passes this queue to *Rebuild*

Rebuild dequeues $\langle$4,0$\rangle$ and $\langle$13,0$\rangle$. Since both are constants, it adds them and enqueues $\langle$17,0$\rangle$. Next, it dequeues $\langle$17,0$\rangle$, and $\langle$t,1$\rangle$;. creates the operation17 + t; assigns the result to a new name, say n_0; and enqueues $\langle n_0$,1$\rangle$. Finally, *Rebuild* dequeues $\langle n_0$,1$\rangle$. and $\langle$s,1$\rangle$ and creates the operation n_0 + s Since the queue is now empty, *Rebuld* assigns the

```
n₀ ← 17 + t
t₃ ← n₀ + s
```

$$\begin{aligned}
t_1 &\leftarrow 13 + s\\
t_2 &\leftarrow t_1 + t\\
t_3 &\leftarrow t_2 + 4\\
t_4 &\leftarrow t_3 \times u\\
t_5 &\leftarrow 3 \times t_4\\
t_6 &\leftarrow v \times t_5\\
t_7 &\leftarrow w + x\\
t_8 &\leftarrow t_7 + y\\
t_9 &\leftarrow t_8 + z\\
t_{10} &\leftarrow t_3 \times t_7\\
t_{11} &\leftarrow t_3 + t_9
\end{aligned}$$

(a) Original Code



(b) Trees in the Code

$$\begin{aligned}
t_1 &\leftarrow 13 + s\\
t_2 &\leftarrow t_1 + t\\
\boxed{t_3} &\leftarrow t_2 + 4\\
t_4 &\leftarrow t_3 \times u\\
t_5 &\leftarrow 3 \times t_4\\
\boxed{t_6} &\leftarrow v \times t_5\\
\boxed{t_7} &\leftarrow w + x\\
t_8 &\leftarrow t_7 + y\\
t_9 &\leftarrow t_8 + z\\
\boxed{t_{10}} &\leftarrow t_3 \times t_7\\
\boxed{t_{11}} &\leftarrow t_3 + t_9
\end{aligned}$$

(c) Finding Roots

FIGURE 8.10 Detailed Example of Tree-Height Balancing

result to the tree's root, t_3. The code is shown in the margin.

Rebuild returns; *Balance* returns; and control is back in the call to *Flatten* on t_{11}. *Flatten* continues, until it hits t_7, and recurs on *Balance*(t_7).

$t_7 \leftarrow x + w$

For t_7, *Flatten* produces the queue: $\{\langle x,1\rangle,\langle w,1\rangle\}$. *Rebuild* dequeues $\langle x,1\rangle$ and $\langle w,1\rangle\}$; creates the operation $x+w$; and, since the queue is empty, assigns the result to the root, t_7, as shown in the margin.

$n_1 \leftarrow z + y$
$n_2 \leftarrow n_1 + t_7$
$t_{11} \leftarrow n_2 + t_3$

Rebuild returns; *Balance* returns; and control returns to *Flatten*(t_{11}). *Flatten* returns the queue: $\{\langle z,1\rangle,\langle y,1\rangle,\langle t_7,2\rangle\langle t_3,2\rangle\}$. *Rebuild* creates the operation $z+y$ and assigns its value to n_1. Next, it then creates the operation $n_1 + t_7$ and assigns its value to n_2. Finally, it creates the operation $n_2 + t_3$ and assigns its value to the tree's root, t_{11}.

Rebuild returns; *Balance* returns; and control flows back to the while loop. Phase 2 calls *Balance*(t_7), which finds that t_7 has been processed. It then calls *Balance*(t_3), which finds that t_3 has been processed.

$t_{10} \leftarrow t_7 \times t_3$

Next, the while loop invokes *Balance*(t_{10}). *Flatten* recurs on *Balance* for both t_3 and t_7. In both cases, the tree has already been processed. At that point, *Flatten* returns the queue: $\{\langle t_7,2\rangle, \langle t_3,2\rangle\}$. *Rebuild* creates the operation $t_7 \times t_3$ and, since the queue is now empty, assigns it to the tree's root, t_{10}. *Rebuild* returns. *Balance* returns.

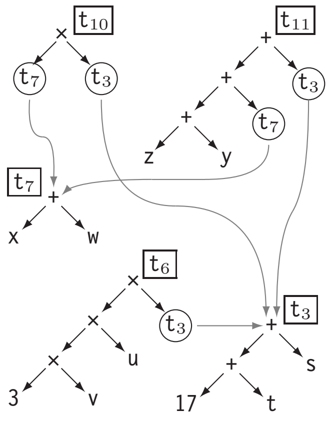
$n_3 \leftarrow 3 \times v$
$n_4 \leftarrow n_3 \times u$
$t_6 \leftarrow n_4 \times t_3$

The next, and final, iteration of the while loop invokes *Balance*(t_6). *Flatten* encounters t_3, which causes it to recur with *Balance*(t_3); since t_3 has been processed, the call returns immediately. *Flatten* returns the queue: $\{\langle 3,0\rangle,\langle v,1\rangle,\langle u,1\rangle,\langle t_3,2\rangle\}$. From this queue, *Rebuild* creates the operations shown in the margin. *Rebuild* returns; *Balance* returns; and the while loop terminates. Figure 8.11 shows the final result.

The difference between the forests of expression trees in Figures 8.10.b and 8.11.b may not be obvious. To appreciate the restruc-

$$
\begin{array}{lcl}
n_0 & \leftarrow & 17 + t \\
t_3 & \leftarrow & n_0 + s \\
t_7 & \leftarrow & x + w \\
n_1 & \leftarrow & z + y \\
n_2 & \leftarrow & n_1 + t_7 \\
t_{11} & \leftarrow & n_2 + t_3 \\
t_{10} & \leftarrow & t_7 \times t_3 \\
n_3 & \leftarrow & 3 \times v \\
n_4 & \leftarrow & n_3 \times u \\
t_6 & \leftarrow & n_4 \times t_3
\end{array}
$$

(a) Transformed Code



(b) Trees in the Transformed Code

FIGURE 8.11 Code Structure after Balancing

tured tree, compare the height of the tree for t_6, including the tree for t_3. In the original code, the tree has height seven where the restructured tree has height four. Similarly, the original tree for t_{11} has height five where the restructured tree has height four.

As a final point, the reader may observe that the individual trees in Figure 8.11.b do not look balanced. The algorithm creates balance across the entire block, rather than across the forests. This effect is due to a single line of code in *Flatten* from Figure 8.8. If *var* is a root, *Flatten* enqueues it with the rank it was previously assigned in *Balance*. If we modified that line of code to enqueue it with a rank of one, each of the subtrees would be balanced. The drawing might look better, but the overall height of trees that reference shared subtrees would be taller and the code would expose less ILP.

This effect is a simple example of the difference between optimizing for a local effect and optimizing over a larger context.