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.

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.

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.

**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 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 \( t_3 \).

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

<table>
<thead>
<tr>
<th>Adder 0</th>
<th>Adder 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( t_1 \leftarrow a + b )</td>
</tr>
<tr>
<td>2</td>
<td>( t_2 \leftarrow t_1 + c )</td>
</tr>
<tr>
<td>3</td>
<td>( t_3 \leftarrow t_2 + d )</td>
</tr>
<tr>
<td>4</td>
<td>( t_4 \leftarrow t_3 + e )</td>
</tr>
<tr>
<td>5</td>
<td>( t_5 \leftarrow t_4 + f )</td>
</tr>
<tr>
<td>6</td>
<td>( t_6 \leftarrow t_5 + g )</td>
</tr>
<tr>
<td>7</td>
<td>( t_7 \leftarrow t_6 + h )</td>
</tr>
</tbody>
</table>

(a) Left-Associative Tree

<table>
<thead>
<tr>
<th>Adder 0</th>
<th>Adder 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( t_1 \leftarrow a + b )</td>
</tr>
<tr>
<td>2</td>
<td>( t_3 \leftarrow e + f )</td>
</tr>
<tr>
<td>3</td>
<td>( t_5 \leftarrow t_1 + t_2 )</td>
</tr>
<tr>
<td>4</td>
<td>( t_7 \leftarrow t_5 + t_6 )</td>
</tr>
</tbody>
</table>

(b) Balanced Tree

**Figure 8.6** Schedules from Trees in Figure 8.5
// All operations have the form “$T_i \leftarrow L_i \text{Op}_i R_i$”

// Phase 1: build a queue, Roots, of the candidate trees

Roots ← new empty queue

for $i \leftarrow 0$ to $n - 1$

    Rank($T_i$) ← -1

    if $\text{Op}_i$ is commutative and associative and $T_i$ is exposed then

        mark $T_i$ as a root

        Enqueue(Roots, $T_i$, precedence of $\text{Op}_i$)

// 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

shown in the margin. Notice that $t_2$ is defined once and used twice. The trees for $t_2$ and $n$ are too small to benefit from tree-height balancing, while the tree for $n$, with four operands, can be balanced.

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 to indicate 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.

**Phase 1: Finding Candidate Trees**

Phase 1 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
8.4 Local Optimization

**Balance(root)**  // Create balanced tree from its root, \( T_i \)

if \( \text{Rank(root)} \geq 0 \) then
    return  // already processed this tree

\( q \leftarrow \text{new empty queue} \)  // Flatten the tree, then rebuild it

\( \text{Rank(root)} \leftarrow \text{Flatten}(L_i,q) + \text{Flatten}(R_i,q) \)

\( \text{Rebuild(root, q, Op_i)} \)

**Flatten(node,q)**  // Flatten computes a rank for node & builds the queue

if node is a constant then  // Constant is a leaf in the tree
    \( \text{Rank(node)} \leftarrow 0 \)
    \( \text{Enqueue(q,node,Rank(node))} \)

else if node is upward exposed then  // UE is a leaf in the tree
    \( \text{Rank(node)} \leftarrow 1 \)
    \( \text{Enqueue(q,node,Rank(node))} \)

else if node is a root  // Root of another tree is a leaf
    \( \text{Balance(node)} \)  // Balance sets \( \text{Rank(node)} \)
    \( \text{Enqueue(q,mpde,Rank(node))} \)

else  // interior node for \( T_i \leftarrow L_i \text{Op}_i R_i \)
    \( \text{Rank(node)} \leftarrow \text{Flatten}(L_i,q) + \text{Flatten}(R_i,q) \)

return \( \text{Rank(node)} \)

---

**Figure 8.8**  Tree-Height Balancing Algorithm: Balance and Flatten

the root’s value must be exposed.

To determine if the value \( T_i \) defined by operation \( i \) is exposed, the compiler must understand where \( T_i \) is used. The compiler can compute \textsc{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 \( T \leftarrow L \text{Op}_i R \) in block \( B \), \( T \) is a root if and only if:

- \( \text{Op}_i \) is both commutative and associative and
- \( (T \text{ is used more than once in } B \text{ or } T \in \text{LiveOut}(B)) \)

Phase 1 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 \( \times \) has precedence two.
\textbf{Rebuild}(\textit{root, q, op}) \quad // Build a balanced expression

\begin{algorithmic}
  \While{\textit{q} is not empty}
    \State \textit{NewL} $\leftarrow$ \text{Dequeue}(\textit{q}) \quad // Get a left operand
    \State \textit{NewR} $\leftarrow$ \text{Dequeue}(\textit{q}) \quad // Get a right operand
    \If{\textit{NewL} and \textit{NewR} are both constants}
      \State // evaluate and fold the result
      \State \textit{NewT} $\leftarrow$ \text{Fold}(\textit{op, NewL, NewR})
      \If{\textit{q} is empty}
        \State create the op \textbf{"root $\leftarrow$ NewT"}
        \State \text{Rank}(\textit{root}) = 0
      \Else
        \State \text{Enqueue}(\textit{q, NewT, 0})
        \State \text{Rank}(\textit{NewT}) = 0
      \EndIf
    \ElseIf{\textit{q} is empty}
      \State // not a constant; name the result
      \State \textit{NewT} $\leftarrow$ \textit{root}
    \Else
      \State \textit{NewT} $\leftarrow$ new name
      \State create the op \textbf{"NewT $\leftarrow$ NewL op NewR"}
      \State \text{Rank}(\textit{NewT}) $\leftarrow$ \text{Rank}(\textit{NewL}) + \text{Rank}(\textit{NewR})
      \If{\textit{q} is not empty} \quad // More ops in \textit{q} $\Rightarrow$ add \textit{NewT} to \textit{q}
        \State \text{Enqueue}(\textit{q, NewT, Rank(\textit{NewT}))}
      \EndIf
    \EndIf
  \EndWhile
\end{algorithmic}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure8_9.png}
\caption{Tree-Height Balancing Algorithm: Rebuild}
\end{figure}

\textbf{Phase 2: Rebuilding the Block in Balanced Form}

Phase 2 takes the queue of candidate-tree roots and builds, from each root, an approximately balanced tree. Phase 2 starts with a while loop that calls \textit{Balance} on each candidate tree root. \textit{Balance}, \textit{Flatten}, and \textit{Rebuild} implement phase two, as shown in Figures 8.8 and 8.9.

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

\textit{Flatten} is straightforward. It recursively walks the candidate tree, assigns each node a rank, and adds them to the queue. \textit{Flatten} assigns ranks directly to leaf nodes: zero for a constant and one for a leaf. If \textit{Flatten} encounters a root, it calls \textit{Balance} on that root; \textit{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 \textit{Balance} calls \textit{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.

The expressions trees and Roots queue for the ongoing example are shown in the margin. Phase 2 will pull the roots from the queue and pass them to Balance. Neither m nor t₂ have enough operands to allow the algorithm to improve the code.

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 t₆, t₁₀, and t₁₁ are live on exit from the block.

The block contains several intertwined computations. Panel (b) shows the five distinct expression trees in the block. The values of t₃ and t₇ are each used multiple times, so we show them as distinct trees. The connections to t₃ and t₇ 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 t₃ and t₇. The roots appear in boxes in panels (b) and (c). At the end of phase 1, Roots contains: \{ ⟨t₁₁,1⟩, ⟨t₇,1⟩, ⟨t₃,1⟩, ⟨t₁₀,2⟩, ⟨t₆,2⟩ \}.

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 t₁₁. Balance calls Flatten, which builds a queue. When Flatten encounters t₃, which is a root, it invokes Balance(t₃).

Applied to t₃, Balance first calls Flatten which produces the queue: \{ ⟨4,0⟩, ⟨13,0⟩, ⟨t,1⟩, ⟨s,1⟩ \}. Balance passes this queue to Rebuild.

Rebuild dequeues ⟨4,0⟩ and ⟨13,0⟩. Since both are constants, it adds them and enqueues ⟨17,0⟩. Next, it dequeues ⟨17,0⟩ and ⟨t,1⟩; creates the operation 17 + t; assigns the result to a new name, say n₀; and
enqueues \(\langle n_0, 1 \rangle\). Finally, \texttt{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, \texttt{Rebuild} assigns the result to the tree’s root, \(t_3\). The code is shown in the margin.

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

For \(t_7\), \texttt{Flatten} produces the queue: \{\(\langle x, 1 \rangle, \langle w, 1 \rangle\}\}. \texttt{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.

\texttt{Rebuild} returns; \texttt{Balance} returns; and control returns to \texttt{Flatten}(\(t_{12}\)).

\texttt{Flatten} returns the queue: \{\(\langle z, 1 \rangle, \langle y, 1 \rangle, \langle t_7, 2 \rangle, \langle t_3, 2 \rangle\}\}. \texttt{Rebuild} creates the operation \(z + y\) and assigns its value to \(n_1\). Next, it 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, \(t_{11}\).

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

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

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

**Figure 8.10** Detailed Example of Tree-Height Balancing

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n_0 \leftarrow 17 + t)</td>
<td>-</td>
</tr>
<tr>
<td>(t_3 \leftarrow n_0 + s)</td>
<td>-</td>
</tr>
<tr>
<td>(t_7 \leftarrow x + w)</td>
<td>-</td>
</tr>
<tr>
<td>(n_1 \leftarrow z + y)</td>
<td>-</td>
</tr>
<tr>
<td>(n_2 \leftarrow n_1 + t_7)</td>
<td>-</td>
</tr>
<tr>
<td>(t_{11} \leftarrow n_2 + t_3)</td>
<td>-</td>
</tr>
<tr>
<td>(t_{10} \leftarrow t_7 \times t_3)</td>
<td>-</td>
</tr>
<tr>
<td>(t_1 \leftarrow 13 + s)</td>
<td>-</td>
</tr>
<tr>
<td>(t_2 \leftarrow t_1 + t)</td>
<td>-</td>
</tr>
<tr>
<td>(t_3 \leftarrow t_2 + 4)</td>
<td>-</td>
</tr>
<tr>
<td>(t_4 \leftarrow t_3 \times u)</td>
<td>-</td>
</tr>
<tr>
<td>(t_5 \leftarrow 3 \times t_4)</td>
<td>-</td>
</tr>
<tr>
<td>(t_6 \leftarrow v \times t_5)</td>
<td>-</td>
</tr>
<tr>
<td>(t_7 \leftarrow w + x)</td>
<td>-</td>
</tr>
<tr>
<td>(t_8 \leftarrow t_7 + y)</td>
<td>-</td>
</tr>
<tr>
<td>(t_9 \leftarrow t_8 + z)</td>
<td>-</td>
</tr>
<tr>
<td>(t_{10} \leftarrow t_3 \times t_7)</td>
<td>-</td>
</tr>
<tr>
<td>(t_{11} \leftarrow t_3 + t_9)</td>
<td>-</td>
</tr>
</tbody>
</table>
The difference between the forests of expression trees in Figures 8.10.b and 8.11.b may not be obvious. To appreciate the restructured 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.